Technische Universität München

Zentrum Mathematik

Regression Models for Ordinal Valued Time Series: Estimation and Applications in Finance

Gernot Müller

Vollständiger Abdruck der von der Fakultät für Mathematik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Herbert Spohn
Prüfer der Dissertation:	1. Univ.-Prof. Claudia Czado, Ph.D.
	2. Univ.-Prof. Dr. Ludwig Fahrmeir, Ludwig-Maximilians-Universität München

Die Dissertation wurde am 7.4.2004 bei der Technischen Universität München eingereicht und durch die Fakultät für Mathematik am 23.6.2004 angenommen.

Acknowledgement

Foremost, I thank Prof. Dr. Claudia Czado for her constant support and encouragement and for many valuable suggestions and fruitful discussions during the last three years. I am also very grateful for many opportunities to travel and getting in touch with distinguished scientists in the fields of statistics and mathematical finance.

I want to thank the Deutsche Forschungsgemeinschaft for their financial support via the Sonderforschungsbereich 386.

Further I would like to thank my friends and colleagues at Munich University of Technology for the very comfortable working atmosphere.

Last but not least I thank my parents for their support in all situations and all my friends for their encouragement especially during the last months.

Abstract

Price changes arising in high-frequency financial data usually take on only values which are integer multiples of a certain amount, for example multiples of one sixteenth of a dollar. Therefore, the price changes represent an ordinal valued time series. Many of the common models cannot take this feature into account while also covering other features of such time series such as the dependency on covariates. Here two new models for ordinal valued time series with covariates are introduced. The first can be considered as an autoregressive extension of the common ordered probit model, the second as a discretized version of a stochastic volatility model. We investigate whether the estimation of the model parameters can be done by Markov Chain Monte Carlo (MCMC) methods. It is shown that in both cases standard MCMC algorithms have bad convergence properties. Therefore two grouped move multigrid Monte Carlo (GM-MGMC) samplers are developed which estimate the parameters accurate and fast. By applying both models to intraday data from the IBM stock at the New York Stock Exchange interesting dependencies of the price changes on covariates are detected and quantified. Implementations of the GM-MGMC samplers in C++ are provided.

Zusammenfassung

Preisveränderungen bei hochfrequenten Finanzdaten nehmen gewöhnlich nur Werte an, die ganzzahlige Vielfache von zum Beispiel einem 16tel Dollar sind. Daher stellen sie eine ordinale Zeitreihe dar. Viele der üblichen Modelle können diese Eigenschaft nicht gleichzeitig mit anderen Eigenschaften wie Abhängigkeiten von Kovariablen berücksichtigen. Hier werden zwei neue Modelle für ordinale Zeitreihen mit Kovariablen eingeführt. Das erste Modell ist ein autoregressives ordered probit Modell, das zweite ein diskretisiertes stochastisches Volatilitätsmodell. Da in beiden Fällen standard Markov Chain Monte Carlo Algorithmen schlechte Konvergenz zeigen, werden grouped move multigrid Monte Carlo (GM-MGMC) Sampler entwickelt, die die Parameter genau und schnell schätzen. Durch die Anwendung beider Modelle auf intraday Daten der IBM Aktie werden Abhängigkeiten der Preisveränderungen von Kovariablen quantifiziert. Implementierungen der GM-MGMC Sampler in C++ werden zur Verfügung gestellt.

Contents

Chapter 1

Introduction

Price changes arising in high-frequency financial data usually take on only few values which are integer multiples of a certain amount, for example multiples of one sixteenth of a dollar. Therefore the price changes over time represent an ordinal valued time series. Moreover, the price changes may depend on covariates such as the transaction volume or the time between trades. In addition, the volatility of the price change process is not constant, since one can observe periods with high price fluctuations which are followed by relative quite periods and vice versa. This effect is called volatility clustering.

Up to now, many models cannot take all these features of the price change process into account. Moreover, only few effort has been made to develop parameter driven models for such time series. In contrast to observation driven models, parameter driven models usually assume a latent process which does not depend on past observations. In general, parameter driven models allow easier interpretations than observation driven models. For price changes observation driven models were proposed by Hausman, Lo, and MacKinlay (1992) and Rydberg and Shephard (2003). Hausman, Lo, and MacKinlay (1992) applied the common ordered probit model to price changes of the IBM stock, although this model cannot capture a possibly present autoregressive structure in the latent process. Rydberg and Shephard (2003) suggested a decomposition model, where the price change is assumed to be a product of three random variables, namely of a price change indicator, of the direction and of the absolute value of the price change.

Many other models for high-frequency data do not focus on the price changes itself, but on the durations between the transactions. Recent developments in modeling durations are for example the ACD model of Engle and Russell (1998) and the many extensions such as the fractionally integrated ACD model of Jasiak (1998), the SCD model of

Bauwens and Veredas (2004), and the log-ACD model of Bauwens and Giot (2000). For a global overview about models for high-frequency financial data including models for returns and durations see Bauwens and Giot (2001) or Dacorogna et al. (2001).

The aim of this thesis is to develop new parameter driven models which cover all important features of the price change process. Efficient MCMC (Markov chain Monte Carlo) estimation procedures are to be provided for fitting the new models to specific data sets. By applying the models to data sets from the New York Stock Exchange, we want to find important covariates and to quantify their impact on the price changes.

We point out that the aim of modeling the price change process is mainly to understand the structure of the data. This knowledge then can be used for example by governing boards of exchanges which may adapt the trading process. However, the prediction of future price changes and the development of trading strategies based on predictions will in general be impossible for the following reason: Long-range dependencies (in this context 'long' means a period of about 10 transactions or more) do probably not exist (cf. for example Hausman et al. (1992) or Rydberg and Shephard (2003)) and therefore the prediction of price changes more than 10 steps ahead cannot be very accurate. Since, however, usually up to 20 transactions take place per minute and it will take half a minute in minimum to realize an order, a profitable trading strategy based on the microstructure of the price process cannot be developed.

In Chapter 2 we first summarize some basic results from statistics which are used in the subsequent chapters. In particular, we introduce important results in the context of Bayesian inference and Markov chain Monte Carlo methods. Moreover, we outline the ideas of Bayes factors for model selection and of non-standard statistical methods such as auxiliary particle filters.

In Chapter 3 we consider a model, which can be viewed as an extension of the common ordered probit (OP) model since it has the features of this model but also allows for an autoregressive time structure in the data. We call this model **autoregressive ordered probit (AOP) model**. For fitting the AOP model to a specific data set, we first develop a standard MCMC algorithm. This algorithm has a structure similar to the algorithm by Albert and Chib (1993) for the common OP model and exhibits similar problems in convergence and mixing. However, a method by Liu and Sabatti (2000) is applied to achieve better results. This method uses randomly drawn elements from an appropriate transformation group to make additional transformation steps in each MCMC iteration. We found a special transformation group which allowed for the development of a GM-MGMC (grouped move multi-grid Monte Carlo) sampler as introduced

by Liu and Sabatti (2000). The use of this GM-MGMC sampler leads to much more satisfying results as is shown in a simulation study.

In Chapter 4 we develop an estimation procedure for the marginal likelihood of the AOP model. This allows for the computation of Bayes factors to decide whether the AOP model can fit a given data set better than other models. Since the estimation of the marginal likelihood also requires a filtering procedure we provide an auxiliary particle filter for the new model. Auxiliary particle filters were introduced by Pitt and Shephard (1999). Since the OP model is a submodel of the AOP model, the algorithms can be easily simplified and applied also to the common ordered probit model. Therefore, for given data we are able to compare in particular the fit of the AOP model to the fit of the OP model.

In Chapter 5 we apply the AOP model to absolute price changes of the IBM stock on December 4, 2000, at the New York Stock Exchange (NYSE). This data set consists of 2000 observations. We search for significant covariates and quantify the autoregressive dependencies between the latent variables. By computing the Bayes factor of the AOP against the common OP model we show that the AOP model fits the data set decisively better than the OP model.

In Chapter 6 we introduce a model which is a discretized version of a stochastic volatility (SV) model. It can be applied to ordinal valued time series and therefore is called **ordinal-response stochastic volatility (OSV) model**. We consider a hybrid MCMC sampler for fitting the OSV model to specific data sets. As for the AOP model, there arise problems in convergence and mixing. Therefore we again develop a grouped move step and construct a GM-MGMC sampler for the OSV model. In a simulation study we examine the improvement in convergence.

In Chapter 7 the OSV model is applied to the price changes of the IBM stock in a period of nine days. This data set consists of more than 22000 observations. We detect significant covariates and quantify their impact on the price change process. The results are compared with theoretical considerations of the trading process which have been undertaken by Diamond and Verrecchia (1987), Easley and O'Hara (1987), and Tauchen and Pitts (1983).

In Chapter 8 we consider an extension of the OSV model which allows for Student-t distributed errors and which therefore is called **OSVt** model. We answer the question whether it is more accurate to use errors from a t-distribution with unknown degrees of freedom instead of normally distributed errors, as is often the case in modeling financial

time series since the t-distribution has heavier tails than the normal distribution. The GM-MGMC algorithm is derived from the corresponding algorithm for the OSV model developed in Chapter 7. In a simulation study we show that the GM-MGMC sampler leads to accurate estimates for all parameters, in particular for the unknown degrees of freedom of the t-distribution. Finally we fit the OSVt model to the same data set which is investigated in Chapter 7.

In Appendix A the Nelder-Mead method, a minimization algorithm from numerics, is described. The Appendices B and C contain C++ implementations of the GM-MGMC samplers for the AOP and the OSV/OSVt model, respectively. These implementations use procedures for random variable generation and related things and a matrix class which can be found in Appendices D and E, respectively.

Chapter 2

Statistical Foundations

Here we summarize some foundations from statistics and stochastics which are used in the following chapters. After stating results for random variable generation, we recall the basic ideas behind Bayesian inference and the Markov chain Monte Carlo (MCMC) methods which are important especially for Chapters 3, 6, and 8. Then we deal with auxiliary particle filters and Bayes factors which are used in Chapters 4 and 5. We summarize the definitions and some properties of the ordered probit model, the stochastic volatility model, and the family of state space models, which serve as basic models. Finally we have a look at the features of stock price changes, since all non-simulated data sets investigated in the following chapters represent such data. Since the Nelder-Mead optimization procedure which is used in Chapter 6 is an algorithm from numerics, it is not described here, but in the Appendix.

We note that all symbols and abbreviations which appear in the text are summarized in an index after the appendix. Mostly we use the terms distribution and density interchangeably.

2.1 Random variable generation

A good way to construct efficient simulation methods is to represent a distribution as a mixture of other distributions. Let $f(x)$ denote the density of a random variable X and $g(x, y)$ the joint density of X and another random variable Y. If Y is continuous, f can

be represented as the marginal of the density g in the form

$$f(x) = \int g(x, y)\, dy.$$

Another useful representation is

$$f(x) = \int h_1(x|y) h_2(y)\, dy \tag{2.1}$$

where h_1 and h_2 are the conditional and the marginal density of $X|Y = y$ and Y, respectively. For a discrete variable Y, representation (2.1) corresponds to

$$f(x) = \sum_y f_y(x) p_y$$

with component distributions $f_y(x)$ and probabilities $p_y := P(Y = y)$.

These equations induce directly several well-known possibilities to draw from f which we summarize in the following Algorithm:

Algorithm 2.1 Random variable generation via mixture representations

1. If Y is continuous and the joint distribution $g(x, y)$ is simple to simulate from, generate a sample (x, y) from the joint distribution and obtain the sample x from f as the first component of (x, y).

2. If Y is discrete and the component distributions $f_y(x)$ are simple to simulate from, choose y with probability p_y and generate an observation from f_y.

3. If the conditional distribution h_1 of $X|Y = y$ is simple to simulate from as well as the marginal distribution h_2 of Y, generate a sample y from the marginal distribution of Y and then generate x from the conditional distribution of $X|Y = y$.

A well-known example for the third possibility is the sampling from the Student-t distribution with ν degrees of freedom, which can be done by first drawing y from a χ^2_ν distribution and then x from a normal distribution depending on y.

Another often used method for drawing random variables is **rejection sampling**. Given a density of interest f, the first requirement is the determination of a density g and a constant M such that

$$f(x) \leq Mg(x)$$

on the support of f. Then the procedure given in Algorithm 2.2 is called rejection sampling:

Algorithm 2.2 Rejection sampling

1. Generate $x \sim g$ and $u \sim \text{Unif}(0, 1)$.
2. If $u \leq f(x)/(Mg(x))$, consider x as a sample from f.
3. Else, return to 1.

The proof, that this algorithm produces a sample from f, is given in Robert and Casella (2000), Chapter 2.

Now we consider the special case that one wants to sample from a **truncated distribution** f_S, which is proportional to a distribution f on a subset S of the support of f and zero elsewhere. Here one can use $g = f$ and set M to the proportionality constant, which equals $f_S(x)/f(x)$ for all $x \in S$. Then we have, as required,

$$f_S(x) = \frac{f_S(x)}{f(x)} f(x) = Mg(x)$$

on the support of f_S. Now consider the fraction

$$\frac{f_S(x)}{Mf(x)}$$

which is evaluated in Step 2 of Algorithm 2.2. On the support of f_S the fraction equals 1, since on S we have $M = f_S(x)/f(x)$. If $x \notin S$, then $f_S(x) = 0$, so that the fraction equals 0. Hence, for a target distribution f_S which is a truncated version of another distribution f, rejection sampling reduces to sampling from the non-truncated distribution f until the sample lies in S. We note that although this is a simple method to draw from truncated distributions, it can take a long time to get a sample, when the probability under f to get a sample in S is small.

For more details on random variable generation we refer to Robert and Casella (2000).

2.2 Bayesian inference

Here we introduce the basic ideas of Bayesian inference. A comprehensive treatment of this topic is given for example in Gelman et al. (1995).

In contrast to the classical frequentist approach, from a Bayesian perspective there is no fundamental distinction between random variables and parameters of a statistical

model: all are considered random quantities. Let $\boldsymbol{y}$ denote the vector of observations, and $\boldsymbol{\theta}$ the vector of all parameters and latent (i.e. unobservable) variables in the model. Formal inference then requires setting up a joint probability distribution $p(\boldsymbol{y}, \boldsymbol{\theta})$ over all random quantities. This joint distribution comprises two parts: a **prior distribution** $\pi(\boldsymbol{\theta})$ which represents the uncertainty about $\boldsymbol{\theta}$ before observing $\boldsymbol{y}$, and a **likelihood** $f(\boldsymbol{y}|\boldsymbol{\theta})$. Specifying $\pi(\boldsymbol{\theta})$ and $f(\boldsymbol{y}|\boldsymbol{\theta})$ gives a **full probability model**, in which

$$\pi(\boldsymbol{y}, \boldsymbol{\theta}) = f(\boldsymbol{y}|\boldsymbol{\theta})\pi(\boldsymbol{\theta}).$$

Since the data $\boldsymbol{y}$ contains information about $\boldsymbol{\theta}$ one can use $\boldsymbol{y}$ to update the information about $\boldsymbol{\theta}$ by determining the distribution of $\boldsymbol{\theta}$ conditional on $\boldsymbol{y}$. Using the Bayes theorem, this distribution is given by

$$\pi(\boldsymbol{\theta}|\boldsymbol{y}) = \frac{f(\boldsymbol{y}|\boldsymbol{\theta})\pi(\boldsymbol{\theta})}{\int f(\boldsymbol{y}|\boldsymbol{\theta})\pi(\boldsymbol{\theta})\, d\boldsymbol{\theta}}. \tag{2.2}$$

It is called **posterior distribution** of $\boldsymbol{\theta}$, and is the object of all Bayesian inference.

The evaluation of the integral in Equation (2.2) is the source of most of the practical difficulties in Bayesian inference, especially in high dimensions. In most applications, analytic evaluation is impossible. Numerical evaluation is difficult and inaccurate in more than about 20 dimensions. Therefore one often uses **Monte Carlo** integration or **Markov chain Monte Carlo** (MCMC) methods (cf. Section 2.3) to approximate the posterior distribution. These methods also take advantage of the fact, that the integral in Equation (2.2) equals $f(\boldsymbol{y})$ and therefore does not depend on $\boldsymbol{\theta}$, which leads to the fundamental proportionality

$$\pi(\boldsymbol{\theta}|\boldsymbol{y}) \propto f(\boldsymbol{y}|\boldsymbol{\theta})\pi(\boldsymbol{\theta}). \tag{2.3}$$

Now we introduce some further notations for the prior and posterior distributions. A prior or posterior density is called **proper**, if it integrates to any positive finite value c. If $c = 1$, it is called **normalized**, otherwise **unnormalized**. Of course, in the latter case it easily can be renormalized by multiplication by c^{-1}. A prior or posterior density is called **improper**, if it integrates to ∞. Improper priors often lead to proper posterior distributions, but they always should be used with care. A prior with $\pi(\boldsymbol{\theta}) \propto \mathbb{1}_S(\boldsymbol{\theta})$, $\boldsymbol{\theta} \in \mathbb{R}^k$, $S \subset \mathbb{R}^k$, is called **noninformative on the support** S, or shortly **noninformative**. However, we point out that also the choice of the support S may reflect some prior information about $\boldsymbol{\theta}$. Usually the prior distribution is taken from a parametric family, for example the family of normal distributions. The parameters which parameterize this family (in the normal distribution case the mean and variance) and which determine the exact prior distribution, are called **hyperparameters**.

As mentioned above, the object of Bayesian inference is the posterior distribution for the vector $\boldsymbol{\theta}$. Here one is interested in location measures such as the **posterior mean** and the **posterior mode** and in scale measures such as the **posterior standard deviation**. We note that, according to (2.3), the posterior mode estimate formally coincides with the maximum likelihood estimate when a noninformative prior is used.

Instead of the confidence intervals considered in the classical approach one can determine **credible intervals** for each component θ_j of $\boldsymbol{\theta}$. A $100(1-\alpha)\%$ credible interval for a parameter $\theta_j \in \mathbb{R}$ is an interval $I := [I_{\text{left}}, I_{\text{right}}]$ for which

$$\int_I \pi(\theta_j|\boldsymbol{y})\, d\theta_j = 1-\alpha \tag{2.4}$$

where $\pi(\theta_j|\boldsymbol{y})$ denotes the marginal posterior for θ_j. For a parameter which can take on only values from a discrete set A one can use the condition

$$\sum_{i\in B} \pi(\theta_j = i\,|\boldsymbol{y}) \geq 1-\alpha$$

for a set $B \subset A$ instead of (2.4). In general, for each parameter θ_j there exist many intervals which satisfy the corresponding condition. However, usually one chooses I_{left} to be the $\alpha/2$-quantile and I_{right} to be the $(1-\alpha/2)$-quantile of the corresponding marginal posterior distribution.

In the following section we summarize the basic concepts of the MCMC methods which will be used later for the approximation of posterior densities.

2.3 Markov chain Monte Carlo (MCMC) methods

In this section we give a short introduction to Markov chain Monte Carlo (MCMC) methods which are used in the following chapters to draw samples from posterior distributions. We start by describing the two underlying concepts: Monte Carlo integration and Markov chains. Then we describe the basic MCMC algorihms: The Metropolis-Hastings algorithm and its most important special case, the Gibbs sampler. Finally we summarize the idea of reduced runs of a Gibbs sampler, a concept which employs the original Gibbs sampler with only slight modifications to sample from other distributions than the posterior distribution. For a detailed introduction to MCMC we refer to Gilks et al. (1996) and Robert and Casella (2000).

2.3.1 Classical Monte Carlo integration

The classical Monte Carlo integration is a simulation-based method to evaluate an integral of the form

$$\mathbf{E}_f[h(X)] = \int_{\mathcal{X}} h(x)f(x)dx$$

where f is a density and h a function which transforms the random variable X. This integral can be approximated by generating a sample $x_1, \ldots, x_M$ from the density f and then computing the average

$$\bar{h}_M = \frac{1}{M}\sum_{m=1}^{M} h(x_m)$$

since $\bar{h}_M$ converges almost surely to $\mathbf{E}_f[h(X)]$ by the Strong Law of Large Numbers (cf. Breiman (1992), Ch. 3). Moreover, when h^2 has a finite expectation under f, the expression

$$\frac{\bar{h}_M - \mathbf{E}_f[h(X)]}{\sqrt{v_M}}$$

with $v_M = M^{-2}\sum_{m=1}^{M}[h(x_m) - \bar{h}_M]^2$ converges in distribution to the standard normal distribution $N(0,1)$ by the Central Limit Theorem (cf. Breiman (1992), Ch. 8). This can be used for the construction of confidence bounds on the approximation of $\mathbf{E}_f[h(X)]$.

2.3.2 Markov chains

A **Markov chain** is a collection of random variables or random vectors $\{\boldsymbol{X}_i \,|\, i \in M\}$ where usually $M = \mathbb{N}$. The evolution of the Markov chain on a space $\Omega \subset \mathbb{R}^p$ is governed by the **transition kernel**

$$\begin{aligned} P(\boldsymbol{x}, A) &:= \Pr(\boldsymbol{X}_{i+1} \in A \,|\, \boldsymbol{X}_i = \boldsymbol{x}, \boldsymbol{X}_j, j < i) \\ &= \Pr(\boldsymbol{X}_{i+1} \in A \,|\, \boldsymbol{X}_i = \boldsymbol{x}) \qquad (\boldsymbol{x} \in \Omega,\ A \subset \Omega) \end{aligned} \tag{2.5}$$

which embodies the Markov assumption that the distribution of each succeeding state in the sequence, given the current and the past states, depends only on the current state.

In general, the transition kernel has both a continuous and a discrete component. For some function $p : \Omega \times \Omega \to [0, \infty)$, the kernel can be expressed as

$$P(\boldsymbol{x}, d\boldsymbol{y}) = p(\boldsymbol{x}, \boldsymbol{y})d\boldsymbol{y} + r(\boldsymbol{x})\mathbb{1}_{d\boldsymbol{y}}(\boldsymbol{x}) \tag{2.6}$$

where $p(\boldsymbol{x}, \boldsymbol{x}) = 0$ and $r(\boldsymbol{x}) = 1 - \int_\Omega p(\boldsymbol{x}, \boldsymbol{y})d\boldsymbol{y}$. Therefore the transition from $\boldsymbol{x}$ to $\boldsymbol{y}$ occurs according to $p(\boldsymbol{x}, \boldsymbol{y})$, and the transition from $\boldsymbol{x}$ to $\boldsymbol{x}$ occurs with probability $r(\boldsymbol{x})$.

Following Equation (2.5), the transition kernel provides the distribution of $\boldsymbol{X}_{i+1}$ given that $\boldsymbol{X}_i = \boldsymbol{x}$. The nth-step-ahead transition kernel is given by

$$P^{(n)}(\boldsymbol{x}, A) = \int_{\Omega} P^{(n-1)}(\boldsymbol{y}, A) P(\boldsymbol{x}, d\boldsymbol{y})$$

where $P^{(1)}(\boldsymbol{x}, d\boldsymbol{y}) = P(\boldsymbol{x}, d\boldsymbol{y})$ and

$$P(\boldsymbol{x}, A) = \int_{A} P(\boldsymbol{x}, d\boldsymbol{y}).$$

Under certain conditions stated below the distribution given by the nth iterate of the transition kernel converges to the **invariant distribution** π^* for $n \to \infty$. This invariant distribution satisfies

$$\pi^*(d\boldsymbol{y}) = \int_{\Omega} P(\boldsymbol{x}, d\boldsymbol{y}) \pi(\boldsymbol{x})\, d\boldsymbol{x} \tag{2.7}$$

where π is the density of π^* with respect to the Lebesque measure, i.e. $\pi^*(d\boldsymbol{y}) = \pi(\boldsymbol{y}) d\boldsymbol{y}$. The invariance condition states that if $\boldsymbol{X}_i$ is distributed according to π^*, then all subsequent elements of the chain are also distributed according to π^*.

A Markov chain is called **reversible** if the function $p(\boldsymbol{x}, \boldsymbol{y})$ in Equation (2.6) satisfies

$$f(\boldsymbol{x}) p(\boldsymbol{x}, \boldsymbol{y}) = f(\boldsymbol{y}) p(\boldsymbol{y}, \boldsymbol{x}) \quad \forall \boldsymbol{x}, \boldsymbol{y}$$

for a density $f(\,\cdot\,)$. If this condition holds, then $f(\,\cdot\,)$ is the density of an invariant distribution. This follows from Equations (2.6) and (2.7) since for all sets $A \subset \Omega$

$$\begin{aligned}
\int P(\boldsymbol{x}, A) f(\boldsymbol{x})\, d\boldsymbol{x} &= \iint_A p(\boldsymbol{x}, \boldsymbol{y}) f(\boldsymbol{x})\, d\boldsymbol{y}\, d\boldsymbol{x} + \int_A r(\boldsymbol{x}) f(\boldsymbol{x})\, d\boldsymbol{x} \\
&= \int_A \int p(\boldsymbol{y}, \boldsymbol{x}) f(\boldsymbol{y})\, d\boldsymbol{x}\, d\boldsymbol{y} + \int_A r(\boldsymbol{x}) f(\boldsymbol{x})\, d\boldsymbol{x} \\
&= \int_A \left[1 - r(\boldsymbol{y})\right] f(\boldsymbol{y})\, d\boldsymbol{y} + \int_A r(\boldsymbol{x}) f(\boldsymbol{x})\, d\boldsymbol{x} \\
&= \int_A f(\boldsymbol{y})\, d\boldsymbol{y}.
\end{aligned}$$

Another important notion is that of π^*-irreducibility, where π^* is a probability measure. This is the requirement that the chain is able to visit all sets with positive probability under π^* from any starting point in Ω. Formally, a Markov chain is said to be **π^*-irreducible** if for every $\boldsymbol{x} \in \Omega$

$$\pi^*(A) > 0 \Longrightarrow P(\boldsymbol{X}_i \in A \,|\, \boldsymbol{X}_0 = \boldsymbol{x}) > 0$$

for some $i \geq 1$.

A Markov chain is called **aperiodic** if there does not exist a partition $(\Omega_0, \ldots, \Omega_{k-1})$ of Ω for some $k \geq 2$ such that

$$P(\boldsymbol{X}_i \in \Omega_{i \bmod k} \,|\, \boldsymbol{X}_0 \in \Omega_0) = 1 \quad \forall\, i.$$

Hence, the aperiodicity of a chain ensures that the chain does not cycle through a finite number of sets.

These definitions allow us to state the following results, which form the basis for Markov chain Monte Carlo methods. The first result gives conditions under which a strong law of large numbers holds. For a proof see Tierney (1994), Corollary 1 and Theorem 3. The second result gives conditions under which the probability density of the Mth iterate converges to its unique, invariant density (cf. Tierney (1994), Theorem 1).

Theorem 2.1
Suppose $\{\boldsymbol{X}_i\}$ is a π^*-irreducible, aperiodic Markov chain with transition kernel $P(\,\cdot\,,\,\cdot\,)$ and invariant distribution π^*. If $P(\boldsymbol{x},\,\cdot\,)$ is absolutely continuous with respect to π^* for all $\boldsymbol{x} \in \Omega$, then π^* is the unique invariant distribution of $P(\,\cdot\,,\,\cdot\,)$ and for all π^*-integrable real-valued functions h,

$$\frac{1}{M}\sum_{i=1}^{M} h(\boldsymbol{X}_i) \longrightarrow \int h(\boldsymbol{x})\pi(\boldsymbol{x})\, d\boldsymbol{x} \qquad \text{as } M \to \infty,\ \text{a.s.}$$

Theorem 2.2
Suppose $\{\boldsymbol{X}_i\}$ is a π^*-irreducible, aperiodic Markov chain with transition kernel $P(\,\cdot\,,\,\cdot\,)$ and invariant distribution π^*. Then for π^*-almost every $\boldsymbol{x} \in \Omega$, and all sets A

$$\parallel P^M(\boldsymbol{x}, A) - \pi^*(A) \parallel \longrightarrow 0 \qquad \text{as } M \to \infty$$

where $\parallel \cdot \parallel$ denotes the total variation distance.

The MCMC methods considered in the following sections always produce chains which have an invariant distribution by construction. Therefore the existence of the invariant distribution does not have to be checked in any particular application of MCMC methods. The other conditions appearing in Theorems 2.1 and 2.2 can also easily be satisfied, as is stated in the following section. More convergence results in the context of Markov chain Monte Carlo methods can be found in Robert and Casella (2000).

2.3.3 Metropolis-Hastings algorithm

The Metropolis-Hastings (MH) method is a general MCMC method to produce sample variates from a given multivariate target density. In the Bayesian context the target

density is usually the posterior density. Like other MCMC methods the MH-algorithm produces a realization of a Markov chain which has the given target density as invariant distribution.

The MH method employs a proposal density which is used to supply a proposal value and a probability of move. This probability is based on the ratio of the target density (evaluated at the proposal value in the numerator and the current value in the denominator) times the ratio of the proposal density (at the current value in the numerator and the proposal value in the denominator). Since only ratios of the target density are involved, knowledge of the normalizing constant of the target density is not required.

In particular, the goal is to simulate from the d-dimensional distribution π^* that has density $\pi(\boldsymbol{\theta})$, $\boldsymbol{\theta} \in \mathbb{R}^d$, with respect to some dominating measure. To define the algorithm, let $q(\boldsymbol{\theta}^\bullet, \boldsymbol{\theta}^\circ)$ denote the proposal density which is used to supply a proposal value $\boldsymbol{\theta}^\circ$ given the current value $\boldsymbol{\theta}^\bullet$, and let $\alpha(\boldsymbol{\theta}^\bullet, \boldsymbol{\theta}^\circ)$ denote the function

$$\alpha(\boldsymbol{\theta}^\bullet, \boldsymbol{\theta}^\circ) := \begin{cases} \min\left[\dfrac{\pi(\boldsymbol{\theta}^\circ) q(\boldsymbol{\theta}^\circ, \boldsymbol{\theta}^\bullet)}{\pi(\boldsymbol{\theta}^\bullet) q(\boldsymbol{\theta}^\bullet, \boldsymbol{\theta}^\circ)}, 1\right] & \text{if } \pi(\boldsymbol{\theta}^\bullet) q(\boldsymbol{\theta}^\bullet, \boldsymbol{\theta}^\circ) > 0, \\ 1 & \text{otherwise.} \end{cases}$$

In algorithmic form, the simulated values are obtained by the following recursive procedure.

Algorithm 2.3 Metropolis-Hastings algorithm

1. Specify an initial value $\boldsymbol{\theta}^{(0)}$.
2. Repeat for $i = 1, \ldots, M$
   - Draw the proposal value $\boldsymbol{\theta}^\circ$ from $q(\boldsymbol{\theta}^{(i-1)}, \cdot)$.
   - Draw a sample $u^{(i)}$ from the uniform distribution Unif$(0, 1)$.
   - Let
   $$\boldsymbol{\theta}^{(i)} := \begin{cases} \boldsymbol{\theta}^\circ & \text{if } u^{(i)} \leq \alpha(\boldsymbol{\theta}^{(i-1)}, \boldsymbol{\theta}^\circ), \\ \boldsymbol{\theta}^{(i-1)} & \text{otherwise.} \end{cases}$$
3. Return the values $\{\boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(2)}, \ldots, \boldsymbol{\theta}^{(M)}\}$.

The Markov chain produced by the MH algorithm has under general conditions the limiting distribution π^*. Therefore, the variates are from π^* only in the limit as the number of iterations becomes large but, in practice, after an initial **burn-in** phase the

chain is assumed to have converged and subsequent values are taken as approximate draws from π^*.

There exist many conditions which can be shown to be sufficient for irreducibility and aperiodicity of the MH-chain. For example, the following quite unrestrictive condition is given in Roberts and Tweedie (1996).

Proposition 2.1
Assume that the target density π is bounded and positive on every compact set of its support S. If there exist positive numbers ε and δ such that

$$q(\boldsymbol{\theta}^\bullet, \boldsymbol{\theta}^\circ) > \varepsilon \qquad \text{if } \| \boldsymbol{\theta}^\bullet - \boldsymbol{\theta}^\circ \| < \delta,$$

then the MH-chain is irreducible and aperiodic.

For more conditions which guarantee irreducibility and aperiodicity see Robert and Casella (2000), Chapters 4 and 6.

In applications when the dimension of $\boldsymbol{\theta}$ is large it is preferable to construct the Markov chain by first grouping $\boldsymbol{\theta}$ into n blocks, i.e. $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_n)$, where $\boldsymbol{\theta}_k \in \mathbb{R}^{d_k}$, and sampling each block, conditioned on the remaining blocks, by the MH-algorithm. Let $\{q_k(\boldsymbol{\theta}_k^\bullet, \boldsymbol{\theta}_k^\circ | (\boldsymbol{\theta}_1^\bullet, \ldots, \boldsymbol{\theta}_{k-1}^\bullet, \boldsymbol{\theta}_{k+1}^\bullet, \ldots, \boldsymbol{\theta}_n^\bullet)) \,|\, 1 \le k \le n\}$ denote a collection of proposal densities. Then the **multiple-block** MH-algorithm can be summarized as follows.

Algorithm 2.4 Multiple-block Metropolis-Hastings algorithm

1. Specify an initial value $\boldsymbol{\theta}^{(0)} = (\boldsymbol{\theta}_1^{(0)}, \ldots, \boldsymbol{\theta}_n^{(0)})$.

2. Repeat for $i = 1, \ldots, M$
   Repeat for $k = 1, \ldots, n$

   - Draw the proposal value $\boldsymbol{\theta}_k^\circ$ from $q_k(\boldsymbol{\theta}_k^{(i-1)}, \cdot \mid \boldsymbol{\theta}_{-k}^{(i)})$, where

   $$\boldsymbol{\theta}_{-k}^{(i)} := (\boldsymbol{\theta}_1^{(i)}, \ldots, \boldsymbol{\theta}_{k-1}^{(i)}, \boldsymbol{\theta}_{k+1}^{(i-1)}, \ldots, \boldsymbol{\theta}_n^{(i-1)}).$$

   - Draw a sample $u_k^{(i)}$ from Unif$(0, 1)$.

   - Let

   $$\boldsymbol{\theta}_k^{(i)} := \begin{cases} \boldsymbol{\theta}_k^\circ & \text{if } u_k^{(i)} \le \alpha_k(\boldsymbol{\theta}_k^{(i-1)}, \boldsymbol{\theta}_k^\circ | \boldsymbol{\theta}_{-k}^{(i)}), \\ \boldsymbol{\theta}_k^{(i-1)} & \text{otherwise,} \end{cases}$$

where

$$\alpha_k(\boldsymbol{\theta}_k^{(i-1)}, \boldsymbol{\theta}_k^{\circ}|\boldsymbol{\theta}_{-k}^{(i)}) := \begin{cases} \min\left[\dfrac{\pi(\boldsymbol{\theta}_k^{\circ}, \boldsymbol{\theta}_{-k}^{(i)}) q_k(\boldsymbol{\theta}_k^{\circ}, \boldsymbol{\theta}_k^{(i-1)}|\boldsymbol{\theta}_{-k}^{(i)})}{\pi(\boldsymbol{\theta}_k^{(i-1)}, \boldsymbol{\theta}_{-k}^{(i)}) q_k(\boldsymbol{\theta}_k^{(i-1)}, \boldsymbol{\theta}_k^{\circ}|\boldsymbol{\theta}_{-k}^{(i)})}, 1\right] \\ \qquad \text{if } \pi(\boldsymbol{\theta}_k^{(i-1)}, \boldsymbol{\theta}_{-k}^{(i)}) q_k(\boldsymbol{\theta}_k^{(i-1)}, \boldsymbol{\theta}_k^{\circ}|\boldsymbol{\theta}_{-k}^{(i)}) > 0, \\ 1 \qquad \text{otherwise.} \end{cases} \tag{2.8}$$

3. Return the values $\{\boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(2)}, \ldots, \boldsymbol{\theta}^{(M)}\}$.

As for example in Chib and Greenberg (1994), the proposal densities q_k can be tailored to the actual values of $\boldsymbol{\theta}$ and can vary across iterations.

2.3.4 Gibbs sampler

The Gibbs sampler is a special case of the multiple-block MH-algorithm. Here one uses the **full conditional** distribution defined by

$$\pi(\,\cdot\,|\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_{k-1}, \boldsymbol{\theta}_{k+1}, \ldots, \boldsymbol{\theta}_n), \qquad k = 1, \ldots, n,$$

as proposal density for $\boldsymbol{\theta}_k$. Obviously this proposal density is independent from the previous value $\boldsymbol{\theta}_k^{(i-1)}$ of $\boldsymbol{\theta}_k$. Since $\pi(\boldsymbol{\theta}_k|\boldsymbol{\theta}_{-k}) \propto \pi(\boldsymbol{\theta}_k, \boldsymbol{\theta}_{-k})$ the acceptance probability in Equation (2.8) becomes

$$\alpha_k(\boldsymbol{\theta}_k^{(i-1)}, \boldsymbol{\theta}_k^{\circ}|\boldsymbol{\theta}_{-k}^{(i)}) \;=\; \min\left\{\frac{\pi(\boldsymbol{\theta}_k^{\circ}, \boldsymbol{\theta}_{-k}^{(i)})\pi(\boldsymbol{\theta}_k^{(i-1)}, \boldsymbol{\theta}_{-k}^{(i)})}{\pi(\boldsymbol{\theta}_k^{(i-1)}, \boldsymbol{\theta}_{-k}^{(i)})\pi(\boldsymbol{\theta}_k^{\circ}, \boldsymbol{\theta}_{-k}^{(i)})}, 1\right\} \;=\; 1$$

which means that each proposed value is accepted with probability 1. Hence the Gibbs sampler can be summarized as follows.

Algorithm 2.5 Gibbs sampler

1. Specify an initial value $\boldsymbol{\theta}^{(0)} = (\boldsymbol{\theta}_1^{(0)}, \ldots, \boldsymbol{\theta}_n^{(0)})$.
2. Repeat for $i = 1, \ldots, M$
   - Generate $\boldsymbol{\theta}_1^{(i)}$ from $\pi(\,\cdot\,|\boldsymbol{\theta}_2^{(i-1)}, \ldots, \boldsymbol{\theta}_n^{(i-1)})$.
   - Generate $\boldsymbol{\theta}_2^{(i)}$ from $\pi(\,\cdot\,|\boldsymbol{\theta}_1^{(i)}, \boldsymbol{\theta}_3^{(i-1)}, \ldots, \boldsymbol{\theta}_n^{(i-1)})$.

     $\vdots$

- Generate $\boldsymbol{\theta}_n^{(i)}$ from $\pi(\cdot|\boldsymbol{\theta}_1^{(i)}, \ldots, \boldsymbol{\theta}_{n-1}^{(i)})$.

3. Return the values $\{\boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(2)}, \ldots, \boldsymbol{\theta}^{(M)}\}$.

When not all full conditionals can be determined explicitly, one can update some blocks by an MH-step. Such multiple-block MH-algorithms in which only some of the blocks are sampled using the full conditional distributions are called **hybrid (MCMC) algorithms**.

2.3.5 Reduced runs

After burn-in an iteration i of the Gibbs sampler generates a vector $(\boldsymbol{\theta}_1^{(i)}, \ldots, \boldsymbol{\theta}_n^{(i)})$ which is considered as a sample from the full posterior distribution $\pi(\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_n|\boldsymbol{y})$. Here again all components $\boldsymbol{\theta}_j$ are allowed to be 1- or multi-dimensional.

If $\{k_1, \ldots, k_m\}$ is a subset of $\{1, \ldots, n\}$ and one is interested in a sample from $\pi(\boldsymbol{\theta}_{k_1}, \ldots, \boldsymbol{\theta}_{k_m}|\boldsymbol{y})$, it is sufficient to take the sample $(\boldsymbol{\theta}_1^{(i)}, \ldots, \boldsymbol{\theta}_n^{(i)})$, and to consider the vector $(\boldsymbol{\theta}_{k_1}^{(i)}, \ldots, \boldsymbol{\theta}_{k_m}^{(i)})$ as a sample from $\pi(\boldsymbol{\theta}_{k_1}, \ldots, \boldsymbol{\theta}_{k_m}|\boldsymbol{y})$. This follows directly from Algorithm 2.1, 1. Therefore one can use the same Gibbs sampler to produce samples from $\pi(\boldsymbol{\theta}_{k_1}, \ldots, \boldsymbol{\theta}_{k_m}|\boldsymbol{y})$.

Now suppose that $(k_1, \ldots, k_m, k_{m+1}, \ldots, k_l, k_{l+1}, \ldots, k_n)$ is any permutation of the first n positive integers $(1, 2, 3, \ldots, n)$. If $\boldsymbol{\theta}_{k_{m+1}}^{\diamond}, \ldots, \boldsymbol{\theta}_{k_l}^{\diamond}$ are known and one is interested in a sample from $\pi(\boldsymbol{\theta}_{k_1}, \ldots, \boldsymbol{\theta}_{k_m}|\boldsymbol{\theta}_{k_{m+1}}^{\diamond}, \ldots, \boldsymbol{\theta}_{k_l}^{\diamond}, \boldsymbol{y})$, one can use a so-called **reduced run** of the Gibbs sampler. That means that one fixes the values $\boldsymbol{\theta}_{k_{m+1}}^{\diamond}, \ldots, \boldsymbol{\theta}_{k_l}^{\diamond}$ and runs the original Gibbs sampler without updating $\boldsymbol{\theta}_{k_{m+1}}, \ldots, \boldsymbol{\theta}_{k_l}$. We emphasize that not only the components with indices $k_1, \ldots, k_m$ must be updated, but all with indices $k_{l+1}, \ldots, k_n$, too. Of course, each iteration of the reduced run of the Gibbs sampler delivers a sample from

$$\pi(\boldsymbol{\theta}_{k_1}, \ldots, \boldsymbol{\theta}_{k_m}, \boldsymbol{\theta}_{k_{l+1}}, \ldots, \boldsymbol{\theta}_{k_n}|\boldsymbol{\theta}_{k_{m+1}}^{\diamond}, \ldots, \boldsymbol{\theta}_{k_l}^{\diamond}, \boldsymbol{y}),$$

so that one has only to discard the components $k_{l+1}, \ldots, k_n$ to get a sample from

$$\pi(\boldsymbol{\theta}_{k_1}, \ldots, \boldsymbol{\theta}_{k_m}|\boldsymbol{\theta}_{k_{m+1}}^{\diamond}, \ldots, \boldsymbol{\theta}_{k_l}^{\diamond}, \boldsymbol{y}).$$

The reduced runs provide a simple method to use the original Gibbs sampler with only slight modifications for sampling from other distributions than the posterior distribution. As described in Chib (1995), one can take advantage from reduced runs for example in the estimation of marginal likelihoods.

2.4 General strategies for improving MCMC

One possibility to improve the convergence of the MCMC chains is to use **block updates** where several lower-dimensional updates are combined to a higher-dimensional joint update. As Gilks et al. (1996) point out especially blocking of highly correlated components into a higher-dimensional component may improve mixing. However, sampling from higher-dimensional distributions can be difficult and time expensive, so that often one must search for other methods to speed up the convergence.

Another possibility is to use random permutations of the **updating order**, since a fixed order is not necessary. Moreover, not all components need to be updated in each iteration. To improve mixing, Zeger and Karim (1991) suggest to update highly correlated components more frequently than other components.

In Sections 3.3 and 6.3 we use a method which is often difficult to apply, but which may heavily speed up the convergence. It is based on the following theorem:

Theorem 2.3 (Liu and Sabatti (2000))
If Γ *is a locally compact group of transformations defined on the sample space* $\boldsymbol{S}$, L *its left-Haar measure,* $\boldsymbol{w} \in \boldsymbol{S}$ *follows a distribution with density* π, *and* $\gamma \in \Gamma$ *is drawn from* $\pi(\gamma(\boldsymbol{w}))|J_\gamma(\boldsymbol{w})|L(d\gamma)$, *with* $J_\gamma(\boldsymbol{w}) = \det\left(\partial\gamma(\boldsymbol{w})/\partial\boldsymbol{w}\right)$, $\partial\gamma(\boldsymbol{w})/\partial\boldsymbol{w}$ *the Jacobian matrix, then* $\boldsymbol{w}^* = \gamma(\boldsymbol{w})$ *has density* π, *too.*

In this context, a locally compact group Γ of transformations on $\boldsymbol{S}$ has the following properties:

- Γ is a locally compact space.
- Γ is a group with respect to the composition operation.
- The group operations $(\gamma_1, \gamma_2) \to \gamma_1\gamma_2$ and $\gamma \to \gamma^{-1}$ are continuous.

A measure L is called **left-Haar measure** with respect to the transformation group Γ if

$$L(B) = L(\gamma B)$$

for all $\gamma \in \Gamma$ and for all measurable subsets B of Γ. For details on left-Haar measures we refer to Rao (1987).

A well-known locally compact transformation group is the **translation group on S along an arbitrary direction e**,

$$\Gamma_{\text{trans}} := \left\{\gamma \in \mathbb{R}^1 : \gamma(\boldsymbol{w}) = \boldsymbol{w} + \gamma\boldsymbol{e} = (w_1 + \gamma e_1, \ldots, w_d + \gamma e_d)\right\},$$

with left-Haar measure $L(d\gamma) = d\gamma$. Following the Theorem of Liu and Sabatti (2000), γ has to be drawn from $\pi(\boldsymbol{w} + \gamma\boldsymbol{e})$ in this case. Another example is the **scale group on S**,

$$\Gamma_{\text{scale}} := \{\gamma > 0 : \gamma(\boldsymbol{w}) = \gamma\boldsymbol{w} = (\gamma w_1, \ldots, \gamma w_d)\},$$

with left-Haar measure $L(d\gamma) = \gamma^{-1}d\gamma$. Here γ has to be drawn from $\gamma^{d-1}\pi(\gamma\boldsymbol{w})$.

In the MCMC context, one can use the Theorem of Liu and Sabatti (2000) by applying the transformation γ to a group $\boldsymbol{w}$ of parameters, since the sample $\gamma(\boldsymbol{w})$ can be considered to be from the same distribution as the original sample $\boldsymbol{w}$. The move $\boldsymbol{w} \to \gamma(\boldsymbol{w})$ is called **grouped move step**. It has to be inserted in each iteration of the underlying MCMC algorithm. The resulting algorithm is called **GM-MGMC (grouped move multi-grid Monte Carlo) sampler**. Whether a GM-MGMC sampler shows a better convergence than the parent MCMC algorithm also depends on the choice of the transformation group Γ and the distribution π. Of course, they should always be chosen so that on the one hand the problematic parameters are transformed and on the other hand the distribution $\pi(\gamma(\boldsymbol{w}))|J_\gamma(\boldsymbol{w})|L(d\gamma)$ allows to draw samples very fast.

Finally we emphasize that a grouped move step is not a block update, since the components of $\boldsymbol{w}$ are **deterministically** transformed by a **randomly** drawn transformation element γ.

2.5 Auxiliary particle filters

In this section we describe some simulation-based filtering methods. For the following chapters a special filtering procedure turns out to be useful which is called auxiliary particle filtering. This method was introduced and investigated by Pitt and Shephard (1999) as an extension of particle filters. The particle filtering method has been used for example by Gordon et al. (1993) and Kitagawa (1996) on non-Gaussian state space models.

2.5.1 The filtering problem

We consider a time series y_t, $t = 1, \ldots, T$, which is supposed to be conditionally independent given an unobserved state y_t^* which is itself assumed to be Markovian. Let $\mathcal{F}_t$ denote the vector of observations until time t, i.e. $\mathcal{F}_t := (y_1, \ldots, y_t)$. Then the assumptions imply that, for instance, $f(y_t|y_t^*, \mathcal{F}_{t-1}) = f(y_t|y_t^*)$ and $f(y_t^*|y_{t-1}^*, \mathcal{F}_{t-1}) = f(y_t^*|y_{t-1}^*)$, which simplifies the following equations. The state evolution is initialized by some density $f(y_0^*)$.

Filtering means to learn about the state y_t^* given contemporaneously available information. We do this by estimating the density (or probability distribution function) $f(y_t^*|\mathcal{F}_t)$, $t = 1, \ldots, T$. Filtering can be viewed as a procedure with two stages which are repeatedly applied. Starting from $f(y_{t-1}^*|\mathcal{F}_{t-1})$ one first produces the prediction density

$$f(y_t^*|\mathcal{F}_{t-1}) = \int f(y_t^*|y_{t-1}^*) f(y_{t-1}^*|\mathcal{F}_{t-1})\, dy_{t-1}^* \tag{2.9}$$

and secondly moves to the filtering density via Bayes theorem,

$$f(y_t^*|\mathcal{F}_t) = \frac{f(y_t|y_t^*) f(y_t^*|\mathcal{F}_{t-1})}{f(y_t|\mathcal{F}_{t-1})} \tag{2.10}$$

where $f(y_t|\mathcal{F}_{t-1}) = \int f(y_t|y_t^*) f(y_t^*|\mathcal{F}_{t-1})\, dy_t^*$. In most cases the integrals in Equations (2.9) and (2.10) cannot be solved analytically, so numerical or simulation-based methods must be used, for example particle filters which are discussed in the following.

2.5.2 Particle filters

Particle filters are the class of simulation filters that recursively approximate the filtering random variable $y_{t-1}^*|\mathcal{F}_{t-1}$ by 'particles' $y_{t-1}^{*1}, \ldots, y_{t-1}^{*M}$ with discrete probability mass $\pi_{t-1}^1, \ldots, \pi_{t-1}^M$, respectively. That means that a continuous variable is approximated by a discrete one with random support. In the literature often all of the π_{t-1}^m are assumed to equal $1/M$.

Since the particles are considered to be samples from $f(y_{t-1}^*|\mathcal{F}_{t-1})$, one can approximate the prediction density (2.9) by

$$\hat{f}(y_t^*|\mathcal{F}_{t-1}) := \sum_{m=1}^{M} f(y_t^*|y_{t-1}^{*m}) \pi_{t-1}^m \tag{2.11}$$

which is called the **empirical prediction density**. This can be combined with the density $f(y_t|y_t^*)$ to get, up to proportionality,

$$\hat{f}(y_t^*|\mathcal{F}_t) \propto f(y_t|y_t^*) \sum_{m=1}^{M} f(y_t^*|y_{t-1}^{*m})\pi_{t-1}^m \tag{2.12}$$

which is called the **empirical filtering density**. Particle filters then sample from this density to produce new particles $y_t^{*1}, \ldots, y_t^{*M}$ with weights $\pi_t^1, \ldots, \pi_t^M$. This procedure can be iterated through the data.

There are several methods to sample from the filtering density $f(y_t^*|\mathcal{F}_t)$, for instance, sampling/importance resampling (SIR), rejection sampling (cf. Section 2.1), or MCMC. For our purposes the SIR method is the most useful one, and therefore we do not discuss the others here. Details to importance sampling can be found for example in Robert and Casella (2000), Section 3.3.

A SIR-based particle filter has the form given in Algorithm 2.6 which is evident from Equation (2.10). For a full filtering procedure this algorithm has to be applied successively for $t = 1, \ldots, T$. To get started one has to draw a sample $\{y_0^{*1}, \ldots, y_0^{*M}\}$ from $f(y_0^*)$.

Algorithm 2.6 SIR-based particle filter

1. Draw $y_t^{*1}, \ldots, y_t^{*R}$ from $f(y_t^*|\mathcal{F}_{t-1})$.
2. For each $r \in \{1, \ldots, R\}$ evaluate the weights

$$w_r := \frac{f(y_t|y_t^{*r})f(y_t^{*r}|\mathcal{F}_{t-1})}{f(y_t^{*r}|\mathcal{F}_{t-1})} = f(y_t|y_t^{*r})$$

   and compute $\pi_r := w_r / \sum_{j=1}^{R} w_j$.
3. Resample among $\{y_t^{*r}|r = 1, \ldots, R\}$ using the associated probabilities $\{\pi_r\}$ to produce a sample of size M from $f(y_t^*|\mathcal{F}_t)$.

This method requires $R \gg M$. It samples from $f(y_t^*|\mathcal{F}_t)$ by making 'blind' proposals $y_t^{*1}, \ldots, y_t^{*R}$ from $f(y_t^*|\mathcal{F}_{t-1})$, ignoring the fact that y_t is known. A particle filter is called **adapted** if it makes proposals that take into account the value of y_t. An adapted SIR-based particle filter has the following structure.

Algorithm 2.7 Adapted SIR-based particle filter

1. Draw $y_t^{*1}, \ldots, y_t^{*R}$ from $g(y_t^*|\mathcal{F}_t)$, where g is some proposal density.

2. For each $r \in \{1, \ldots, R\}$ evaluate the weights

$$w_r := \frac{f(y_t|y_t^{*r}) f(y_t^{*r}|\mathcal{F}_{t-1})}{g(y_t^{*r}|\mathcal{F}_t)}$$

and compute $\pi_r := w_r / \sum_{j=1}^{R} w_j$.

3. Resample among $\{y_t^{*r} | r = 1, \ldots, R\}$ using the associated probabilities $\{\pi_r\}$ to produce a sample of size M from $f(y_t^*|\mathcal{F}_t)$.

This method looks very attractive. However, we know from Equation (2.11) that, for a particle filter, $\hat{f}(y_t^*|\mathcal{F}_{t-1}) = \sum_{m=1}^{M} f(y_t^*|y_{t-1}^{*m})\pi_{t-1}^m$. Therefore one must evaluate at least $M \cdot R$ densities to generate M samples from $f(y_t^*|\mathcal{F}_t)$. Given that M and R are typically very large, this implies that the adapted SIR-based particle filter is very computer-intensive and not practicable for many models. In the following section we consider a method which does not require so many density evaluations.

2.5.3 Auxiliary particle filters

The aim of the considered filtering procedures is a fast sampling from the density $f(y_t^*|\mathcal{F}_t)$, which can up to proportionality be approximated by (2.12). One can rewrite this expression in the form

$$f(y_t^*|\mathcal{F}_t) \propto \sum_{m=1}^{M} f(y_t|y_t^*) f(y_t^*|y_{t-1}^{*m}) \pi_{t-1}^m$$

and define the densities $f(y_t^*, m|\mathcal{F}_t)$, $m = 1, \ldots, M$, by the condition

$$f(y_t^*, m|\mathcal{F}_t) \propto f(y_t|y_t^*) f(y_t^*|y_{t-1}^{*m}) \pi_{t-1}^m.$$

Of course, m serves as an index on the mixture in Equation (2.11).

Now a sample from $f(y_t^*|\mathcal{F}_t)$ can be produced by first drawing from the joint density $f(y_t^*, m|\mathcal{F}_t)$ and then discarding the second component. Since m is present simply to aid the task of simulation, it is called an **auxiliary variable**. Particle filters of this type are called **auxiliary particle filters**.

As in the case without auxiliary variable, one can use SIR, rejection sampling, or MCMC methods for sampling from $f(y_t^*, m|\mathcal{F}_t)$. Again we consider only the SIR-based method.

Algorithm 2.8 SIR-based auxiliary particle filter

1. Make R proposals (y_t^{*r}, m^r) from $g(y_t^*, m|\mathcal{F}_t)$, where g is some proposal density.

2. For each $r \in \{1, \ldots, R\}$ evaluate the weights

$$w_r := \frac{f(y_t|y_t^{*r}) f(y_t^{*r}|y_{t-1}^{*m^r})}{g(y_t^{*r}, m^r|\mathcal{F}_t)} \tag{2.13}$$

   and compute $\pi_r := w_r / \sum_{j=1}^R w_j$.

3. Resample among $\{y_t^{*r} | r = 1, \ldots, R\}$ using the associated probabilities $\{\pi_r\}$ to produce a sample of size M from $f(y_t^*|\mathcal{F}_t)$.

The auxiliary SIR-based particle filter is an adaptable and very flexible method, since the proposal density g can depend on y_t and y_{t-1}^{*m}. g should be chosen so that the weights w_r are as equal as possible. If one can achieve exactly equal weights, the procedure is called **fully adapted** to the model.

Since we only use the SIR-based auxiliary particle filter in the subsequent chapters, we refer to Pitt and Shephard (1999) for more details about other auxiliary particle filters.

2.6 Model selection via Bayes factors

In this section we introduce some basic notions in the context of Bayes factors and consider some estimation strategies which follow the papers by Chib (1995) and Chib and Jeliazkov (2001).

2.6.1 Bayes factor and marginal likelihood

We consider the problem of comparing a collection of models $\{M_1, \ldots, M_L\}$ which reflect competing hypotheses about the data. We suppose that each model M_l is characterized by a model-specific parameter vector $\boldsymbol{\theta}_l \in \Theta_l \subset \mathbb{R}^{d_l}$ of dimension d_l. Let $f(\boldsymbol{y}|\boldsymbol{\theta}_l, M_l)$ denote the likelihood for the model M_l.

Bayesian model selection proceeds by pairwise comparison of the models $M_1, \ldots, M_L$ through their **posterior odds** ratio, which for any two models M_i and M_j is defined by

B_{ij}	Evidence against model j
$1 - 3.2$	Not worth more than a bare mention
$3.2 - 10$	Substantial
$10 - 100$	Strong
> 100	Decisive

Table 2.1: Jeffrey's Bayes factor scale with B_{ij} denoting the Bayes factor for model i versus model j.

$P(M_i|\boldsymbol{y})/P(M_j|\boldsymbol{y})$. This ratio can also be written as

$$\frac{P(M_i|\boldsymbol{y})}{P(M_j|\boldsymbol{y})} = \frac{P(M_i, \boldsymbol{y})}{P(M_j, \boldsymbol{y})} = \frac{P(M_i)}{P(M_j)} \cdot \frac{m(\boldsymbol{y}|M_i)}{m(\boldsymbol{y}|M_j)} \tag{2.14}$$

where

$$m(\boldsymbol{y}|M_l) = \int f(\boldsymbol{y}|\boldsymbol{\theta}_l, M_l)\pi_l(\boldsymbol{\theta}_l|M_l)\, d\boldsymbol{\theta}_l \tag{2.15}$$

is the marginal likelihood of M_l, with $\pi_l(\boldsymbol{\theta}_l|M_l)$ denoting the prior for $\boldsymbol{\theta}_l$ in model M_l. The first fraction on the right-hand side of (2.14) is known as the **prior odds** and the second as the **Bayes factor**.

When the prior odds on the models M_i and M_j is equal to one, the Bayes factor and the posterior odds are equal. If this holds for all $i, j \in \{1, \ldots, L\}$, model selection is done by finding an appropriate estimate of the marginal likelihood for each competing model and then computing the Bayes factors for all pairs of competing models.

The strength of evidence in favor of model M_i versus model M_j can then be evaluated according to the Bayes factor scale in Table 2.1, which was first proposed by Jeffreys (1961).

2.6.2 Estimation strategies

The calculation of the marginal likelihood can be a difficult task in complex models. Of course, the analytic evaluation of the integral in Equation (2.15) is almost never possible. Furthermore, because the marginal likelihood is obtained by integrating the sampling density $f(\boldsymbol{y}|\boldsymbol{\theta}_l, M_l)$ with respect to the prior distribution of the parameters, and not the posterior distribution, the posterior MCMC output from the simulation cannot be used directly for estimation.

An alternative approach is given in Chib (1995). This approach is based on another

representation of the marginal likelihood that is amenable to calculation by MCMC methods. Because the marginal likelihood is the normalizing constant of the posterior density $\pi_l(\boldsymbol{\theta}_l|\boldsymbol{y}, M_l)$, one can write

$$m(\boldsymbol{y}|M_l) = \frac{f(\boldsymbol{y}|\boldsymbol{\theta}_l, M_l)\pi_l(\boldsymbol{\theta}_l|M_l)}{\pi_l(\boldsymbol{\theta}_l|\boldsymbol{y}, M_l)} \tag{2.16}$$

which is referred to as the **basic marginal likelihood identity**. Evaluating the right-hand side of this identity at some appropriate point $\boldsymbol{\theta}_l^\diamond$ and taking logarithms one obtains the expression

$$\log m(\boldsymbol{y}|M_l) = \log f(\boldsymbol{y}|\boldsymbol{\theta}_l^\diamond, M_l) + \log \pi_l(\boldsymbol{\theta}_l^\diamond|M_l) - \log \pi_l(\boldsymbol{\theta}_l^\diamond|\boldsymbol{y}, M_l) \tag{2.17}$$

from which the marginal likelihood can be estimated by finding an estimate of the posterior ordinate $\pi_l(\boldsymbol{\theta}_l^\diamond|\boldsymbol{y}, M_l)$, and of the likelihood ordinate $f(\boldsymbol{y}|\boldsymbol{\theta}_l^\diamond, M_l)$, when this expression is not calculable directly. For estimation efficiency, the point $\boldsymbol{\theta}_l^\diamond$ is generally taken to be a high-density point in the support of the posterior. Here the posterior mean or the posterior mode can be an appropriate choice.

When the model M_l includes latent variables, another marginal likelihood identity often turns out to be useful. In this case the parameter vector $\boldsymbol{\theta}_l$ consists of a classical parameter vector $\boldsymbol{\psi}_l$ and of the vector $\boldsymbol{z}_l$ of the latent variables, $\boldsymbol{\theta}_l = (\boldsymbol{\psi}_l, \boldsymbol{z}_l)$. Therefore Equation (2.16) becomes

$$m(\boldsymbol{y}|M_l) = \frac{f(\boldsymbol{y}|\boldsymbol{\psi}_l^\diamond, \boldsymbol{z}_l^\diamond, M_l)\pi_l(\boldsymbol{\psi}_l^\diamond, \boldsymbol{z}_l^\diamond|M_l)}{\pi_l(\boldsymbol{\psi}_l^\diamond, \boldsymbol{z}_l^\diamond|\boldsymbol{y}, M_l)}.$$

This identity is not that useful because it requires the computation of the posterior ordinate $\pi_l(\boldsymbol{\psi}_l^\diamond, \boldsymbol{z}_l^\diamond|\boldsymbol{y}, M_l)$ whose dimension can easily run into the hundreds, if not thousands. The same is true for the alternative representation

$$m(\boldsymbol{y}|M_l) = \frac{f(\boldsymbol{y}, \boldsymbol{z}_l^\diamond|\boldsymbol{\psi}_l^\diamond, M_l)\pi_l(\boldsymbol{\psi}_l^\diamond|M_l)}{\pi_l(\boldsymbol{\psi}_l^\diamond, \boldsymbol{z}_l^\diamond|\boldsymbol{y}, M_l)}.$$

To avoid this problem one can integrate out the latent variables $\boldsymbol{z}_l$ and use the identity

$$m(\boldsymbol{y}|M_l) = \frac{f(\boldsymbol{y}|\boldsymbol{\psi}_l^\diamond, M_l)\pi_l(\boldsymbol{\psi}_l^\diamond|M_l)}{\pi_l(\boldsymbol{\psi}_l^\diamond|\boldsymbol{y}, M_l)} \tag{2.18}$$

which usually leads to a quite efficient estimation of the marginal likelihood since the latent variables do not appear any more.

2.7 Basic models

Here we consider some well-known models which play a role in the following chapters. First we introduce the ordered probit model which will be extended in Chapter 3 by an autoregressive component. Then we have a look at stochastic volatility models for a continuous response. Finally we consider a general form of state space models and provide the corresponding Kalman recursions and the simulation smoother by De Jong and Shephard (1995).

2.7.1 Ordered probit model

In the **ordered probit (OP) model** it is assumed that $\boldsymbol{y} = (y_1, \ldots, y_n)'$ is a vector of n independent ordinal random variables and that y_i, $i = 1, \ldots, n$, takes on the value $k \in \{1, \ldots, K\}$ with probability

$$p_{ik} := \Phi(c_k + \boldsymbol{x}_i'\boldsymbol{\beta}) - \Phi(c_{k-1} + \boldsymbol{x}_i'\boldsymbol{\beta})$$

where $\Phi(\cdot)$ denotes the cumulative distribution function of the standard normal distribution. The **cutpoints** c_k have to satisfy the **order condition** $-\infty =: c_0 < c_1 < \ldots < c_{K-1} < c_K := \infty$. $\boldsymbol{x}_i$ is a $p \times 1$ column vector of covariates and $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_p)'$ a $p \times 1$ vector of regression coefficients.

An equivalent representation of the ordered probit model was used by Albert and Chib (1993). They introduced **latent variables** y_i^* such that

$$\begin{aligned} y_i = k \quad \Longleftrightarrow \quad & y_i^* \in [c_{k-1}, c_k)\,, \\ & y_i^* \;=\; \boldsymbol{x}_i'\boldsymbol{\beta} + \varepsilon_i^*, \end{aligned}$$

for $k = 1, \ldots, K$ and $i = 1, \ldots, n$, where all errors $\varepsilon_i^* \sim N(0, 1)$ are assumed to be independent. Albert and Chib (1993) took advantage of this latent variable representation for the construction of a Gibbs sampler to estimate the cutpoints c_k and the regression coefficients $\beta_1, \ldots, \beta_p$.

We note that if the expression $\boldsymbol{x}_i'\boldsymbol{\beta}$ contains an intercept, one has to fix one of the cutpoints for reasons of identifiability. Usually the cutpoint c_1 is fixed to 0.

2.7.2 Stochastic volatility models

Stochastic volatility (SV) models are used for modeling time series with non-constant volatility where the volatility shows a stochastic behavior. Such time series occur for example in finance. So far, all stochastic volatility models have been designed for a continuous response. In Chapter 6, however, we will introduce a SV model for ordinal-valued time series.

The **basic SV model** for continuous variables $y_1, \ldots, y_T$ is defined by the following equations (cf. Taylor (1986)):

$$y_t = \exp(h_t^*/2)\varepsilon_t^*, \tag{2.19}$$

$$h_t^* = \mu + \phi(h_{t-1}^* - \mu) + \sigma\eta_t^*, \tag{2.20}$$

where all the variables ε_t^* and η_t^* are assumed to be independent and standard normally distributed. The latent variables h_t^* represent the **log-volatilities** which form an AR(1)-process with independent errors of mean 0 and standard deviation $\sigma > 0$.

As an extension of the basic SV model, covariate vectors $\boldsymbol{x}_t$ and $\boldsymbol{z}_t$ with corresponding regression parameter vectors $\boldsymbol{\beta}$ and $\boldsymbol{\alpha}$ can be inserted in Equations (2.19) and (2.20), respectively:

$$y_t = \boldsymbol{x}_t'\boldsymbol{\beta} + \exp(h_t^*/2)\varepsilon_t^*, \tag{2.21}$$

$$h_t^* = \mu + \boldsymbol{z}_t'\boldsymbol{\alpha} + \phi(h_{t-1}^* - \mu) + \sigma\eta_t^*. \tag{2.22}$$

Obviously, the covariate vector $\boldsymbol{x}_t$ changes the mean of y_t given $\boldsymbol{\beta}$ and h_t^*, whereas the covariate vector $\boldsymbol{z}_t$ has an impact on the log-volatility h_t^*.

Another extension is to replace the standard normally distributed errors ε_t^* by Student-t distributed errors. Also jump components can be inserted. There exist many papers which deal with SV models. For example, Jacquier et al. (1994) have proposed a Bayesian treatment of SV models. MCMC methods for SV models with both t-distributed errors and a jump component have been developed by Chib et al. (2002).

2.7.3 State space models and associated algorithms

State space models are a flexible family of models which can be used in many applications. The corresponding techniques were originally developed in connection with the control of linear systems. However, since there exist state space representations for many

well-known models, for example autoregressive moving average (ARMA) processes, these techniques can also be used in other contexts. Details on state space models and the derivation of the Kalman recursions can be found in Harvey (1989).

We consider the multivariate **Gaussian state space model** defined by

$$\begin{aligned} \boldsymbol{y}_t &= \boldsymbol{c}_t + Z_t \boldsymbol{x}_t + G_t \boldsymbol{u}_t, && t = 1, \ldots, T, && (2.23) \\ \boldsymbol{x}_{t+1} &= \boldsymbol{d}_t + S_t \boldsymbol{x}_t + H_t \boldsymbol{u}_t, && \boldsymbol{u}_t \sim N_{n+m}(\boldsymbol{0}, I) \text{ i.i.d.}, && (2.24) \end{aligned}$$

with initial state

$$\boldsymbol{x}_1 \sim N_m(\boldsymbol{x}_{1|0}, P_{1|0}). \tag{2.25}$$

The vectors $\boldsymbol{y}_t \in \mathbb{R}^n$ are called **observations**, whereas the vectors $\boldsymbol{x}_t \in \mathbb{R}^m$ represent the (unobserved) **states**. Accordingly, (2.23) is called **observation equation** and (2.24) **state equation**. The **system vectors** $\boldsymbol{c}_t \in \mathbb{R}^n$ and $\boldsymbol{d}_t \in \mathbb{R}^m$ are assumed known as well as the **system matrices** $Z_t \in \mathbb{R}^{n \times m}$, $G_t \in \mathbb{R}^{n \times (n+m)}$, $S_t \in \mathbb{R}^{m \times m}$, and $H_t \in \mathbb{R}^{m \times (n+m)}$. For simplicity we assume here that $G_t H_t' = 0$, and we define $\Sigma_t^y := G_t G_t'$ and $\Sigma_t^x := H_t H_t'$. The process is initiated by the state vector $\boldsymbol{x}_1$ which is normally distributed with known mean $\boldsymbol{x}_{1|0} \in \mathbb{R}^m$ and known covariance matrix $P_{1|0} \in \mathbb{R}^{m \times m}$.

We are interested in the distribution $f(\boldsymbol{y}_t | \boldsymbol{y}_1, \ldots, \boldsymbol{y}_{t-1})$ of $\boldsymbol{y}_t$ given the observations $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_{t-1}$. Since we assume normally distributed errors $\boldsymbol{u}_t$ this distribution is also multivariate normal. The same holds for the distribution $f(\boldsymbol{x}_t | \boldsymbol{y}_1, \ldots, \boldsymbol{y}_{t-1})$ of $\boldsymbol{x}_t$ given $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_{t-1}$. We denote the means and covariance matrices of these conditional distributions by

$$\begin{aligned} \boldsymbol{y}_{t|t-1} &:= \mathrm{E}(\boldsymbol{y}_t | \boldsymbol{y}_1, \ldots, \boldsymbol{y}_{t-1}), & F_{t|t-1} &:= \mathrm{Cov}(\boldsymbol{y}_t | \boldsymbol{y}_1, \ldots, \boldsymbol{y}_{t-1}), \\ \boldsymbol{x}_{t|t-1} &:= \mathrm{E}(\boldsymbol{x}_t | \boldsymbol{y}_1, \ldots, \boldsymbol{y}_{t-1}), & P_{t|t-1} &:= \mathrm{Cov}(\boldsymbol{x}_t | \boldsymbol{y}_1, \ldots, \boldsymbol{y}_{t-1}). \end{aligned}$$

The following **Kalman recursions** allow for the computation of these quantities, successively for $t = 1, \ldots, T$. Recall that $\boldsymbol{x}_{1|0}$ and $P_{1|0}$ are known.

$$\begin{aligned} \boldsymbol{y}_{t|t-1} &= \boldsymbol{c}_t + Z_t \boldsymbol{x}_{t|t-1}, && (2.26) \\ \boldsymbol{v}_t &:= \boldsymbol{y}_t - \boldsymbol{y}_{t|t-1}, && (2.27) \\ F_{t|t-1} &= Z_t P_{t|t-1} Z_t' + \Sigma_t^y, && (2.28) \\ K_t &:= S_t P_{t|t-1} Z_t' F_{t|t-1}^{-1}, && (2.29) \\ L_t &:= S_t - K_t Z_t, && (2.30) \\ \boldsymbol{x}_{t+1|t} &= \boldsymbol{d}_t + S_t \boldsymbol{x}_{t|t-1} + K_t \boldsymbol{v}_t, && (2.31) \\ P_{t+1|t} &= S_t P_{t|t-1} L_t' + \Sigma_t^x. && (2.32) \end{aligned}$$

The vectors $\boldsymbol{v}_t$ are called **innovations**. Although one could insert the explicit expressions for the matrices K_t and L_t in Equations (2.31) and (2.32), these matrices should be used to reduce the number of matrix multiplications required.

If the new observation $\boldsymbol{y}_t$ becomes available, the estimates for the corresponding state can be updated. In particular, $\boldsymbol{x}_t$ given the observations $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_t$ is normally distributed with mean $\boldsymbol{x}_{t|t}$ and covariance matrix $P_{t|t}$. These quantities are provided by the following **updating equations**:

$$\begin{aligned} \boldsymbol{x}_{t|t} &= \boldsymbol{x}_{t|t-1} + P_{t|t-1} Z_t' F_{t|t-1}^{-1} \boldsymbol{v}_t, && (2.33) \\ P_{t|t} &= P_{t|t-1} - P_{t|t-1} Z_t' F_{t|t-1}^{-1} Z_t P_{t|t-1}. && (2.34) \end{aligned}$$

Inserting Equations (2.33) and (2.29) in Equation (2.31) we get a shorter expression for $\boldsymbol{x}_{t+1|t}$:

$$\begin{aligned} \boldsymbol{x}_{t+1|t} &= \boldsymbol{d}_t + S_t \boldsymbol{x}_{t|t-1} + K_t v_t \\ &= \boldsymbol{d}_t + S_t(\boldsymbol{x}_{t|t} - P_{t|t-1} Z_t' F_{t|t-1}^{-1} \boldsymbol{v}_t) + S_t P_{t|t-1} Z_t' F_{t|t-1}^{-1} \boldsymbol{v}_t \\ &= \boldsymbol{d}_t + S_t \boldsymbol{x}_{t|t}. \end{aligned}$$

Similarly, inserting (2.34) and (2.30) in (2.32) we get

$$\begin{aligned} P_{t+1|t} &= S_t P_{t|t-1} L_t' + \Sigma_t^x \\ &= S_t P_{t|t-1} S_t' - S_t P_{t|t-1} Z_t' K_t' + \Sigma_t^x \\ &= S_t P_{t|t-1} S_t' - S_t P_{t|t-1} Z_t' F_{t|t-1}^{-1} Z_t P_{t|t-1} S_t' + \Sigma_t^x \\ &= S_t P_{t|t} S_t' + \Sigma_t^x. \end{aligned}$$

Therefore, alternatively to Equations (2.26) up to (2.32) one can use the recursions

$$\begin{aligned} \boldsymbol{y}_{t|t-1} &= \boldsymbol{c}_t + Z_t \boldsymbol{x}_{t|t-1}, && (2.35) \\ \boldsymbol{v}_t &= \boldsymbol{y}_t - \boldsymbol{y}_{t|t-1}, && (2.36) \\ F_{t|t-1} &= Z_t P_{t|t-1} Z_t' + \Sigma_t^y, && (2.37) \\ N_t &:= P_{t|t-1} Z_t' F_{t|t-1}^{-1}, && (2.38) \\ \boldsymbol{x}_{t|t} &= \boldsymbol{x}_{t|t-1} + N_t \boldsymbol{v}_t, && (2.39) \\ P_{t|t} &= P_{t|t-1} - N_t Z_t P_{t|t-1}, && (2.40) \\ \boldsymbol{x}_{t+1|t} &= \boldsymbol{d}_t + S_t \boldsymbol{x}_{t|t}, && (2.41) \\ P_{t+1|t} &= S_t P_{t|t} S_t' + \Sigma_t^x. && (2.42) \end{aligned}$$

which provide additionally the state estimate $\boldsymbol{x}_{t|t}$.

The system vectors and matrices of the state space model (2.23) to (2.25) are typically parameterized by some hyperparameters, which we combine in a vector $\boldsymbol{\theta}$. Now we show how the Kalman recursions can be used to compute the likelihood for a specific value of $\boldsymbol{\theta}$. Since

$$f(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_T|\boldsymbol{\theta}) = f(\boldsymbol{y}_1|\boldsymbol{\theta}) \prod_{t=2}^{T} f(\boldsymbol{y}_t|\boldsymbol{y}_1, \ldots, \boldsymbol{y}_{t-1}, \boldsymbol{\theta}) \tag{2.43}$$

and the distribution of $\boldsymbol{y}_t$, given the past observations, is multivariate normal with mean $\boldsymbol{y}_{t|t-1}$ and covariance matrix $F_{t|t-1}$ we get immediately

$$\begin{aligned} \log f(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_T|\boldsymbol{\theta}) &= -\frac{nT}{2}\log(2\pi) - \frac{1}{2}\sum_{t=1}^{T}\log(\det(F_{t|t-1})) \\ &\quad -\frac{1}{2}\sum_{t=1}^{T}(\boldsymbol{y}_t - \boldsymbol{y}_{t|t-1})' F_{t|t-1}^{-1}(\boldsymbol{y}_t - \boldsymbol{y}_{t|t-1}). \end{aligned}$$

Using the innovations $\boldsymbol{v}_t$ defined in (2.27) this equation can be simplified to

$$\log f(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_T|\boldsymbol{\theta}) = -\frac{nT}{2}\log(2\pi) - \frac{1}{2}\sum_{t=1}^{T}\log(\det(F_{t|t-1})) - \frac{1}{2}\sum_{t=1}^{T}\boldsymbol{v}_t' F_{t|t-1}^{-1}\boldsymbol{v}_t. \tag{2.44}$$

This representation is known as the **prediction error decomposition** of the log likelihood. The covariance matrices $F_{t|t-1}$ as well as the innovations $\boldsymbol{v}_t$ depend on the parameters contained in $\boldsymbol{\theta}$. We emphasize that therefore the maximization of the log likelihood by a numerical optimization procedure requires a full run of the Kalman recursions for each value of $\boldsymbol{\theta}$ at which the log likelihood is evaluated.

In the following we deal with the problem of how to generate a sample from the multivariate normal posterior distribution

$$f(Z_1\boldsymbol{x}_1, \ldots, Z_T\boldsymbol{x}_T|\boldsymbol{y}_1, \ldots, \boldsymbol{y}_T, \boldsymbol{\theta})$$

where $\boldsymbol{\theta}$ denotes the vector of parameters in the model. This can be done using the **simulation smoother** of De Jong and Shephard (1995), which involves first running the Kalman recursions (2.35) to (2.42), storing $\{\boldsymbol{v}_t, F_{t|t-1}, N_t\}$ for each $t = 1, \ldots, T$, and then running the following backward recursions for $t = T, \ldots, 1$, where $\boldsymbol{r}_T = \boldsymbol{0}$ and

$M_T = 0$:

$$
\begin{aligned}
D_t &= F_{t|t-1}^{-1} + N_t' S_t' M_t S_t N_t, && (2.45)\\
\boldsymbol{b}_t &= F_{t|t-1}^{-1} \boldsymbol{v}_t - N_t' S_t' \boldsymbol{r}_t, && (2.46)\\
Q_t &= \Sigma_t^y - \Sigma_t^y D_t \Sigma_t^y, && (2.47)\\
\boldsymbol{\kappa}_t &\sim N_n(\mathbf{0}, Q_t), && (2.48)\\
A_t &= \Sigma_t^y (D_t Z_t - N_t' S_t' M_t S_t), && (2.49)\\
\boldsymbol{r}_{t-1} &= Z_t' F_{t|t-1}^{-1} \boldsymbol{v}_t + (S_t' - Z_t' N_t' S_t') \boldsymbol{r}_t - A_t' Q_t^{-1} \boldsymbol{\kappa}_t, && (2.50)\\
M_{t-1} &= Z_t' F_{t|t-1}^{-1} Z_t' + (S_t' - Z_t' N_t' S_t') M_t (S_t - S_t N_t Z_t) + A_t' Q_t^{-1} A_t, && (2.51)\\
\boldsymbol{\xi}_t &= \boldsymbol{y}_t - \boldsymbol{c}_t - \Sigma_t^y \boldsymbol{b}_t - \boldsymbol{\kappa}_t. && (2.52)
\end{aligned}
$$

Then $\boldsymbol{\xi}_t$ is a sample from $f(Z_t \boldsymbol{x}_t | Z_{t+1} \boldsymbol{x}_{t+1}, \ldots, Z_T \boldsymbol{x}_T, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_T, \boldsymbol{\theta})$. Since

$$
f(Z_1 \boldsymbol{x}_1, \ldots, Z_T \boldsymbol{x}_T | \boldsymbol{y}_1, \ldots, \boldsymbol{y}_T, \boldsymbol{\theta}) = \prod_{t=1}^{T} f(Z_t \boldsymbol{x}_t | Z_{t+1} \boldsymbol{x}_{t+1}, \ldots, Z_T \boldsymbol{x}_T, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_T, \boldsymbol{\theta})
$$

this implies directly that $(\boldsymbol{\xi}_1, \ldots, \boldsymbol{\xi}_T)$ is a sample from $f(Z_1 \boldsymbol{x}_1, \ldots, Z_T \boldsymbol{x}_T | \boldsymbol{y}_1, \ldots, \boldsymbol{y}_T, \boldsymbol{\theta})$. Note that $\boldsymbol{\kappa}_t$ in Equation (2.48) is drawn randomly from a multivariate normal distribution.

We finally remark that the computation of some matrices appearing in the Kalman recursions and in the simulation smoother as for example A_t is not absolutely necessary, since one could insert the corresponding expressions in the other equations. However, since these matrices appear in more than one equation, one can save computation time, since the corresponding expressions only have to be computed once. Of course, the number of multiplications can further be reduced, for example by storing $(S_t - S_t N_t Z_t)$ in an additional matrix.

2.8 Features of the price change process

The data which will be investigated in Chapters 5, 7, and 8 is high-frequency data collected at the New York Stock Exchange (NYSE). In high-frequency finance, one considers all transactions of one or more stocks or derivatives which occur in a certain period. The structure of such data differs heavily from data which is collected only once a day but over a long period of two or more years. Usually one only has to consider few days to collect enough data, since more important stocks are traded up to 500 times per hour, so that one observes sometimes over 3000 transactions per day.

We are interested in modeling the high-frequency behavior of price changes of stocks. The first important feature of such price changes is that they take on only values which are integer multiples of a certain amount, called tick-size. On June 24, 1997 the tick size at NYSE was changed from 1/8\$ to 1/16\$, and since January 29, 2001, the tick size is 1/100\$. In the following chapters we only consider data collected between these two dates, therefore the tick-size is always one sixteenth of a dollar. However, mostly only few integer multiples occur: In about 99% of the transactions the lie between $-3/16$\$ to $+3/16$\$. Therefore one observes an **ordinal valued time series**, and the use of continuous response models as for example common stochastic volatility models (cf. Taylor (1986) or Chib et al. (2002)) seems not to be adequate.

Another important feature of the price changes is that they may depend on several **covariates**. This follows from theoretical considerations about the trading process. For example, Diamond and Verrecchia (1987) give theoretical reasons for the assumption that longer periods between consecutive transactions usually lead to a higher volatility of the price change process. Easley and O'Hara (1987) and Tauchen and Pitts (1983) give reasons for the dependency on the transaction volume.

The third important feature of the price change process is that the transaction times are **not equidistant**. One can take this into account for example by using an appropriate covariate which contains the time elapsed since the last transaction. Then one can search for the impact of this covariate on the process. Another possibility is to use continuous-time models where one assumes that the process is observed at non-equidistant time points.

Fourth, looking at the price changes one can observe periods with higher price fluctuations which are followed by relative quite periods and vice versa. This effect is called **volatility clustering**. It also implies that the volatility of the price change process is **non-constant**.

To capture these features one can use either a single model for the **signed** price changes or a decomposition strategy where the price changes are decomposed, for example into the product of the **absolute** values and the sign of the price changes. An even finer decomposition is used by Rydberg and Shephard (2003). The AOP model which will be introduced in the following chapter can model the absolute price changes whereas the the OSV and OSVt models considered in Chapters 6, 7, and 8 capture the structure of the signed price changes. For a comprehensive treatment on market microstructure theory we now refer to O'Hara (1995).

Chapter 3

The Autoregressive Ordered Probit Model

In this chapter we introduce a model which can be considered as an autoregressive extension of the ordered probit model. After formulating the model in Section 3.1 we develop a standard Gibbs sampler for parameter estimation in Section 3.2. However this algorithm exhibits bad convergence properties. To get a sampling method with a better convergence behavior we utilize in Section 3.3 a special transformation group on the sample space which allows to develop a grouped move multi-grid Monte Carlo (GM-MGMC) sampler (cf. Section 2.4). A simulation study is given in Section 3.4 to demonstrate the substantial improvement by this new algorithm.

3.1 Model formulation

We assume that we can observe a discrete response time series $\{y_t, t = 1, \ldots, T\}$, where y_t takes on only K different values, and a $(p+1)$-dimensional vector $\boldsymbol{x}_t = (1, x_{t1}, \ldots, x_{tp})'$ of real-valued covariates for each $t \in \{1, \ldots, T\}$. To model the time dependency in $\{y_t, t = 1, \ldots, T\}$ we assume that there exists an underlying unobserved autoregressive real-valued time series $\{y_t^*, t = 1, \ldots, T\}$ which produces the discrete values y_t by thresholding. In particular, the following latent variable representation holds:

$$y_t = k \iff y_t^* \in [c_{k-1}, c_k), \qquad k \in \{1, \ldots, K\}, \tag{3.1}$$

$$y_t^* = \boldsymbol{x}_t'\boldsymbol{\beta} + \phi y_{t-1}^* + \varepsilon_t^*, \qquad t \in \{1, \ldots, T\}, \tag{3.2}$$

where $-\infty = c_0 < c_1 < \ldots < c_{K-1} < c_K = \infty$ are unknown cutpoints, and $\boldsymbol{\beta} = (\beta_0, \ldots, \beta_p)'$ is a vector of unknown regression cofficients. All latent variables (except parameters) are marked with an asterisk *. We assume that $\varepsilon_t^* \sim N(0, \delta^2)$ i.i.d. Since the vector of covariates contains an intercept, we have to fix c_1 for reasons of identifiability. In particular, we set $c_1 = 0$. For the same reasons we have to fix the variance δ^2, since otherwise we could multiply $\boldsymbol{c} := (c_2, \ldots, c_{K-1})'$, $\boldsymbol{\beta}$ and $\boldsymbol{y}^* := (y_0^*, \ldots, y_T^*)'$ by a positive constant without changing the likelihood. Therefore we assume $\delta^2 = 1$. It remains to estimate the latent variables $y_t^*, t = 0, \ldots, T$, the cutpoints $c_j, j = 2, \ldots, K-1$, the regression parameters $\beta_j, j = 0, \ldots, p$ and the autoregressive parameter ϕ.

For the following we introduce the notations

$$\begin{aligned} \boldsymbol{\theta} &:= (\beta_0, \ldots, \beta_p, \phi)', \\ \boldsymbol{y} &:= (y_1, \ldots, y_T)', \\ \boldsymbol{y}_{-t}^* &:= (y_0^*, \ldots, y_{t-1}^*, y_{t+1}^*, \ldots, y_T^*)', \\ \text{and} \qquad \boldsymbol{c}_{-k} &:= (c_2, \ldots, c_{k-1}, c_{k+1}, \ldots, c_{K-1})'. \end{aligned}$$

Univariate normal distributions that are truncated to an interval $[a, b]$ are denoted by $N_{[a,b]}(\mu, \sigma^2)$. For the n-dimensional normal distribution with mean $\boldsymbol{\mu}$ and covariance matrix Σ we write $N_n(\boldsymbol{\mu}, \Sigma)$.

3.2 Standard Gibbs sampler

Now we develop an MCMC algorithm that allows us to draw approximate samples from the posterior distribution $\pi(\boldsymbol{c}, \boldsymbol{\theta}, \boldsymbol{y}^* | \boldsymbol{y})$. In the following we use the notations $f(\cdot)$ and $f(\cdot|\cdot)$ for distributions (densities) and conditional distributions (densities), respectively. For prior and posterior distributions we also use the notations $\pi(\cdot)$ and $\pi(\cdot|\cdot)$, respectively.

For the Bayesian approach we have to specify prior distributions for $\boldsymbol{c}$, $\boldsymbol{\theta}$ and y_0^*. We assume

$$\pi(y_0^*, \boldsymbol{\theta}, \boldsymbol{c}) \quad \propto \quad \exp\left\{ -\frac{1}{2} \left[\sigma^{-2} (y_0^*)^2 + \tau^{-2} \boldsymbol{\beta}' \boldsymbol{\beta} + \rho^{-2} \phi^2 \right] \right\} \cdot \mathbb{1}_{\{0 < c_2 < \ldots < c_{K-1} < C\}} \tag{3.3}$$

where σ, τ, ρ and C are known hyperparameters. We assume all parameters to be a priori independent, except of the vector $\boldsymbol{c}$ for which the order condition has to be fulfilled. We choose normal priors for y_0^*, $\boldsymbol{\beta}$ and ϕ, respectively, and a noninformative prior on the set $\{0 < c_2 < \ldots < c_{K-1} < C\}$ for the cutpoints. We can take large values

for σ, τ and ρ, when there is little prior information about y_0^* and $\boldsymbol{\theta}$. At this point we also redefine $c_K := C$ for notational convenience.

Since the regression parameters and the autoregressive parameter can be updated in one block only the updates for the latent variables differ from the updates for the ordered probit model considered in Albert and Chib (1993). Of course, all full conditional distributions are proportional to the joint distribution $f(\boldsymbol{y}^*, \boldsymbol{y}, \boldsymbol{\beta}, \phi, \boldsymbol{c})$, so we can always take advantage of the factorization

$$\begin{aligned} f(\boldsymbol{y}^*, \boldsymbol{y}, \boldsymbol{\beta}, \phi, \boldsymbol{c}) &= \left[\prod_{t=1}^{T} f(y_t^*, y_t | y_0^*, \ldots, y_{t-1}^*, y_1, \ldots, y_{t-1}, \boldsymbol{\beta}, \phi, \boldsymbol{c})\right] \pi(y_0^*, \boldsymbol{\beta}, \phi, \boldsymbol{c}) \\ &= \left[\prod_{t=1}^{T} f(y_t | y_t^*, \boldsymbol{c}) f(y_t^* | y_{t-1}^*, \boldsymbol{\beta}, \phi)\right] \pi(y_0^*, \boldsymbol{\beta}, \phi, \boldsymbol{c}). \end{aligned} \tag{3.4}$$

3.2.1 Latent variable update

Since the joint density $f(\boldsymbol{y}^* | \boldsymbol{y}, \boldsymbol{\beta}, \phi, \boldsymbol{c})$ of the latent variables is proportional to $f(\boldsymbol{y}^*, \boldsymbol{y}, \boldsymbol{\beta}, \phi, \boldsymbol{c})$ and y_0^* is a priori $N(0, \sigma^2)$-distributed and independent of the other parameters, it follows from Equation (3.4) that

$$\begin{aligned} f(y_0^* | \boldsymbol{y}, \boldsymbol{y}_{-0}^*, \boldsymbol{\beta}, \phi, \boldsymbol{c}) &\propto f(y_1^* | y_0^*, \boldsymbol{\beta}, \phi) \pi(y_0^*) \\ &\propto \exp\left\{-\frac{1}{2}\left[(y_1^* - \boldsymbol{x}_1'\boldsymbol{\beta} - \phi y_0^*)^2 + (y_0^*)^2 \sigma^{-2}\right]\right\} \\ &\propto \exp\left\{-\frac{1}{2}\left[(y_0^*)^2\left(\phi^2 + \sigma^{-2}\right) - 2y_0^*\left[\phi\left(y_1^* - \boldsymbol{x}_1'\boldsymbol{\beta}\right)\right]\right]\right\}, \end{aligned}$$

so that

$$y_0^* \mid \boldsymbol{y}, \boldsymbol{y}_{-0}^*, \boldsymbol{\beta}, \phi, \boldsymbol{c} \sim N\left(\frac{\phi(y_1^* - \boldsymbol{x}_1'\boldsymbol{\beta})}{\phi^2 + \sigma^{-2}}, \frac{1}{\phi^2 + \sigma^{-2}}\right).$$

Furthermore, for $t = 1, \ldots, T-1$, we can see from Equation (3.4) that

$$\begin{aligned} f(y_t^* | \boldsymbol{y}, \boldsymbol{y}_{-t}^*, \boldsymbol{\beta}, \phi, \boldsymbol{c}) &\propto f(y_t | y_t^*, \boldsymbol{c}) f(y_t^* | y_{t-1}^*, \boldsymbol{\beta}, \phi) f(y_{t+1}^* | y_t^*, \boldsymbol{\beta}, \phi) \\ &\propto \mathbb{1}_{\{y_t^* \in [c_{y_t-1}, c_{y_t})\}} \exp\left\{-\frac{1}{2}\left[\left(y_t^* - \boldsymbol{x}_t'\boldsymbol{\beta} - \phi y_{t-1}^*\right)^2 + \left(y_{t+1}^* - \boldsymbol{x}_{t+1}'\boldsymbol{\beta} - \phi y_t^*\right)^2\right]\right\} \\ &\propto \mathbb{1}_{\{y_t^* \in [c_{y_t-1}, c_{y_t})\}} \exp\left\{-\frac{1}{2}\left[(y_t^*)^2 - 2y_t^*(\boldsymbol{x}_t'\boldsymbol{\beta} + \phi y_{t-1}^*) + \phi^2 (y_t^*)^2 - 2y_t^* \phi (y_{t+1}^* - \boldsymbol{x}_{t+1}'\boldsymbol{\beta})\right]\right\} \\ &= \mathbb{1}_{\{y_t^* \in [c_{y_t-1}, c_{y_t})\}} \exp\left\{-\frac{1}{2}\left[(y_t^*)^2(1+\phi^2) - 2y_t^*\left[\phi(y_{t+1}^* - \boldsymbol{x}_{t+1}'\boldsymbol{\beta}) + (\boldsymbol{x}_t'\boldsymbol{\beta} + \phi y_{t-1}^*)\right]\right]\right\}. \end{aligned}$$

Therefore $f(y_t^*|\boldsymbol{y}, \boldsymbol{y}_{-t}^*, \boldsymbol{\beta}, \phi, \boldsymbol{c})$ $(t = 1, \ldots, T-1)$ is a truncated normal distribution, in particular

$$y_t^*|\boldsymbol{y}, \boldsymbol{y}_{-t}^*, \boldsymbol{\beta}, \phi, \boldsymbol{c} \sim N_{[c_{y_t-1}, c_{y_t})}\left(\frac{\phi(y_{t+1}^* - \boldsymbol{x}_{t+1}'\boldsymbol{\beta}) + (\boldsymbol{x}_t'\boldsymbol{\beta} + \phi y_{t-1}^*)}{1+\phi^2}, \frac{1}{1+\phi^2}\right).$$

For $f(y_T^*|\boldsymbol{y}, \boldsymbol{y}_{-T}^*, \boldsymbol{\beta}, \phi, \boldsymbol{c})$ we get from Equation (3.4)

$$\begin{aligned} f(y_T^*|\boldsymbol{y}, \boldsymbol{y}_{-T}^*, \boldsymbol{\beta}, \phi, \boldsymbol{c}) &\propto f(y_T|y_T^*, \boldsymbol{c}) f(y_T^*|y_{T-1}^*, \boldsymbol{\beta}, \phi) \\ &\propto \mathbb{1}_{\{y_T^* \in [c_{y_T-1}, c_{y_T})\}} \exp\left\{-\frac{1}{2}\left[\left(y_T^* - \boldsymbol{x}_T'\boldsymbol{\beta} - \phi y_{T-1}^*\right)^2\right]\right\} \end{aligned}$$

which is again a truncated normal distribution, in particular

$$y_T^*|\boldsymbol{y}, \boldsymbol{y}_{-T}^*, \boldsymbol{\beta}, \phi, \boldsymbol{c} \sim N_{\left[c_{y_T-1}, c_{y_T}\right)}\left(\boldsymbol{x}_T'\boldsymbol{\beta} + \phi y_{T-1}^*, 1\right).$$

3.2.2 Joint regression and autoregressive parameter update

Here we can update all the parameters β_j, $j = 0, \ldots, p$, and ϕ in one block. The derivation of the full conditional is completely analogous to the ordered probit model (cf. Albert and Chib (1993)).

Defining $\boldsymbol{b}_t' := (\boldsymbol{x}_t', y_{t-1}^*)$ for $t = 1, \ldots, T$ and $D := \operatorname{diag}(\tau^{-2}, \ldots, \tau^{-2}, \rho^{-2})$ we can again use Equation (3.4) to see that for $\boldsymbol{\theta} = (\beta_0, \ldots, \beta_p, \phi)'$

$$\begin{aligned} f(\boldsymbol{\theta}|\boldsymbol{y}, \boldsymbol{y}^*, \boldsymbol{c}) &\propto \left[\prod_{t=1}^{T} f(y_t^*|y_{t-1}^*, \boldsymbol{\theta})\right] \pi(\boldsymbol{\theta}) \\ &\propto \exp\left\{-\frac{1}{2}\left[\sum_{t=1}^{T}(y_t^* - \boldsymbol{x}_t'\boldsymbol{\beta} - \phi y_{t-1}^*)^2 + \sum_{j=0}^{p}\tau^{-2}\beta_j + \rho^{-2}\phi^2\right]\right\} \\ &= \exp\left\{-\frac{1}{2}\left[\sum_{t=1}^{T}(y_t^* - \boldsymbol{b}_t'\boldsymbol{\theta})^2 + \boldsymbol{\theta}' D \boldsymbol{\theta}\right]\right\} \\ &\propto \exp\left\{-\frac{1}{2}\left[\sum_{t=1}^{T}(-y_t^*\boldsymbol{b}_t'\boldsymbol{\theta} - \boldsymbol{b}_t'\boldsymbol{\theta} y_t^* + \boldsymbol{b}_t'\boldsymbol{\theta}\boldsymbol{b}_t'\boldsymbol{\theta}) + \boldsymbol{\theta}' D \boldsymbol{\theta}\right]\right\} \\ &= \exp\left\{-\frac{1}{2}\left[-2\left(\sum_{t=1}^{T} y_t^*\boldsymbol{b}_t'\right)\boldsymbol{\theta} + \boldsymbol{\theta}'\left[\left(\sum_{t=1}^{T}\boldsymbol{b}_t\boldsymbol{b}_t'\right) + D\right]\boldsymbol{\theta}\right]\right\}. \end{aligned} \quad (3.5)$$

Using the $T \times (p+2)$-matrix Z defined by

$$Z := \begin{bmatrix} \boldsymbol{b}_1' \\ \vdots \\ \boldsymbol{b}_T' \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1p} & y_0^* \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{T1} & \cdots & x_{Tp} & y_{T-1}^* \end{bmatrix}$$

we can derive from Equation (3.5) that

$$\boldsymbol{\theta}|\boldsymbol{y}, \boldsymbol{y}^*, \boldsymbol{c} \sim N_{p+2}(\Sigma Z' \boldsymbol{y}^*_{-0}, \Sigma)$$

which is a $(p+2)$-dimensional normal distribution with covariance matrix $\Sigma := (Z'Z + D)^{-1}$.

3.2.3 Cutpoint parameter update

We are now interested in the full conditionals $f(c_k|\boldsymbol{y}, \boldsymbol{y}^*, \boldsymbol{\beta}, \phi, \boldsymbol{c}_{-k})$ for $k \in \{2, \ldots, K-1\}$. From Equation (3.4) it follows that

$$\begin{aligned} f(c_k|\boldsymbol{y}, \boldsymbol{y}^*, \boldsymbol{\beta}, \phi, \boldsymbol{c}_{-k}) &\propto \left[\prod_{t=1}^{T} f(y_t|y_t^*, \boldsymbol{c})\right] \pi(\boldsymbol{c}) \\ &= \left[\prod_{t=1}^{T} \mathbb{1}_{\left\{y_t^* \in \left[c_{y_t-1}, c_{y_t}\right)\right\}}\right] \mathbb{1}_{\{0<c_2<\ldots<c_{K-1}<C\}} \\ &= \left[\prod_{\{t=1,\ldots,T|y_t=k\}} \mathbb{1}_{\{y_t^* \in [c_{k-1}, c_k)\}}\right] \left[\prod_{\{t=1,\ldots,T|y_t=k+1\}} \mathbb{1}_{\{y_t^* \in [c_k, c_{k+1})\}}\right] \mathbb{1}_{\{0<c_2<\ldots<c_{K-1}<C\}}. \end{aligned}$$

Obviously this is an uniform distribution for c_k which is unequal zero if and only if all the following conditions are fulfilled:

$$\begin{aligned} c_k &> y_t^* \text{ for all } t \in \{1, \ldots, T\}, \text{ where } y_t = k, \\ c_k &> c_{k-1}, \\ c_k &< y_t^* \text{ for all } t \in \{1, \ldots, T\}, \text{ where } y_t = k+1, \\ c_k &< c_{k+1}. \end{aligned}$$

We conclude that $f(c_k|\boldsymbol{y}, \boldsymbol{y}^*, \boldsymbol{\beta}, \phi, \boldsymbol{c}_{-k})$ is an uniform distribution in the interval (l_k, r_k), where

$$l_k := \max\left\{c_{k-1}, \max_{t=1,\ldots,T}\{y_t^*|y_t = k\}\right\}, \tag{3.6}$$

$$r_k := \min\left\{c_{k+1}, \min_{t=1,\ldots,T}\{y_t^*|y_t = k+1\}\right\}. \tag{3.7}$$

3.3 GM-MGMC sampler

Simulation experiments with the standard Gibbs sampler developed in Section 3.2 show that the produced MCMC-chains converge very slowly to the region around the true

value especially for the cutpoints c_j and the regression intercept β_0 (cf. Subsection 3.3.2). This behavior was also observed by Cowles (1996) for the independent multinomial case, which can be explained as follows. The parameter c_j is drawn from a uniform distribution with boundaries l_j and r_j given in Equations (3.6) and (3.7). If the observed data set is large (e.g. $T = 2000$), the difference $r_j - l_j$ is very small, so c_j has very little room to move in one iteration. Therefore we look for some possibilities to speed up the convergence of the standard Gibbs sampler.

One general possibility is, of course, to update some variables in one block, for example all latent variables y_t^*. In this case we have to draw a sample of a $(T+1)$-dimensional truncated normal distribution instead of drawing $T+1$ samples from univariate truncated normal distributions. However, to get a sample from a multivariate truncated normal distribution one has to employ a Gibbs sampler itself (cf. Geweke (1991) or Robert (1995)). Simulations show that one reaches an improvement in convergence, but the computational cost is very high. So if one uses this method fewer iterations are needed for a comparable result, but the time used for each iteration increases in such a way that the overall improvement is negligible. In addition this blocking of the latent variables does not involve the update of the cutpoints c_k which seems to be most important. Therefore, we use now a method that was proposed by Liu and Sabatti (2000).

3.3.1 Development of an appropriate grouped-move-step

Using the same notation, we apply the Theorem by Liu and Sabatti (2000) given in Section 2.4 where $\boldsymbol{w}$ is a vector with $T + p + K + 1$ components, namely

$$\boldsymbol{w} = (y_0^*, \ldots, y_T^*, \beta_0, \ldots, \beta_p, c_2, \ldots, c_{K-1}, \phi)\,,$$

and π is the posterior density of $\boldsymbol{w}$. The difficulty in the choice of a suitable transformation group is to find one where the resulting distribution allows to draw samples very fast. Unfortunately, in our problem standard transformation groups as the translation group or the scale group do not lead to an easy sampling distribution. Therefore we use the group

$$\Gamma_m := \{\gamma > 0 : \gamma(\boldsymbol{w}) = (\gamma w_1, \ldots, \gamma w_m, w_{m+1}, \ldots, w_d)\}$$

which we call a **partial scale group on $\boldsymbol{S}$**. Here only m components are transformed, the others remain fixed. The left-Haar measure for this group is again $\gamma^{-1}d\gamma$ as for the (total) scale group. We easily compute $\det\left(\partial\gamma(\boldsymbol{w})/\partial\boldsymbol{w}\right) = \gamma^m$. Therefore

$$\pi(\gamma(\boldsymbol{w}))|J_\gamma(\boldsymbol{w})|L(d\gamma) = \gamma^{m-1}\pi(\gamma\boldsymbol{w})\,d\gamma.$$

In order to get an easy sampling distribution for our problem we take $m = T + p + K$ and let only the parameter ϕ remain fixed. Therefore

$$\gamma(\boldsymbol{w}) = (\gamma y_0^*, \ldots, \gamma y_T^*, \gamma\beta_0, \ldots, \gamma\beta_p, \gamma c_2, \ldots, \gamma c_{K-1}, \phi)\,.$$

The posterior distribution in our problem is given by

$$\begin{aligned}
\pi(\boldsymbol{w}|\boldsymbol{y}) &= \pi(y_0^*, \ldots, y_T^*, \boldsymbol{\beta}, \boldsymbol{c}, \phi | y_1, \ldots, y_T) \\
&\propto \exp\left\{-\frac{1}{2}\left[\sum_{t=1}^{T}(y_t^* - \boldsymbol{\beta}'\boldsymbol{x}_t - \phi y_{t-1}^*)^2 + \sigma^{-2}(y_0^*)^2 + \tau^{-2}\boldsymbol{\beta}'\boldsymbol{\beta} + \rho^{-2}\phi^2\right]\right\} \cdot \\
&\quad \cdot \left[\prod_{t=1}^{T} \mathbb{1}_{[c_{y_t-1}, c_{y_t})}(y_t^*)\right] \mathbb{1}_{\{0<c_2<\ldots<c_{K-1}<C\}}.
\end{aligned}$$

The density $\gamma^{m-1}\pi(\gamma\boldsymbol{w}|\boldsymbol{y})$ is therefore proportional to

$$\begin{aligned}
&\gamma^{m-1} \exp\left\{-\frac{1}{2}\left[\sum_{t=1}^{T}(\gamma y_t^* - \gamma\boldsymbol{\beta}'\boldsymbol{x}_t - \phi\gamma y_{t-1}^*)^2 + \sigma^{-2}(\gamma y_0^*)^2 + \tau^{-2}\gamma^2\boldsymbol{\beta}'\boldsymbol{\beta} + \rho^{-2}\phi^2\right]\right\} \cdot \\
&\quad \cdot \left[\prod_{t=1}^{T} \mathbb{1}_{[\gamma c_{y_t-1}, \gamma c_{y_t})}(\gamma y_t^*)\right] \mathbb{1}_{\{0<\gamma c_2<\ldots<\gamma c_{K-1}<C\}} \\
&\propto \gamma^{m-1} \exp\left\{-\frac{1}{2}\gamma^2\left[\sum_{t=1}^{T}(y_t^* - \boldsymbol{\beta}'\boldsymbol{x}_t - \phi y_{t-1}^*)^2 + \sigma^{-2}(y_0^*)^2 + \tau^{-2}\boldsymbol{\beta}'\boldsymbol{\beta}\right]\right\} \cdot \\
&\quad \cdot \left[\prod_{t=1}^{T} \mathbb{1}_{[c_{y_t-1}, c_{y_t})}(y_t^*)\right] \mathbb{1}_{\{0<\gamma c_2<\ldots<\gamma c_{K-1}<C\}}. \qquad (3.8)
\end{aligned}$$

For all $\gamma > 0$ we have the equivalence

$$\begin{aligned}
&[0 < \gamma c_2 < \ldots < \gamma c_{K-1} < C \text{ and } c_{K-1} < C] \\
&\qquad \Longleftrightarrow \quad \left[0 < c_2 < \ldots < c_{K-1} < C \text{ and } \gamma^2 < C^2/c_{K-1}^2\right].
\end{aligned}$$

Since expression (3.8) is considered to be a density for γ (up to a normalizing constant), and since during all updates of the MCMC sampler the condition $0 < c_2 < \ldots < c_{K-1} < C$ is always fulfilled, this equivalence leads to the proportionality

$$\begin{aligned}
\mathbb{1}_{\{0<\gamma c_2<\ldots<\gamma c_{K-1}<C\}} &\propto \mathbb{1}_{\{0<\gamma c_2<\ldots<\gamma c_{K-1}<C\}}\mathbb{1}_{\{c_{K-1}<C\}} \\
&= \mathbb{1}_{\{0<c_2<\ldots<c_{K-1}<C\}}\mathbb{1}_{\{\gamma^2<C^2/c_{K-1}^2\}} \\
&\propto \mathbb{1}_{\{\gamma^2<C^2/c_{K-1}^2\}}.
\end{aligned}$$

Therefore expression (3.8) simplifies to

$$\left(\gamma^2\right)^{\frac{m-1}{2}} \exp\left\{-\frac{1}{2}\gamma^2\left[\sum_{t=1}^{T}(y_t^* - \boldsymbol{\beta}'\boldsymbol{x}_t - \phi y_{t-1}^*)^2 + \sigma^{-2}(y_0^*)^2 + \tau^{-2}\boldsymbol{\beta}'\boldsymbol{\beta}\right]\right\} \mathbb{1}_{\{\gamma^2<C^2/c_{K-1}^2\}}$$

which is proportional to a Gamma distribution $\Gamma(a,b)$ truncated to $\left(0, C^2/c_{K-1}^2\right)$ for γ^2 with parameters

$$a = \frac{T+K+p+1}{2}, \tag{3.9}$$

$$b = \frac{\sum_{t=1}^{T}(y_t^* - \boldsymbol{x}_t'\boldsymbol{\beta} - \phi y_{t-1}^*)^2 + \sigma^{-2}(y_0^*)^2 + \tau^{-2}\boldsymbol{\beta}'\boldsymbol{\beta}}{2}. \tag{3.10}$$

Here the $\Gamma(a,b)$ density is given by $f_{\Gamma(a,b)}(x) = b^a x^{a-1} e^{-bx}/\Gamma(a)$ for $x \geq 0$.

One can easily sample from the truncated Gamma distribution by rejection sampling (cf. Section 2.1). Of course, if one chooses a prior for $\boldsymbol{c}$ with infinite support, i.e. $C = \infty$, one has to draw from the corresponding non-truncated Gamma distribution.

In this way we get a new algorithm that lies in the class of the grouped move multigrid Monte Carlo (GM-MGMC) algorithms (Liu and Sabatti (2000)). Each iteration consists of the following two parts:

Algorithm 3.1 One iteration of the GM-MGMC sampler for the AOP model

1. **MCMC-Step**: Generate an iteration from the standard Gibbs-sampler using
   - latent variable update,
   - joint regression and autoregressive parameter update,
   - cutpoint parameter update,

   to get $\boldsymbol{y}_{cur}^*, \boldsymbol{\beta}_{cur}, \phi_{cur}, \boldsymbol{c}_{cur}$ as current values.

2. **GM-Step**: Draw γ^2 from $\Gamma(a,b)$ truncated to $\left(0, C^2/c_{cur,K-1}^2\right)$ with a and b defined in (3.9) and (3.10) respectively, and update the current values by multiplication with the group element $\gamma = \sqrt{\gamma^2}$,

$$\begin{aligned} \boldsymbol{y}_{new}^* &\leftarrow \gamma \boldsymbol{y}_{cur}^*, \\ \boldsymbol{\beta}_{new} &\leftarrow \gamma \boldsymbol{\beta}_{cur}, \\ \boldsymbol{c}_{new} &\leftarrow \gamma \boldsymbol{c}_{cur}. \end{aligned}$$

Note that ϕ_{cur} does not need to be updated since it remains unchanged under the partial scale group.

3.3.2 An illustration: Standard sampler against GM-MGMC

We now illustrate the improvement in convergence which is achieved by adding the GM-step presented in the previous subsection to the standard sampler. For this pur-

pose we first simulated two covariates x_{t1} and x_{t2} independently from $N(-1, 1)$ and $N(-0.25, 0.18^2)$, respectively, for $t = 1, \ldots, 2000$. Using these covariates we then simulated one data set of length $T = 2000$ from the autoregressive ordered probit model with parameters $c_2 = 1.2$, $c_3 = 2.2$, $c_4 = 3.1$, $c_5 = 4.1$, $c_6 = 5.3$, $\beta_0 = 2.9$, $\beta_1 = -0.6$, $\beta_2 = 9.0$ and $\phi = 0.5$.

We run both the standard Gibbs sampler and the GM-MGMC sampler for the simulated data set for 15000 iterations. The starting values for the cutpoints $c_2, \ldots, c_6$ are $2, 4, 6, 8, 10$, respectively, and 0.0 for each of the regression coefficients. Here and in the following section we always use the hyperparameters $C = \infty$, $\sigma = 1$, $\tau = 100$, and $\rho = 0.5$ for the prior distributions of $\boldsymbol{c}$, y_0^*, β_j $(j = 0, 1, 2)$, and ϕ, respectively (cf. Equation (3.3)).

Figures 3.1 and 3.2 demonstrate the fundamental superiority of the GM-MGMC sampler. As can be seen from Figure 3.1 the chains for the cutpoints produced by the standard Gibbs sampler move very slowly to the regions around the true values which are indicated by the horizontal lines. The sharp drop in the chains of c_4, c_5, and c_6 from the standard Gibbs sampler at the beginning can be explained by the fact that the samples of $\{y_t^* | y_t \in \{4, 5, 6\}\}$ lie close to the lower limit of the corresponding intervals. Hence the cutpoints c_4, c_5, and c_6 have more room to move in the first iterations. We can guess that it will take thousands of iterations until the chains for the cutpoints produced by the standard Gibbs sampler will have converged. The same holds for the chains for the regression coefficients, given in Figure 3.2. The chains produced by the GM-MGMC sampler are also given in Figures 3.1 and 3.2. They converge within about only 20 iterations, and this holds for both the cutpoints and for the regression coefficients. Finally we note that the chain for the autoregressive parameter ϕ converges quite fast for both samplers and is therefore not shown here.

We further determined the autocorrelations in the chains after a burn-in period of 5000 iterations for both samplers. As can be seen from Figure 3.3, the GM-MGMC sampler is better also from this point of view, since the autocorrelations in the GM-MGMC chains are much smaller than in the chains from the standard sampler.

3.4 Simulation study

Chen, Shao, and Ibrahim (2000) point out that GM-MGMC algorithms do not always guarantee faster convergence than the parent MCMC algorithm. Therefore we now test

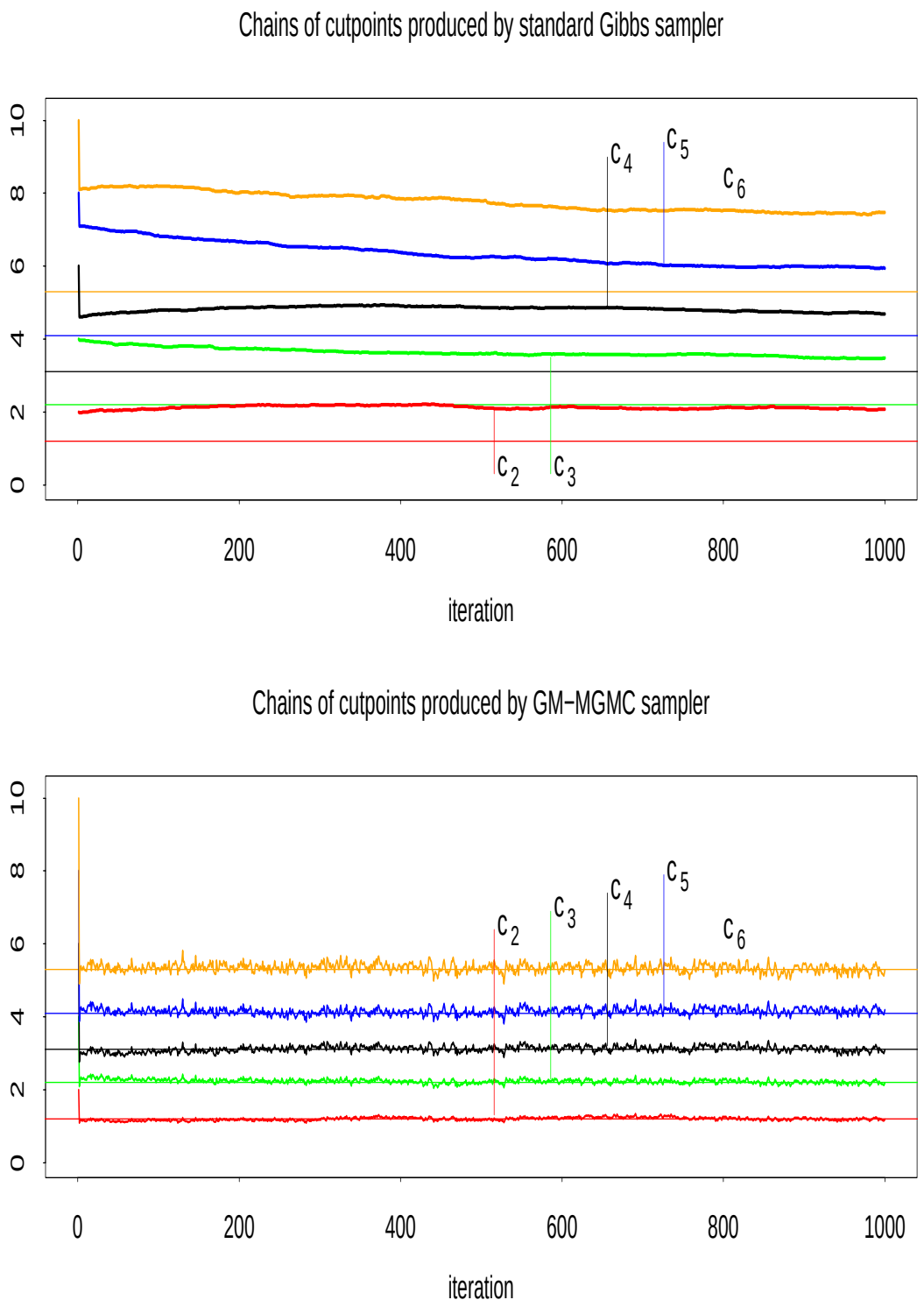

Figure 3.1: First 1000 iterations of chains for cutpoints produced by standard Gibbs sampler (above) and GM-MGMC sampler (below). The horizontal thin lines indicate the true values.

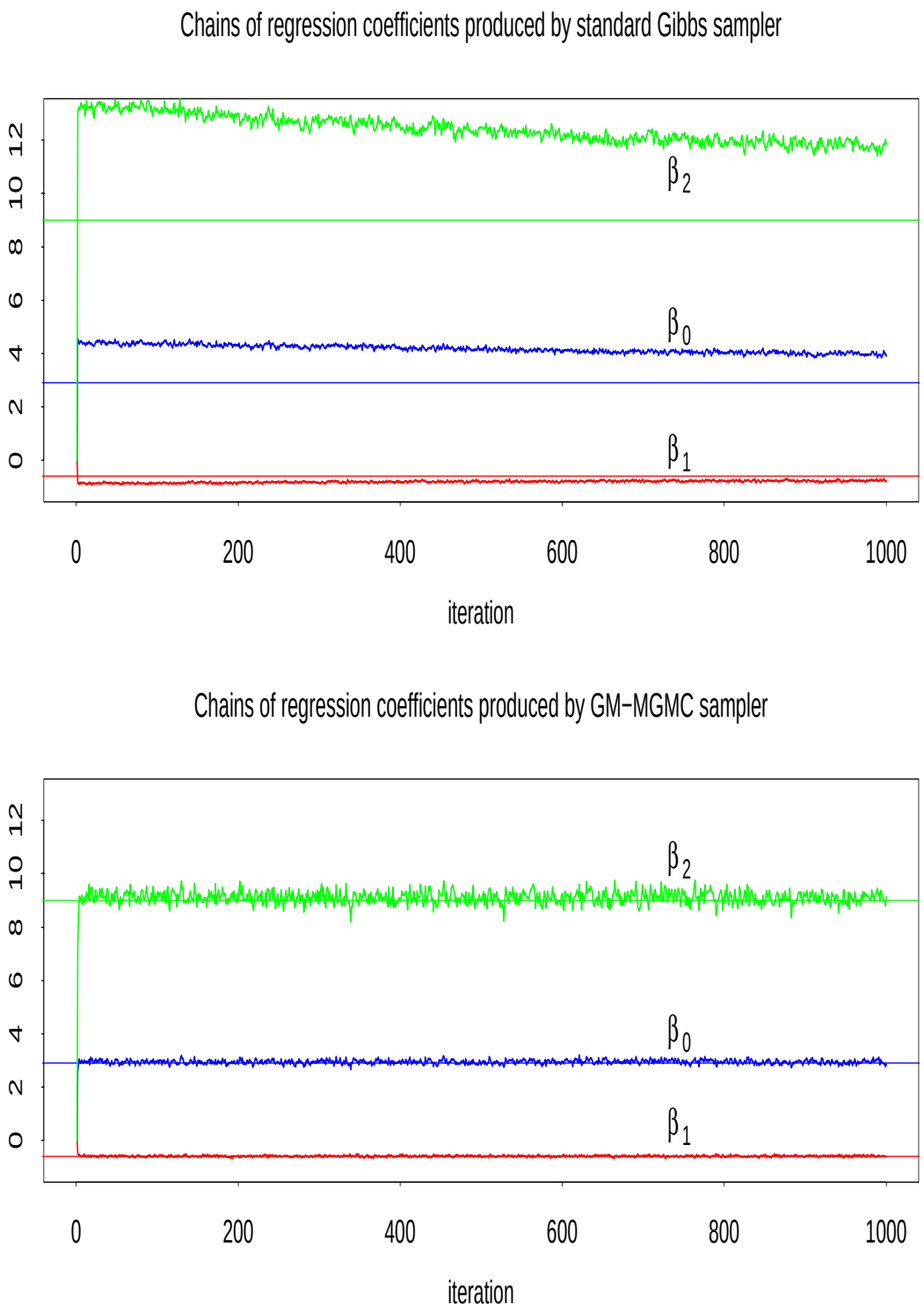

Figure 3.2: First 1000 iterations of chains for regression coefficients produced by standard Gibbs sampler (above) and GM-MGMC sampler (below). The horizontal thin lines indicate the true values.

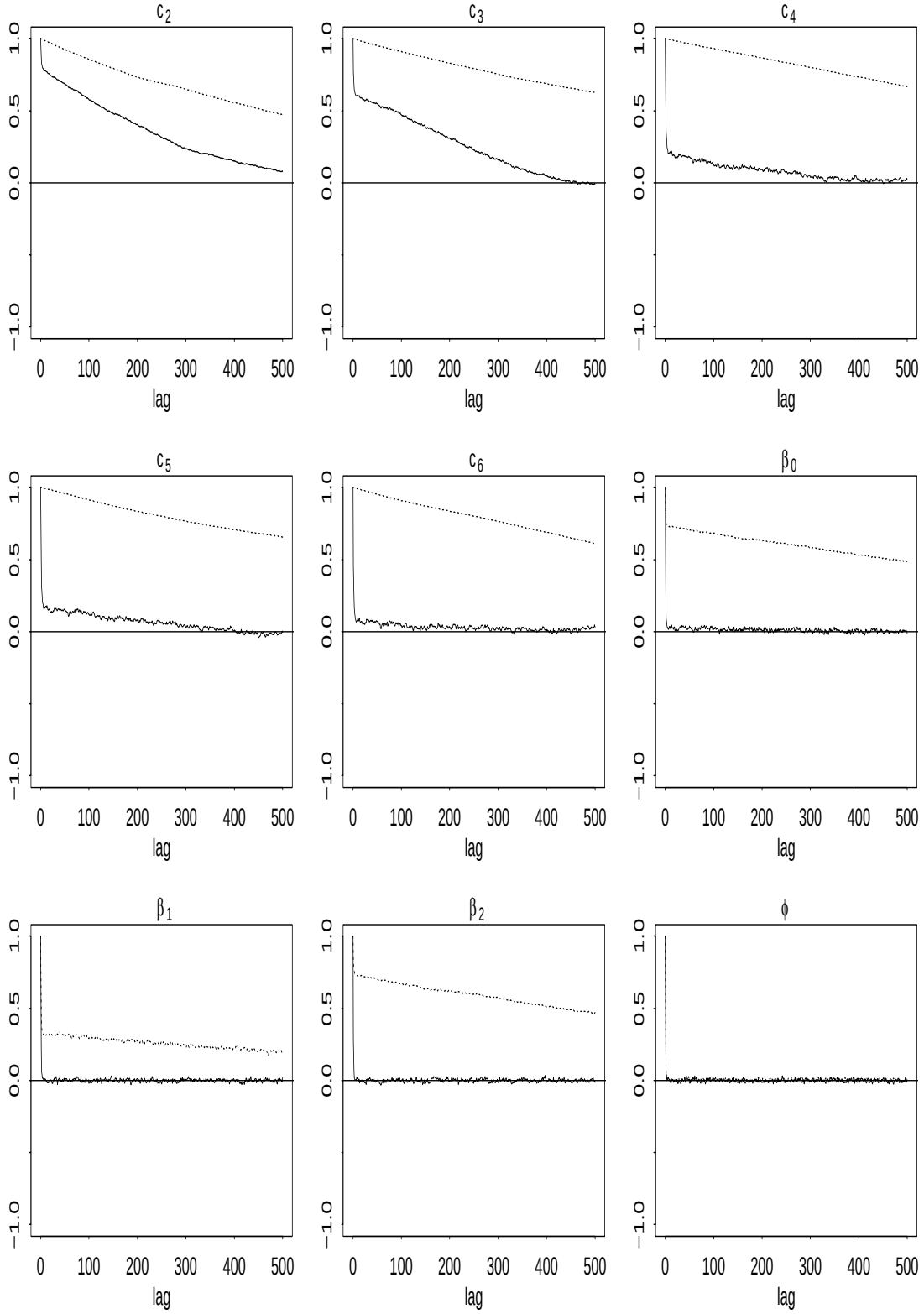

Figure 3.3: Autocorrelations of chains produced by standard Gibbs sampler (dotted) and GM-MGMC sampler (solid).

this GM-MGMC algorithm in different situations and investigate especially the accuracy of the posterior mean estimates.

3.4.1 Design of the simulation study

First we mention that the smaller the number K of categories is the better the parameter estimates are when the GM-MGMC sampler is used. Therefore we investigate the behavior of this Gibbs sampler for data sets with a for our practical concerns relative high number of categories, $K = 7$. Further we take $T = 2000$.

We investigate the behavior of the GM-MGMC Gibbs sampler for six different parameter sets that differ in the sign of the autoregressive parameter ϕ, in the frequencies that occur in the categories, and in the covariates. See Table 3.1 for specific choices made. Especially concerning the frequencies we are interested in whether the parameter estimates are better if the frequencies in the categories are nearly identical than in situations where the majority of the observations lies in one or two categories and only few observations in the other categories. This is important for many practical problems, for example for our application in Chapter 5. There more than 80% of the observations lie in the categories 1 and 2, and only 4.2% in category 4. In the following, 'very different' frequencies means that there is at least one category with more than 40% of the observations and at least one category with less than 5% of the observations. 'Nearly identical' means that in each category lie between 11% and 18% of the observations.

The parameter sets A to D are connected in the sense that in each we use two covariates generated from normal distributions with specified mean and variance values, and that there appear all combinations of positive/negative ϕ and nearly identical/very different frequencies. In parameter set E, covariate x_{t1} is an exponential trend with

Parameter set	ϕ	Frequencies in categories	Covariates
A	negative	nearly identical	2 (normally distributed)
B	positive	nearly identical	2 (normally distributed)
C	negative	very different	2 (normally distributed)
D	positive	very different	2 (normally distributed)
E	negative	nearly identical	2 (exp.trend,Bernoulli(0.6))
F	negative	nearly identical	4 (normally distributed)

Table 3.1: Design of the simulation study.

	x_{t1}		x_{t2}		x_{t3}		x_{t4}	
Parameter set	mean	stdd.	mean	stdd.	mean	stdd.	mean	stdd.
A	-0.40	2.20	0.02	0.02				
B	-1.00	1.00	-0.25	0.18				
C	0.00	0.10	0.30	0.90				
D	-1.00	0.90	-0.25	0.11				
F	0.00	0.80	1.60	1.10	0.40	0.60	-0.50	0.40

Table 3.2: Means and standard deviations of the normal distributions from which the covariates are generated for parameter sets A, B, C, D, and F.

Parameter set	c_2	c_3	c_4	c_5	c_6	β_0	β_1	β_2	β_3	β_4	ϕ
A	1.3	2.4	3.5	4.6	5.7	4.9	0.8	-60.0			-0.2
B	1.2	2.2	3.1	4.1	5.3	2.9	-0.6	9.0			0.5
C	0.7	1.7	3.0	4.0	4.7	2.9	7.9	0.6			-0.3
D	0.8	1.9	3.5	4.7	5.5	3.3	-0.6	9.0			0.4
E	0.6	1.3	2.4	3.1	3.7	2.7	0.2	-1.1			-0.3
F	0.7	1.8	3.5	4.6	5.3	2.8	-0.8	0.4	0.5	0.4	-0.3

Table 3.3: Settings of cutpoints, regression coefficients, and autoregressive parameter.

$x_{t1} = e^{t/T} = e^{t/2000}$ and $x_{t2},\ t = 1, \ldots, 2000$ is drawn from a Bernoulli-distribution with success probability 0.6. In parameter set F we use four covariates generated from normal distributions.

For each parameter set we first chose the value of the autoregressive parameter ϕ. Then we tried not only different cutpoints to get nearly identical or very different frequencies in the categories, but also different standard deviations of the normal distributions the covariates were generated from. This is because otherwise we would have had very small or very high values for the cutpoint $c_{K-1} = c_6$, which would mean that the influence of the noise would have been very large or very small. The means and standard deviations of the normal distributions used in parameter sets A, B, C, D, and F are given in Table 3.2. Figure 3.4 shows the densities of $\beta_j x_{tj},\ j = 1, 2$, the sum $\boldsymbol{x}_t'\boldsymbol{\beta}$, and the error component ε_t^* for parameter sets A to D. In the fourth and fifth column one can compare the influence of $\boldsymbol{x}_t'\boldsymbol{\beta}$ and of the noise ε_t^*. The specific settings of the cutpoints, regression coefficients, and autoregressive parameter are given in Table 3.3.

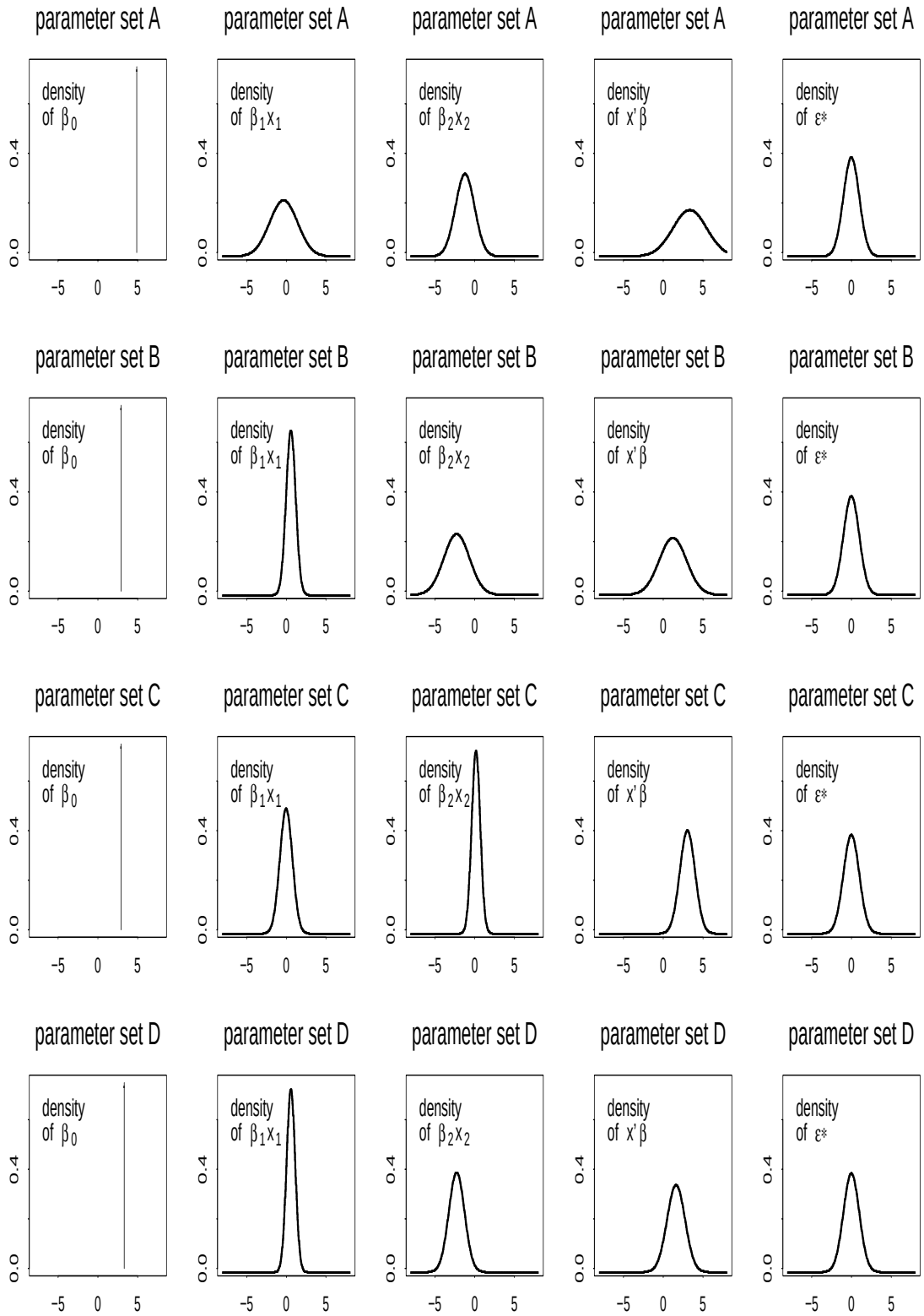

Figure 3.4: Generating densities for $\beta_j x_{tj}$, $j = 1, 2$, in the parameter sets A to D, densities for the sum $\boldsymbol{x}_t'\boldsymbol{\beta}$ and the error component $\varepsilon_t^* \sim N(0, 1)$.

The simulation consists of the following three steps:

1. Generation of one design matrix per parameter set.
2. Simulation of 100 data sets per parameter set using the design of step 1.
3. 15000 iterations of the GM-MGMC Gibbs sampler for each of the data sets.

3.4.2 Results of the simulation study

The computing time of the C++ program is about 0.029 seconds per iteration in presence of two covariates and about 0.045 seconds in presence of four covariates on an UltraSPARC III Cu 900 Mhz processor, so that we had an over-all computing time of about 80 hours. As starting values we took $\beta_j = 0$, $j = 0, \ldots, p$, $\phi = 0$, $(c_2, c_3, c_4, \ldots) = (2, 4, 6, \ldots)$, and randomly drawn values uniformly distributed between c_{y_t-1} and c_{y_t} for y_t^*, $t = 1, \ldots, 2000$. After running the GM-MGMC sampler we estimated the parameters by the posterior means using the iterations 3001 to 15000, and computed estimates of the relative bias, the standard deviation for the relative bias, the relative MSE, and the standard deviation for the relative MSE for each parameter in each parameter set. The estimates of the relative bias ($\widehat{\mathrm{B}}_{\mathrm{rel}}$), the relative MSE ($\widehat{\mathrm{MSE}}_{\mathrm{rel}}$), and of the corresponding standard deviations are defined as

$$\begin{aligned}
\widehat{\mathrm{B}}_{\mathrm{rel}} &:= \frac{1}{\psi}\frac{1}{R}\sum_{r=1}^{R}\left(\hat{\psi}_r - \psi\right), \\
\widehat{\mathrm{MSE}}_{\mathrm{rel}} &:= \frac{1}{\psi^2}\frac{1}{R}\sum_{r=1}^{R}\left(\hat{\psi}_r - \psi\right)^2, \\
\widehat{\mathrm{std}}(\widehat{\mathrm{B}}_{\mathrm{rel}}) &:= \frac{1}{\psi}\sqrt{\frac{1}{R(R-1)}\sum_{r=1}^{R}\left[\hat{\psi}_r - \psi - \psi\widehat{\mathrm{B}}_{\mathrm{rel}}\right]^2}, \\
\widehat{\mathrm{std}}(\widehat{\mathrm{MSE}}_{\mathrm{rel}}) &:= \frac{1}{\psi^2}\sqrt{\frac{1}{R(R-1)}\sum_{r=1}^{R}\left[\left(\hat{\psi}_r - \psi\right)^2 - \psi^2\widehat{\mathrm{MSE}}_{\mathrm{rel}}\right]^2},
\end{aligned}$$

where ψ is the parameter to be estimated and $\hat{\psi}_r$ the posterior mean estimate of ψ for the r^{th} data set based on 12000 iterations of the GM-MGMC algorithm. Here we used $\widehat{\mathrm{B}}_{\mathrm{rel}} = \psi^{-1}\hat{B}$, where $\hat{B}$ is the common bias estimate, and the independence of the estimated posterior means for different data sets.

Table 3.4 contains the estimates of the relative bias, relative MSE, and their standard deviations for the model parameters in the parameter sets A,B,C,D,E, and F. The au-

	A	B	C	D	E	F
	relative bias					
c_2	-0.0144	-0.0017	-0.0015	0.0010	0.0032	0.0186
c_3	-0.0036	-0.0039	-0.0011	0.0017	0.0026	0.0022
c_4	0.0065	-0.0025	0.0021	0.0009	0.0031	0.0053
c_5	0.0060	0.0011	0.0039	0.0010	0.0026	0.0071
c_6	0.0012	-0.0034	0.0072	0.0036	0.0017	-0.0064
β_0	0.0047	-0.0068	-0.0063	0.0035	0.0075	-0.0010
β_1	-0.0041	-0.0114	-0.0032	0.0014	0.0031	0.0036
β_2	-0.0030	-0.0037	0.0254	0.0045	0.0057	0.0045
β_3						-0.0325
β_4						0.0117
ϕ	-0.0273	0.0026	0.0063	0.0005	0.0005	0.0107
	standard deviation for relative bias					
c_2	0.0027	0.0031	0.0039	0.0001	0.0003	0.0070
c_3	0.0023	0.0021	0.0024	0.0002	0.0003	0.0031
c_4	0.0016	0.0019	0.0019	0.0001	0.0003	0.0019
c_5	0.0009	0.0016	0.0019	0.0001	0.0003	0.0015
c_6	0.0015	0.0013	0.0019	0.0004	0.0002	0.0012
β_0	0.0010	0.0029	0.0018	0.0004	0.0007	0.0035
β_1	0.0023	0.0028	0.0032	0.0001	0.0003	0.0022
β_2	0.0017	0.0023	0.0025	0.0005	0.0006	0.0044
β_3						0.0089
β_4						0.0102
ϕ	0.0029	0.0025	0.0042	0.0000	0.0000	0.0048
	relative MSE					
c_2	0.000907	0.000969	0.001513	0.001295	0.005365	0.005120
c_3	0.000547	0.000445	0.000590	0.000877	0.002018	0.000941
c_4	0.000290	0.000377	0.000353	0.000257	0.001294	0.000382
c_5	0.000121	0.000245	0.000364	0.000221	0.000854	0.000272
c_6	0.000225	0.000174	0.000422	0.000649	0.000458	0.000185
β_0	0.000128	0.000885	0.000353	0.001061	0.002759	0.001172
β_1	0.000531	0.000904	0.001034	0.002384	0.015712	0.000492
β_2	0.000310	0.000559	0.001254	0.000503	0.005167	0.001926
β_3						0.008809
β_4						0.010432
ϕ	0.001599	0.000632	0.001774	0.001164	0.001637	0.002347
	standard deviation for relative MSE					
c_2	0.000095	0.000114	0.000181	0.000130	0.000536	0.000736
c_3	0.000066	0.000042	0.000060	0.000088	0.000202	0.000115
c_4	0.000037	0.000035	0.000043	0.000026	0.000129	0.000042
c_5	0.000013	0.000033	0.000031	0.000022	0.000085	0.000033
c_6	0.000042	0.000020	0.000029	0.000065	0.000046	0.000037
β_0	0.000021	0.000078	0.000041	0.000106	0.000276	0.000149
β_1	0.000051	0.000092	0.000122	0.000238	0.001571	0.000066
β_2	0.000028	0.000074	0.000112	0.000050	0.000517	0.000171
β_3						0.000885
β_4						0.001636
ϕ	0.000132	0.000077	0.000205	0.000116	0.000164	0.000304

Table 3.4: Estimates of the relative bias, relative MSE, and their standard deviations for model parameters in the parameter sets A, B, C, D, E, and F.

toregressive parameter ϕ is always estimated quite well. The relative bias is in most cases between -1% and +1% for the cutpoints as well as for the regression parameters. The relative MSE is less than 0.001 for most of the parameters in the parameter sets A to D. That means that on average the estimates are less than 3% away from the true values. The relative MSE is worse for the parameter sets E and F. The worst value we get for β_1 in set E which is the regression parameter for the exponential trend. Further we like to mention that all these estimates are similar if one uses the iterations 5001 to 15000 instead of iterations 3001 to 15000.

For all data sets and parameter sets the autocorrelations in the chains show a similar behavior as in the example of Section 3.3.2 and are therefore not shown once more. Sometimes the autocorrelations oscillate around zero, but the absolute values of the autocorrelations decline similar to the curves in Figure 3.3. Note that the data set used in Section 3.3.2 is simulated with the parameters of parameter set B.

We conclude that the GM-MGMC Gibbs sampler works very well in most situations, especially there is no difference in the performance whether the autoregressive parameter is positive or negative and whether there are categories with very different frequencies or not. As expected, the fewer covariates we use, the better is the estimation. The autocorrelations in the chains produced by the GM-MGMC Gibbs sampler are explicitly better than those in the chains produced by the standard Gibbs sampler.

Chapter 4

Estimation of the Marginal Likelihood for the Autoregressive Ordered Probit Model

In this chapter we further deal with the autoregressive ordered probit model introduced in Chapter 3. An interesting question for the application of a certain model to a given data set is always whether this model fits the data set better or worse than other ones. To answer this question one can use for example Bayes factors (cf. Section 2.6) which require the estimation of the marginal likelihood for the data set under the model. In this chapter we therefore provide an estimation procedure for the marginal likelihood for the autoregressive ordered probit model defined by Equations (3.1) and (3.2).

Since in this model latent variables are involved, we use the marginal likelihood identity (cf. Section 2.6.2) in the form of Equation (2.18), which can here be written in the form

$$m(\boldsymbol{y}) = \frac{f(\boldsymbol{y}|\boldsymbol{c}^\diamond, \boldsymbol{\theta}^\diamond)\pi(\boldsymbol{c}^\diamond, \boldsymbol{\theta}^\diamond)}{\pi(\boldsymbol{c}^\diamond, \boldsymbol{\theta}^\diamond|\boldsymbol{y})} \tag{4.1}$$

where $\boldsymbol{c}^\diamond$ and $\boldsymbol{\theta}^\diamond = (\beta_0^\diamond, \ldots, \beta_p^\diamond, \phi^\diamond)'$ are corresponding posterior mean estimates. In the following sections we first consider a filtering procedure and then the estimation and computation of the three factors which appear on the right-hand side in Equation (4.1). $\mathcal{F}_t$ will always denote the vector of observations until time t, i.e. $\mathcal{F}_t := (y_1, \ldots, y_t)$.

4.1 Filtering

In this section we present a fully adapted SIR-based auxiliary particle filter for the autoregressive ordered probit model. This filtering procedure is important for the estimation of the likelihood ordinate $f(\boldsymbol{y}|\boldsymbol{c}^\diamond, \boldsymbol{\theta}^\diamond)$ and is part of Algorithm 4.2 presented later.

The general form of a SIR-based auxiliary particle filter is given by Algorithm 2.8 in Section 2.5.3. Using the notation of that section we choose

$$\begin{aligned} g(y_t^*, m|\mathcal{F}_t, \boldsymbol{c}, \boldsymbol{\theta}) &\propto f(y_t|y_t^*, \boldsymbol{c}, \boldsymbol{\theta}) f(y_t^*|y_{t-1}^{*m}, \boldsymbol{c}, \boldsymbol{\theta}) \\ &= \mathbb{1}_{\{y_t^* \in [c_{y_t-1}, c_{y_t})\}} N(y_t^*|\boldsymbol{x}_t'\boldsymbol{\beta} + \phi y_{t-1}^{*m}, 1) \end{aligned}$$

as proposal density. This immediately implies that the numerator equals the denominator in Equation (2.13) and therefore the weights given by Equation (2.13) all equal 1. Hence they are all the same and the algorithm is fully adapted to the model (cf. Section 2.5.3). Therefore the resampling step 3 of Algorithm 2.8 is not necessary and we just have to sample from the proposal density. Since

$$g(y_t^*, m|\mathcal{F}_t, \boldsymbol{c}, \boldsymbol{\theta}) = g(y_t^*|m, \mathcal{F}_t, \boldsymbol{c}, \boldsymbol{\theta}) g(m|\mathcal{F}_t, \boldsymbol{c}, \boldsymbol{\theta})$$

this can be done by first choosing $m \in \{1, \ldots, M\}$ with probability $g(m|\mathcal{F}_t, \boldsymbol{c}, \boldsymbol{\theta})$ and then sampling from $g(y_t^*|m, \mathcal{F}_t, \boldsymbol{c}, \boldsymbol{\theta})$. Since

$$\begin{aligned} g(m|\mathcal{F}_t, \boldsymbol{c}, \boldsymbol{\theta}) &= \int g(y_t^*, m|\mathcal{F}_t, \boldsymbol{c}, \boldsymbol{\theta})\, dy_t^* \\ &\propto \int \mathbb{1}_{\{y_t^* \in [c_{y_t-1}, c_{y_t})\}} N(y_t^*|\boldsymbol{x}_t'\boldsymbol{\beta} + \phi y_{t-1}^{*m}, 1)\, dy_t^* \qquad (4.2) \\ &= \int_{c_{y_t-1}}^{c_{y_t}} N(y_t^*|\boldsymbol{x}_t'\boldsymbol{\beta} + \phi y_{t-1}^{*m}, 1)\, dy_t^* \\ &= \Phi(c_{y_t} - \boldsymbol{x}_t'\boldsymbol{\beta} - \phi y_{t-1}^{*m}) - \Phi(c_{y_t-1} - \boldsymbol{x}_t'\boldsymbol{\beta} - \phi y_{t-1}^{*m}) =: v_m \qquad (4.3) \end{aligned}$$

where $\Phi(\cdot)$ denotes the probability function of the standard normal distribution, we must first evaluate the weights v_m appearing in Equation (4.3) for $m = 1, \ldots, M$, and then compute the probabilities $\psi_m := v_m / \sum_{j=1}^M v_j$ which are associated to the numbers $1, \ldots, M$. We note that the proportionality constant needed in Equation (4.2) is independent of y_t^* and m, so we can use the weights v_m directly without taking into account any normalizing constants.

After drawing m with probability ψ_m we then have to sample from

$$g(y_t^*|m, \mathcal{F}_t, \boldsymbol{c}, \boldsymbol{\theta}) \propto \mathbb{1}_{\{y_t^* \in [c_{y_t-1}, c_{y_t})\}} N(y_t^*|\boldsymbol{x}_t'\boldsymbol{\beta} + \phi y_{t-1}^{*m}, 1)$$

which is a normal distribution with mean $\boldsymbol{x}_t'\boldsymbol{\beta} + \phi y_{t-1}^{*m}$ and variance 1, truncated to $[c_{y_t-1}, c_{y_t})$.

The following Algorithm 4.1 summarizes the necessary steps to produce a sample $\{y_t^{*1}, \ldots, y_t^{*M}\}$ from $f(y_t^*|\mathcal{F}_t, \boldsymbol{c}, \boldsymbol{\theta})$ when a sample $\{y_{t-1}^{*1}, \ldots, y_{t-1}^{*M}\}$ from $f(y_{t-1}^*|\mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\theta})$ is given. For a full filtering procedure this algorithm has to be applied successively for $t = 1, \ldots, T$. To get started one has to draw a sample $\{y_0^{*1}, \ldots, y_0^{*M}\}$ from $f(y_0^*|\mathcal{F}_0, \boldsymbol{c}, \boldsymbol{\theta}) = f(y_0^*|\boldsymbol{c}, \boldsymbol{\theta})$. This density is just the $N(0, \sigma^2)$-density, since the prior of y_0^* is chosen to be independent of $\boldsymbol{c}$ and $\boldsymbol{\theta}$.

Algorithm 4.1 Fully adapted SIR-based auxiliary particle filter

1. Given $\{y_{t-1}^{*1}, \ldots, y_{t-1}^{*M}\}$ from $f(y_{t-1}^*|\mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\theta})$, calculate the weights

   $$v_m := \Phi(c_{y_t} - \boldsymbol{x}_t'\boldsymbol{\beta} - \phi y_{t-1}^{*m}) - \Phi(c_{y_t-1} - \boldsymbol{x}_t'\boldsymbol{\beta} - \phi y_{t-1}^{*m}), \qquad m \in \{1, \ldots, M\},$$

   and the corresponding probabilities $\psi_m := v_m / \sum_{j=1}^M v_j$. Set $k = 1$.

2. Choose $m \in \{1, \ldots, M\}$ with probability ψ_m and sample y_t^{*k} from

   $$N_{[c_{y_t-1}, c_{y_t})}(\boldsymbol{x}_t'\boldsymbol{\beta} + \phi y_{t-1}^{*m}, 1).$$

3. If $k < M$, increment k to $k + 1$ and go to Step 2.
   If $k = M$, consider $\{y_t^{*1}, \ldots, y_t^{*M}\}$ as a sample from $f(y_t^*|\mathcal{F}_t, \boldsymbol{c}, \boldsymbol{\theta})$.

4.2 Estimation of the likelihood ordinate

Since $f(\boldsymbol{y}|\boldsymbol{c}, \boldsymbol{\theta})$ can be decomposed into

$$f(\boldsymbol{y}|\boldsymbol{c}, \boldsymbol{\theta}) = \prod_{t=1}^{T} f(y_t|\mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\theta})$$

the likelihood ordinate $f(\boldsymbol{y}|\boldsymbol{c}, \boldsymbol{\theta})$ can be estimated by estimating all the one-step ahead densities $f(y_t|\mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\theta})$, $t = 1, \ldots, T$. The following procedure takes advantage of this decomposition. Although we want to evaluate $f(\boldsymbol{y}|\boldsymbol{c}^\diamond, \boldsymbol{\theta}^\diamond)$ we suppress the '$\diamond$' for notational convenience.

Algorithm 4.2 Estimation of the likelihood function

1. Set $t = 1$, initialize $\boldsymbol{c}, \boldsymbol{\theta}$ and obtain a sample $y_0^{*1}, \ldots, y_0^{*M}$ from $f(y_0^*|\boldsymbol{c}, \boldsymbol{\theta})$. Since the prior of y_0^* is chosen to be independent of $\boldsymbol{c}$ and $\boldsymbol{\theta}$, this density is just the $N(0, \sigma^2)$-density.

2. For each value y_{t-1}^{*m} from $f(y_{t-1}^*|\mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\theta})$ draw

$$y_t^{*(m)} \sim N(\boldsymbol{x}_t'\boldsymbol{\beta} + \phi y_{t-1}^{*m}, 1).$$

The set $\{y_t^{*(m)},\ m = 1, \ldots, M\}$ can be considered to be a sample from the density $f(y_t^*|\mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\theta})$. This follows directly from Algorithm 2.1, 3., with $h_1 = f(y_t^*|y_{t-1}^*, \mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\theta})$ and $h_2 = f(y_{t-1}^*|\mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\theta})$.

3. Estimate the one-step ahead density $f(y_t|\mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\theta})$ by

$$\hat{f}(y_t|\mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\theta}) := \frac{1}{M}\sum_{m=1}^{M} \mathbb{1}_{\left\{y_t^{*(m)} \in \left[c_{y_t-1}, c_{y_t}\right)\right\}}. \tag{4.4}$$

This estimate can be derived from the equation

$$f(y_t|\mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\theta}) = \int f(y_t|y_t^*, \mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\theta}) f(y_t^*|\mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\theta})\, dy_t^*.$$

Since $\{y_t^{*(m)},\ m = 1, \ldots, M\}$ represents a sample from $f(y_t^*|\mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\theta})$ (cf. Step 2) and

$$f(y_t|y_t^*, \mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\theta}) = f(y_t|y_t^*, \boldsymbol{c}) = \mathbb{1}_{\left\{y_t^* \in \left[c_{y_t-1}, c_{y_t}\right)\right\}}$$

the integral can be approximated by the classical Monte Carlo integration (cf. Section 2.3.1) by averaging the expressions $\mathbb{1}_{\{y_t^{*(m)} \in [c_{y_t-1}, c_{y_t})\}}$.

Of course, the estimator in Equation (4.4) will be zero when all the $y_t^{*(m)}$'s lie outside the interval $[c_{y_t-1}, c_{y_t})$. This would nullify the complete likelihood ordinate. Therefore M should be chosen not too small to avoid this problem and to achieve a sufficiently high accuracy.

4. Apply the filtering procedure in Algorithm 4.1 to obtain a sample $y_t^{*1}, \ldots, y_t^{*M}$ from $f(y_t^*|\mathcal{F}_t, \boldsymbol{c}, \boldsymbol{\theta})$.

5. If $t < T$, increment t to $t + 1$ and go to Step 2.

6. Return the estimated log likelihood ordinate

$$\log \hat{f}(\boldsymbol{y}|\boldsymbol{c}, \boldsymbol{\theta}) := \sum_{t=1}^{T} \log \hat{f}(y_t|\mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\theta}).$$

4.3 Computation of the prior ordinate

Since we assumed

$$\pi(\boldsymbol{c},\boldsymbol{\theta}) \propto \exp\left(-\frac{1}{2}\left[\tau^{-2}\boldsymbol{\beta}'\boldsymbol{\beta}+\rho^{-2}\phi^2\right]\right)\cdot \mathbb{1}_{\{0<c_2<\ldots<c_{K-1}<C\}}$$

with known hyperparameters τ, ρ, C, we can evaluate the factor $\pi(\boldsymbol{c}^\diamond,\boldsymbol{\theta}^\diamond)$ by multiplying the three factors

$$\begin{aligned}\pi(\boldsymbol{\beta}^\diamond) &= \textstyle\prod_{j=0}^{p} N(\beta_j^\diamond|0,\tau^2),\\ \pi(\phi^\diamond) &= N(\phi^\diamond|0,\rho^2),\\ \pi(\boldsymbol{c}^\diamond) &\propto \mathbb{1}_{\{0<c_2^\diamond<\ldots<c_{K-1}^\diamond<C\}}.\end{aligned}$$

Whereas the first and second factor can be evaluated directly with the density function of the normal distribution, the third factor demands the computation of the normalizing constant, which is the inverse of the volume of the $(K-2)$-dimensional subset $\{(c_2,\ldots,c_{K-1}) \mid 0<c_2<\ldots<c_{K-1}<C\}$ of $\mathbb{R}^{K-2}$. The volume of this subset can easily be derived to equal

$$\int_0^C\int_{c_2}^C\ldots\int_{c_{K-2}}^C dc_{K-1}\ldots dc_2 = \frac{1}{(K-2)!}C^{K-2}.$$

Therefore $\pi(\boldsymbol{c}^\diamond) = (K-2)!/C^{K-2}$.

4.4 Estimation of the posterior ordinate

The aim of this section is to find an estimate for $\pi(\boldsymbol{c}^\diamond,\boldsymbol{\theta}^\diamond|\boldsymbol{y})$. Since this expression can be decomposed into

$$\pi(\boldsymbol{c},\boldsymbol{\theta}|\boldsymbol{y}) = \pi(\boldsymbol{\theta}|\boldsymbol{c},\boldsymbol{y})\pi(c_{K-1}|c_2,\ldots,c_{K-2},\boldsymbol{y})\pi(c_{K-2}|c_2,\ldots,c_{K-3},\boldsymbol{y})\cdots\pi(c_2|\boldsymbol{y})$$

it is sufficient to estimate the $K-1$ factors

$$\pi(c_2^\diamond|\boldsymbol{y}),\ \ldots,\ \pi(c_{K-2}^\diamond|c_2^\diamond,\ldots,c_{K-3}^\diamond,\boldsymbol{y}),\ \pi(c_{K-1}^\diamond|c_2^\diamond,\ldots,c_{K-2}^\diamond,\boldsymbol{y}),\ \pi(\boldsymbol{\theta}^\diamond|\boldsymbol{c}^\diamond,\boldsymbol{y}). \tag{4.5}$$

For this purpose we use in the following the techniques described in Section 2.3.5. We point out here that all full conditional distributions appearing in this section are known from Section 3.2, including the normalizing constants, so that the evaluation is straightforward.

First we consider the estimation of $\pi(c_2^\diamond|\boldsymbol{y})$. Running the full Gibbs sampler, we get M samples

$$(c_2^{[m]}, \ldots, c_{K-1}^{[m]}, \boldsymbol{\theta}^{[m]}, \boldsymbol{y}^{*[m]}) \sim \pi(\boldsymbol{c}, \boldsymbol{\theta}, \boldsymbol{y}^*|\boldsymbol{y}), \qquad m = 1, \ldots, M.$$

Therefore we can estimate $\pi(c_2^\diamond|\boldsymbol{y})$ using the Monte Carlo integration principle by evaluating and averaging M full conditionals for c_2:

$$\hat{\pi}(c_2^\diamond|\boldsymbol{y}) := \frac{1}{M}\sum_{m=1}^{M} f(c_2^\diamond|c_3^{[m]}, \ldots, c_{K-1}^{[m]}, \boldsymbol{\theta}^{[m]}, \boldsymbol{y}^{*[m]}, \boldsymbol{y})$$

where

$$\begin{aligned} f(c_2^\diamond|c_3^{[m]}, \ldots, c_{K-1}^{[m]}, \boldsymbol{\theta}^{[m]}, \boldsymbol{y}^{*[m]}, \boldsymbol{y}) &= \left(r_2^{[m]} - l_2^{[m]}\right)^{-1} \mathbb{1}_{\left\{c_2^\diamond \in \left(l_2^{[m]}, r_2^{[m]}\right)\right\}} \\ \text{with} \qquad l_2^{[m]} &:= \max\left\{0, \max_{t=1,\ldots,T}\left\{y_t^{*[m]} \,\middle|\, y_t = 2\right\}\right\} \\ \text{and} \qquad r_2^{[m]} &:= \min\left\{c_3^{[m]}, \min_{t=1,\ldots,T}\left\{y_t^{*[m]} \,\middle|\, y_t = 3\right\}\right\}. \end{aligned}$$

The other $K-2$ factors appearing in (4.5) are all estimated by reduced runs of the Gibbs sampler, which we discuss now.

To estimate $\pi(c_3^\diamond|c_2^\diamond, \boldsymbol{y})$ we use a reduced run where c_2 is fixed at $c_2^\diamond$ and all components but c_2 are updated in each iteration. So we get J samples $(c_3^{[j]}, \ldots, c_{K-1}^{[j]}, \boldsymbol{\theta}^{[j]}, \boldsymbol{y}^{*[j]})$, $j = 1, \ldots, J$, from $\pi(c_3, \ldots, c_{K-1}, \boldsymbol{\theta}, \boldsymbol{y}^*|c_2^\diamond, \boldsymbol{y})$. Again we can now estimate $\pi(c_3^\diamond|c_2^\diamond, \boldsymbol{y})$ using the Monte Carlo integration principle by evaluating and averaging J full conditionals for c_3:

$$\hat{\pi}(c_3^\diamond|c_2^\diamond, \boldsymbol{y}) := \frac{1}{J}\sum_{j=1}^{J} f(c_3^\diamond|c_2^\diamond, c_4^{[j]}, \ldots, c_{K-1}^{[j]}, \boldsymbol{\theta}^{[j]}, \boldsymbol{y}^{*[j]}, \boldsymbol{y})$$

where

$$\begin{aligned} f(c_3^\diamond|c_2^\diamond, c_4^{[j]}, \ldots, c_{K-1}^{[j]}, \boldsymbol{\theta}^{[j]}, \boldsymbol{y}^{*[j]}, \boldsymbol{y}) &= \left(r_3^{[j]} - l_3^{[j]}\right)^{-1} \mathbb{1}_{\left\{c_3^\diamond \in \left(l_3^{[j]}, r_3^{[j]}\right)\right\}} \\ \text{with} \qquad l_3^{[j]} &:= \max\left\{c_2^\diamond, \max_{t=1,\ldots,T}\left\{y_t^{*[j]} \,\middle|\, y_t = 3\right\}\right\} \\ \text{and} \qquad r_3^{[j]} &:= \min\left\{c_4^{[j]}, \min_{t=1,\ldots,T}\left\{y_t^{*[j]} \,\middle|\, y_t = 4\right\}\right\}. \end{aligned}$$

In the next reduced run c_2 and c_3 are fixed at $c_2^\diamond$ and $c_3^\diamond$, respectively. With the same arguments as before one gets

$$\hat{\pi}(c_4^\diamond|c_2^\diamond, c_3^\diamond, \boldsymbol{y}) := \frac{1}{L}\sum_{l=1}^{L} f(c_4^\diamond|c_2^\diamond, c_3^\diamond, c_5^{[l]}, \ldots, c_{K-1}^{[l]}, \boldsymbol{\theta}^{[l]}, \boldsymbol{y}^{*[l]}, \boldsymbol{y})$$

where $f(c_4^\diamond|c_2^\diamond, c_3^\diamond, c_5^{[l]}, \ldots, c_{K-1}^{[l]}, \boldsymbol{\theta}^{[l]}, \boldsymbol{y}^{*[l]}, \boldsymbol{y})$ is again density of an uniform distribution. This principle carries over to the remaining factors in (4.5). Finally, the last factor, $\pi(\boldsymbol{\theta}^\diamond|\boldsymbol{c}^\diamond, \boldsymbol{y})$, is estimated using a reduced run where the cutpoints c_k are fixed to $c_k^\diamond$, $k = 2, \ldots, K-1$, respectively, and only $\boldsymbol{\theta}$ and $\boldsymbol{y}^*$ are updated in each iteration. Here we get

$$\hat{\pi}(\boldsymbol{\theta}^\diamond|\boldsymbol{c}^\diamond, \boldsymbol{y}) := \frac{1}{N}\sum_{n=1}^{N} f(\boldsymbol{\theta}^\diamond|\boldsymbol{c}^\diamond, \boldsymbol{y}^{*[n]}, \boldsymbol{y})$$

where $f(\boldsymbol{\theta}^\diamond|\boldsymbol{c}^\diamond, \boldsymbol{y}^{*[n]}, \boldsymbol{y})$ is the $N_{p+2}(\boldsymbol{\theta}^\diamond|\Sigma^{[n]}Z^{[n]\prime}\boldsymbol{y}_{-0}^{*[n]}, \Sigma^{[n]})$-density

$$\begin{aligned} \text{with} \qquad \Sigma^{[n]} &:= \left(Z^{[n]\prime}Z^{[n]} + \operatorname{diag}(\tau^{-2}, \ldots, \tau^{-2}, \rho^{-2})\right)^{-1} \\ \text{and} \qquad Z^{[n]} &:= \begin{bmatrix} 1 & x_{11} & \cdots & x_{1p} & y_0^{*[n]} \\ \vdots & \vdots & & \vdots & \vdots \\ 1 & x_{T1} & \cdots & x_{Tp} & y_{T-1}^{*[n]} \end{bmatrix}. \end{aligned}$$

We point out that it is very important to use only reduced runs from the standard Gibbs sampler without grouped moved steps. This is because all reduced runs require at least the fixing of the cutpoint $c_2 = c_2^\diamond$. Therefore it is not possible to multiply all cutpoints $c_2, \ldots, c_{K-1}$ by an element (unequal 1) from the partial scale group. This, however, would be done by a grouped move step. The requirement to use the standard Gibbs sampler without grouped move steps seems to be a major disadvantage for our estimating procedure, but this is not really the case. The grouped move steps were used mainly to speed up the convergence for the cutpoints. However, from the full run of the GM-MGMC sampler we have already good estimates for all parameters including the cutpoints, which can be used as starting values in the reduced runs, so that the convergence problem does not play a role any more.

Finally we note that an alternative approach to estimate the posterior ordinate was developed by Ritter and Tanner (1992). This method is based on the invariance condition of the Gibbs chain and only requires draws from the full Gibbs run, where all full conditionals must be known including the normalizing constants. However, the method by Ritter and Tanner (1992) is only precise when the posterior is low-dimensional and the model does not contain latent variables (cf. Chib and Jeliazkov (2001)). Therefore, for the autoregressive ordered probit model we will get more accurate results with the decomposition used here.

Chapter 5

Application of the Autoregressive Ordered Probit Model to High-Frequency Finance

We now want to detect and to quantify the influence of covariates information on price changes of stocks. For this we choose data of the IBM stock traded on Dec 4, 2000 at the NYSE. We first consider the data set and conduct a short exploratory analysis for the covariates. To detect whether the data contains an autoregressive structure we then fit the autoregressive ordered probit model to the observations. Furthermore we also fit the common ordered probit model (cf. Section 2.7.1) to the data and finally investigate using Bayes factors how big the benefit is of using the autoregressive ordered probit model instead of the ordered probit model.

5.1 Data description

The IBM stock is a very frequently traded stock (about 400 transactions per hour), so that we have enough data even if we do not use data from the first minutes after opening and the last minutes before closing which might exhibit a different behavior. On the Dec 4, 2000, between 9:35:59 am and 2:42:40 pm there was a total of 2001 transactions of the IBM stock at the NYSE. The data is taken from the TAQ2 database of the NYSE, which contains the following covariates:

- TIMEDIFF, the time elapsed between two following transactions in seconds,

price change	category	frequency
0\$	1	859
$\pm 1/16$\$	2	782
$\pm 2/16$\$	3	275
$\leq -3/16$\$ and $\geq 3/16$\$	4	84

Table 5.1: Absolute price changes: associated categories and observed frequencies.

- SIZE, the volume of the transaction.

As response y_t we take the absolute values of the price differences. The price differences take on only values which are integer multiples of 1/16 US\$, and 99.5% of them lie between $-3/16$ US\$ and $+3/16$ US\$. The absolute price differences are a reasonable quantity to consider, since a price difference can be decomposed into the product of the absolute price change and the direction of the price change. These two factors of the time series may not only depend on different covariates but may also demand for a different modeling. We note, that a decomposition of the price change into three factors (activity, direction of price change, and absolute price change) was considered in Rydberg and Shephard (2003).

Because we consider only the absolute values of the price changes from one transaction to the next one, we associate the signed price changes to the response categories as shown in Table 5.1.

5.2 Exploratory analysis

First we want to conduct an exploratory analysis to choose appropriate transformations of the covariates. However, two problems arise here. The first problem arises since the response variable is discrete and takes on only few values. Therefore ordinary scatter plots are not informative, especially when the regressor is also discrete or categorical. Instead, we group the covariate data in intervals of the same length or use categories and then compute the average response per interval or per category. Now we can look for a linear (quadratic, logarithmic ...) relationship.

This relationship, however, is between y_t in Equation (3.1) and $\boldsymbol{x}_t$ in Equation (3.2). Therefore linearity can be destroyed by the (a priori unknown) cutpoints c_k, which is the second problem. Only when the cutpoints are estimated to be nearly equidistant, we

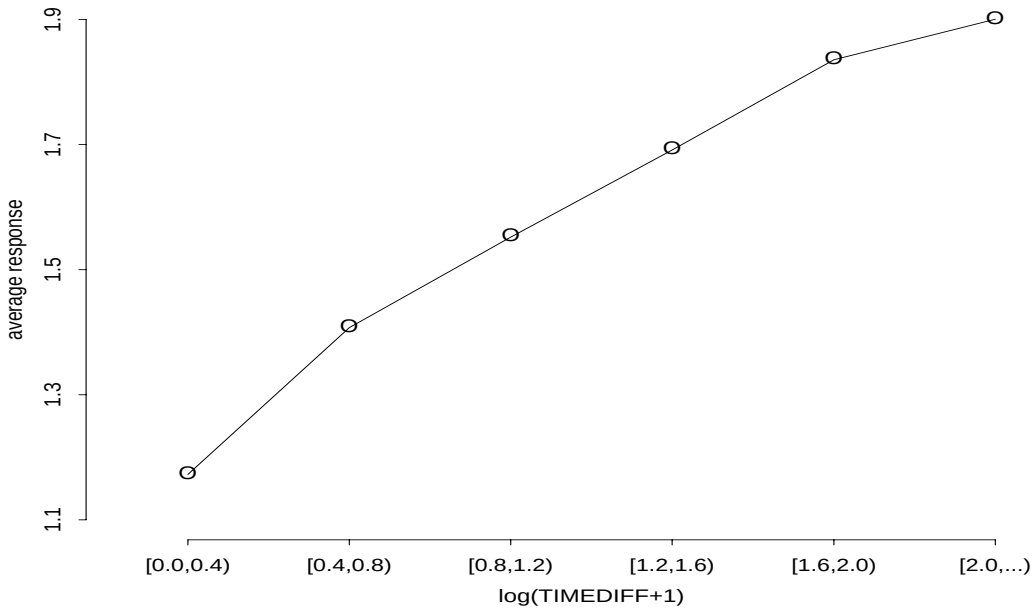

Figure 5.1: Relationship between response and the covariate log(TIMEDIFF+1).

have a validation for the chosen transformation of the covariate. Otherwise one should use other transformations that take into account the different distances between the cutpoints. At this point further research will be necessary to develop iterative methods for choosing appropriate transformations.

Considering the covariate TIMEDIFF it is useful to take the logarithm of TIMEDIFF plus 1 to get a nearly linear dependency of the response. Of course, by adding 1 we avoid the value log(0). As described above, we group the data in intervals of the same length and compute the average response for each interval. The result can be seen in Figure 5.1. The relationship is quite linear. For small (logarithmic) time differences we have an average response of 1.17, for big ones an average response of 1.90. The relatively high difference of $1.90 - 1.17 = 0.73$ is a first hint at the strong significance of this covariate.

Considering the covariate SIZE it turns out again to be useful to take the logarithm to get a nearly linear dependency of the response. Again we group the data in intervals and compute the average response for each interval. A plot of the average response per category is given in Figure 5.2. The relationship is quite linear. However, the difference between the maximal and minimal average response is only $1.87 - 1.71 = 0.16$. So we expect that log(SIZE) is not as significant as log(TIMEDIFF+1).

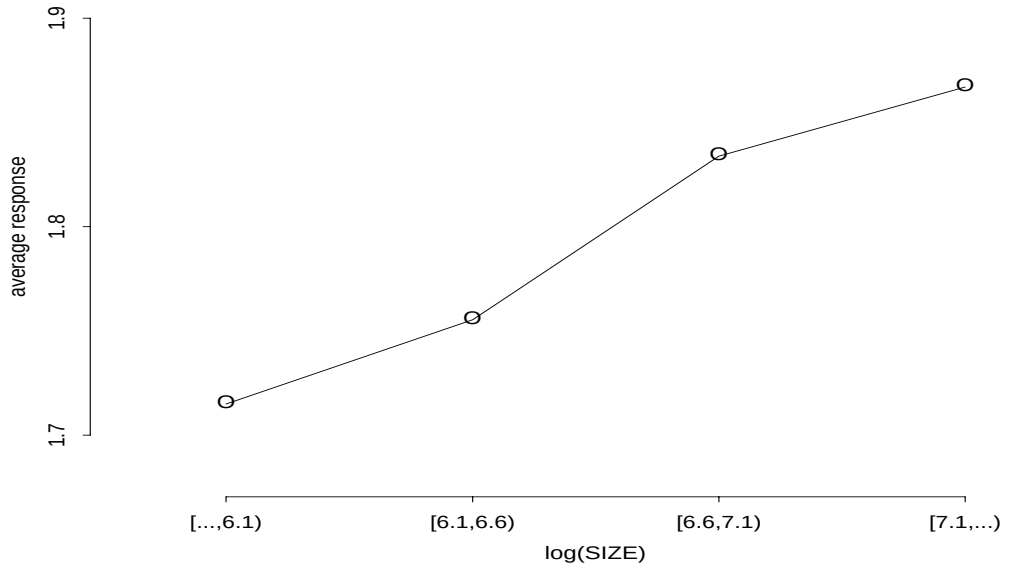

Figure 5.2: Relationship between response and the covariate log(SIZE).

5.3 Model estimation

We now fit the autoregressive ordered probit model defined in Equations (3.1) and (3.2) to the data. Since we are also interested in a comparison of this model to the ordered probit model without an autoregressive component, we made all computations for both models. We abbreviate these models by AOP (Autoregressive Ordered Probit) and OP (Ordered Probit), respectively. In particular, the latent structures of these two models are given by

$$\begin{aligned} \text{AOP:} \quad & y_t^* = \beta_0 + \beta_1 \cdot \log(\text{TIMEDIFF}+1)_t + \beta_2 \cdot \log(\text{SIZE})_t + \phi y_{t-1}^* + \varepsilon_t^*, \\ \text{OP:} \quad & y_t^* = \beta_0 + \beta_1 \cdot \log(\text{TIMEDIFF}+1)_t + \beta_2 \cdot \log(\text{SIZE})_t + \varepsilon_t^*. \end{aligned}$$

We first consider the specification of the hyperparameters τ (standard deviation of prior for the regression coefficients), σ (standard deviation of prior for y_0^*), ρ (standard deviation of prior for ϕ), and C (maximum for cutpoint c_3). Since the variance of the error term ε_t^* is fixed to 1 and the cutpoint c_1 is fixed to 0, we do not expect a very high value for the cutpoint c_3 and for $|y_0^*|$. Otherwise the error term ε_t^* would have hardly any impact on the time series, and therefore the data could be nearly deterministically explained by the covariates and the autoregression. Therefore we choose $C = 10$ and $\sigma^2 = 1.0$. For the same reason we do not expect an extreme intercept β_0. Since the

Parameter	Prior distribution
$\boldsymbol{c}$	Uniform on $\{(c_2, c_3) \mid 0 < c_2 < c_3 < 10\}$
y_0^*	N(0,1)
β_0	N(0,10)
β_1	N(0,10)
β_2	N(0,10)
ϕ	N(0,0.1)

Table 5.2: Prior distributions for the cutpoints $\boldsymbol{c}$, for the latent variable y_0^*, for the regression coefficients β_j, and for the autoregressive parameter ϕ.

logarithms of (TIMEDIFF+1) and SIZE have values between 0 and 10.8, we also do not expect extreme values for $\beta_j, j = 1, 2$. Therefore we choose $\tau^2 = 10$ which leads here to a sufficiently uninformative prior for the regression coefficients. The autoregressive component in the AOP model is expected to be present, but not too large, so the choice of $\rho^2 = 0.1$ seems to be adequate. Table 5.2 summarizes the chosen prior distributions. By using other hyperparameter values we have seen that the posterior estimates are not very prior-sensitive. This also can be expected because of the large number of observations.

For the estimation of the OP model the corresponding GM-MGMC sampler of Liu and Sabatti (2000) is used. The GM-MGMC sampler for the OP model can be derived in a natural way from the GM-MGMC sampler for the AOP model since the OP model is a submodel of the AOP model. The partial scale group used for the sampler in the AOP case then simplifies to a (total) scale group since the parameter ϕ does not occur in the OP model.

We now run both GM-MGMC samplers for 15000 iterations and discard the first 5000 iterations for burn-in. From the simulation study in Section 3.4 we know that this leads to very accurate estimates. The results are summarized in Table 5.3. It shows the posterior mean estimates for the cutpoints, the regression coefficients, and the autoregressive parameter using the iterations 5001 to 15000 of the GM-MGMC sampler together with their corresponding estimated standard deviations and 90% credible intervals.

We conclude that the intercept, log(TIMEDIFF+1), and log(SIZE) are all significant in both models, as well as the autoregressive component in the AOP model with posterior mean estimate 0.1362. Because of the positive sign of the estimates for β_1 we conclude that the more time elapses since the last transaction, the higher the expected price change is. The same holds for the transaction volume: The more stocks are traded, the

	Autoregressive Ordered Probit			Ordered Probit		
	estimate	std.err.	90% cred.int.	estimate	std.err.	90% cred.int.
c_2	1.1231	0.0344	(1.0664,1.1798)	1.1172	0.0327	(1.0648,1.1710)
c_3	1.9593	0.0540	(1.8733,2.0496)	1.9381	0.0534	(1.8509,2.0278)
β_0	-0.5567	0.1437	(-0.7918,-0.3224)	-0.5179	0.1442	(-0.7527,-0.2813)
β_1	0.2350	0.0328	(0.1821,0.2888)	0.2278	0.0324	(0.1743,0.2805)
β_2	0.0368	0.0197	(0.0043,0.0688)	0.0368	0.0196	(0.0044,0.0695)
ϕ	0.1362	0.0277	(0.0913,0.1818)			

Table 5.3: Posterior mean estimates and corresponding estimated standard deviations and 90% posterior credible intervals for parameters in Models AOP and OP.

higher the expected price change is.

Figure 5.3 shows the estimated posterior marginal densities for the parameters in the AOP model. The densities are unimodal and quite symmetric, so that the posterior mean estimates represent a high density point. This property is used in the next section. The estimated marginal densities for the parameters in the OP model have the same shape and are therefore not shown. Figure 5.4 finally shows the estimated autocorrelations in the chains produced by the GM-MGMC samplers after the burn-in period of 5000 iterations for both the AOP model (solid curves) and the OP model (dotted curves). The autocorrelations decline quite fast for all parameters and differ hardly between the two models.

5.4 Bayes factor of AOP against OP model

Since the posterior mean estimate of ϕ in the AOP model is 0.1362 and the corresponding 90% credible interval is far away from 0, we can assume the presence of an autoregressive structure in the IBM data. From this point of view we prefer the AOP model to the OP model for this data set. However, so far we do not know, how big the benefit is of using the AOP model instead of the simpler OP model. Therefore we now estimate the marginal likelihoods for the data under the two models to determine an estimate for the Bayes factor of the AOP against the OP model.

We follow Chapter 4 and estimate the likelihood ordinate, the posterior ordinate, and the prior ordinate for both models. Since AOP and OP are nested models and, in

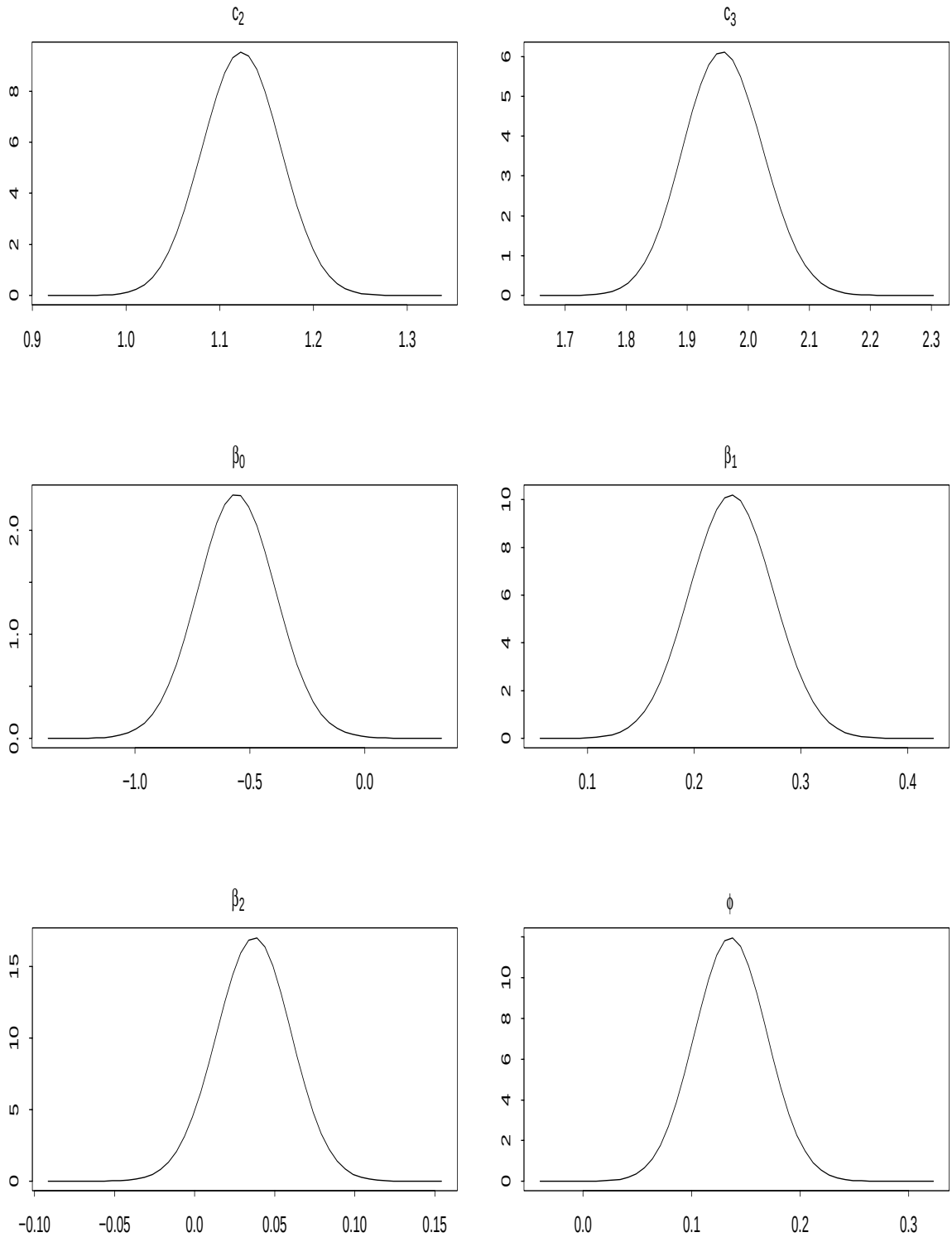

Figure 5.3: Estimated posterior marginal densities for parameters of autoregressive ordered probit model.

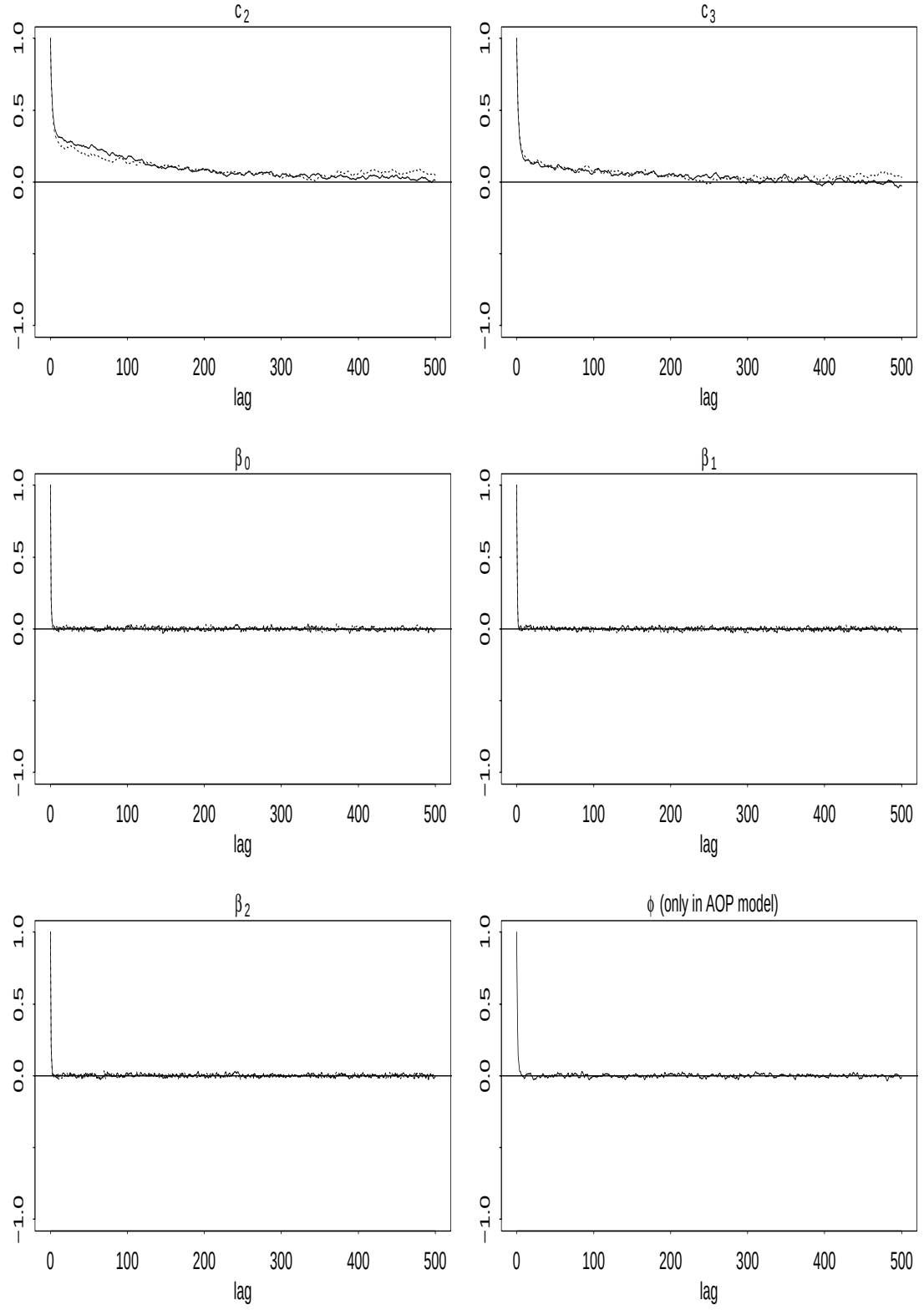

Figure 5.4: Autocorrelations of chains produced by the GM-MGMC samplers for the AOP model (solid) and the OP model (dotted). Visible difference only for cutpoints.

	Autoregressive Ordered Probit	Ordered Probit
$\log f(\boldsymbol{y}\|\boldsymbol{c}^\diamond, \boldsymbol{\theta}^\diamond)$	-2233.3235	-2244.9539
$\log \pi(\boldsymbol{c}^\diamond, \boldsymbol{\theta}^\diamond)$	-10.0014	-10.1388
$\log \pi(\boldsymbol{c}^\diamond, \boldsymbol{\theta}^\diamond\|\boldsymbol{y})$	15.4477	13.0259
$\log m(\boldsymbol{y})$	-2258.7726	-2268.1186
Bayes factor of AOP against OP	exp(-2258.7726+2268.1186) = **11452.92**	

Table 5.4: Estimated log likelihood ordinate, log prior ordinate, log posterior ordinate and marginal log likelihood for Models AOP and OP. Bayes Factor of AOP against OP.

particular, OP is a submodel of AOP, we can use the estimation methods for the AOP model also for the OP model. Of course, the algorithms presented in Chapter 4 simplify in this case. For example, since y_t^* is independent from y_{t-1}^* in the OP model, a filtering procedure as for the AOP model is not necessary here and Algorithm 4.1 reduces to sampling M times from $N_{[c_{y_t-1}, c_{y_t})}(\boldsymbol{x}_t'\boldsymbol{\beta}, 1)$. Furthermore, since the OP model does not contain the autoregressive component of the AOP model, $\boldsymbol{\theta} = \boldsymbol{\beta}$ and the prior ordinate $\pi(\boldsymbol{c}^\diamond, \boldsymbol{\theta}^\diamond)$ equals $\pi(\boldsymbol{c}^\diamond)\pi(\boldsymbol{\beta}^\diamond)$.

For the fixed parameters $\boldsymbol{c}^\diamond$ and $\boldsymbol{\theta}^\diamond$ appearing in Equation (4.1) one should use high density points (cf. Section 2.6.2). Hence we take the corresponding posterior mean estimates given in Table 5.3. We run the Algorithms 4.1 and 4.2 with $M = 30000$ particles. This takes about 130 minutes for the AOP model on an UltraSPARC III Cu 900 Mhz processor. The choice of M seems to be sufficiently high since we got nearly the same results for $M = 10000$ and $M = 20000$. The computation of the prior ordinate can be done exactly. Here we only mention that since the hyperparameter C was chosen to be 10 and we have 4 response categories, we get $\pi(\boldsymbol{c}^\diamond) = 2/C^2 = 0.02$. For the means which have to be computed for estimating the posterior ordinate, we use 10000 samples from the full and the reduced runs each. Table 5.4 summarizes the results.

We get a Bayes factor of 11452.92 for the AOP against the OP model. Following the Bayes factor scale in Table 2.1, we conclude that the autoregressive ordered probit model fits the IBM data decisively better than the ordered probit model.

Chapter 6

Stochastic Volatility Model for Ordinal Valued Time Series

From Section 2.8 we know that the volatility of the signed price change process varies over time. Moreover one can observe periods with important price fluctuations followed by relative quiet periods and vice versa (volatility clustering). These features cannot be covered by the autoregressive ordered probit (AOP) model (cf. Chapter 3) since we assume a constant variance of the noise term ε_t^* in Equation (3.2). Therefore we applied in Chapter 5 the AOP model only to the absolute price changes and not to the signed ones which also take the directions of the price changes into account.

In the following chapters we now want to model the signed price change process. Therefore we must allow for a non-constant volatility term. One large class of models which contains a non-constant volatility term is the class of stochastic volatility (SV) models (cf. Section 2.7.2). However, the common SV models are used for continuous data, so that we cannot apply them directly to the signed price change process.

Therefore in this chapter we first introduce a new model which can be applied to ordinal valued time series and which allows for a non-constant volatility like common SV models. Then we describe an Hybrid MCMC algorithm to estimate the parameters in the model. This algorithm exhibits convergence problems similar to the standard sampler of Section 3.2 for the AOP model, and therefore we develop appropriate grouped move steps to speed up the convergence. Simulations will show the fundamental improvement which is achieved by using these grouped move steps.

6.1 Model formulation

To cover the main features of the price change process as ordinal response, non-constant volatility, and dependence on covariates we introduce now the **Ordinal-response Stochastic Volatility (OSV) Model** defined by the following three equations:

$$\begin{aligned} y_t = k \quad \Leftrightarrow \quad y_t^* &\in [c_{k-1}, c_k)\,, && (6.1)\\ y_t^* &= \boldsymbol{x}_t'\boldsymbol{\beta} + \exp(h_t^*/2)\varepsilon_t^*, && (6.2)\\ h_t^* &= \mu + \boldsymbol{z}_t'\boldsymbol{\alpha} + \phi(h_{t-1}^* - \mu - \boldsymbol{z}_{t-1}'\boldsymbol{\alpha}) + \sigma\eta_t^*, && (6.3) \end{aligned}$$

where $\varepsilon_t^* \overset{\text{i.i.d.}}{\sim} N(0,1)$ independent of $\eta_t^* \overset{\text{i.i.d.}}{\sim} N(0,1)$, $k \in \{1,\ldots,K\}$, $t \in \{1,\ldots,T\}$, and $h_0^* = \mu$. $\boldsymbol{x}_t$ and $\boldsymbol{z}_t$ are vectors of covariates, for $t = 0$ we set

$$\boldsymbol{z}_0 := (0,\ldots,0)'. \tag{6.4}$$

The parameters $c_1,\ldots,c_{K-1}$ are cutpoints. For notational convenience we set $c_0 := -\infty$ and $c_K := +\infty$. Additionally we define $\boldsymbol{y} := (y_1,\ldots,y_T)$.

We call the latent variables y_t^* the continuous versions of the observations y_t. Given the covariate vector $\boldsymbol{x}_t$ and the latent variable h_t^*, y_t^* is normally distributed with conditional mean $\boldsymbol{x}_t'\boldsymbol{\beta}$ and conditional variance $\exp(h_t^*)$. The log-volatilities h_t^* form an autoregressive process of order one with impact of another covariate vector $\boldsymbol{z}_t$. Since the expression $\boldsymbol{z}_{t-1}'\boldsymbol{\alpha}$ is subtracted in the brace in Equation (6.3), the vector $\boldsymbol{z}_t$ has an impact only on h_t^* but not on future log-volatilities h_s^*, $s > t$.

We now consider the problem of identifiability of the unknown parameters in this model. If we want to allow that $\boldsymbol{x}_t'\boldsymbol{\beta}$ contains an intercept β_0, we have to fix one of the cutpoints, for example the cutpoint c_1. If we would not fix it, we could get an equivalent model by taking some real-valued constant and adding this constant to all y_t^*'s, to all c_k's, and to β_0. Therefore we set $c_1 = 0$. Furthermore, we have to fix another cutpoint or, alternatively, the intercept μ in the log-volatility evolution equation (6.3). If we would not fix any of these, we would get an equivalent model by first taking some constant $C > 0$ and then using the parameter transformations $\tilde{c}_k := Cc_k$, $\tilde{y}_t^* := Cy_t^*$, $\tilde{\boldsymbol{\beta}} := C\boldsymbol{\beta}$, $\tilde{h}_t^* := h_t^* + 2\log C$, and $\tilde{\mu} := \mu + 2\log C$. Since some of the equations in Section 6.3, where we develop the GM-MGMC sampler, would not longer be true if we would fix another cutpoint, we fix μ. Obviously a large value for μ would heavily increase the volatility $\exp(h_t^*)$ and therefore the cutpoints would also become very large. For this we set $\mu = -0.6$ which leads to non-extreme parameter estimates in the data sets considered later.

After fixing c_1 and μ the parameters and latent variables to estimate are the

- cutpoints $\boldsymbol{c} := (c_2, \ldots, c_{K-1})$,
- latent continuous versions of y_t, $\boldsymbol{y}^* := (y_1^*, \ldots, y_T^*)$,
- log-volatilities $\boldsymbol{h}^* := (h_1^*, \ldots, h_T^*)$,
- regression parameters $\boldsymbol{\beta} = (\beta_0, \ldots, \beta_p)'$ for the means of the latent variables,
- regression parameters $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_q)'$ for the log-volatilities,
- autoregressive parameter ϕ in the log-volatility Equation (6.3),
- standard deviation σ in the log-volatility Equation (6.3).

We emphasize that $\boldsymbol{x}_t = (1, x_{t1}, \ldots, x_{tp})'$ contains an intercept, whereas $\boldsymbol{z}_t = (z_{t1}, \ldots, z_{tq})'$ does not, since Equation (6.3) already contains the (fixed) intercept μ.

6.2 A hybrid MCMC algorithm

For estimating the parameters in the OSV model we describe in this section a Hybrid MCMC algorithm. The algorithm consists of two parts, where the second part uses ideas of the MCMC sampler developed in Chib, Nardari, and Shephard (2002). However, there are several differences to the sampler of Chib, Nardari, and Shephard (2002). First, the SV model considered there uses the equation $h_t^* = \mu + \boldsymbol{z}_t'\boldsymbol{\alpha} + \phi(h_{t-1}^* - \mu) + \sigma\eta_t^*$ instead of Equation (6.3), so that the covariate vector $\boldsymbol{z}_t$ also has an impact on future log-volatilities h_s^* for $s > t$. Secondly, in the sampler of Chib et al. (2002) the prior distributions for $\boldsymbol{\alpha}$, ϕ, and σ do not appear while updating these parameters, in spite of the fact that on page 284 of Chib et al. (2002) informative normal, beta, and log-normal priors are assumed, respectively. This seems to be incorrect, as well as some of the recursions for the block update of $\boldsymbol{h}^*$, where often a factor ϕ is missing. Last, we use a multivariate normal proposal distribution instead of the multivariate t-distribution in Chib et al. (2002) for the update of $\boldsymbol{\alpha}$, ϕ, and σ. Here, of course, both possibilities are admissible.

Now we turn back to the parts of the Hybrid MCMC algorithm described here. In the first part the regression parameter vector $\boldsymbol{\beta}$, the latent variables y_t^*, $t = 1, \ldots, T$, and the cutpoints $c_2, \ldots, c_{K-1}$ are updated. For the update of the remaining parameters in the second part we switch to a state space approximation of the latent process (6.2) and

i	q_i	m_i	v_i^2
1	0.00730	-11.40039	5.79596
2	0.10556	-5.24321	2.61369
3	0.00002	-9.83726	5.17950
4	0.04395	1.50746	0.16735
5	0.34001	-0.65098	0.64009
6	0.24566	0.52478	0.34023
7	0.25750	-2.35859	1.26261

Table 6.1: Parameters of seven-component Gaussian mixture to approximate the distribution of $\log \varepsilon_t^{*2}$.

(6.3). We now first describe this state space approximation, then we specify the prior distributions and the updates in the Hybrid MCMC sampler.

6.2.1 State space approximation of the latent process

Obviously, Equation (6.2) is equivalent to

$$\log\left(y_t^* - \boldsymbol{x}_t'\boldsymbol{\beta}\right)^2 = h_t^* + \log \varepsilon_t^{*2}. \tag{6.5}$$

The distribution of $\log \varepsilon_t^{*2}$ can be approximated by a seven-component mixture of normals, as in Kim, Shephard, and Chib (1998). In particular,

$$\log \varepsilon_t^{*2} \approx \sum_{i=1}^{7} q_i u_t^{*(i)}$$

where $u_t^{*(i)}$ is normally distributed with mean m_i and variance v_i^2 independent of t. Moreover, the random variables $\{u_t^{*(i)} \,|\, t = 1, \ldots, T, i = 1, \ldots, 7\}$ are independent. The quantity q_i denotes the probability that the mixture component i occurs. These probabilities are also independent of t and are given in Table 6.1 together with the corresponding means and variances.

Now let $s_t \in \{1, \ldots, 7\}$ denote the component of the mixture that occurs at time t and let $\pi(s_t)$ denote the prior for s_t, where $\pi(s_t = i) = q_i$. Defining $\tilde{y}_t^* := \log\left(y_t^* - \boldsymbol{x}_t'\boldsymbol{\beta}\right)^2$, Equation (6.5) leads to the following state space approximation of the latent process (6.2) and (6.3):

$$\tilde{y}_t^* = h_t^* + u_t^{*(s_t)}, \tag{6.6}$$

$$h_t^* = \mu + \boldsymbol{z}_t'\boldsymbol{\alpha} + \phi(h_{t-1}^* - \mu - \boldsymbol{z}_{t-1}'\boldsymbol{\alpha}) + \sigma\eta_t^*. \tag{6.7}$$

We will use this approximation for sampling ϕ, σ, $\boldsymbol{\alpha}$, and h_t^*, $t = 1, \ldots, T$. Of course, we now have to sample in addition the unknown mixture indices s_t for $t = 1, \ldots, T$. For notational convenience we define $\tilde{\boldsymbol{y}}^* := (\tilde{y}_1^*, \ldots, \tilde{y}_T^*)$, $\boldsymbol{s} := (s_1, \ldots, s_T)$, and $\boldsymbol{s}_{-t} := (s_1, \ldots, s_{t-1}, s_{t+1}, \ldots, s_T)$.

6.2.2 Prior distributions

For the Bayesian approach we now specify the prior distributions for $\boldsymbol{c}$, $\boldsymbol{\beta}$, h_0^*, $\boldsymbol{\alpha}$, ϕ, and σ. We assume prior independence so that the joint prior density can be written as

$$\pi(\boldsymbol{c}, \boldsymbol{\beta}, h_0^*, \boldsymbol{\alpha}, \phi, \sigma) = \pi(\boldsymbol{c})\pi(\boldsymbol{\beta})\pi(h_0^*)\pi(\alpha_1) \cdots \pi(\alpha_q)\pi(\phi)\pi(\sigma).$$

For $\boldsymbol{\beta}$ we choose a multivariate normal prior distribution, for h_0^* the Dirac measure at μ, and for the remaining parameters noninformative priors. In particular,

$$\begin{aligned}
\pi(\boldsymbol{c}) &= \mathbb{1}_{\{0<c_2<\ldots<c_{K-1}<C\}}, \\
\pi(\boldsymbol{\beta}) &= N_{p+1}(\boldsymbol{\beta} \mid \boldsymbol{b}_0, B_0), \\
\pi(h_0^*) &= \mathbb{1}_{\{h_0^*=\mu\}}, \\
\pi(\alpha_j) &= \mathbb{1}_{(-C_\alpha, C_\alpha)}(\alpha_j), \qquad j = 1, \ldots, q, \\
\pi(\phi) &= \mathbb{1}_{(-1,1)}(\phi), \\
\pi(\sigma) &= \mathbb{1}_{(0, C_\sigma)}(\sigma),
\end{aligned}$$

where $C > 0$, $C_\alpha > 0$, and $C_\sigma > 0$ are (known) hyperparameters, as well as the mean vector $\boldsymbol{b}_0$ and the covariance matrix B_0.

In the following sections we specify the MCMC updates.

6.2.3 Regression parameter update ($\boldsymbol{\beta}$-update)

Given $\boldsymbol{\beta}$ and h_t^*, the latent variable y_t^* is normally distributed with mean $\boldsymbol{x}_t'\boldsymbol{\beta}$ and vari-

ance $\exp(h_t^*)$. Together with the $N_{p+1}(\boldsymbol{b}_0, B_0)$ prior for $\boldsymbol{\beta}$ we get

$$\begin{aligned}
f(\boldsymbol{\beta}|\boldsymbol{y}, \boldsymbol{c}, \boldsymbol{y}^*, \boldsymbol{h}^*, \boldsymbol{\alpha}, \phi, \sigma) &= f(\boldsymbol{\beta}|\boldsymbol{y}^*, \boldsymbol{h}^*) \\
&\propto f(\boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*) \\
&= \prod_{t=1}^{T} f(y_t^*|\boldsymbol{\beta}, h_t^*) \cdot \pi(\boldsymbol{\beta}) \\
&\propto \exp\left\{-\frac{1}{2}\left[\sum_{t=1}^{T} \frac{(y_t^* - \boldsymbol{x}_t'\boldsymbol{\beta})^2}{\exp(h_t^*)} + (\boldsymbol{\beta} - \boldsymbol{b}_0)' B_0^{-1} (\boldsymbol{\beta} - \boldsymbol{b}_0)\right]\right\} \\
&\propto \exp\left\{-\frac{1}{2}\left[\sum_{t=1}^{T} \frac{-2y_t^* \boldsymbol{x}_t'\boldsymbol{\beta}}{\exp(h_t^*)} - 2\boldsymbol{b}_0' B_0^{-1}\boldsymbol{\beta} + \sum_{t=1}^{T} \frac{\boldsymbol{\beta}'\boldsymbol{x}_t\boldsymbol{x}_t'\boldsymbol{\beta}}{\exp(h_t^*)} + \boldsymbol{\beta}' B_0^{-1}\boldsymbol{\beta}\right]\right\} \\
&= \exp\left\{-\frac{1}{2}\left[-2\left(\sum_{t=1}^{T} \frac{y_t^* \boldsymbol{x}_t'}{\exp(h_t^*)} + \boldsymbol{b}_0' B_0^{-1}\right)\boldsymbol{\beta} + \boldsymbol{\beta}'\left(\sum_{t=1}^{T} \frac{\boldsymbol{x}_t\boldsymbol{x}_t'}{\exp(h_t^*)} + B_0^{-1}\right)\boldsymbol{\beta}\right]\right\}.
\end{aligned}$$

From this we conclude that $\boldsymbol{\beta}|\boldsymbol{y}, \boldsymbol{c}, \boldsymbol{y}^*, \boldsymbol{h}^*, \boldsymbol{\alpha}, \phi, \sigma \sim N_{p+1}(\boldsymbol{b}, B)$ where

$$\begin{aligned}
B &:= \left(\sum_{t=1}^{T} \frac{\boldsymbol{x}_t\boldsymbol{x}_t'}{\exp(h_t^*)} + B_0^{-1}\right)^{-1} \\
\text{and} \qquad \boldsymbol{b} &:= B\left(\sum_{t=1}^{T} \frac{\boldsymbol{x}_t y_t^*}{\exp(h_t^*)} + B_0^{-1}\boldsymbol{b}_0\right).
\end{aligned}$$

6.2.4 Latent variable update (y_t^*-update)

Since

$$\begin{aligned}
f(y_t^*|\boldsymbol{y}, \boldsymbol{c}, \boldsymbol{y}_{-t}^*, \boldsymbol{\beta}, \boldsymbol{h}^*, \boldsymbol{\alpha}, \phi, \sigma) &= f(y_t^*|y_t, \boldsymbol{c}, \boldsymbol{\beta}, h_t^*) \propto f(y_t^*, y_t, \boldsymbol{c}, \boldsymbol{\beta}, h_t^*) \\
&\propto f(y_t|y_t^*, \boldsymbol{c}) \cdot f(y_t^*|\boldsymbol{\beta}, h_t^*) \\
&\propto \mathbb{1}_{[c_{y_t-1}, c_{y_t})}(y_t^*) \exp\left\{-\frac{1}{2}\frac{(y_t^* - \boldsymbol{x}_t'\boldsymbol{\beta})^2}{\exp(h_t^*)}\right\}
\end{aligned}$$

draw $y_t^*,\ t = 1, \ldots, T$, from the univariate truncated normal distribution

$$N_{[c_{y_t-1}, c_{y_t})}\left(y_t^* \mid \boldsymbol{x}_t'\boldsymbol{\beta}, \exp(h_t^*)\right).$$

6.2.5 Cutpoint parameter update (c_k-update)

Investigating the full conditionals $f(c_k|\boldsymbol{y}, \boldsymbol{c}_{-k}, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*, \boldsymbol{\alpha}, \phi, \sigma)$ for $k \in \{2, \ldots, K-1\}$ we can see that

$$f(c_k|\boldsymbol{y}, \boldsymbol{c}_{-k}, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*, \boldsymbol{\alpha}, \phi, \sigma) \;=\; f(c_k|\boldsymbol{y}, \boldsymbol{c}_{-k}, \boldsymbol{y}^*) \;\propto\; f(\boldsymbol{y}, \boldsymbol{c}, \boldsymbol{y}^*) \;=\; \left[\prod_{t=1}^{T} f(y_t|y_t^*, \boldsymbol{c})\right] \pi(\boldsymbol{c}).$$

From this decomposition it follows that

$$\begin{aligned} f(c_k|\boldsymbol{y}, \boldsymbol{c}_{-k}, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*, \boldsymbol{\alpha}, \phi, \sigma) \;&\propto\; \left[\prod_{t=1}^{T} \mathbb{1}_{\left[c_{y_t-1}, c_{y_t}\right)}(y_t^*)\right] \mathbb{1}_{\{0<c_2<\ldots<c_{K-1}<C\}} \\ = \;\left[\prod_{\{t=1,\ldots,T|y_t=k\}} \mathbb{1}_{[c_{k-1}, c_k)}(y_t^*)\right] &\left[\prod_{\{t=1,\ldots,T|y_t=k+1\}} \mathbb{1}_{[c_k, c_{k+1})}(y_t^*)\right] \mathbb{1}_{\{0<c_2<\ldots<c_{K-1}<C\}}. \end{aligned}$$

Therefore we have the same expressions as in Subsection 3.2.3 and can conclude that $f(c_k|\boldsymbol{y}, \boldsymbol{y}^*, \boldsymbol{\beta}, \phi, \boldsymbol{c}_{-k})$ is an uniform distribution in the interval (l_k, r_k), where

$$l_k \;:=\; \max\left\{c_{k-1}, \max_{t=1,\ldots,T}\{y_t^*|y_t = k\}\right\}, \tag{6.8}$$

$$r_k \;:=\; \min\left\{c_{k+1}, \min_{t=1,\ldots,T}\{y_t^*|y_t = k+1\}\right\}. \tag{6.9}$$

6.2.6 Mixture index update (s_t-update)

For the updates of the remaining parameters we switch to the state space approximation (6.6) and (6.7) of the latent process. Therefore we now compute

$$\tilde{y}_t^* = \log(y_t^* - \boldsymbol{x}_t'\boldsymbol{\beta})^2, \qquad t = 1, \ldots, T,$$

and consider for the remaining updates only the system

$$\tilde{y}_t^* \;=\; h_t^* + u_t^{(s_t)}, \tag{6.10}$$

$$h_t^* \;=\; \mu + \boldsymbol{z}_t'\boldsymbol{\alpha} + \phi(h_{t-1}^* - \mu - \boldsymbol{z}_{t-1}'\boldsymbol{\alpha}) + \sigma\eta_t^*. \tag{6.11}$$

Since the mixture indices $\{s_t,\ t = 1, \ldots, T\}$ are conditionally independent, we have on the one hand

$$f(s_t|\tilde{\boldsymbol{y}}^*, \boldsymbol{h}^*, \boldsymbol{s}_{-t}, \boldsymbol{\alpha}, \phi, \sigma) \;=\; f(s_t|\tilde{y}_t^*, h_t^*) \;\propto\; f(s_t, \tilde{y}_t^*, h_t^*) \;=\; f(\tilde{y}_t^*|h_t^*, s_t)\pi(s_t).$$

On the other hand, given h_t^* and s_t, $\tilde{y}_t^*$ is normally distributed with mean $h_t^* + m_{s_t}$ and variance $v_{s_t}^2$. We conclude that

$$f(s_t|\tilde{\boldsymbol{y}}^*, \boldsymbol{h}^*, \boldsymbol{s}_{-t}, \boldsymbol{\alpha}, \phi, \sigma) \;\propto\; N(\tilde{y}_t^*|h_t^* + m_{s_t}, v_{s_t}^2)\,\pi(s_t).$$

This, of course, is no standard distribution. For the update of s_t we therefore first have to evaluate the seven densities $N(h_t^* + m_i, v_i^2)$, $i = 1, \ldots, 7$, each at the point $\tilde{y}_t^*$, resulting in the values

$$r_{t,i} \;:=\; \frac{1}{\sqrt{2\pi}v_i} \exp\left\{ -\frac{(\tilde{y}_t^* - h_t^* - m_i)^2}{2\,v_i^2} \right\}.$$

Then we must draw $s_t \in \{1, \ldots, 7\}$ according to the probabilities

$$f(s_t = i \,|\, \tilde{\boldsymbol{y}}^*, \boldsymbol{h}^*, \boldsymbol{s}_{-t}, \boldsymbol{\alpha}, \phi, \sigma) = \frac{r_{t,i}\, q_i}{\sum_{k=1}^{7} r_{t,k}\, q_k}, \qquad i = 1, \ldots, 7.$$

6.2.7 $(\boldsymbol{\alpha}, \phi, \sigma)$ joint update and log-volatility update ($\boldsymbol{h}$-update)

Here we take again advantage of the state space representation which allows for the application of the Kalman recursions, the prediction error decomposition and the simulation smoother described in Section 2.7. If we compare the model Equations (2.23), (2.24) and (6.10), (6.11) and take into account that we have univariate observations and states here, we see that the notations correspond to each other in the following way (the notations from Section 2.7 are given on the left-hand side of each assignment):

$$\begin{aligned}
\boldsymbol{y}_t &= \tilde{y}_t^*, & \boldsymbol{c}_t &= m_{s_t}, & Z_t &\equiv 1, & G_t &= (v_{s_t}, 0)', \\
\boldsymbol{x}_t &= h_t^*, & \boldsymbol{d}_t &= \mu + \boldsymbol{z}_{t+1}'\boldsymbol{\alpha} - \phi(\mu + \boldsymbol{z}_t'\boldsymbol{\alpha}), & S_t &\equiv \phi, & H_t &\equiv (0, \sigma)'.
\end{aligned}$$

Since the dimensions both of the observation and of the state equation equal one in this context, the matrices appearing in the model equations and in the Kalman recursions shrink to real valued numbers (except G_t and H_t). Therefore we will substitute the upper case letters for the matrices by the corresponding lower case letters in the following. Moreover, we will denote the innovations by e_t instead of v_t to avoid confusions with the mixture variances v_{s_t}.

Now define $\boldsymbol{\theta} := (\boldsymbol{\alpha}', \phi, \sigma)'$. Since $f(\boldsymbol{h}^*, \boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s}) = f(\boldsymbol{h}^*|\boldsymbol{\theta}, \tilde{\boldsymbol{y}}^*, \boldsymbol{s}) f(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s})$ we will draw a sample from $f(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s})$ and then use this sample for a block update of $\boldsymbol{h}^*$.

First we consider the sampling from $f(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s})$ which is done by a Metropolis-Hastings (MH) step. Let $\boldsymbol{\theta}^\bullet$ denote the current value of $\boldsymbol{\theta}$. The MH step requires

- the specification of an appropriate proposal density $q(\,\cdot\,)$ for $\boldsymbol{\theta}$,

- sampling a proposal $\boldsymbol{\theta}^\circ$ from this proposal density,
- the evaluation of the target density $f(\,\cdot\,|\tilde{\boldsymbol{y}}^*, \boldsymbol{s})$ at $\boldsymbol{\theta}^\circ$ and $\boldsymbol{\theta}^\bullet$ (at least up to a normalizing constant),
- the evaluation of the proposal density $q(\,\cdot\,)$ at $\boldsymbol{\theta}^\circ$ and $\boldsymbol{\theta}^\bullet$ (at least up to a normalizing constant), and
- finally accepting the proposal $\boldsymbol{\theta}^\circ$ with probability

$$\alpha(\boldsymbol{\theta}^\bullet, \boldsymbol{\theta}^\circ|\tilde{\boldsymbol{y}}^*, \boldsymbol{s}) = \min\left\{\frac{f(\boldsymbol{\theta}^\circ|\tilde{\boldsymbol{y}}^*, \boldsymbol{s})}{f(\boldsymbol{\theta}^\bullet|\tilde{\boldsymbol{y}}^*, \boldsymbol{s})}\frac{q(\boldsymbol{\theta}^\bullet)}{q(\boldsymbol{\theta}^\circ)}, 1\right\}. \tag{6.12}$$

  If the proposal $\boldsymbol{\theta}^\circ$ is rejected, the current value $\boldsymbol{\theta}^\bullet$ is retained as the next draw.

First we show how $f(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s})$ can be evaluated up to a normalizing constant. Since

$$\begin{aligned} f(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s}) &\propto f(\tilde{y}_1^*, \ldots, \tilde{y}_T^*, \boldsymbol{\theta}, \boldsymbol{s}) \\ &= f(\tilde{y}_1^*|\boldsymbol{\theta}, \boldsymbol{s})\left[\prod_{t=2}^{T} f(\tilde{y}_t^*|\tilde{y}_1^*, \ldots, \tilde{y}_{t-1}^*, \boldsymbol{\theta}, \boldsymbol{s})\right]\pi(\boldsymbol{\theta}, \boldsymbol{s}) \\ &\propto f(\tilde{y}_1^*|\boldsymbol{\theta}, \boldsymbol{s})\left[\prod_{t=2}^{T} f(\tilde{y}_t^*|\tilde{y}_1^*, \ldots, \tilde{y}_{t-1}^*, \boldsymbol{\theta}, \boldsymbol{s})\right]\pi(\boldsymbol{\theta}) \end{aligned}$$

we have, for some unknown normalizing constant $d \in \mathbb{R}$ independent of $\boldsymbol{\theta}$,

$$f(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s}) = d \cdot f(\tilde{y}_1^*|\boldsymbol{\theta}, \boldsymbol{s})\left[\prod_{t=2}^{T} f(\tilde{y}_t^*|\tilde{y}_1^*, \ldots, \tilde{y}_{t-1}^*, \boldsymbol{\theta}, \boldsymbol{s})\right]\pi(\boldsymbol{\theta}). \tag{6.13}$$

Defining

$$g(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s}) := f(\tilde{y}_1^*|\boldsymbol{\theta}, \boldsymbol{s})\left[\prod_{t=2}^{T} f(\tilde{y}_t^*|\tilde{y}_1^*, \ldots, \tilde{y}_{t-1}^*, \boldsymbol{\theta}, \boldsymbol{s})\right] \tag{6.14}$$

we can write Equation (6.13) in the form

$$f(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s}) = d \cdot g(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s}) \cdot \pi(\boldsymbol{\theta}),$$

or, equivalently,

$$\log f(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s}) - \log d = \log g(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s}) + \log \pi(\boldsymbol{\theta}).$$

The second term on the right-hand side of this equation can be evaluated directly using the known prior densities for $\boldsymbol{\alpha}$, ϕ, and σ. Comparing (2.43) and (6.14) we see that for

the evaluation of the first term we can apply the prediction error decomposition given by Equation (2.44) which has here the following form:

$$\log g(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s}) = -\frac{T}{2}\log(2\pi) - \frac{1}{2}\sum_{t=1}^{T}\log f_{t|t-1} - \frac{1}{2}\sum_{t=1}^{T}\frac{e_t^2}{f_{t|t-1}}. \tag{6.15}$$

e_t and $f_{t|t-1}$ are given by the Kalman recursions. Recall, that since $\pi(h_0^*) = \mathbb{1}_{\{h_0^*=\mu\}}$ and since $\boldsymbol{z}_0 = (0, \ldots, 0)'$ (cf. Equation (6.4)), h_1^* given h_0^* is normally distributed with mean $h_{1|0} := \mu + \boldsymbol{z}_1'\boldsymbol{\alpha}$ and variance $p_{1|0} := \sigma$. These quantities serve as initial values for the following recursions which correspond to Equations (2.35) to (2.42) applied to state space model (6.10) and (6.11):

$$\begin{aligned}
y_{t|t-1} &:= m_{s_t} + h_{t|t-1}, && (6.16)\\
e_t &:= \tilde{y}_t^* - y_{t|t-1}, && (6.17)\\
f_{t|t-1} &:= p_{t|t-1} + v_{s_t}^2, && (6.18)\\
n_t &:= p_{t|t-1} f_{t|t-1}^{-1}, && (6.19)\\
h_{t|t} &:= h_{t|t-1} + n_t e_t, && (6.20)\\
p_{t|t} &:= (1 - n_t) p_{t|t-1}, && (6.21)\\
h_{t+1|t} &:= \mu + \boldsymbol{z}_{t+1}'\boldsymbol{\alpha} + \phi(h_{t|t} - \mu - \boldsymbol{z}_t'\boldsymbol{\alpha}), && (6.22)\\
p_{t+1|t} &:= \phi^2 p_{t|t} + \sigma^2. && (6.23)
\end{aligned}$$

Therefore, the target density can be evaluated at $\boldsymbol{\theta}^\circ$ and $\boldsymbol{\theta}^\bullet$ up to the normalizing constant d, which, of course, cancels down in the first fraction of expression (6.12).

Next we specify the proposal density. We take a $(q+2)$-dimensional normal distribution for $\boldsymbol{\theta}$ where the mean is determined by the maximum likelihood estimator $\boldsymbol{m}$ of the target density $f(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s})$. As covariance matrix W we take the negative inverse of the Hessian matrix V of $[\log g(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s}) + \log \pi(\boldsymbol{\theta})]$ at $\boldsymbol{m}$. In particular,

$$\begin{aligned}
\boldsymbol{m} &:= \arg\max_{\boldsymbol{\theta}} \left[\log g(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s}) + \log \pi(\boldsymbol{\theta})\right],\\
W &:= -V^{-1} = -\left\{ \frac{\partial^2 \left[\log g(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s}) + \log \pi(\boldsymbol{\theta})\right]}{\partial\boldsymbol{\theta}\partial\boldsymbol{\theta}'} \bigg|_{\boldsymbol{\theta}=\boldsymbol{m}} \right\}^{-1}.
\end{aligned}$$

$\boldsymbol{m}$ is found by numerical minimization of $-[\log g(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s}) + \log \pi(\boldsymbol{\theta})]$. We do this using the Nelder-Mead minimization algorithm, which is described in detail in Appendix A. It requires neither analytically nor numerically determined derivatives of the function to be minimized. Instead, starting with an initial simplex, it generates in each iteration a new simplex until a stopping criterion is fulfilled. How the next simplex is constructed

depends only on the values of the function to minimize at the vertices of the current simplex. Therefore, for each vertex, $g(\cdot\,|\tilde{\boldsymbol{y}}^*, \boldsymbol{s})$ has to be evaluated running the Kalman recursions and applying the prediction error decomposition (6.15). We pass only the current value $\boldsymbol{\theta}^\bullet = (\theta_1^\bullet, \ldots, \theta_{q+2}^\bullet)$ to our implementation of the Nelder-Mead algorithm, which itself generates an initial simplex determined by the $q+3$ vertices

$$\begin{bmatrix} \theta_1^\bullet \\ \theta_2^\bullet \\ \vdots \\ \theta_{q+2}^\bullet \end{bmatrix} \quad \begin{bmatrix} \theta_1^\bullet + a \\ \theta_2^\bullet \\ \vdots \\ \theta_{q+2}^\bullet \end{bmatrix} \quad \begin{bmatrix} \theta_1^\bullet \\ \theta_2^\bullet + a \\ \vdots \\ \theta_{q+2}^\bullet \end{bmatrix} \quad \cdots \quad \begin{bmatrix} \theta_1^\bullet \\ \theta_2^\bullet \\ \vdots \\ \theta_{q+2}^\bullet + a \end{bmatrix}.$$

for some constant $a \in \mathbb{R}$, $a \neq 0$. The algorithm finally returns the maximum likelihood estimator $\boldsymbol{m}$. The Hessian matrix V at $\boldsymbol{m}$ is then evaluated by numerically approximating the derivatives on an equally spaced grid. To justify the transformation of the Hessian matrix appearing in the definition of W we note that the Hessian matrix V at the maximum $\boldsymbol{m}$ is negative definit, hence $-V$ positive definit as well as $W = -V^{-1}$, as it has to be to serve as a covariance matrix. Moreover, high values on the diagonal of the Hessian matrix V indicate sharply declining values around the maximum, which correspond to small values in the diagonal of the covariance matrix. Therefore the negative inverse W of V is used as covariance matrix.

We now summarize the Metropolis-Hastings step to draw a sample from $f(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s})$:

1. Find the maximum likelihood estimator $\boldsymbol{m}$ of $[\log g(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s}) + \log \pi(\boldsymbol{\theta})]$ using the Nelder-Mead algorithm where $\log g(\boldsymbol{\theta}|\tilde{\boldsymbol{y}}^*, \boldsymbol{s})$ is evaluated running the Kalman recursions (6.16) to (6.23) and applying the prediction error decomposition (6.15).

2. Approximate the Hessian matrix V at $\boldsymbol{m}$ and calculate $W = -V^{-1}$.

3. Draw a proposal $\boldsymbol{\theta}^\circ$ from the $(q+2)$-dimensional normal distribution $q(\boldsymbol{\theta}) = N_{q+2}(\boldsymbol{\theta}|\boldsymbol{m}, W)$.

4. Accept the proposal $\boldsymbol{\theta}^\circ$ with probability

$$\alpha(\boldsymbol{\theta}^\bullet, \boldsymbol{\theta}^\circ|\tilde{\boldsymbol{y}}^*, \boldsymbol{s}) = \min\left\{\frac{g(\boldsymbol{\theta}^\circ|\tilde{\boldsymbol{y}}^*, \boldsymbol{s})}{g(\boldsymbol{\theta}^\bullet|\tilde{\boldsymbol{y}}^*, \boldsymbol{s})}\frac{\pi(\boldsymbol{\theta}^\circ)}{\pi(\boldsymbol{\theta}^\bullet)}\frac{q(\boldsymbol{\theta}^\bullet)}{q(\boldsymbol{\theta}^\circ)}, 1\right\}.$$

   If $\boldsymbol{\theta}^\circ$ is rejected, retain $\boldsymbol{\theta}^\bullet$ as the next draw.

Next we consider the block update of $\boldsymbol{h}$ from $f(\boldsymbol{h}|\boldsymbol{\theta}, \tilde{\boldsymbol{y}}^*, \boldsymbol{s})$. Let $\boldsymbol{\theta}^\bullet = (\boldsymbol{\alpha}^{\bullet\prime}, \phi^\bullet, \sigma^\bullet)'$ denote the sample from the MH step before. Sampling can be done using the simulation

smoother of De Jong and Shephard (1995) which is described in Section 2.7. It requires running the Kalman recursions (6.16) to (6.23) with $\boldsymbol{\alpha} = \boldsymbol{\alpha}^\bullet$, $\phi = \phi^\bullet$, and $\sigma = \sigma^\bullet$, storing e_t, $f_{t|t-1}$ and n_t for each $t = 1, \ldots, T$, and finally running the following backward recursions for $t = T, \ldots, 1$ which correspond to Equations (2.45) to (2.52). Initially set $r_T = 0$ and $m_T = 0$.

$$
\begin{aligned}
d_t &:= f_{t|t-1}^{-1} + \phi^2 n_t^2 m_t, && (6.24)\\
b_t &:= f_{t|t-1}^{-1} e_t - \phi n_t r_t, && (6.25)\\
q_t &:= v_{s_t}^2 - v_{s_t}^4 d_t, && (6.26)\\
\kappa_t &\sim N(0, q_t), && (6.27)\\
a_t &:= v_{s_t}^2 (d_t - \phi^2 n_t m_t), && (6.28)\\
r_{t-1} &:= f_{t|t-1}^{-1} e_t + (\phi - \phi n_t) r_t - a_t q_t^{-1} \kappa_t, && (6.29)\\
m_{t-1} &:= f_{t|t-1}^{-1} + (\phi - \phi n_t)^2 m_t + a_t^2 q_t^{-1}, && (6.30)\\
\xi_t &:= \tilde{y}_t^* - m_{s_t} - v_{s_t}^2 b_t - \kappa_t. && (6.31)
\end{aligned}
$$

Following Section 2.7, the vector $(\xi_1, \ldots, \xi_T)$ can be considered as a sample from the conditional distribution $f(h_1^*, \ldots, h_T^* | \boldsymbol{\theta}, \tilde{\boldsymbol{y}}^*, \boldsymbol{s})$.

6.3 GM-MGMC sampler

Simulations with the Hybrid MCMC sampler developed in Section 6.2 show that the produced chains especially for the cutpoints c_k and the regression intercept β_0 converge very slowly to the region around the true values (cf. Section 6.3.2). We encountered these problems already for the standard Gibbs sampler in the autoregressive ordered probit model of Chapter 3. There we achieved a fast convergence by an appropriate grouped move step which was inserted after one complete iteration of the standard Gibbs sampler.

In the following we develop again a grouped move step based on the Theorem of Liu and Sabatti (2000) (cf. Section 2.4). Inserting this step in the Hybrid MCMC sampler of Section 6.2 we have an algorithm which falls again in the class of the GM-MGMC samplers.

6.3.1 Development of an appropriate grouped move step

As was pointed out in Section 2.4 the difficulty in developing a suitable grouped move step is how to choose the distribution π and the transformation group Γ which appear in the Theorem of Liu and Sabatti (2000) (cf. Section 2.4). They should be chosen in such a way that on the one hand the problematic parameters are transformed and that on the other hand the distribution $\pi(\gamma(\boldsymbol{x}))|J_\gamma(\boldsymbol{x})|L(d\gamma)$ allows to draw samples very fast. For the AOP model we succeeded by using the posterior distribution and the partial scale group for π and Γ, respectively.

Here, however, it seems to be impossible to find a transformation group which satisfies these conditions when π is the full posterior distribution. Therefore we apply the Theorem of Liu and Sabatti (2000) not to the full posterior distribution, but on a certain conditional one. In particular, we set

$$\boldsymbol{w} := (y_1^*, \ldots, y_T^*, c_2, \ldots, c_{K-1}, \beta_0, \ldots, \beta_p)\,,$$

combine the remaining parameters to

$$\boldsymbol{R} := (\boldsymbol{h}^*, \boldsymbol{\alpha}, \phi, \sigma)\,,$$

and let $\pi(\boldsymbol{w})$ be the conditional distribution $f(\boldsymbol{w}|\boldsymbol{y}, \boldsymbol{R})$. With $\mathcal{F}_t$ denoting the observations until time t, i.e. $\mathcal{F}_t = (y_1, \ldots, y_t)$, we see that

$$\begin{aligned} f(y_1^*, \ldots, y_t^*, \boldsymbol{y}, \boldsymbol{c}, \boldsymbol{\beta}, \boldsymbol{R}) &= f(y_t|y_1^*, \ldots, y_t^*, \mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\beta}, \boldsymbol{R}) f(y_t^*|y_1^*, \ldots, y_{t-1}^*, \mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\beta}, \boldsymbol{R}) \cdot \\ &\quad \cdot f(y_1^*, \ldots, y_{t-1}^*, \mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\beta}, \boldsymbol{R}) \\ &= f(y_t|y_t^*, \boldsymbol{c}) f(y_t^*|\boldsymbol{\beta}, \boldsymbol{R}) f(y_1^*, \ldots, y_{t-1}^*, \mathcal{F}_{t-1}, \boldsymbol{c}, \boldsymbol{\beta}, \boldsymbol{R}). \end{aligned} \tag{6.32}$$

Using Equation (6.32) recursively we have the following proportionality for the conditional distribution $\pi(\boldsymbol{w})$:

$$\begin{aligned} \pi(\boldsymbol{w}) &= f(y_1^*, \ldots, y_T^*, \boldsymbol{c}, \boldsymbol{\beta}|\boldsymbol{y}, \boldsymbol{R}) \\ &\propto f(y_1^*, \ldots, y_T^*, \boldsymbol{y}, \boldsymbol{c}, \boldsymbol{\beta}, \boldsymbol{R}) \\ &= \left[\prod_{t=1}^{T} f(y_t|y_t^*, \boldsymbol{c})\right] \left[\prod_{t=1}^{T} f(y_t^*|\boldsymbol{\beta}, \boldsymbol{R})\right] \pi(\boldsymbol{c}, \boldsymbol{\beta}, \boldsymbol{R}). \end{aligned}$$

Since we assume $\boldsymbol{c}$, $\boldsymbol{\beta}$, and $\boldsymbol{R}$ to be a priori independent of each other, a $(p+1)$-dimensional normal distribution for $\boldsymbol{\beta}$ with mean vector $\boldsymbol{b}_0$ and covariance matrix B_0

and noninformative priors for the cutpoints, we get

$$\begin{aligned}\pi(\boldsymbol{w}) &\propto \left[\prod_{t=1}^{T} f(y_t|y_t^*, \boldsymbol{c})\right] \pi(\boldsymbol{c}) \left[\prod_{t=1}^{T} f(y_t^*|\boldsymbol{\beta}, \boldsymbol{R})\right] \pi(\boldsymbol{\beta}) \\ &= \left[\prod_{t=1}^{T} \mathbb{1}_{[c_{y_t-1}, c_{y_t})}(y_t^*)\right] \mathbb{1}_{\{0<c_2<\ldots<c_{K-1}<C\}} \\ &\quad \cdot \exp\left\{-\frac{1}{2}\left[\sum_{t=1}^{T} \frac{(y_t^* - \boldsymbol{x}_t'\boldsymbol{\beta})^2}{\exp(h_t^*)} + (\boldsymbol{\beta}-\boldsymbol{b}_0)' B_0^{-1}(\boldsymbol{\beta}-\boldsymbol{b}_0)\right]\right\}. \qquad (6.33)\end{aligned}$$

At this point we now have to set the mean of the normal prior for $\boldsymbol{\beta}$ to zero, i.e. $\pi(\boldsymbol{\beta}) = N_{p+1}(\boldsymbol{\beta} \mid \mathbf{0}, B_0)$. Otherwise some of the transformations in the following equations cannot be made. Using this prior for $\boldsymbol{\beta}$, expression (6.33) simplifies to

$$\pi(\boldsymbol{w}) \propto \left[\prod_{t=1}^{T} \mathbb{1}_{[c_{y_t-1}, c_{y_t})}(y_t^*)\right] \mathbb{1}_{\{0<c_2<\ldots<c_{K-1}<C\}} \exp\left\{-\frac{1}{2}\left[\sum_{t=1}^{T} \frac{(y_t^* - \boldsymbol{x}_t'\boldsymbol{\beta})^2}{\exp(h_t^*)} + \boldsymbol{\beta}' B_0^{-1}\boldsymbol{\beta}\right]\right\}.$$

In order to get an easy sampling distribution we now use the scale group

$$\Gamma = \{\gamma > 0 : \gamma(\boldsymbol{w}) = (\gamma w_1, \ldots, \gamma w_d)\}$$

with $\gamma^{-1}d\gamma$ as left-Haar measure. In this case γ has to be drawn from $\gamma^{d-1}\pi(\gamma\boldsymbol{w})$, where d denotes the dimension of $\boldsymbol{w}$. Since $\boldsymbol{w}$ contains all the latent variables y_t^*, $t = 1, \ldots, T$, the cutpoints c_k, $k = 2, \ldots, K-1$, and β_j, $j = 0, \ldots, p$, we have $d = T + K + p - 1$. Therefore we get the proportionality

$$\begin{aligned}\gamma^{d-1}\pi(\gamma\boldsymbol{w}) &\propto \gamma^{T+K+p-2} \left[\prod_{t=1}^{T} \mathbb{1}_{[\gamma c_{y_t-1}, \gamma c_{y_t})}(\gamma y_t^*)\right] \mathbb{1}_{\{0<\gamma c_2<\ldots<\gamma c_{K-1}<C\}} \\ &\quad \cdot \exp\left\{-\frac{1}{2}\left[\sum_{t=1}^{T} \frac{(\gamma y_t^* - \boldsymbol{x}_t'\gamma\boldsymbol{\beta})^2}{\exp(h_t^*)} + \gamma\boldsymbol{\beta}' B_0^{-1}\gamma\boldsymbol{\beta}\right]\right\}. \qquad (6.34)\end{aligned}$$

For all $\gamma > 0$ we have the equivalence

$$\begin{aligned}&[0 < \gamma c_2 < \ldots < \gamma c_{K-1} < C \text{ and } c_{K-1} < C] \\ &\qquad \Longleftrightarrow \quad \left[0 < c_2 < \ldots < c_{K-1} < C \text{ and } \gamma^2 < C^2/c_{K-1}^2\right].\end{aligned}$$

Since expression (6.34) is considered to be a density for γ (up to a normalizing constant), and since during all updates of the Hybrid MCMC sampler the condition $0 < c_2 < \ldots < c_{K-1} < C$ is always fulfilled, this equivalence leads to the proportionality

$$\begin{aligned}\mathbb{1}_{\{0<\gamma c_2<\ldots<\gamma c_{K-1}<C\}} &\propto \mathbb{1}_{\{0<\gamma c_2<\ldots<\gamma c_{K-1}<C\}} \mathbb{1}_{\{c_{K-1}<C\}} \\ &= \mathbb{1}_{\{0<c_2<\ldots<c_{K-1}<C\}} \mathbb{1}_{\{\gamma^2<C^2/c_{K-1}^2\}} \\ &\propto \mathbb{1}_{\{\gamma^2<C^2/c_{K-1}^2\}}.\end{aligned}$$

Therefore expression (6.34) simplifies to

$$\gamma^{T+K+p-2}\left[\prod_{t=1}^{T}\mathbb{1}_{[c_{y_t-1},c_{y_t})}(y_t^*)\right]\mathbb{1}_{\{\gamma^2<C^2/c_{K-1}^2\}}\exp\left\{-\frac{1}{2}\gamma^2\left[\sum_{t=1}^{T}\frac{(y_t^*-\boldsymbol{x}_t'\boldsymbol{\beta})^2}{\exp(h_t^*)}+\boldsymbol{\beta}'B_0^{-1}\boldsymbol{\beta}\right]\right\}$$
$$\propto \left(\gamma^2\right)^{\frac{T+K+p-2}{2}}\exp\left\{-\frac{1}{2}\gamma^2\left[\sum_{t=1}^{T}\frac{(y_t^*-\boldsymbol{x}_t'\boldsymbol{\beta})^2}{\exp(h_t^*)}+\boldsymbol{\beta}'B_0^{-1}\boldsymbol{\beta}\right]\right\}\mathbb{1}_{\{\gamma^2<C^2/c_{K-1}^2\}}.$$

If one chooses a prior for $\boldsymbol{c}$ with infinite support, i.e. $C=\infty$, this expression is proportional to a Gamma distribution $\Gamma(a,b)$ for γ^2 with parameters

$$a = \frac{T+K+p}{2}, \tag{6.35}$$

$$b = \frac{1}{2}\left[\sum_{t=1}^{T}\frac{(y_t^*-\boldsymbol{x}_t'\boldsymbol{\beta})^2}{\exp(h_t^*)}+\boldsymbol{\beta}'B_0^{-1}\boldsymbol{\beta}\right], \tag{6.36}$$

where the $\Gamma(a,b)$ density is given by $f_{\Gamma(a,b)}(x)=b^a x^{a-1}e^{-bx}/\Gamma(a),\ x\geq 0$.

If a finite support for $\boldsymbol{c}$ is chosen, i.e. $C<\infty$, one gets a Gamma distribution for γ^2 with the same parameters as before, however truncated to $(0, C^2/c_{K-1}^2)$. Of course, one can easily sample also from this truncated Gamma distribution by rejection sampling (cf. Section 2.1).

If γ^2 is drawn from the (truncated) Gamma distribution with a and b given in (6.35) and (6.36), respectively, the Theorem by Liu and Sabatti (2000) guarantees that $\gamma\boldsymbol{w}=\sqrt{\gamma^2}\boldsymbol{w}$ can be considered as a sample from $\pi(\boldsymbol{w})=f(\boldsymbol{w}|\boldsymbol{y},\boldsymbol{R})$, if $\boldsymbol{w}$ itself is a sample from this conditional distribution. Such a sample is given directly after the updates of $\boldsymbol{\beta}$, y_t^*, $t=1,\ldots,T$, and c_k, $k=2,\ldots,K-1$. Therefore we insert the corresponding grouped move step exactly at this point in each iteration of the Hybrid MCMC sampler from Section 6.2. This results in a GM-MGMC sampler, where each iteration consists of the following steps:

Algorithm 6.1 One iteration of the GM-MGMC sampler for the OSV model

1. **MCMC-Step (Part 1)**

   - Draw $\boldsymbol{\beta}$ from $(p+1)$-variate normal.
   - Draw y_t^*, $t=1,\ldots,T$, from truncated univariate normals.
   - Draw c_k, $k=2,\ldots,K-1$, from Unif(l_k,r_k) where

$$l_k = \max\{c_{k-1}, \max_{t=1,\ldots,T}\{y_t^*|y_t=k\}\},$$
$$r_k = \min\{c_{k+1}, \min_{t=1,\ldots,T}\{y_t^*|y_t=k+1\}\}.$$

Get $\boldsymbol{\beta}_{cur}, \boldsymbol{y}^*_{cur}, \boldsymbol{c}_{cur}$ as current values.

2. **GM-Step**
   Draw γ^2 from the (truncated) $\Gamma(a, b)$ distribution with a and b defined in (6.35) and (6.36), respectively, and update $\boldsymbol{\beta}^*_{cur}$, $\boldsymbol{y}^*_{cur}$, $\boldsymbol{c}_{cur}$ by multiplication with the group element $\gamma = \sqrt{\gamma^2}$,

$$\begin{aligned} \boldsymbol{\beta}_{new} &\leftarrow \gamma \boldsymbol{\beta}_{cur}, \\ \boldsymbol{y}^*_{new} &\leftarrow \gamma \boldsymbol{y}^*_{cur}, \\ \boldsymbol{c}_{new} &\leftarrow \gamma \boldsymbol{c}_{cur}. \end{aligned}$$

3. **MCMC-Step (Part 2)**

   - Compute $\tilde{y}^*_t = \log(y^*_t - \boldsymbol{x}'_t \boldsymbol{\beta})^2$ for $t = 1, \ldots, T$.
   - Draw s_t, $t = 1, \ldots, T$, proportional to $\Pr(s_t) N(\tilde{y}^*_t | h^*_t + m_{s_t}, v^2_{s_t})$.
   - Draw $(\boldsymbol{\alpha}, \phi, \sigma)$ via Metropolis-Hastings step; use ML-estimates of $(\boldsymbol{\alpha}, \phi, \sigma)$ to find an adequate multivariate normal proposal (see Section 6.2.7).
   - Draw $\boldsymbol{h}^*$ in one block using the simulation smoother of De Jong and Shephard (1995) (see Section 6.2.7).

6.3.2 An illustration: Hybrid MCMC against GM-MGMC

We now illustrate the fundamental improvement which is achieved by adding the GM-step to the Hybrid MCMC sampler of Section 6.2. Here and in the following we always use the hyperparameters $C = \infty$, $\boldsymbol{b}_0 = \boldsymbol{0}$, $B_0 = \mathrm{diag}(10, \ldots, 10)$, $C_\alpha = 10^6$, $C_\sigma = 10$, so that the prior distributions are

$$\begin{aligned} \pi(\boldsymbol{c}) &= \mathbb{1}_{\{0 < c_2 < \ldots < c_{K-1} < \infty\}}, \\ \pi(\boldsymbol{\beta}) &= N_{p+1}(\boldsymbol{\beta} \,|\, \boldsymbol{0}, \mathrm{diag}(10, \ldots, 10)), \\ \pi(\alpha_j) &= \mathbb{1}_{\{-10^6 < \alpha_j < 10^6\}}, \qquad j = 1, \ldots, q, \\ \pi(\phi) &= \mathbb{1}_{\{-1 < \phi < 1\}}, \\ \pi(\sigma) &= \mathbb{1}_{\{0 < \sigma < 10\}}. \end{aligned}$$

We simulate an OSV process of length $T = 22000$ where we allow for $K = 7$ response categories. For the log-volatility Equation (6.3) we use a two-dimensional covariate vector $\boldsymbol{z}_t$. The two components are exactly the covariates from the IBM data which will be used in Chapter 7. The simulation parameters in the log-volatility equation are

set to $\alpha_1 = 0.25$, $\alpha_2 = 0.15$, $\phi = 0.90$ and $\sigma = 0.20$. Using these parameters we first simulate the log-volatility process $\{h_t^* \,|\, t = 1, \ldots, T\}$. The covariate vector $\boldsymbol{x}_t$ in the equation for the latent variables y_t^* also has two components. The first corresponds to the intercept and is always 1, the second is the lagged response y_{t-1}. The simulation parameters are set to $\beta_0 = 3.50$ and $\beta_1 = -0.30$. To generate the response, we choose the cutpoints as $c_2 = 0.90$, $c_3 = 1.80$, $c_4 = 2.75$, $c_5 = 3.65$, and $c_6 = 4.50$. We note that these simulation parameters are chosen close to the estimated values for the IBM data presented in Chapter 7. Therefore, by showing that the GM-MGMC sampler works very well for these parameters and covariates, we get a first justification that the GM-MGMC algorithm works well for the IBM data.

We run both the Hybrid MCMC sampler from Section 6.2 and the GM-MGMC sampler for 4000 iterations. As starting values for the cutpoints $c_2, \ldots, c_6$ we choose 2.0, 4.0, 6.0, 8.0, 10.0, respectively, 0.0 for each of the regression coefficients α_j, 0.8 for ϕ, and 0.3 for σ. Since each iteration starts with the $\boldsymbol{\beta}$-update, starting values for β_j, $j = 0, \ldots, p$, are not necessary.

Figures 6.1 to 6.4 demonstrate the fundamental superiority of the GM-MGMC sampler. They show the first 1000 iterations of both the Hybrid MCMC and the GM-MGMC sampler for the cutpoints $c_2, \ldots, c_6$, the regression coefficients β_0, β_1, α_1, and α_2, and the parameters ϕ and σ. As can be seen from Figure 6.1 the chains for the cutpoints produced by the Hybrid MCMC sampler move very slowly around the starting values and do hardly move to the regions around the true values which are indicated by the horizontal lines. It is not clear, whether the chains for the cutpoints produced by the Hybrid MCMC sampler will converge at all, and when they do so, it will take thousands of iterations. The same holds for the observed chains for the regression coefficients, given in Figures 6.2 and 6.3, respectively. The observed chains of ϕ and σ are given in Figure 6.4. For the Hybrid MCMC sampler, they stabilize at completely wrong values. Moreover, the observed chains for α_1, α_2, ϕ, and σ do not move after iteration 5: The proposal values in the MH-step were never accepted. This, however, is probably a consequence of the fact, that the other parameters are also estimated wrongly.

The chains produced by the GM-MGMC sampler are also given in the Figures 6.1 to 6.4. They converge within about only 100 iterations for the cutpoints as well as for the regression coefficients and the parameters ϕ and σ. An interesting effect occurs at iterations 2 to 13 of the parameters α_1, α_2, ϕ, and σ, which do not move during these 11 iterations. This effect is typical for the burn-in period and can be explained as follows. The parameters α_1, α_2, ϕ, and σ are drawn by a Metropolis-Hastings step (cf. Section

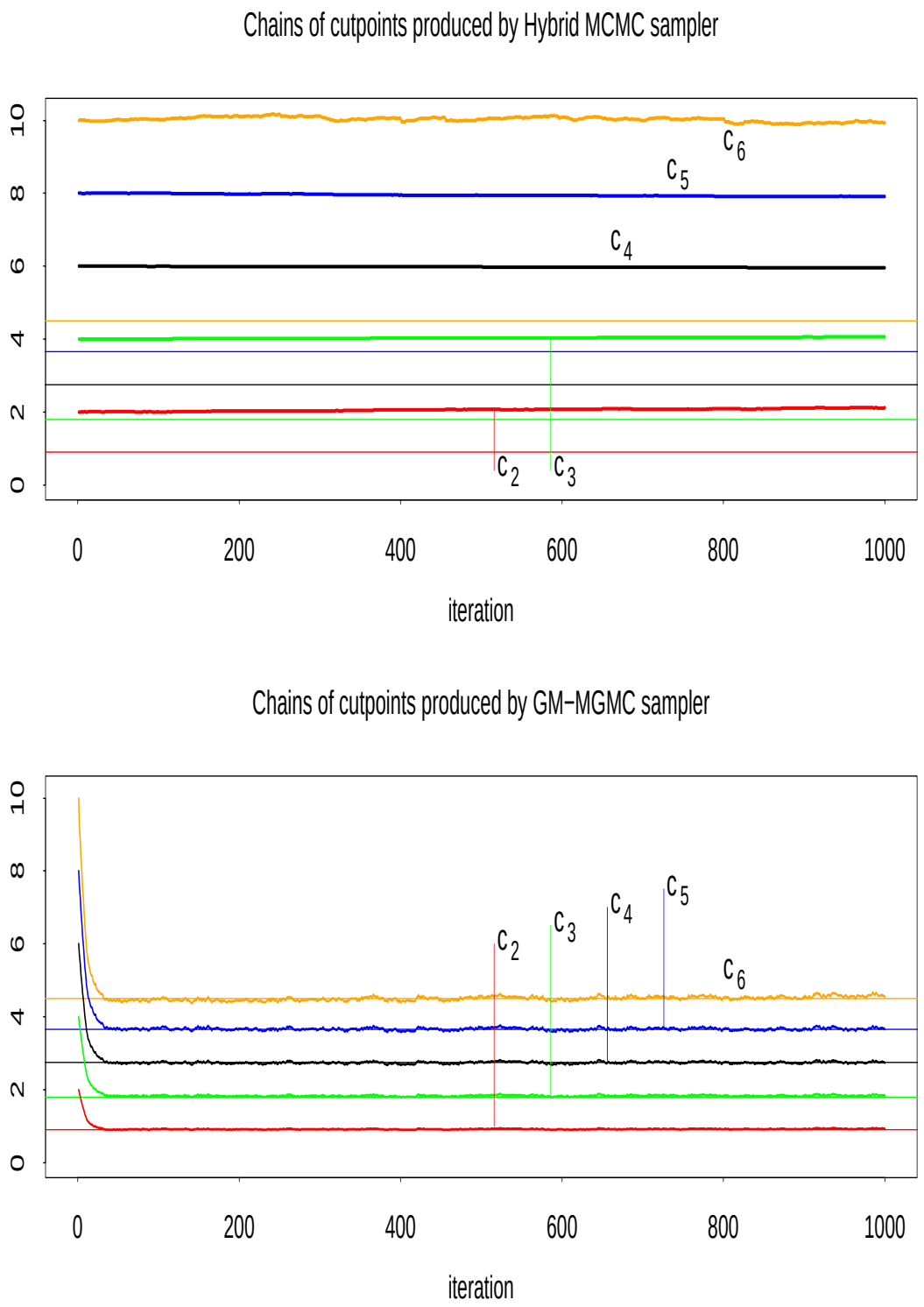

Figure 6.1: First 1000 iterations of chains for cutpoints produced by Hybrid MCMC sampler (above) and GM-MGMC sampler (below). The horizontal thin lines indicate the true values.

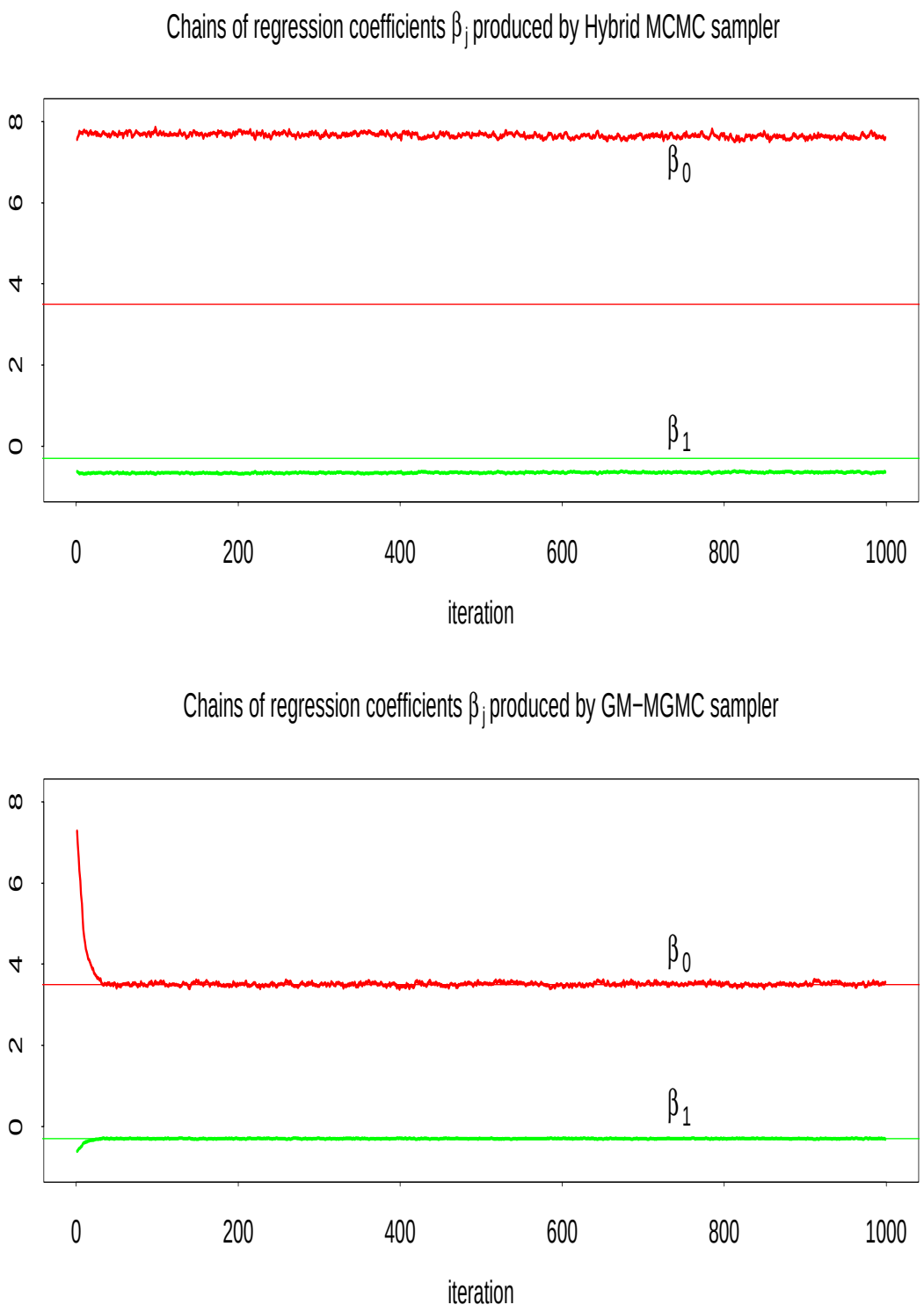

Figure 6.2: First 1000 iterations of chains for regression coefficients β_j produced by Hybrid MCMC sampler (above) and GM-MGMC sampler (below). The horizontal thin lines indicate the true values.

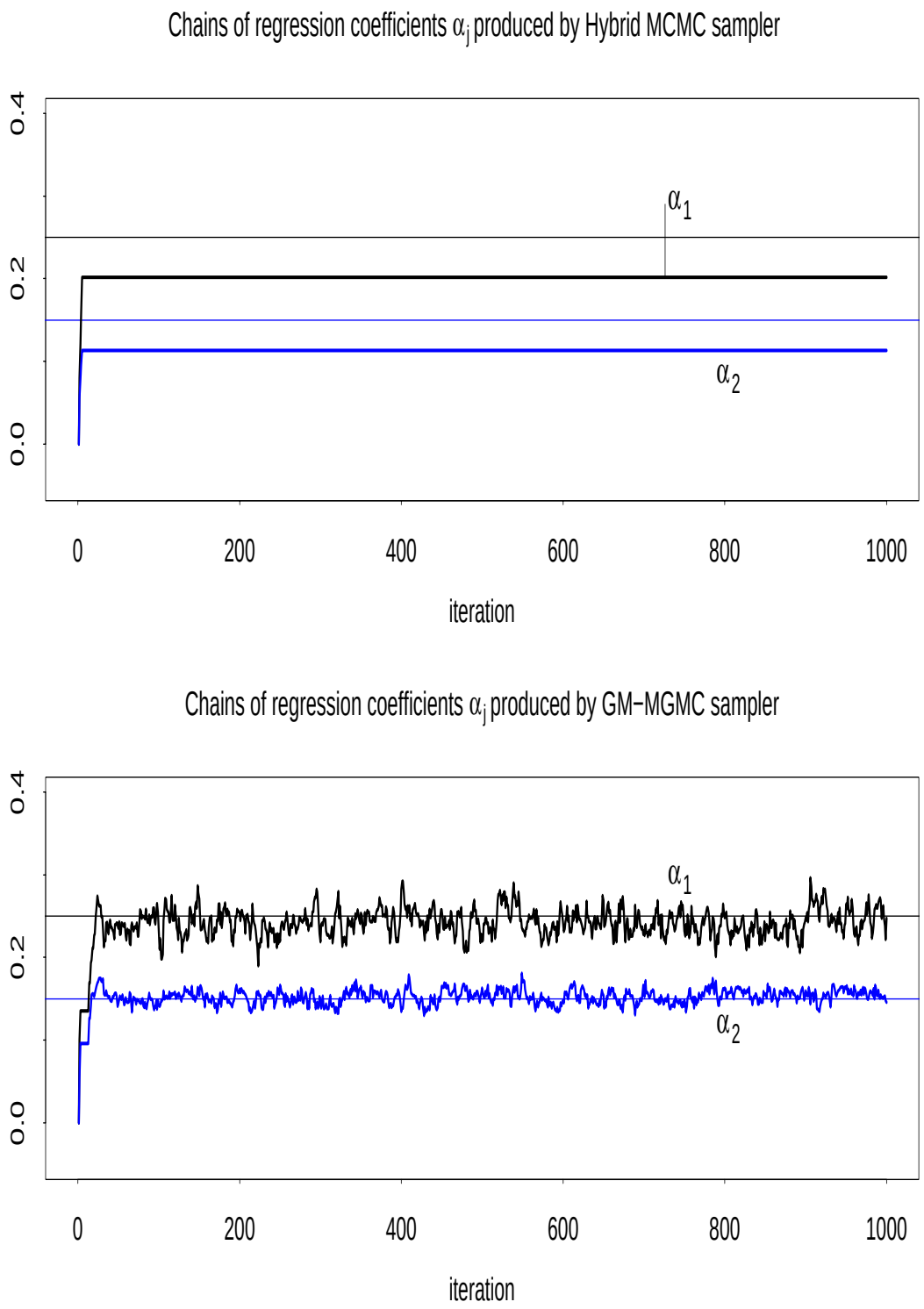

Figure 6.3: First 1000 iterations of chains for regression coefficients α_j produced by Hybrid MCMC sampler (above) and GM-MGMC sampler (below). The horizontal thin lines indicate the true values.

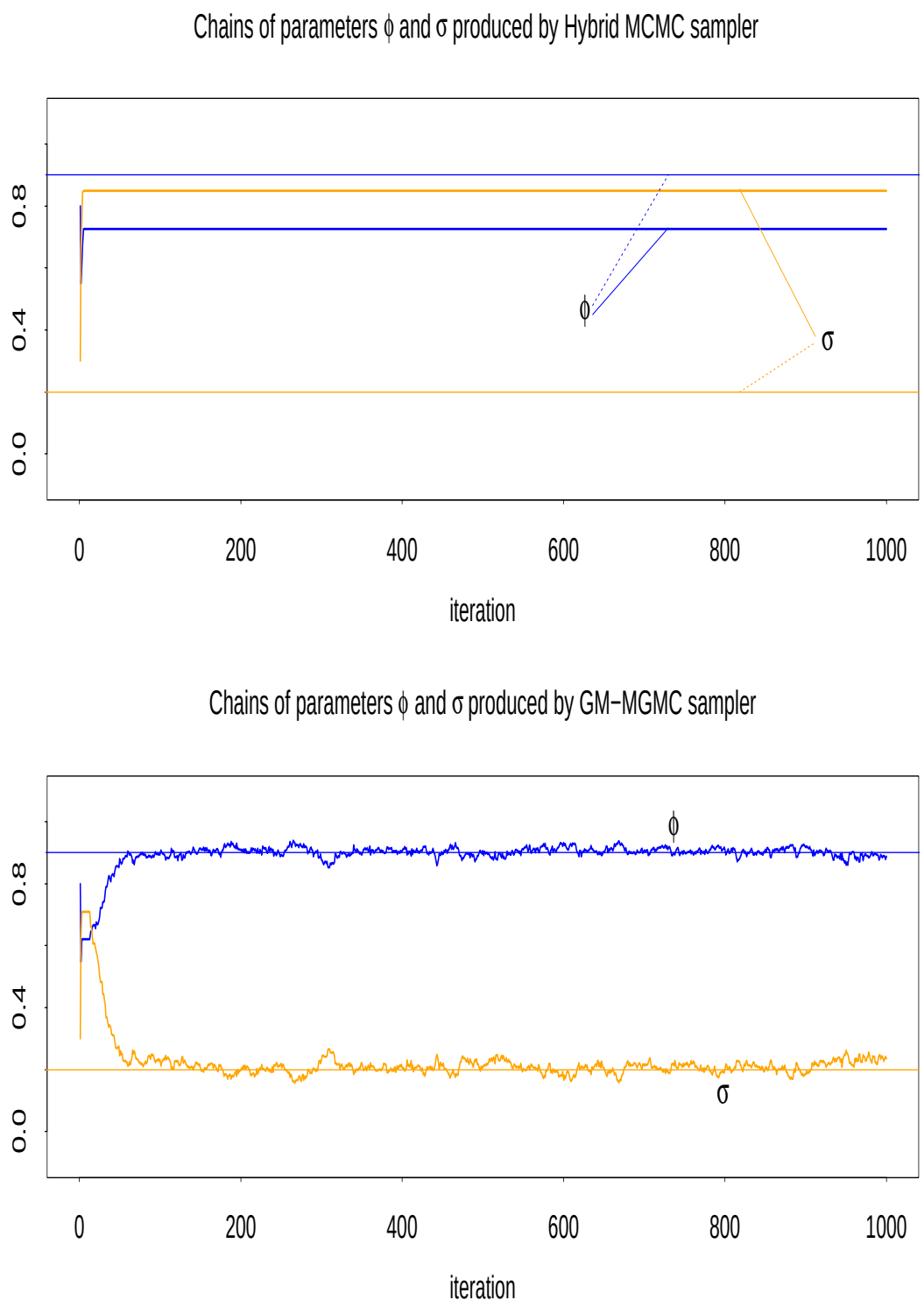

Figure 6.4: First 1000 iterations of chains for parameters ϕ and σ produced by Hybrid MCMC sampler (above) and GM-MGMC sampler (below). The horizontal thin lines indicate the true values.

6.2). During the iterations 2 to 13, the proposal values for these parameters were never accepted. However, when the chains of the other parameters get closer to the true values (cf. Figures 6.1 and 6.2), the multivariate normal proposal for $\boldsymbol{\theta} = (\alpha_1, \alpha_2, \phi, \sigma)$ approximates the target density very good, and the chains of the components of $\boldsymbol{\theta}$ start to converge. After the first 100 iterations, the average acceptance rate is about 90%. Since the proposal density for $\boldsymbol{\theta}$ is adapted to the target density very carefully, this leads to a fast mixing in the whole support of the target density.

We further investigate the empirical autocorrelations for the lags 0 to 200 in the observed chains after a burn-in period of 1000 iterations for both samplers. As can be seen from Figure 6.5, the GM-MGMC sampler is better also from this point of view since the empirical autocorrelations in the GM-MGMC chains decline very fast. In contrast, in the chains produced by the Hybrid MCMC sampler they decline very slowly and are greater than 0.5 even at lag 200. Furthermore, the empirical autocorrelations for the chains α_1, α_2, ϕ, and σ produced by the Hybrid MCMC sampler do not exist, since these four chains did not move during the iterations 1001 to 4000. Therefore, for these parameters, Figure 6.5 only contains the autocorrelations for the chains produced by the GM-MGMC sampler.

Summarizing the results of this section, we conclude that the Hybrid MCMC sampler of Section 6.2 does not work well and therefore hardly can be used for fitting the OSV model to a data set. In contrast, the GM-MGMC sampler works very well, since the chains converge very rapidly and the autocorrelations in the chains are small. However, an open interesting question is, how accurate the parameter estimates are when we run the GM-MGMC sampler several times. This will be investigated in the next section.

6.4 Simulation study

Here we conduct a simulation study to assess the accuracy of the posterior mean estimates by the GM-MGMC sampler. In the Sections 6.4.1 and 6.4.2 we consider two settings which differ in the choice of the covariates, the length of the data, and the values of the simulation parameters. The second setting is interesting for the application in Chapter 7 since the covariates in the simulated data set are taken from the IBM data set investigated there.

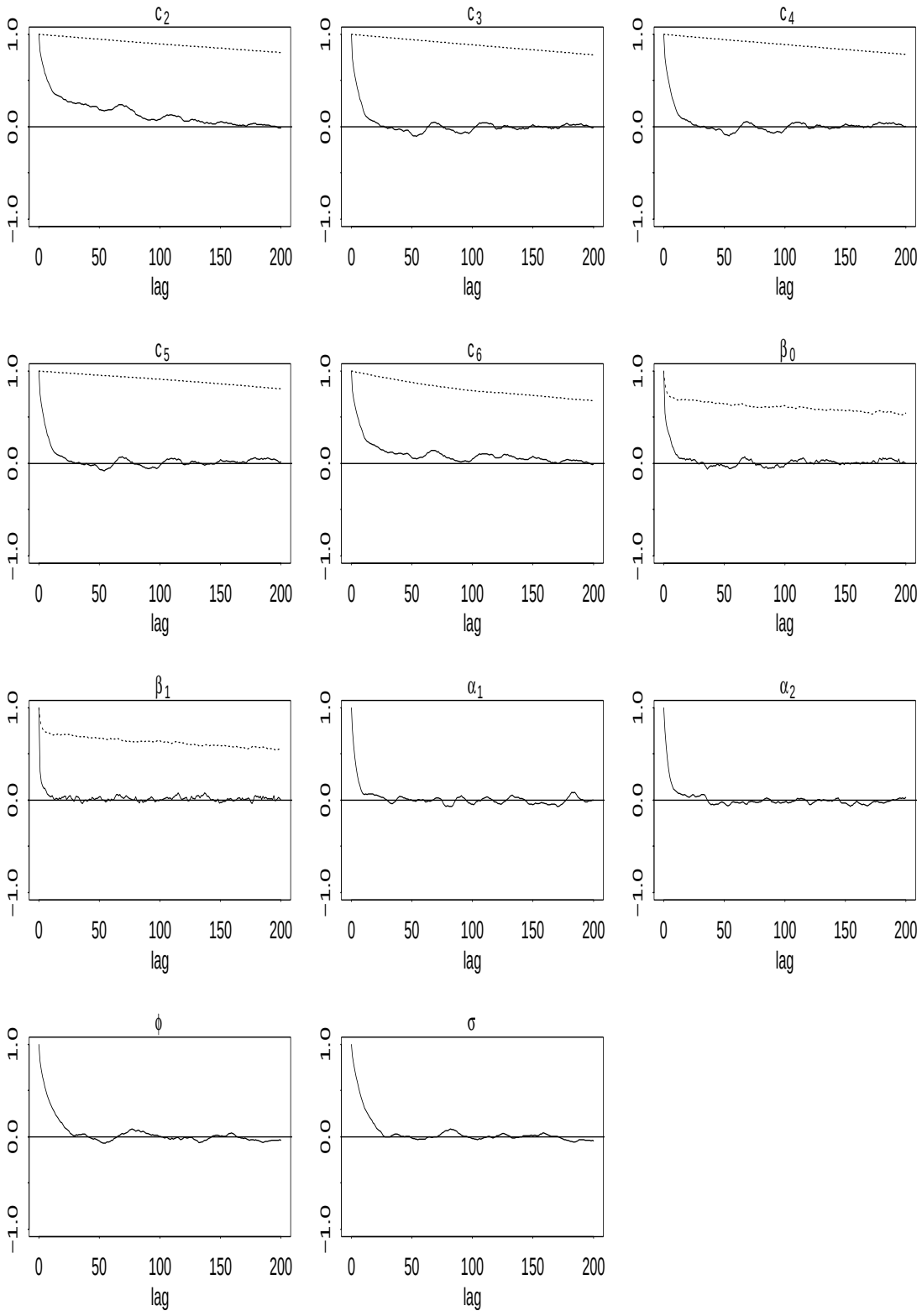

Figure 6.5: Autocorrelations of chains produced by Hybrid MCMC sampler (dotted) and GM-MGMC sampler (solid).

ϕ	σ	β_0	β_1	α_1	c_2	c_3	c_4	c_5	c_6
0.90	0.25	2.80	-0.30	0.80	0.60	1.20	1.75	2.35	3.00

Table 6.2: Parameters for Simulation Setting 1.

6.4.1 Setting 1

Here the prior distributions from Section 6.3.2 are used. 20 data sets are simulated, each of length 8000, with response categories $1, \ldots, 7$. First we simulate the covariate vector $\boldsymbol{z}_t$ for the log-volatility equation. In this setting $\boldsymbol{z}_t$ is assumed to be one-dimensional and is therefore denoted by the corresponding non-bold letter z_t. To have a covariate not too similar to a normal distributed random variable we simulate uniformly distributed errors

$$a_t \sim \text{Unif}(-0.1, 0.1) \text{ i.i.d.}, \qquad t = 1, \ldots, 8000,$$

and use these errors to compute

$$z_t = 0.16 \cdot \sin(t/40)^2 + a_t, \qquad t = 1, \ldots, 8000.$$

The covariate z_t is used for all 20 data sets. The other covariate vector consists of the component for the intercept and the lagged observation, i.e. $\boldsymbol{x}_t = (1, y_{t-1})'$. This is reasonable choice for high-frequency financial data (cf. Chapter 7). Of course, using the lagged observation as covariate, one must compute y_t directly after simulating y_t^* since y_t is needed for the simulation of y_{t+1}^*.

The parameter choices used for simulation are given in Table 6.2. They are close to the ones which we found in high-frequency data investigated (cf. Chapter 7). The cutpoints are chosen so that the frequencies in the categories also correspond to those in high-frequency data, where only few observations lie in the categories 1 and 7, and many observations in category 4.

We computed posterior mean estimates by running the GM-MGMC sampler for 4000 iterations each, where the first 1000 iterations were discarded for burn-in. Running the GM-MGMC sampler for such data sets takes about 2.1 seconds per iteration on an UltraSPARC III Cu 900 Mhz processor. Therefore, running 4000 iterations for each of the 20 data sets takes about 47 hours.

Table 6.3 gives the means and standard deviations of the posterior mean estimates across the 20 samples, based on 4000 GM-MGMC draws, discarding the first 1000. Since the means of the posterior mean estimates are almost identical to the true values for all

	true	mean	std. dev.		true	mean	std. dev.
ϕ	0.90	0.9010	0.0100	c_2	0.60	0.6069	0.0255
σ	0.25	0.2562	0.0239	c_3	1.20	1.2095	0.0266
β_0	2.80	2.8176	0.0569	c_4	1.75	1.7631	0.0519
β_1	-0.30	-0.3041	0.0080	c_5	2.35	2.3560	0.0527
α_1	0.80	0.7872	0.0896	c_6	3.00	2.9898	0.0638

Table 6.3: Means and standard deviations of the posterior mean estimates across the 20 samples in Setting 1 using the GM-MGMC sampler.

ϕ	σ	β_0	β_1	α_1	α_2	c_2	c_3	c_4	c_5	c_6
0.90	0.20	3.50	-0.30	0.25	0.15	0.90	1.80	2.75	3.65	4.50

Table 6.4: Parameters for Simulation Setting 2.

parameters, we conclude that on average we estimate nearly the true parameter values. Moreover, as one can see from the standard deviations, the posterior mean estimates themselves have always been close to the true values for each of the 20 data sets.

6.4.2 Setting 2

For the second parameter setting we use again the same prior distributions as in Section 6.3.2. In contrast to Setting 1, also the covariate vectors and the simulation parameters are the same as in the example of Section 6.3.2. We recall that the covariate vector $\boldsymbol{z}_t$ was two-dimensional and that the covariate vector $\boldsymbol{x}_t$ consists of the component for the intercept and the lagged observation y_{t-1}. Hence, $\boldsymbol{x}_t$ is the same as in Setting 1. As mentioned in Section 6.3.2, the simulation parameters (cf. Table 6.4) lie close to the estimated values for the IBM data in Chapter 7. Since the parameters are estimated very well in this parameter setting, we can conclude that the estimates which we get for the IBM data are also accurate.

Again we simulate 20 data sets. However, the length of each data set is now $T = 22000$, which is close to the length of the IBM data set of Chapter 7 (there $T = 22689$). We computed posterior mean estimates by running the GM-MGMC sampler for 4000 iterations each, where the first 1000 iterations were discarded for burn-in.

Running the GM-MGMC sampler for such data sets takes about six seconds per iteration

	true	mean	std. dev.		true	mean	std. dev.
ϕ	0.90	0.8942	0.0093	c_2	0.90	0.9041	0.0121
σ	0.20	0.2080	0.0150	c_3	1.80	1.8041	0.0124
β_0	3.50	3.5092	0.0281	c_4	2.75	2.7558	0.0186
β_1	-0.30	-0.3011	0.0049	c_5	3.65	3.6581	0.0263
α_1	0.25	0.2513	0.0090	c_6	4.50	4.5147	0.0419
α_2	0.15	0.1524	0.0095				

Table 6.5: Means and standard deviations of the posterior mean estimates across the 20 samples in Setting 2 using the GM-MGMC sampler.

on an UltraSPARC III Cu 900 Mhz processor. Therefore, running 4000 iterations for each of the 20 data sets takes about 133 hours.

Table 6.5 gives the means and standard deviations of the posterior mean estimates across the 20 samples. As in parameter Setting 1, the means of the posterior mean estimates are almost identical to the true values for all parameters. Therefore we conclude that on average we estimate nearly the true parameter values. Moreover, the standard deviations are very small, so that for each of the 20 data sets the posterior mean estimates themselves have always been close to the true values.

Chapter 7

Application of the Ordinal-Response Stochastic Volatility Model to High-Frequency Finance

In this chapter we apply the OSV model to data from the IBM stock in January 2001. In contrast to Chapter 5, where we applied the AOP model to the absolute price changes, we model here the signed price changes.

First, we describe the data set in Section 7.1. Since many papers today deal with models for the log returns of prices of stocks or derivatives, we consider in Section 7.2 the difference between modeling the the signed price changes and modeling the log returns. Finally in Section 7.3 we fit the OSV model to the data.

7.1 Description of IBM data

The data used here are price changes of the IBM stock traded at the New York Stock Exchange (NYSE) from January 9, 2001 to January 25, 2001. In this period we removed data from Mondays and Fridays, and data before 09:50am and after 03:40pm to exclude data which might exhibit a special behavior. Table 7.1 gives the periods contained in our IBM data set.

The prices and therefore also the price differences take on only values which are integer multiples of 1/16 US$. Since price changes of less than $-3/16$$ and more than

day		time	min. price ($)		max. price($)	
Jan 9	(Tuesday)	09.50am - 03.40pm	91	7/16	95	12/16
Jan 10	(Wednesday)	09.50am - 03.40pm	92	5/16	94	15/16
Jan 11	(Thursday)	09.50am - 03.40pm	91	4/16	94	4/16
Jan 16	(Tuesday)	09.50am - 03.40pm	91	13/16	93	11/16
Jan 17	(Wednesday)	09.50am - 03.40pm	94	13/16	97	12/16
Jan 18	(Thursday)	09.50am - 03.40pm	104	3/16	110	0/16
Jan 23	(Tuesday)	09.50am - 03.40pm	107	10/16	109	7/16
Jan 24	(Wednesday)	09.50am - 03.40pm	108	15/16	111	7/16
Jan 25	(Thursday)	09.50am - 03.40pm	109	8/16	111	2/16

Table 7.1: Periods contained in the IBM data set and minimal and maximal prices of IBM stock in each period.

price diff. ($)	$\leq$ -3/16	-2/16	-1/16	0	1/16	2/16	$\geq$ 3/16
response y_t	1	2	3	4	5	6	7
frequency	151	1053	4886	10333	5222	860	184

Table 7.2: Price differences and corresponding response categories together with observed frequencies.

+3/16\$ hardly occur, we deal with them like price changes of $-3/16$\$ and $+3/16$\$, respectively. Therefore, we only observe seven different price changes. We associate these price changes to the categories $1, \ldots, 7$ in a natural way, as can be seen from Table 7.2. Adding up the frequencies in Table 7.2, we see that we have a total of 22689 observations.

The data is taken from the TAQ2 database of the NYSE, which contains the covariates TIMEDIFF (the time which elapses between two subsequent transactions in seconds) and SIZE (the volume of the transaction). From the application of the autoregressive ordered probit model to another IBM data set in Chapter 5 we know that these covariates have an impact on how large a price change is. Therefore we use the same transformations of TIMEDIFF and SIZE as in Chapter 5 for the covariate vector $\boldsymbol{z}_t$ in the log-volatility equation since a high (log-) volatility also can lead to larger price changes. However, we recall that in Section 6.1 we fixed the parameter μ to -0.6 to get non-extreme parameter estimates. Therefore we now center the covariate vector $\boldsymbol{z}_t$ at $\mathbf{0}$ since otherwise the mean of the log-volatilities would no longer be -0.6 but could become much larger or smaller. With TIMEDIFF_t denoting the time which elapses between the transaction at time $t-1$

	TIMEDIFF (seconds)	$z_{\cdot 1}$	SIZE (stocks)	$z_{\cdot 2}$
min	0	-1.9563	100	-2.1339
avg	8.3357	0.0000	2331	0.0000
max	116	2.8059	180000	5.3616

Table 7.3: Minimum, average, and maximum for the two original covariates and their transformations $z_{\cdot 1}$ and $z_{\cdot 2}$.

and the transaction at time t, we have

$$z_{t1} := \log(\text{TIMEDIFF}_t + 1) - \frac{1}{22689} \sum_{k=1}^{22689} [\log(\text{TIMEDIFF}_k + 1)] .$$

By adding 1 to TIMEDIFF_t we avoid the value $\log(0)$. In the same way we center now the second component of $\boldsymbol{z}_t$. In particular, with SIZE_t denoting the transaction volume of the transaction at time t, we have

$$z_{t2} := \log(\text{SIZE}_t) - \frac{1}{22689} \sum_{k=1}^{22689} [\log(\text{SIZE}_k)] .$$

Table 7.3 gives some summary information about these two covariates for the log-volatility equation.

Considering the response one can observe that often a positive price jump is followed by a negative one and vice versa (cf. Figure 7.1). This can be taken into account by using the lagged response as covariate in the model Equation (6.2) since the covariates there have an impact on the mean of the latent variables y_t^*. Therefore, also recalling that $\boldsymbol{x}_t'\boldsymbol{\beta}$ must contain an intercept, we use $\boldsymbol{x}_t := (1, y_{t-1})'$ as covariate vector for Equation (6.2). We note that exploratory analyses for TIMEDIFF and SIZE show that often higher values of these covariates come along with higher price changes, but partly upwards and partly downwards. Therefore we do not expect an impact of these covariates on the mean of the latent variables y_t^*.

Before we fit in Section 7.3 the OSV model to the data, we show that for our data it is reasonable to take the price changes instead of the commonly used log returns as response.

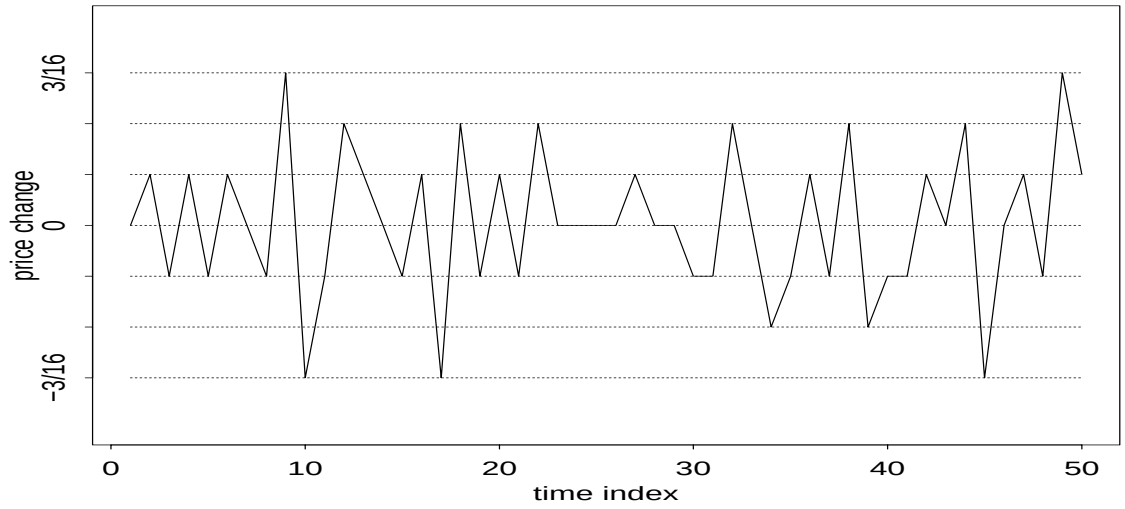

Figure 7.1: Price changes of IBM stock at January 9, 2001 (first 50 transactions after 09:50am).

7.2 Log returns and signed price changes

As mentioned before, in many papers not the price changes itself, but the log returns are modeled. Here we show that at least for our data set it is reasonable to consider the signed priced changes.

First we introduce some notations. Let p_t denote the transaction price at time t. The log return is then defined as

$$r_t := \log\left(\frac{p_t}{p_{t-1}}\right).$$

With d_t denoting the signed price change at time t, i.e. $d_t := p_t - p_{t-1}$, it is evident that

$$r_t = \log\left(\frac{p_{t-1} + d_t}{p_{t-1}}\right) = \log\left(1 + \frac{d_t}{p_{t-1}}\right). \tag{7.1}$$

One important argument for considering the log return is that the log return takes the actual value of the stock into account, in contrast to the price difference. We illustrate this in a short example. Consider two cases (1) and (2) with stock prices $p_{t-1}^{(1)} = 10$ and $p_{t-1}^{(2)} = 100$ at time $t-1$, respectively. Intuitively, one might model the stock prices in such a way that (1) a rise from 10 to 10.5 dollars is as probable as (2) a rise from 100 to 105 dollars since both equals a rise of 5%. This can adequately be modeled by considering the log returns, since

$$r_t^{(1)} = \log(1 + 0.5/10) = \log(1 + 5/100)) = r_t^{(2)}.$$

d_t in dollars	r_t for $p_{t-1}^{\min} = 91.2500$	r_t for $p_{t-1}^{\max} = 111.4375$
-3/16	-0.002057	-0.001684
-2/16	-0.001371	-0.001122
-1/16	-0.000685	-0.000561
0	0.000000	0.000000
1/16	0.000685	0.000561
2/16	0.001369	0.001121
3/16	0.002053	0.001681

Table 7.4: Price changes d_t and corresponding log returns r_t for $p_{t-1}^{\min} = 91.2500$ and $p_{t-1}^{\max} = 111.4375$.

Therefore, if one models the log returns, one models the **relative** price changes. In contrast, if one considers the signed price differences, one gets

$$d_t^{(1)} = 10.5 - 10 \neq 105 - 100 = d_t^{(2)}.$$

But $d_t^{(1)} = d_t^{(2)}$ when in both cases (1) and (2) the price rises by the same amount, for example 50 cents. Therefore, a model for the signed price changes takes only the price changes themselves into account, but not the price at time $t-1$.

Of course, the larger the difference between the prices $p_{t-1}^{(1)}$ and $p_{t-1}^{(2)}$ becomes the larger is the difference between modeling the log returns and modeling the signed price changes.

As can be seen from Table 7.1 the stock prices in our IBM data set lie between $p_{t-1}^{\min} := 91.2500$ and $p_{t-1}^{\max} := 111.4375$ dollars. Using Equation (7.1) we can compute the log returns r_t which correspond to the price changes d_t when the actual price p_{t-1} has these (for our data set) extreme values. The result is shown in Table 7.4. From Equation (7.1) we see that, for known d_t, the log return r_t is a monotone function of p_{t-1}. We conclude that for known d_t the log returns in our data set lie in the intervals given by the second and third column of Table 7.4. These intervals are illustrated in Figure 7.2 together with the corresponding price changes.

We see that the log returns occur in clusters which are clearly separated from each other. For example, log returns between 0.00001 and 0.00055 do not occur. Therefore we doubt whether it makes sense to model the log returns of this data set by a continuous distribution, as would be done by applying common SV models, and prefer the application of the OSV model, which, of course, does not distinguish between the different log returns in each interval.

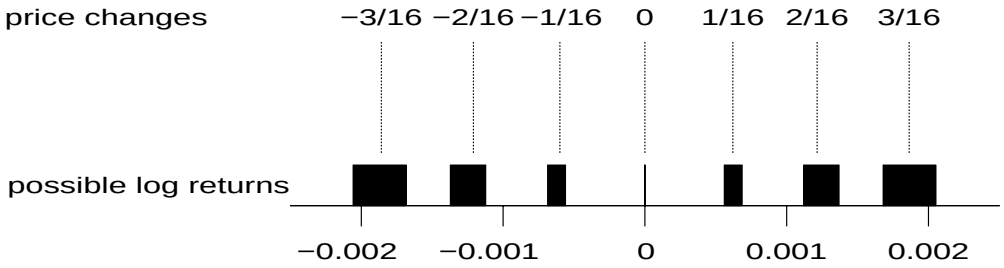

Figure 7.2: Price changes and the corresponding log returns which can occur in the IBM data set.

Finally we note that the rise from 91.2500 to 111.4375 dollars corresponds to a rise by 22 percent. This is a large gain for a period of 16 days. However, especially if one were to consider longer time periods, there can be gains or losses of for example 50 percent or even more. It is obvious, that the log return clusters in Figure 7.2 become narrower when the range of the stock prices in the data set decreases. For small ranges only few log returns will occur, and after appropriate rescaling they will all be very close to the corresponding integer multiple of 1/16. Hence for data sets with small price range the difference between modeling the price differences and modeling the log returns is very small. However, the difference between these modeling approaches increases with the range of the stock prices of the data set to be considered.

7.3 Model estimation

We now fit the OSV model to the data. First we specify the hyperparameters for the prior distributions, then we compute the parameter estimates and investigate what these estimates tell us about the structure of the IBM data set. Finally we have a look at the (log-) volatility estimates.

Parameter	Prior distribution
$\boldsymbol{c}$	Uniform on $\{(c_2, \ldots, c_6) \,\vert\, 0 < c_2 < \ldots < c_6 < \infty\}$
$\boldsymbol{\beta}$	$N_2(\mathbf{0}, \text{diag}(10, 10))$
h_0^*	Dirac(-0.6)
α_1	Uniform on $\{-100 < \alpha_1 < 100\}$
α_2	Uniform on $\{-100 < \alpha_2 < 100\}$
ϕ	Uniform on $\{-1 < \phi < 1\}$
σ	Uniform on $\{0 < \sigma < 10\}$

Table 7.5: Prior distributions for the cutpoints $\boldsymbol{c}$, for the regression coefficient vector $\boldsymbol{\beta}$, for the latent variable h_0^*, for the regression coefficients α_j, and for the parameters ϕ and σ in the log-volatility equation.

7.3.1 Specification of the hyperparameters

We have to specify the hyperparameters C (maximum for cutpoints), $\boldsymbol{b}_0$ (mean for $\boldsymbol{\beta}$-prior), B_0 (covariance matrix for $\boldsymbol{\beta}$-prior), C_α (bound for noninformative α_j-prior, $j = 1, 2$), and C_σ (upper bound for noninformative σ-prior). First, to be able to apply the grouped move step in our GM-MGMC algorithm, we must set $\boldsymbol{b}_0 = \mathbf{0}$ (cf. Section 6.3.1). Moreover, we set $C := \infty$. Of course, we could also choose a finite value, but then rejection sampling would be required in the grouped move step. Since we use the covariate vector $\boldsymbol{x}_t = (1, y_{t-1})'$, the values of this vector lie between 1 and 7 for all t. Therefore we do not expect extreme values for the coefficients β_j, $j = 0, 1$, and the choice $B_0 = \text{diag}(10, 10)$ seems to be appropriate. The values of the components of the covariate vector $\boldsymbol{z}_t$ lie between -2.1339 and 5.3616 for all t (cf. Table 7.3). Since this covariate vector has an impact on the log-volatility, we do not expect extreme values for the coefficients α_j, $j = 1, 2$. The choice of $C_\alpha = 100$ therefore seems to be sufficiently high. Since we do not expect a dominating impact of the error term $\sigma\eta_t^*$ on the log-volatility, the value 10 for C_σ could be adequate. Table 7.5 summarizes the chosen prior distributions.

We checked out by using other hyperparameter values that the posterior estimates are not very prior-sensitive. For example, the less informative $\boldsymbol{\beta}$-prior $N_2(\mathbf{0}, \text{diag}(10^4, 10^4))$ leads to nearly identical estimates. This also can be expected because of the large number of observations.

	estimate	std.err.	90% cred.int.		estimate	std.err.	90% cred.int.
ϕ	0.9061	0.0119	(0.8853,0.9248)	c_2	0.9332	0.0137	(0.9102,0.9554)
σ	0.2230	0.0194	(0.1922,0.2570)	c_3	1.8248	0.0208	(1.7894,1.8587)
β_0	3.5152	0.0463	(3.4402,3.5908)	c_4	2.7609	0.0310	(2.7087,2.8113)
β_1	-0.3073	0.0070	(-0.3188,-0.2962)	c_5	3.6893	0.0443	(3.6140,3.7609)
α_1	0.2599	0.0182	(0.2298,0.2912)	c_6	4.5562	0.0636	(4.4500,4.6623)
α_2	0.1511	0.0090	(0.1363,0.1665)				

Table 7.6: Posterior mean estimates and corresponding estimated standard deviations and 90% posterior credible intervals for parameters in OSV model.

7.3.2 Parameter estimates and conclusions

We now run the GM-MGMC sampler from Section 6.3.1 for 4000 iterations and discard the first 1000 iterations for burn-in. From the simulations in Sections 6.3.2 and 6.4 we know that this leads to very accurate estimates. The results are summarized in Table 7.6. It shows the posterior mean estimates for the parameters ϕ and σ in the log-volatility equation, for the regression coefficients β_0, β_1, α_1, and α_2, and for the cutpoints, using the iterations 1001 to 4000 of the GM-MGMC sampler together with their corresponding estimated standard deviations and 90% credible intervals.

Since the 90% credible intervals for β_0, β_1, α_1, and α_2 are far away from zero, we conclude that the intercept, the lagged observation y_{t-1}, $\log(\text{TIMEDIFF}_t + 1)$, and $\log(\text{SIZE}_t)$ all have a significant impact on the new observation y_t. Also the credible intervals for the autoregressive parameter ϕ and the standard deviation σ are far away from zero. The estimate 0.9061 for ϕ shows the high dependence of the log-volatility h_t^* on the previous log-volatility h_{t-1}^*, but is still away from the nonstationary case $\phi = 1$.

Figure 7.3 shows the estimated posterior marginal densities for the parameters in the OSV model. The densities are unimodal and quite symmetric, so that the posterior mean estimates also represent high density points. Figure 7.4 shows the estimated autocorrelations in the chains produced by the GM-MGMC sampler after the burn-in period of 1000 iterations. The autocorrelations decline quite fast for all parameters. This justifies that no subsampling is required to estimate the standard errors of the estimates.

The negative sign of the estimate -0.3073 for β_1 and the positive signs of the estimates 0.2599 for α_1 and 0.1511 for α_2 lead to the following qualitative conclusions:

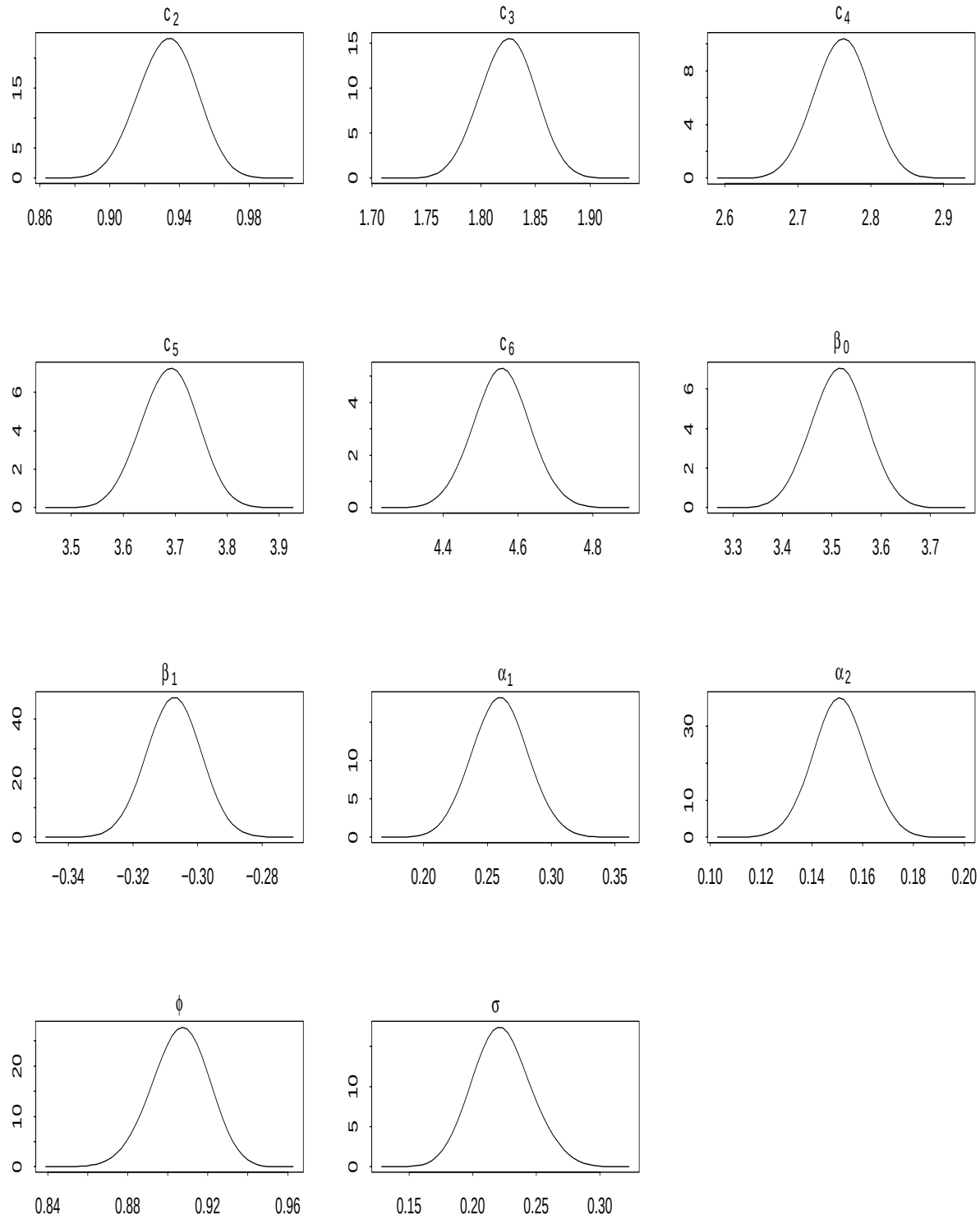

Figure 7.3: Estimated posterior marginal densities for parameters of OSV model.

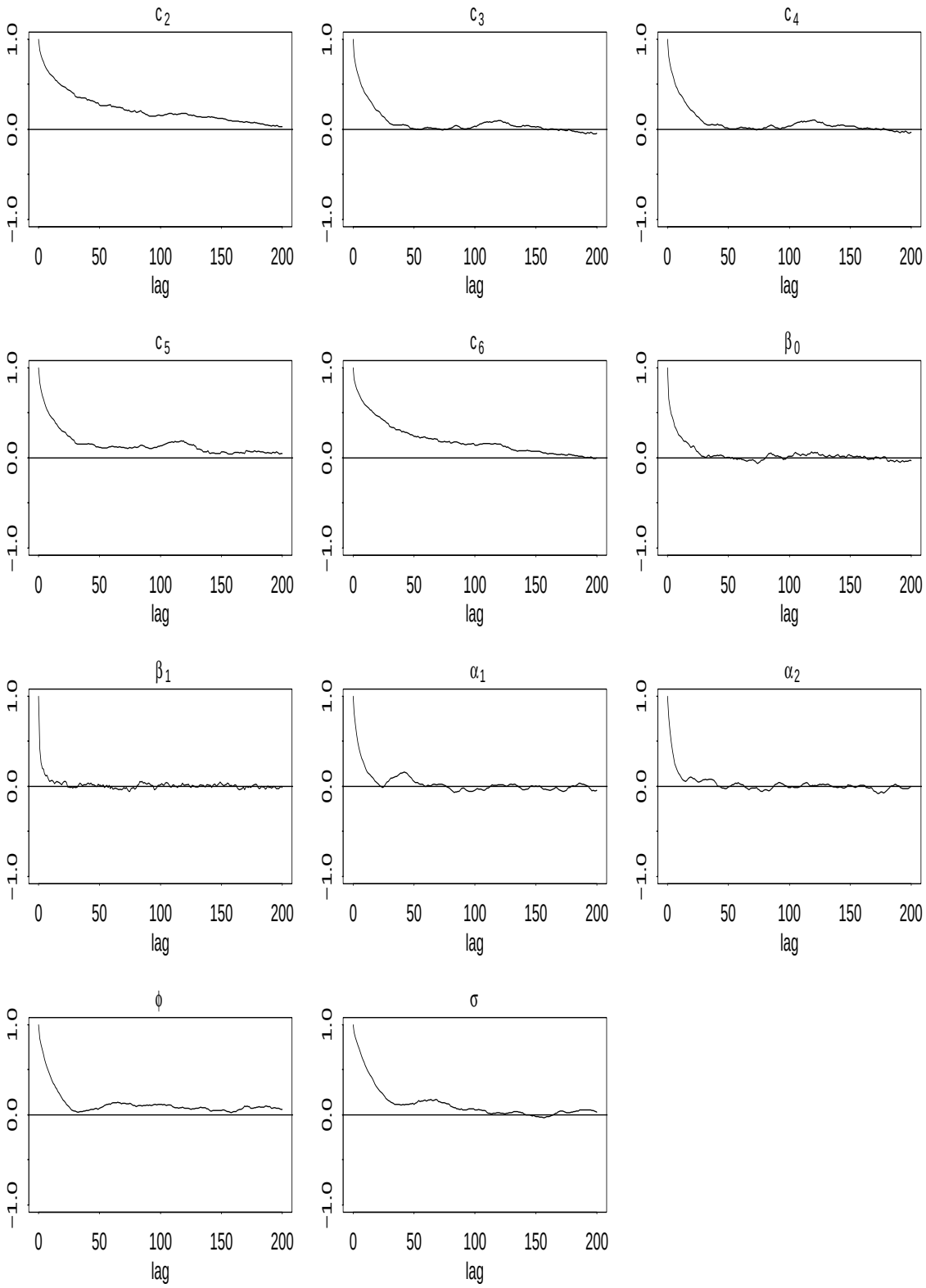

Figure 7.4: Autocorrelations of chains produced by the GM-MGMC sampler for parameters of OSV model.

- Positive price changes are often followed by negative ones and vice versa (this confirms what we have seen directly in Figure 7.1).

- The more time elapses between two subsequent transactions, the higher the (log-) volatility is, or, equivalently, the more time elapses, the higher the probability for a big price change is.

- The more stocks are traded, the higher the (log-) volatility is, or, equivalently, the more stocks are traded, the higher the probability for a big price change is.

These results agree with many publications about theoretical results for the price change process. Diamond and Verrecchia (1987) point out, that periods without transactions can be considered as a hint for the existence of bad news. Because of the prohibition of short-selling, i.e. of selling a stock without owning it, many investors cannot use bad information by selling. Therefore, longer periods between consecutive transactions usually lead to a higher volatility of the price change process. Following Easley and O'Hara (1987), well informed investors usually buy or sell large amounts of stocks in each transaction to take maximal advantage of their informations. Therefore, noninformed market participants associate large transaction volumes with existence of new information and trade themselves. Hence for large transaction volumes one can expect higher volatilities. The same dependence between market informations, transaction volumes, and expected volatility is derived by Tauchen and Pitts (1983).

Using the posterior mean estimates for α_1 and α_2 we can compare the impacts of TIMEDIFF and SIZE on the log-volatilities. From Table 7.3 we know that the time difference always lies between 0 and 116 seconds. For each of these 117 values we compute the corresponding transformation $z_{\cdot 1}$ (cf. Section 7.1) and multiply the resulting value by the posterior mean estimate 0.2599 for α_1. The same is done for the transaction volume with range 100 to 180000 stocks. Here the transformed values $z_{\cdot 2}$ are multiplied by the estimate 0.1511 for α_2.

The result can be seen in Figure 7.5. For the extreme values TIMEDIFF $= 0$ and TIMEDIFF $= 116$, the estimated impacts are about -0.51 and 0.73, respectively. The corresponding estimates for the covariate SIZE are -0.32 and 0.81. Since $0.73 + 0.51 = 1.24$ and $0.81 + 0.32 = 1.13$, we conclude that the covariate TIMEDIFF affects the log-volatility slightly more than the covariate SIZE. Moreover, the impact of both covariates is quite large if one takes the posterior mean estimate 0.2230 for σ into account.

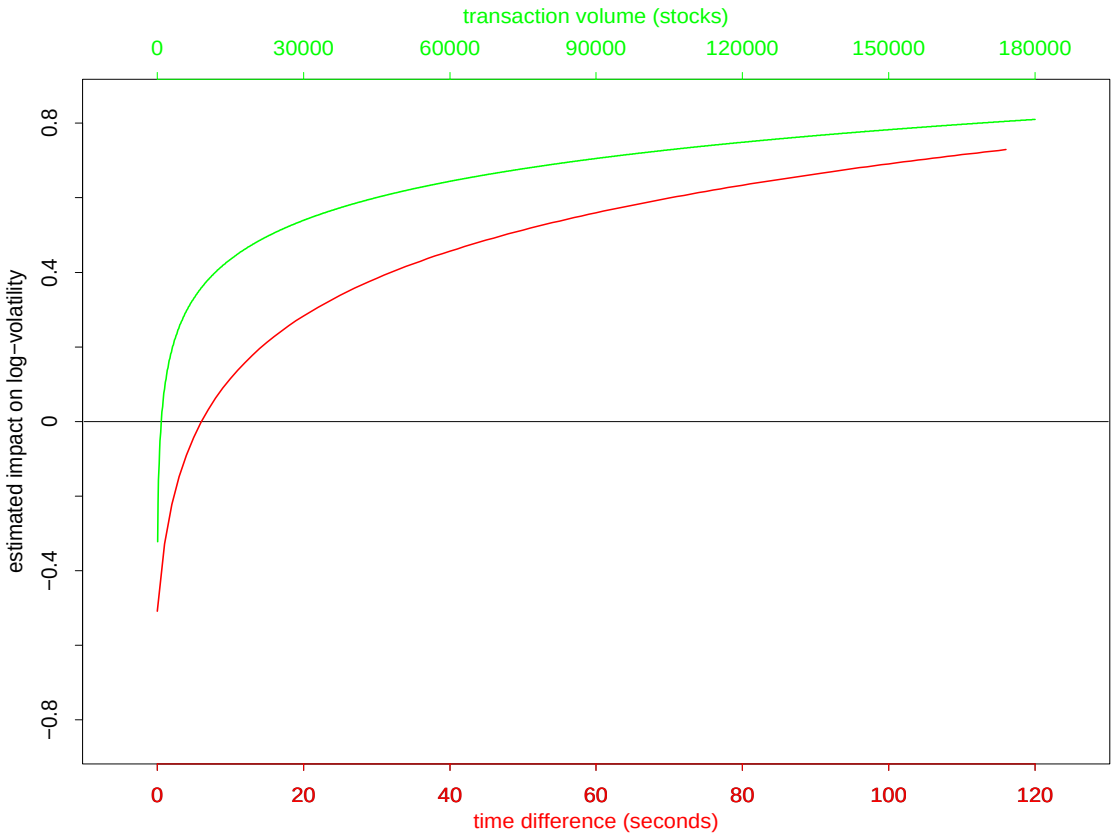

Figure 7.5: Estimated impacts of TIMEDIFF and SIZE on the log-volatilities h_t^*. The estimated joint impact of both covariates is given by adding these individual impacts.

7.3.3 Volatility estimates

We are now interested in detecting periods with higher volatility and periods with lower volatility. From Section 6.1 we know that in the OSV model the log-volatility is not determined uniquely until the additive constant μ is fixed. For computational reasons we fixed this parameter to $= -0.6$, but in this context it may be more intuitive to consider log-volatilities with mean zero. Therefore we now define the **normalized volatility at time** t by

$$v_t^n := \exp\{h_t^* - \mu\} \; = \; \exp\{h_t^* + 0.6\}$$

since the covariates in the log-volatility equation were also centered at zero. In each iteration i the GM-MGMC sampler produces estimates $\hat{h}_{t,i}^*$ of h_t^*, $t = 1, \ldots, 22689$, which, of course, can be used in the following way to get estimates $\hat{v}_t^n$ for the normalized volatilities v_t:

$$\hat{v}_t^n := \frac{1}{3000} \sum_{i=1001}^{4000} \exp\left\{\hat{h}_{t,i}^* + 0.6\right\}.$$

The implementation of the GM-MGMC sampler for the OSV model which is provided in Appendix C also computes the estimates $\hat{v}_t^n$, $t = 1, \ldots, 22689$. Figure 7.6 shows the IBM stock prices and the estimated normalized volatilities for $t = 1, \ldots, 6000$, which corrsponds to the period, January 9, 09:50:00am to January 11, 11:00:36am.

We note that the large price jumps from Tuesday to Wednesday and Wednesday to Thursday (time indices $2772 \to 2773$ and $5277 \to 5278$) are not surprising since we discarded all transactions after 03:40pm and before 09:50am. However, for the computation of the first price difference and the covariate TIMEDIFF_t on a new day we used always the last transaction before 09:50am on this new day.

In Figure 7.7 we zoom into Figure 7.6 from time index 500 to 680. This period also contains the highest volatility peak at observation 602. One can see that the estimated volatilities move rapidly especially when the volatility is high.

To detect not only single peaks of the volatility but also short periods with a high volatility one can compute **moving averages of the estimated volatilities**, defined by

$$\hat{v}_t^{n,[k]} := \frac{1}{2k+1} \sum_{j=t-k}^{t+k} \hat{v}_j^n$$

for $k \in \mathbb{N}_0$. For fixed t this is the average of $2k+1$ values from time index $t-k$ to time index $t+k$. Of course, the higher the parameter k is chosen, the smoother the resulting moving average will be. We computed the values $\hat{v}_t^{n,[k]}$ for $k = 2$ and $t = 503, \ldots, 678$. The result is the thick blue curve in Figure 7.7. Now we can easily detect small periods of higher volatility by checking where the moving average is greater than a chosen limit. In Figure 7.7 we chose the limit 2.1. The periods where the moving average is greater than 2.1 are indicated by the vertical dashed lines. Now considering the corresponding stock prices we see how accurate the volatility estimates are: Exactly during these periods we observe higher price fluctuations of the IBM stock.

Finally in Figure 7.8 we illustrate the impact of the covariates TIMEDIFF and SIZE on the volatility. The blue curve shows the estimated normalized volatilities for the OSV model with these two covariates, whereas the orange curve shows the estimated normalized volatilities for the same OSV model without covariates in the log-volatility equation. The estimated volatilities differ clearly: The curve for the model without covariates is smoother, whereas the model with TIMEDIFF and SIZE detects a much higher variability of the volatilities. We can take this as an additional hint for the significance of the used covariates.

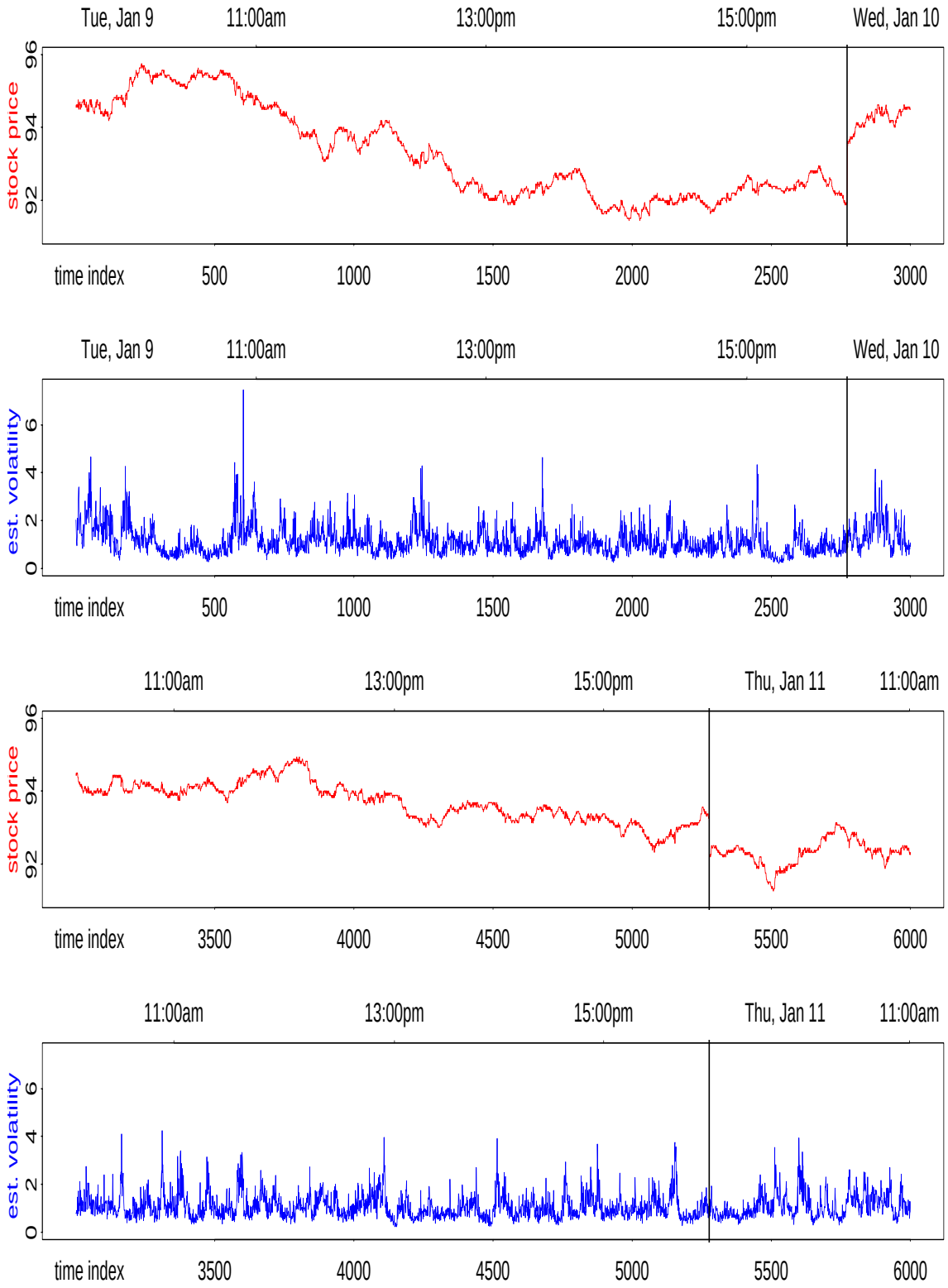

Figure 7.6: IBM stock prices and estimated normalized volatilities on January 9 to 11, 2001.

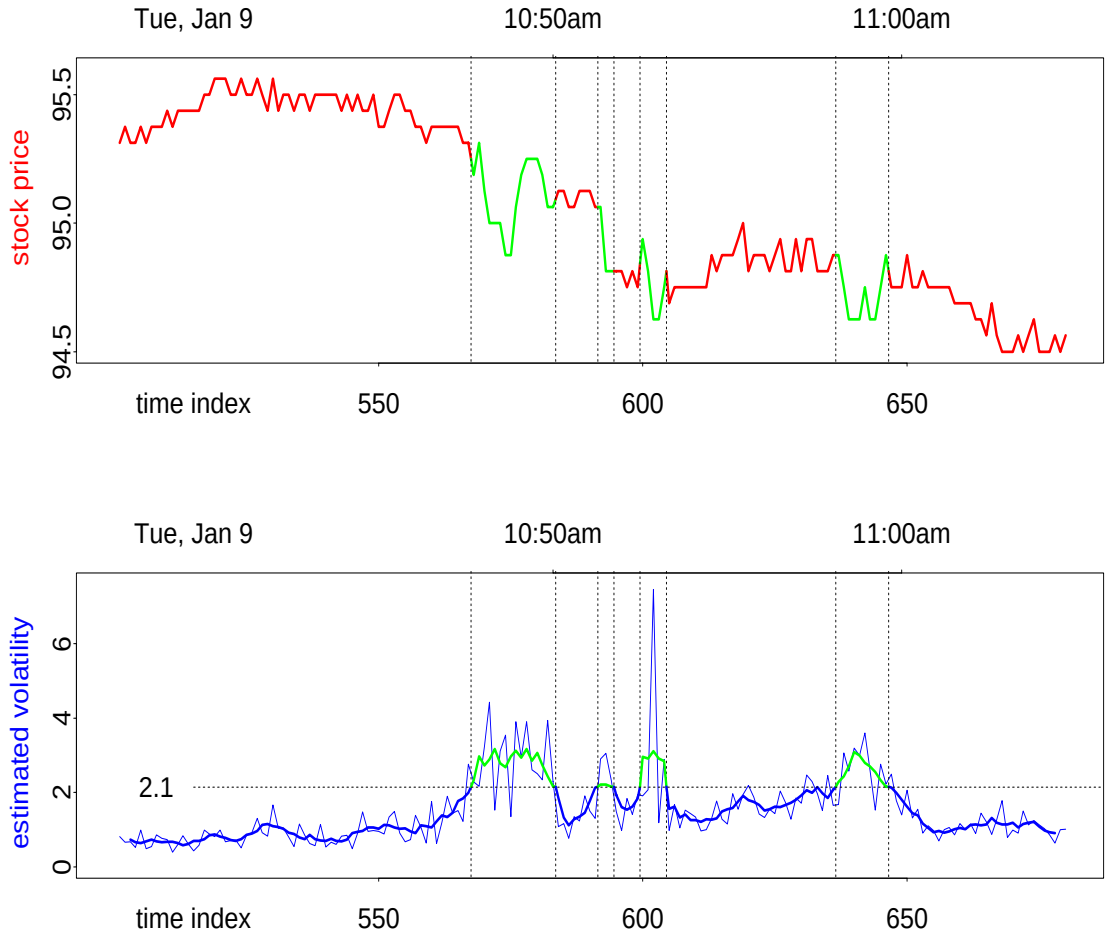

Figure 7.7: IBM stock prices (above) and estimated normalized volatilities v_t^n (below, thin blue curve) between 10:39:10am and 11:03:17am on January 9, 2001. The thick curve (below) shows the moving average $v_t^{n,[2]}$ of the normalized volatilities. It is marked green, when $v_t^{n,[2]} > 2.1$, that is, when a period of high volatility occurs. In these periods also the stock price curve (above) is marked green.

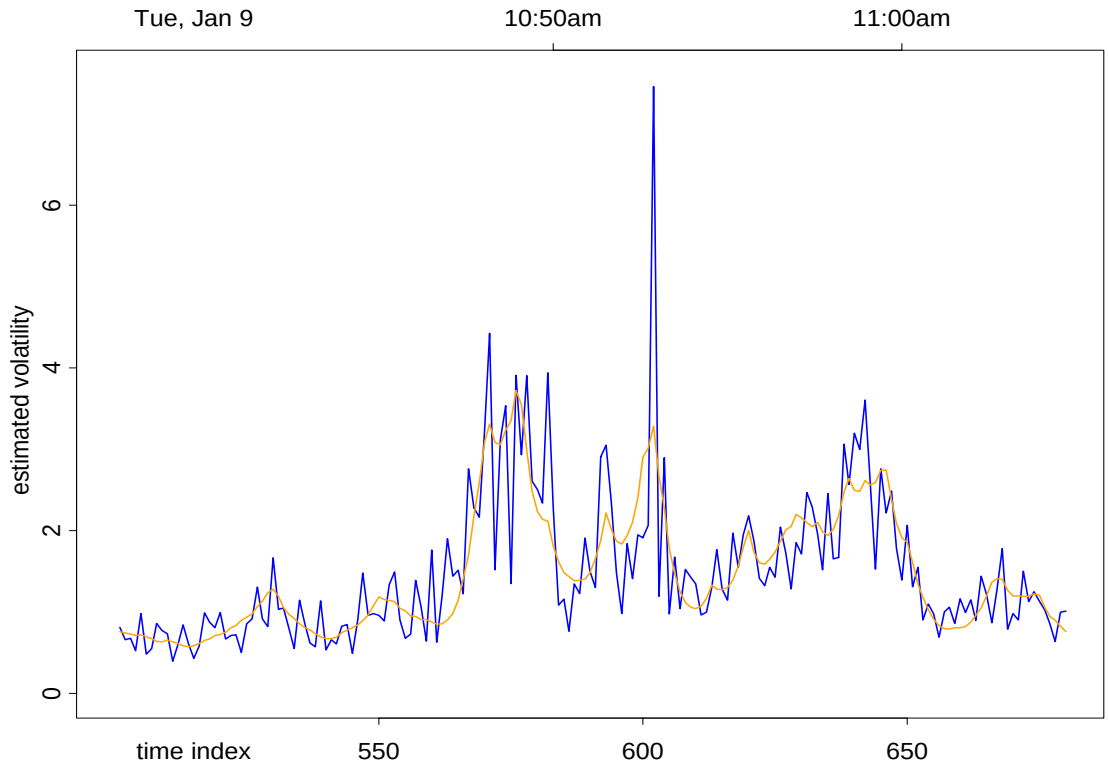

Figure 7.8: Estimated normalized volatilities v_t^n between 10:39:10am and 11:03:17am on January 9, 2001. Blue curve: OSV model with covariates TIMEDIFF and SIZE in the log-volatility equation. Orange curve: OSV model without covariates in the log-volatility equation.

Chapter 8

Ordinal-Response Stochastic Volatility Model with Student-t Errors

After fitting the OSV model to IBM data, we now try to answer the question whether it is really adequate to use normally distributed errors in Equation (6.2). As often in modeling financial time series it may be more accurate to allow for Student-t distributed errors, since the Student-t distribution has heavier tails. Therefore we introduce in Section 8.1 an extension of the OSV model, the **OSVt model**. In Section 8.2 we derive the MCMC updates for fitting the OSVt model to a data set, and investigate in Section 8.3 how the GM step of the GM-MGMC sampler for the OSV model must be modified in the OSVt case. After a simulation study in Section 8.4 we answer in Section 8.5 the question whether the OSVt model should be used for such data or whether the OSV model is sufficiently accurate.

We note that Chib et al. (2002) use the same technique which is applied in the following to extend their SV model to a SVt model with t-distributed errors. Unfortunately, they only mention that they use a 'tailored proposal density' for the MH-update of the unknown degrees of freedom, but do not specify this density precisely. This, however, is the most difficult and important task for the extension of the SV to the SVt model as well as for the extension of the OSV to the OSVt model, as can be seen from Section 8.2.

8.1 The OSVt model

First we recall the three defining equations for the OSV model:

$$\begin{aligned} y_t = k \quad \Leftrightarrow \quad y_t^* &\in [c_{k-1}, c_k)\,, && (8.1)\\ y_t^* &= \boldsymbol{x}_t'\boldsymbol{\beta} + \exp(h_t^*/2)\varepsilon_t^*, && (8.2)\\ h_t^* &= \mu + \boldsymbol{z}_t'\boldsymbol{\alpha} + \phi(h_{t-1}^* - \mu - \boldsymbol{z}_{t-1}'\boldsymbol{\alpha}) + \sigma\eta_t^*, && (8.3) \end{aligned}$$

where, in particular, $\varepsilon_t^* \overset{\text{i.i.d.}}{\sim} N(0,1)$ independent of $\eta_t^* \overset{\text{i.i.d.}}{\sim} N(0,1)$. Now we replace the normal distribution of the errors ε_t^* in Equation (8.2) by a Student-t distribution with ν degrees of freedom. Here we can take advantage from the well-known decomposition of a t-distributed random variable in a product of a normally and a Gamma-distributed random variable, which is also used for the generation of t-distributed random numbers (cf. Robert and Casella (2000)):

$$\varepsilon_t^* \sim N(0,1) \quad \text{independent of} \quad \lambda_t^* \sim \Gamma\left(\frac{\nu}{2}, \frac{\nu}{2}\right) \quad \Longrightarrow \quad \lambda_t^{*-1/2} \cdot \varepsilon_t^* \sim t_\nu(0,1). \tag{8.4}$$

Here $t_\nu(a, b^2)$ denotes the general form of the non-central t-distribution with location parameter a and scale parameter b. Its density is given by

$$f_{t_\nu(a,b^2)}(x) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)\sqrt{\nu\pi}\,b}\left(1 + \frac{1}{\nu}\left(\frac{x-a}{b}\right)^2\right)^{-\frac{\nu+1}{2}}, \qquad x \in \mathbb{R}. \tag{8.5}$$

Using the decomposition in (8.4) we define the **Ordinal-response Stochastic Volatility Model with t-distributed errors (OSVt Model)** by the following three equations:

$$\begin{aligned} y_t = k \quad \Leftrightarrow \quad y_t^* &\in [c_{k-1}, c_k)\,, && (8.6)\\ y_t^* &= \boldsymbol{x}_t'\boldsymbol{\beta} + \exp(h_t^*/2)\lambda_t^{*-1/2}\varepsilon_t^*, && (8.7)\\ h_t^* &= \mu + \boldsymbol{z}_t'\boldsymbol{\alpha} + \phi(h_{t-1}^* - \mu - \boldsymbol{z}_{t-1}'\boldsymbol{\alpha}) + \sigma\eta_t^*, && (8.8) \end{aligned}$$

where $\varepsilon_t^* \overset{\text{i.i.d.}}{\sim} N(0,1)$ independent of $\lambda_t^* \overset{\text{i.i.d.}}{\sim} \Gamma(\nu/2, \nu/2)$. In all other respects we assume the same conditions as for the OSV model.

In addition to the parameters and variables to estimate in the OSV model, now the variables λ_t^* and the parameter ν have to be estimated. From the non-central t-density (8.5) it follows directly that $a + bY \sim t_\nu(a, b^2)$ if $Y \sim t_\nu(0,1)$. Therefore, given $\boldsymbol{\beta}$, h_t^*, and ν, the latent variables y_t^* are non-central t-distributed. In particular,

$$f(y_t^*|\boldsymbol{\beta}, h_t^*, \nu) = t_\nu(y_t^*|\boldsymbol{x}_t'\boldsymbol{\beta}, \exp(h_t^*)). \tag{8.9}$$

However, if in addition λ_t^* is given, y_t^* is again normally distributed with mean $\boldsymbol{x}_t'\boldsymbol{\beta}$ and variance $\exp(h_t^*)\lambda_t^{*-1}$.

For notational convenience we define

$$\begin{aligned} \boldsymbol{\lambda}^* &:= (\lambda_1^*, \ldots, \lambda_T^*) \\ \text{and} \qquad \boldsymbol{\lambda}_{-t}^* &:= (\lambda_1^*, \ldots, \lambda_{t-1}^*, \lambda_{t+1}^*, \ldots, \lambda_T^*). \end{aligned}$$

8.2 Hybrid MCMC updates for OSVt model

Here we develop an Hybrid MCMC sampler for the OSVt model. Since the OSVt model differs only slightly from the OSV model, the derivation of the updates is completely analogous to Section 6.2. Mostly, one has only to replace the term $\exp(h_t^*)$ by $\exp(h_t^*)\lambda_t^{*-1}$. Therefore we just provide the updates without derivations. Of course, in addition we now need an update for the degrees of freedom ν of the t-distribution, and updates for the variables λ_t^*, $t = 1, \ldots, T$. For ν we assume a noninformative prior on the set $\{1, \ldots, 127\}$. This choice will become clear later. Following the model definition, the variables λ_t^* are assumed a priori independent of each other and $\Gamma(\nu/2, \nu/2)$-distributed. Moreover, we assume ν and λ_t^*, $t = 1, \ldots, T$, a priori independent of all other parameters and variables.

The regression parameter update for $\boldsymbol{\beta}$ consists here of sampling from $N_{p+1}(\boldsymbol{b}, B)$ where

$$\begin{aligned} B &= \left(\sum_{t=1}^{T} \frac{\boldsymbol{x}_t \boldsymbol{x}_t'}{\exp(h_t^*)\lambda_t^{*-1}} + B_0^{-1} \right)^{-1} \\ \text{and} \qquad \boldsymbol{b} &= B \left(\sum_{t=1}^{T} \frac{\boldsymbol{x}_t y_t^*}{\exp(h_t^*)\lambda_t^{*-1}} + B_0^{-1}\boldsymbol{b}_0 \right). \end{aligned}$$

For the latent variable update of y_t^* one has to draw from the univariate truncated normal distribution

$$N_{[c_{y_t-1}, c_{y_t})} \left(y_t^* \mid \boldsymbol{x}_t'\boldsymbol{\beta}, \exp(h_t^*)\lambda_t^{*-1} \right).$$

The cutpoint parameter update is not affected by the variables λ_t^*.

Again one can take advantage from a state space approximation of Equations (8.7) and (8.8). However, instead of using $\tilde{y}_t^* = \log(y_t^* - \boldsymbol{x}_t'\boldsymbol{\beta})^2$ as in the OSV model, here one must compute

$$\tilde{y}_t^* := \log(y_t^* - \boldsymbol{x}_t'\boldsymbol{\beta})^2 + \log \lambda_t^*, \qquad t = 1, \ldots, T.$$

Using this definition of $\tilde{y}_t^*$ the updates of the mixture indices s_t, the complete Metropolis-Hastings step for the joint update of $\boldsymbol{\alpha}$, ϕ, and σ, and the $\boldsymbol{h}^*$-update can be done exactly as described in Sections 6.2.6 and 6.2.7.

Next we consider the updates which must be made in addition to the OSV case. Again defining $\boldsymbol{\theta} := (\boldsymbol{\alpha}, \phi, \sigma)$, one can update ν and λ_t^*, $t = 1, \ldots, T$, by first drawing ν from

$$f(\nu \,|\, \boldsymbol{y}, \boldsymbol{c}, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*, \boldsymbol{\theta})$$

and then sampling λ_t^*, $t = 1, \ldots, T$, from

$$f(\lambda_t^* \,|\, \boldsymbol{y}, \boldsymbol{c}, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*, \boldsymbol{\lambda}_{-t}, \nu, \boldsymbol{\theta}).$$

For the update of ν we have

$$\begin{aligned} f(\nu \,|\, \boldsymbol{y}, \boldsymbol{c}, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*, \boldsymbol{\theta}) &= f(\nu \,|\, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*) \\ &\propto f(\nu, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*) \\ &= \left[\prod_{t=1}^{T} f(y_t^* | \boldsymbol{\beta}, h_t^*, \nu)\right] \pi(\nu). \end{aligned}$$

Now using Equation (8.9) and the noninformative prior of ν on the set $\{1, \ldots, 127\}$, we derive that

$$f(\nu \,|\, \boldsymbol{y}, \boldsymbol{c}, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*, \boldsymbol{\theta}) \propto \left[\prod_{t=1}^{T} t_\nu(y_t^* | \boldsymbol{x}_t'\boldsymbol{\beta}, \exp(h_t^*))\right] \mathbb{1}_{\{1,\ldots,127\}}(\nu). \tag{8.10}$$

We emphasize that this target density is discrete with support $\{1, \ldots, 127\}$. Since it is no standard distribution, we draw ν by a Metropolis-Hastings step. The difficulty in developing an MH step is the choice of an appropriate proposal density. It should be from a parametric distribution family where the mode and the variance can be easily adapted to the current data situation to achieve a good approximation of the target density. For this, however, we must determine the global maximum of the (unnormalized) target density given on the right-hand side of (8.10). This can be very time-expensive since we do not have any special information about this function. Here now we can take great advantage of the restriction to the support $\{1, \ldots, 127\}$, as is explained in the following.

When we assume that the target density $f(\,\cdot \,|\, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*)$ only has one global and no other local maximum, we can find

$$m := \arg \max_{\nu=1,\ldots,127} f(\nu \,|\, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*)$$

by evaluating $f(\cdot \,|\, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*)$ at only 13 points. At the beginning, one evaluates $f(\nu = 64 \,|\, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*)$. In the next step, the values $f(\nu = 32 \,|\, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*)$ and $f(\nu = 96 \,|\, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*)$ are computed. Since we assume that there exists only one maximum, we can now restrict the set where the maximum can be:

If $f(\nu = 32 \,\|\, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*) > f(\nu = 64 \,\|\, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*)$,	we know that $m < 64$.
If $f(\nu = 96 \,\|\, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*) > f(\nu = 64 \,\|\, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*)$,	we know that $m > 64$.
Else	we know that $32 < m < 96$.

Therefore by evaluating $f(\cdot \,|\, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*)$ at two additional points, we restrict the set from 127 to 63 elements. With another two points, one can restrict it to 31 elements, and so on. Therefore, m is found after the evaluation of $f(\cdot \,|\, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*)$ at 13 values of ν.

Moreover, since the t-distribution becomes more and more similar to the normal distribution as ν increases, the maximal value 127 for ν seems to be sufficiently large.

There exist several possibilities to choose a proposal with support $\{1, \ldots, 127\}$. For example, one can use the Binomial distribution family $\text{Bin}(126, p)$, $p \in [0, 1]$ with support $\{0, \ldots, 126\}$ and shift it by 1. However, this distribution family only has one free parameter so that one cannot adapt both the mode and the variance. Another possibility is to truncate discrete distributions with infinite support to the set $\{1, \ldots, 127\}$. Since the Poisson and the Geometric distributions are also parameterized by only one parameter, one may suggest the family of the Negative Binomial distributions $\text{NegBin}(\mu, \alpha)$ with density

$$f_{\text{NegBin}(\mu,\alpha)}(x) = \frac{\Gamma(x + \alpha^{-1})}{\Gamma(\alpha^{-1}) x!} \left(\frac{\mu}{\alpha^{-1} + \mu} \right)^x \left(\frac{1}{1 + \alpha\mu} \right)^{\alpha^{-1}} \tag{8.11}$$

$$\propto \frac{\Gamma(x + \alpha^{-1})}{x!} \left(\frac{\mu}{\alpha^{-1} + \mu} \right)^x . \tag{8.12}$$

This density has an infinite discrete support and allows for adapting the mean and the variance by the parameters μ and α. Moreover, the truncation of the Negative Binomial density to $\{1, \ldots, 127\}$ is no problem, since one can use rejection sampling for drawing and since it is not necessary to know the normalizing constant for the MH step. However, the evaluation of expression (8.12) requires the computation of the Gamma function at $x + \alpha^{-1}$ and of $(x!)$. Especially when x is large (e.g. $x = 100$) this may cause numerical problems without using special computation methods. Moreover, since (8.12) must be evaluated for 13 different values of x per iteration, the determination of the maximum will be very time-expensive.

For these reasons we use a proposal distribution $q_{a,b}$ which can be considered as a discretized and truncated version of the Gamma(a,b) distribution and which does not have the disadvantages of the Negative Binomial distribution. In particular, we define $q_{a,b}$ by its unnormalized version $q^u_{a,b}$ with

$$q^u_{a,b}(x) := \int_{x-0.5}^{x+0.5} f^u_{\Gamma(a,b)}(y)\,dy, \qquad x = 1,\ldots,127,$$

where $f^u_{\Gamma(a,b)}(y)$ denotes the value of the unnormalized Gamma(a,b)-density at y:

$$f^u_{\Gamma(a,b)}(y) := y^{a-1}e^{-by}, \qquad x \geq 0.$$

Sampling from $q_{a,b}$ is straight-forward, since one only has to sample from the Gamma(a,b) distribution, until the sample lies in the interval $[0.5, 127.5)$, and then to round the sample to the nearest integer.

For evaluating $q^u_{a,b}$ at x we use the Newton-Cotes formula with 5 nodes:

$$\begin{aligned} q^u_{a,b}(x) \;\approx\; \frac{1}{90}\big[\quad & 7\cdot f^u_{\Gamma(a,b)}(x-0.5) + 32\cdot f^u_{\Gamma(a,b)}(x-0.25) + 12\cdot f^u_{\Gamma(a,b)}(x) \\ & +32\cdot f^u_{\Gamma(a,b)}(x+0.25) + 7\cdot f^u_{\Gamma(a,b)}(x+0.5)\big] \end{aligned}$$

with remainder $-\frac{8}{945}\frac{1}{2^{14}} f^{u\ (6)}_{\Gamma(a,b)}(\xi)$ for some $\xi \in [x-0.5, x+0.5]$.

Now we choose the parameters a and b of the Gamma-distribution, from which the proposal is generated. Since the Gamma(a,b)-distribution has mode $(a-1)/b$ and since m is the value that maximizes the target density, a and b must satisfy the equation $m = (a-1)/b$. The variance $v^2 := a/(b^2)$ of the Gamma-distribution is chosen as

$$v^2 = \begin{cases} 0.05\cdot m^2 & \text{if } 1 \leq m < 10, \\ 0.10\cdot m^2 & \text{if } 10 \leq m < 30, \\ 0.20\cdot m^2 & \text{if } 30 \leq m \leq 127, \end{cases}$$

since for this choice we usually observe acceptance probabilities of about 30 to 60 percent on average. Perhaps a more sophisticated method to choose v^2 may lead to an even better adaption to the target density.

The formulas $(a-1)/b = m$ and $a/(b^2) = v^2$ $(a, b > 0)$ are equivalent to the expressions

$$\begin{aligned} b &= \frac{m + \sqrt{m^2 + 4v^2}}{2v^2}, \\ a &= bm + 1, \end{aligned}$$

which give the parameters of the Gamma(a,b)-distribution from which the proposal distribution is generated.

As mentioned above, starting from the actual value $\nu^\bullet$, the use of this truncated discretized Gamma distribution as proposal distribution allows for a fast sampling of a proposal value ν° and for an accurate and fast computation of the acceptance probability

$$\alpha(\nu^\bullet, \nu^\circ | \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*) = \min \left\{ \frac{f(\nu^\circ | \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*)}{f(\nu^\bullet | \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*)} \frac{q^u_{a,b}(\nu^\bullet)}{q^u_{a,b}(\nu^\circ)}, 1 \right\}.$$

Next we investigate the update of λ_t^*, $t = 1, \ldots, T$. Here we can see that

$$\begin{aligned} f(\lambda_t^* \,|\, \boldsymbol{y}, \boldsymbol{c}, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*, \boldsymbol{\lambda}_{-t}, \nu, \boldsymbol{\theta}) &= f(\lambda_t^* \,|\, y_t^*, \boldsymbol{\beta}, h_t^*, \nu) \\ &\propto f(\lambda_t^*, y_t^*, \boldsymbol{\beta}, h_t^*, \nu) \\ &= f(y_t^* | \boldsymbol{\beta}, h_t^*, \lambda_t^*, \nu) \pi(\lambda_t^*). \end{aligned}$$

Given $\boldsymbol{\beta}$, h_t^*, λ_t^*, and ν, the variable y_t^* is normally distributed with mean $\boldsymbol{x}_t'\boldsymbol{\beta}$ and variance $\exp(h_t^*)\lambda_t^{*-1}$. Moreover, since $\pi(\lambda_t^*)$ is the Gamma$(\nu/2, \nu/2)$-density, we conclude that

$$\begin{aligned} f(\lambda_t^* \,|\, \boldsymbol{y}, \boldsymbol{c}, \boldsymbol{y}^*, \boldsymbol{\beta}, \boldsymbol{h}^*, \boldsymbol{\lambda}_{-t}, \nu, \boldsymbol{\theta}) &\propto \left[\frac{1}{\lambda_t^{*-1/2}} \exp\left\{ -\frac{1}{2} \frac{(y_t^* - \boldsymbol{x}_t'\boldsymbol{\beta})^2}{\exp(h_t^*)\lambda_t^{*-1}} \right\} \right] \lambda_t^{*\frac{\nu}{2}-1} \exp\left\{ -\frac{\nu}{2}\lambda_t^* \right\} \\ &= \lambda_t^{*\frac{1}{2}+\frac{\nu}{2}-1} \exp\left\{ -\left[\frac{1}{2} \frac{(y_t^* - \boldsymbol{x}_t'\boldsymbol{\beta})^2}{\exp(h_t^*)} + \frac{\nu}{2} \right] \lambda_t^* \right\}. \end{aligned}$$

Hence, the variables λ_t^*, $t = 1, \ldots, T$, have to be drawn from Gamma(c, d_t)-distributions, where

$$\begin{aligned} c &= \frac{\nu + 1}{2}, \\ d_t &= \frac{\nu + (y_t^* - \boldsymbol{x}_t'\boldsymbol{\beta})^2 / \exp(h_t^*)}{2}. \end{aligned}$$

8.3 GM-MGMC sampler

The derivation of the GM step for the OSVt model is again analoguous to the OSV case. One uses the same vector

$$\boldsymbol{w} := (y_1^*, \ldots, y_T^*, c_2, \ldots, c_{K-1}, \beta_0, \ldots, \beta_p),$$

but, of course, has now the following vector of remaining parameters:

$$\boldsymbol{R} := (\boldsymbol{h}^*, \boldsymbol{\lambda}^*, \boldsymbol{\alpha}, \phi, \sigma).$$

Again one considers the conditional distribution $f(\boldsymbol{w}|\boldsymbol{y}, \boldsymbol{R})$. The scale group which was used as transformation group in the OSV case leads here also to a (truncated) Gamma(a, b)-distribution for γ^2. However, here the parameter b depends on the variables λ_t^*:

$$a = \frac{T + K + p}{2}, \tag{8.13}$$

$$b = \frac{1}{2}\left[\sum_{t=1}^{T} \frac{(y_t^* - \boldsymbol{x}_t'\boldsymbol{\beta})^2 \lambda_t^*}{\exp(h_t)} + \boldsymbol{\beta}' B_0^{-1} \boldsymbol{\beta}\right]. \tag{8.14}$$

The GM-MGMC sampler for the OSVt model therefore consists of 3 MCMC parts and the GM step which must be inserted after the first MCMC part. Whereas for part 2 one switches to the state space approximation, parts 1 and 3 use the original model equations. The steps of one iteration of the GM-MGMC sampler are summarized in the following Algorithm 8.1.

Algorithm 8.1 One iteration of the GM-MGMC sampler for the OSVt model

1. **MCMC-Step (Part 1)**

   - Draw $\boldsymbol{\beta}$ from $(p+1)$-variate normal.
   - Draw y_t^*, $t = 1, \ldots, T$, from truncated univariate normals.
   - Draw c_k, $k = 2, \ldots, K-1$, from Unif(l_k, r_k) where

   $$\begin{aligned} l_k &= \max\{c_{k-1}, \max_{t=1,\ldots,T}\{y_t^* | y_t = k\}\}, \\ r_k &= \min\{c_{k+1}, \min_{t=1,\ldots,T}\{y_t^* | y_t = k+1\}\}. \end{aligned}$$

   Get $\boldsymbol{\beta}_{cur}, \boldsymbol{y}_{cur}^*, \boldsymbol{c}_{cur}$ as current values.

2. **GM-Step**
   Draw γ^2 from the (truncated) $\Gamma(a, b)$ distribution with a and b defined in (8.13) and (8.14), respectively, and update $\boldsymbol{\beta}_{cur}^*$, $\boldsymbol{y}_{cur}^*$, $\boldsymbol{c}_{cur}$ by multiplication with the group element $\gamma = \sqrt{\gamma^2}$,

   $$\begin{aligned} \boldsymbol{\beta}_{new} &\leftarrow \gamma \boldsymbol{\beta}_{cur}, \\ \boldsymbol{y}_{new}^* &\leftarrow \gamma \boldsymbol{y}_{cur}^*, \\ \boldsymbol{c}_{new} &\leftarrow \gamma \boldsymbol{c}_{cur}. \end{aligned}$$

3. **MCMC-Step (Part 2)**

   - Compute $\tilde{y}_t^* = \log(y_t^* - \boldsymbol{x}_t'\boldsymbol{\beta})^2 + \log \lambda_t^*$.
   - Draw s_t, $t = 1, \ldots, T$, proportional to $\Pr(s_t) N(\tilde{y}_t^* | h_t^* + m_{s_t}, v_{s_t}^2)$.
   - Draw $(\boldsymbol{\alpha}, \phi, \sigma)$ via Metropolis-Hastings step; use ML-estimates of $(\boldsymbol{\alpha}, \phi, \sigma)$ to find an adequate multivariate normal proposal.
   - Draw $\boldsymbol{h}^*$ in one block using the simulation smoother of De Jong and Shephard (1995).

4. **MCMC-Step (Part 3)**

   - Draw ν by a Metropolis-Hastings step; use an ML-estimate of ν to find an adequate discretized and truncated Gamma proposal.
   - Draw λ_t^*, $t = 1, \ldots, T$ from Gamma distributions.

We note that one can use also some modified versions of this GM-MGMC sampler, since, following Section 2.4, not all parameters need to be updated in each iteration. For example, the parameter ν is used only for modeling the tail-behavior of the error distribution for the latent variables y_t^*. Therefore one can omit the update for ν until the other chains have moved away from the starting values towards the area around the true values. From the simulations in Chapter 6 we know that this takes about 50 to 100 iterations. For this one can use for example the starting value $\nu = 10$ in the first 50 iterations and update ν only from iteration 51 on. Since ν is updated in Part 3 and remains unchanged under the GM-step one can use the same GM-step as in the original sampler. The implementation of the GM-MGMC sampler in Appendix C allows also for this modification which we will use in Sections 8.4 and 8.5.

8.4 Simulation study

Here we investigate the accuracy of the posterior mean estimates for the parameter ν in the OSVt model. We do this by two simulation settings where the simulation parameters for ϕ, σ, β_0, β_1, α_1, α_2, and $c_2, \ldots, c_6$ are identical to that chosen in Section 6.4.2. Also the used covariates from IBM data are the same as well as the prior distributions. In addition, we take an uniform prior on the set $\{1, \ldots, 127\}$ for ν. The starting values are also the same as in 6.4.2, for ν we use in both settings the starting value 10.

	true	mean	std. dev.		true	mean	std. dev.
ϕ	0.90	0.8919	0.0171	c_2	0.90	0.9129	0.0178
σ	0.20	0.2152	0.0304	c_3	1.80	1.8169	0.0360
β_0	3.50	3.5181	0.0391	c_4	2.75	2.7788	0.0343
β_1	-0.30	-0.3010	0.0069	c_5	3.65	3.6740	0.0468
α_1	0.25	0.2540	0.0138	c_6	4.50	4.5303	0.0552
α_2	0.15	0.1549	0.0100	ν	**15**	**14.8781**	**2.4203**

Table 8.1: Means and standard deviations of the posterior mean estimates across the 20 samples in Setting 1.

	true	mean	std. dev.		true	mean	std. dev.
ϕ	0.90	0.8928	0.0164	c_2	0.90	0.9167	0.0192
σ	0.20	0.2131	0.0297	c_3	1.80	1.8276	0.0384
β_0	3.50	3.5214	0.0524	c_4	2.75	2.7741	0.0395
β_1	-0.30	-0.3022	0.0075	c_5	3.65	3.6735	0.0525
α_1	0.25	0.2517	0.0136	c_6	4.50	4.5405	0.0670
α_2	0.15	0.1545	0.0098	ν	**100**	**98.2844**	**8.3142**

Table 8.2: Means and standard deviations of the posterior mean estimates across the 20 samples in Setting 2.

In the first simulation setting we choose $\nu = 15$, in the second $\nu = 100$. We simulate 20 data sets for both parameter settings, each of length $T = 22000$, which is close to length of the IBM data set of Section 8.5. We compute the posterior mean estimates by running the GM-MGMC sampler for 4000 iterations each, discarding the first 1000 iterations for burn-in.

Running the GM-MGMC sampler for such data sets takes about 6.8 seconds per iteration on an UltraSPARC III Cu 900 Mhz processor. Therefore, running 4000 iterations for 20 data sets takes about 151 hours.

Tables 8.1 and 8.2 give the means and standard deviations of the posterior mean estimates across the 20 samples for the Settings 1 and 2, respectively. We note that in spite of the fact, that ν is drawn from a discrete distribution, the posterior means for ν will of course not be integers in general. The same is true for the means of the posterior means for ν.

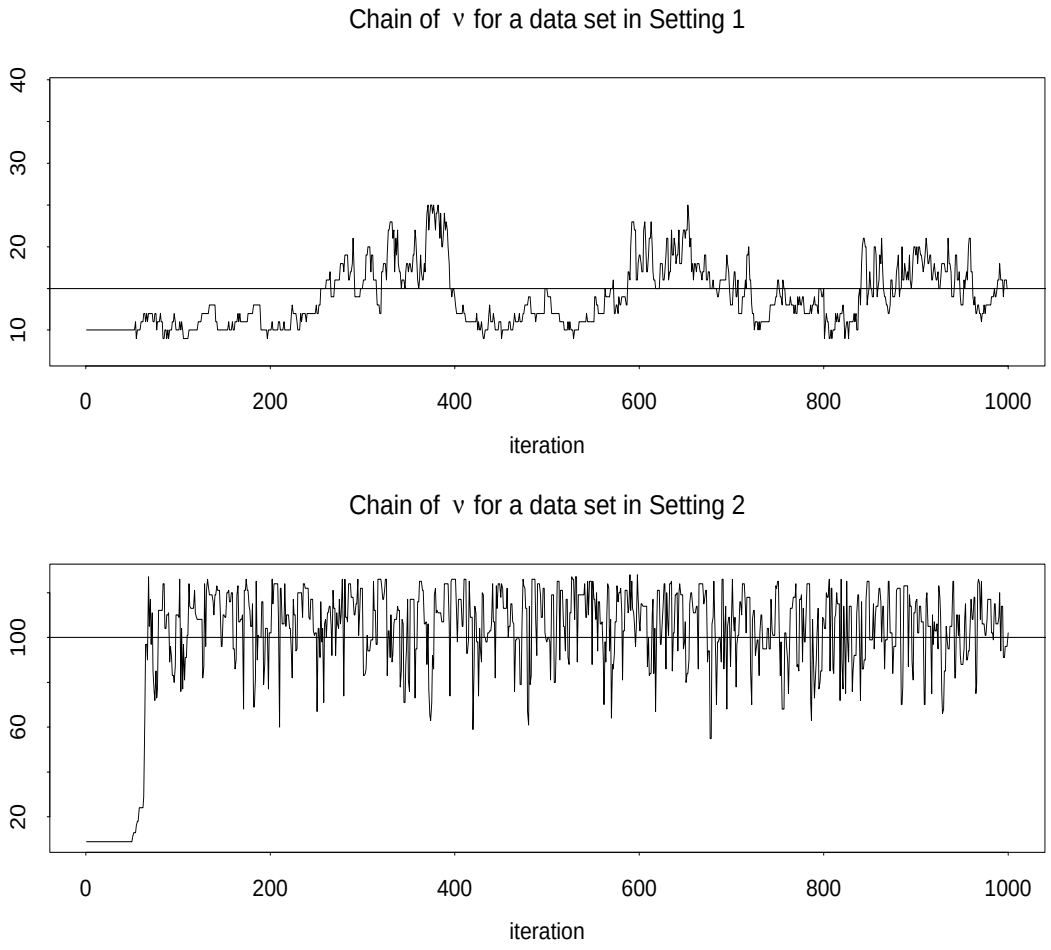

Figure 8.1: First 1000 iterations of chain of ν for both a data set of Setting 1 (above) and Setting 2 (below).

In Setting 1, where we chose the value 15 for ν, the mean of the posterior mean estimates is about 14.9 with standard deviation 2.4. Hence ν was always estimated quite well, which is also true for the other parameters. In Setting 2, where the true value for ν was 100, the mean of the posterior mean estimates is 98.3. Therefore the GM-MGMC sampler estimates ν well on average. However, the standard deviation of about 8.3 is quite large. This may be a consequence of the fact that the t-distribution becomes more and more similar to the normal distribution when the degrees of freedom increase. Therefore one needs much more data to be able to distinguish clearly between t-distributions with high degrees of freedom. The other parameters are all estimated quite well again.

Figure 8.1 shows the chain for ν for both a data set of Setting 1 and 2. In the first 50 iterations ν was fixed at the starting value 10. For Setting 1 the chain moves around

	estimate	std.err.	90% cred.int.		estimate	std.err.	90% cred.int.
ϕ	0.9133	0.0087	(0.8984,0.9275)	c_2	0.9489	0.0095	(0.9333,0.9641)
σ	0.2066	0.0136	(0.1851,0.2294)	c_3	1.8548	0.0163	(1.8280,1.8812)
β_0	3.5388	0.0385	(3.4749,3.6016)	c_4	2.7794	0.0243	(2.7381,2.8183)
β_1	-0.3082	0.0064	(-0.3181,-0.2976)	c_5	3.7217	0.0329	(3.6679,3.7759)
α_1	0.2584	0.0172	(0.2301,0.2861)	c_6	4.6116	0.0476	(4.5351,4.6868)
α_2	0.1482	0.0078	(0.1362,0.1615)	ν	**106.81**	**15.4530**	**(77,126)**

Table 8.3: Posterior mean estimates and corresponding estimated standard deviations and 90% posterior credible intervals for parameters in OSVt model.

more slowly than for Setting 2, however, fast enough to move several times around the whole support of the posterior distribution during 3000 iterations.

We conclude that the GM-MGMC sampler works very well also for the OSVt model. Since both for $\nu = 15$ and $\nu = 100$ the posterior mean estimates are satisfyingly accurate, we will use the GM-MGMC sampler in the following section to fit the OSVt model to the IBM data from Chapter 6.

8.5 Application to IBM data

Here we answer the question whether our IBM high-frequency data set in fact requires modeling with the heavier tailed t-distributed errors. For this we run the GM-MGMC sampler for the OSVt model for 4000 iterations and discard again the first 1000 for burn-in. From the simulations in Section 8.4 we know that this leads quite accurate estimates. The results are summarized in Table 8.3. It shows the posterior mean estimates together with their corresponding estimated standard deviations and 90% credible intervals for the parameters ϕ and σ in the log-volatility equation, for the regression coefficients β_0, β_1, α_1, and α_2, for the cutpoints $c_2, \ldots, c_6$, and for the parameter ν.

Comparing these values to the results for the OSV model in Table 7.6, we see that the posterior mean estimates for the OSV model are nearly identical to the posterior mean estimates in the OSVt model. The posterior mean estimate for the additional parameter ν is about 107. Since a t-distribution with 107 degrees of freedom is already quite close to a normal distribution, we conclude that the usage of t-distributed errors is not really necessary for our IBM data. Therefore we prefer the OSV model.

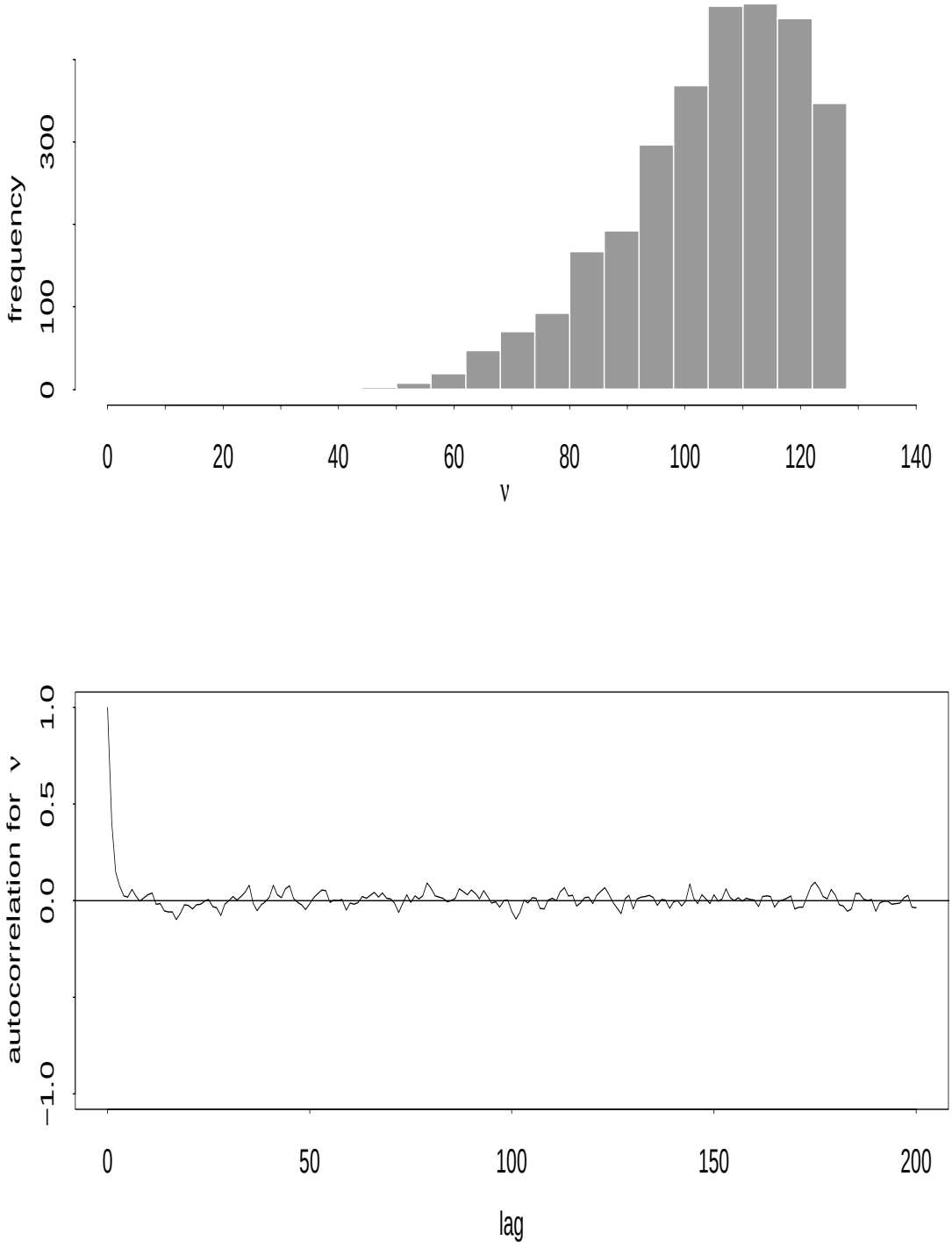

Figure 8.2: Histogramm for estimates of ν for iterations 1001 to 4000 (above). Autocorrelation of chain for ν after iteration 1000 (below).

Figure 8.2 shows an histogram of the estimates for the parameter ν in iterations 1001 to 4000. It suggests that the marginal posterior for ν is unimodal, but of course not symmetric, since the chosen prior does not allow for values greater than 127. Furthermore, Figure 8.2 shows the estimated autocorrelations in the chain for ν after iteration 1000. They decline very fast, and therefore justify that no subsampling is required to estimate the standard error of ν.

Chapter 9

Summary and Conclusion

In this thesis we introduced two regression models for ordinal valued time series: The **autoregressive ordered probit (AOP)** and the **ordinal-response stochastic volatility (OSV)** model. The latter was also extended to the **OSVt** model with t-distributed errors.

The AOP model is an **autoregressive extension of the common ordered probit model**. To fit this model to specific data sets, we developed a **GM-MGMC algorithm**. The grouped move steps which were used in each iteration are based on a special transformation group. We called this group **partial scale group**, since it rescales only a subspace of the support of the posterior distribution. As was shown in a simulation study, this transformation group led to a fundamental improvement of the convergence and mixing properties, compared to a standard Gibbs sampler.

To decide whether the AOP model can fit a given data set better than other models we developed an **estimation procedure for the marginal likelihood** of the AOP model which allows for the computation of **Bayes factors**. In this context we also provided an **auxiliary particle filter** for the AOP model.

We applied the AOP model to price changes of the IBM stock on December 4, 2000, collected at the New York Stock Exchange. However, since the variance of the latent process in the AOP model is assumed to be constant, we used not the signed but the absolute values of the price changes and hence followed a decomposition strategy similar to Rydberg and Shephard (2003). It turned out that the **autoregressive component** of the AOP model is **significant**. Moreover, we quantified the impact of significant covariates: The **time between trades** and the **transaction volume**. In both cases

higher values increase the probability for a large price change. By estimating the corresponding Bayes factor we showed that **the AOP model fits the data set decisively better than the common OP model**.

The OSV model is a **discretized version of a stochastic volatility model** and therefore can be applied to ordinal valued time series. It allows for a non-constant volatility of the continuous latent process from which the discrete response is assumed to be generated. Because of the autoregressive structure in the log-volatility also the volatility clustering effect can be modeled adequately. We developed a **GM-MGMC sampler** for fitting the OSV model to specific data sets. The exceptional feature of the used grouped move step is that one considers a certain **conditional distribution** and therefore has to perform the grouped move step exactly between two specific hybrid MCMC updates in each iteration. A simulation study illustrated the fundamental improvement of the convergence and mixing properties, compared to the hybrid MCMC sampler. As a byproduct we detected a mistake in the paper by Chib et al. (2002). For a certain update in their MCMC algorithm they did not take the corresponding informative prior into account.

We applied the OSV model to the price changes of the IBM stock in a period of nine days in January 2001. The mean of the latent continuous process from which the discrete response is generated significantly increases when the **previous response** decreases and vice versa. The log-volatilities have a **strong autoregressive structure**. The corresponding parameter estimate is about 0.9 and hence large but away from one. Moreover, the **time between trades** and the **transaction volume** are significant covariates for the evolution of the log-volatilities. The signs of the corresponding parameter estimates led to the conclusion, that **the volatility increases when more time between two subsequent transactions elapses**. The same is true for the transaction volume: **The more stocks are traded the higher is the probability for a large price change**. These results agree with theoretical considerations of the trading process which have been undertaken by Diamond and Verrecchia (1987), Easley and O'Hara (1987), and Tauchen and Pitts (1983).

Finally we investigated whether it is more accurate to use t-distributed instead of normally distributed errors. For this we expanded the OSV model to the OSVt model and adapted the GM-MGMC sampler of the OSV model to the OSVt case. The OSVt model was fitted to the same IBM data set as the OSV model. The value of the posterior mean estimate for the degrees of freedom of the t-distribution was about 107. Since this value is quite large, we concluded that **the use of t-distributed errors is not really**

necessary for this data set and that the use of normally distributed errors seems to be adequate.

An open issue is the development of an appropriate model selection criterion for the OSV and OSVt model. Such a criterion would allow for example to quantify the improvement which is achieved by using additional covariates or to compare the OSV(t) model to totally different models. One possibility is to develop an estimation procedure for the marginal likelihood to be able to compute Bayes factors.

Another possibility is to use the Deviance Information Criterion (DIC) for model comparison. However, as was pointed out by DeIorio and Robert in the discussion of the paper Spiegelhalter et al. (2002), the use of the DIC seems problematic for models, where latent variables are involved. They show that several versions of the DIC can be considered which differ in the treatment of the latent variables. Some versions do not use the latent variables, whereas some incorporate them as additional parameters. DeIorio and Robert show that the different strategies can lead to rather different results in model selection. Therefore more investigation is required to answer the question whether and how the DIC criterion can be used properly for the AOP and the OSV(t) model.

A possible extension of the models discussed here is to consider corresponding models in which the parameters are allowed to switch amongst a given number of states according to a hidden Markov process. The basic stochastic volatility model under this assumption has been investigated by So et al. (1998). Second, one can develop continuous time analogues of the AOP and the OSV model. For this one may use the approaches by Elerian et al. (2001) and Eraker (2001). The advantage of such continuous time models is the handling of the non-equidistant transaction times. Instead of using a corresponding covariate one assumes that the time-continuous process is observed at non-equidistant time points. Third, one can extend the models to the multivariate case and apply them to ordinal-valued time series with multivariate response, for example to price changes of several stocks.

Finally we note that the models which have been introduced here can be used also in other situations. For example, another important area where ordinal valued time series occur is medicine. Patients assess the severity of their pain on an ordinal scale, say, on the set $\{0, 1, \ldots, 5\}$. Naturally these data may have an autoregressive structure, and also some covariates such as temperature or humidity may influence the pain severity. If one expects a constant volatility of the process, especially the application of the AOP model could be appropriate.

Appendix A

Nelder-Mead Minimization Algorithm

Here we describe the Nelder-Mead minimization algorithm which is used in Section 6.2 to find an appropriate proposal density for the Metropolis-Hastings step used there. The Nelder-Mead algorithm, first published in Nelder and Mead (1965), is used for unconstrained minimization of a scalar-valued nonlinear function of n real variables. Since it requires only function values and not any derivative information (explicit or implicit), it falls in the class of direct search methods. Especially in lower dimensions the Nelder-Mead algorithm works very well and converges fast to the minimum. For a detailed discussion of the convergence properties we refer to Lagarias et al. (1998).

A large subclass of the direct search methods, including the Nelder-Mead method, maintains at each step a nondegenerate simplex, a geometric figure in n dimensions of nonzero volume that is the convex hull of $n+1$ vertices. Each iteration of a simplex-based direct search method begins with a simplex, specified by its $n+1$ vertices and the associated function values. One or more test points are computed, along with their function values, and the iteration terminates with a new different simplex such that the function values at its vertices satisfy some form of descent condition compared to the previous simplex.

Now we describe the Nelder-Mead algorithm in detail. Aim is the minimization of the real-valued function $f(\boldsymbol{x})$ for $\boldsymbol{x} \in \mathbb{R}^n$. First four scalar parameters must be specified to define a complete Nelder-Mead method. These are the coefficients of **reflection** (ρ), **expansion** (χ), **contraction** (γ), and **shrinkage** (σ). According to the original Nelder-Mead paper, these parameters should satisfy the conditions $\rho > 0$, $\chi > 1$, $\chi > \rho$,

$0 < \gamma < 1$, and $0 < \sigma < 1$. Often used choices of these parameters are

$$\rho = 1, \quad \chi = 2, \quad \gamma = \tfrac{1}{2}, \quad \text{and} \quad \sigma = \tfrac{1}{2}.$$

At the beginning of iteration k, $k \geq 0$, a nondegenerate simplex Δ_k is given, along with its $n+1$ vertices which are points in $\mathbb{R}^n$. First these vertices are ordered and labeled as $\boldsymbol{x}_1^{(k)}, \ldots, \boldsymbol{x}_{n+1}^{(k)}$, such that

$$f_1^{(k)} \leq f_2^{(k)} \leq \ldots \leq f_{n+1}^{(k)}$$

where $f_i^{(k)}$ denotes $f(\boldsymbol{x}_i^{(k)})$. As we are searching for a minimum we refer to $\boldsymbol{x}_1^{(k)}$ as the **best** point or vertex and to $\boldsymbol{x}_{n+1}^{(k)}$ as the **worst**. Accordingly $f_{n+1}^{(k)}$ is referred to as the worst function value, and so on. The result of each iteration is either a single new vertex which replaces the worst vertex $\boldsymbol{x}_{n+1}^{(k)}$ in the set of vertices for the next iteration, or, if a shrink step is performed, a set of n new points which, together with the best point $\boldsymbol{x}_1^{(k)}$, form the new simplex $\Delta_{k+1} \neq \Delta_k$ for the next iteration. Algorithm A.1 summarizes the steps of one iteration of the Nelder-Mead method. For notational convenience we suppress the iteration index $^{(k)}$.

Algorithm A.1 One iteration of the Nelder-Mead method

1. **Order.** Order the $n+1$ vertices to satisfy $f(\boldsymbol{x}_1) \leq f(\boldsymbol{x}_2) \leq \ldots f(\boldsymbol{x}_{n+1})$.

2. **Reflect.** Compute the **reflection point**

$$\boldsymbol{x}_{[r]} := \bar{\boldsymbol{x}} + \rho(\bar{\boldsymbol{x}} - \boldsymbol{x}_{n+1}) = (1+\rho)\bar{\boldsymbol{x}} - \rho\boldsymbol{x}_{n+1}$$

   where $\bar{\boldsymbol{x}} := n^{-1} \sum_{i=1}^{n} \boldsymbol{x}_i$ is the centroid of the n best points.
   Evaluate $f_{[r]} := f(\boldsymbol{x}_{[r]})$.
   If $f_1 \leq f_{[r]} < f_n$, accept the reflected point $\boldsymbol{x}_{[r]}$ and terminate the iteration.

3. **Expand.** If $f_{[r]} < f_1$, calculate the **expansion point**

$$\boldsymbol{x}_{[e]} := \bar{\boldsymbol{x}} + \chi(\boldsymbol{x}_{[r]} - \bar{\boldsymbol{x}}) = \bar{\boldsymbol{x}} + \rho\chi(\bar{\boldsymbol{x}} - \boldsymbol{x}_{n+1}) = (1+\rho\chi)\bar{\boldsymbol{x}} - \rho\chi\boldsymbol{x}_{n+1}$$

   and evaluate $f_{[e]} := f(\boldsymbol{x}_{[e]})$.
   If $f_{[e]} < f_{[r]}$, accept $\boldsymbol{x}_{[e]}$ and terminate the iteration.
   If $f_{[e]} \geq f_{[r]}$, accept $\boldsymbol{x}_{[r]}$ and terminate the iteration.

4. **Contract.** If $f_{[r]} \geq f_n$, perform a **contraction** between $\bar{\boldsymbol{x}}$ and the better of $\boldsymbol{x}_{n+1}$ and $\boldsymbol{x}_{[r]}$.

a. **Outside.** If $f_n \leq f_{[r]} < f_{n+1}$, perform an **outside contraction**: Calculate

$$\boldsymbol{x}_{[c]} := \bar{\boldsymbol{x}} + \gamma(\boldsymbol{x}_{[r]} - \bar{\boldsymbol{x}}) = \bar{\boldsymbol{x}} + \gamma\rho(\bar{\boldsymbol{x}} - \boldsymbol{x}_{n+1}) = (1 + \rho\gamma)\bar{\boldsymbol{x}} - \rho\gamma\boldsymbol{x}_{n+1}$$

and evaluate $f_{[c]} := f(\boldsymbol{x}_{[c]})$.
If $f_{[c]} \leq f_{[r]}$, accept $\boldsymbol{x}_{[c]}$ and terminate the iteration.
If $f_{[c]} > f_{[r]}$, perform a shrink step (step 5).

b. **Inside.** If $f_{[r]} \geq f_{n+1}$, perform an **inside contraction**: Calculate

$$\boldsymbol{x}_{[cc]} := \bar{\boldsymbol{x}} - \gamma(\bar{\boldsymbol{x}} - \boldsymbol{x}_{n+1}) = (1 - \gamma)\bar{\boldsymbol{x}} + \gamma\boldsymbol{x}_{n+1}$$

and evaluate $f_{[cc]} := f(\boldsymbol{x}_{[cc]})$.
If $f_{[cc]} \leq f_{n+1}$, accept $\boldsymbol{x}_{[cc]}$ and terminate the iteration.
If $f_{[cc]} > f_{n+1}$, perform a shrink step (step 5).

5. **Shrink.** Evaluate f at the n points

$$\boldsymbol{v}_i := \boldsymbol{x}_1 + \sigma(\boldsymbol{x}_i - \boldsymbol{x}_1), \qquad i = 2, \ldots, n+1,$$

and consider in the next iteration the simplex which is represented by the (unordered) vertices $\boldsymbol{x}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_{n+1}$.

Figures A.1 and A.2 visualize the effects of reflection, expansion, contraction, and shrinkage for a simplex in two dimensions, i.e. a triangle, using the standard coefficients $\rho = 1$, $\chi = 2$, $\gamma = 0.5$, and $\sigma = 0.5$. We observe that, except in a shrink, the one new vertex always lies on the (extended) line joining $\bar{\boldsymbol{x}}$ and $\boldsymbol{x}_{n+1}$. During an expansion or contraction with the standard coefficients the shape of the simplex performs a noticeable change.

Nelder and Mead (1965) did not mention how to order the vertices in the case of equal function values. Lagarias et al. (1998) provide the following tie-breaking rules, which assign to the new vertex the highest possible index consistent with the relation

$$f(\boldsymbol{x}_1^{(k+1)}) \leq f(\boldsymbol{x}_2^{(k+1)}) \leq \ldots \leq f(\boldsymbol{x}_{n+1}^{(k+1)}).$$

Nonshrink ordering rule. When a nonshrink step occurs, the worst vertex $\boldsymbol{x}_{n+1}^{(k)}$ is discarded. The accepted point created during the kth iteration, denoted by $\boldsymbol{v}^{(k)}$, becomes a new vertex and takes position $j + 1$ in the vertices of Δ_{k+1}, where

$$j := \max_{0 \leq l \leq n} \left\{ l \mid f(\boldsymbol{v}^{(k)}) < f(\boldsymbol{x}_{l+1}^{(k)}) \right\}.$$

All other vertices retain their relative ordering from iteration k.

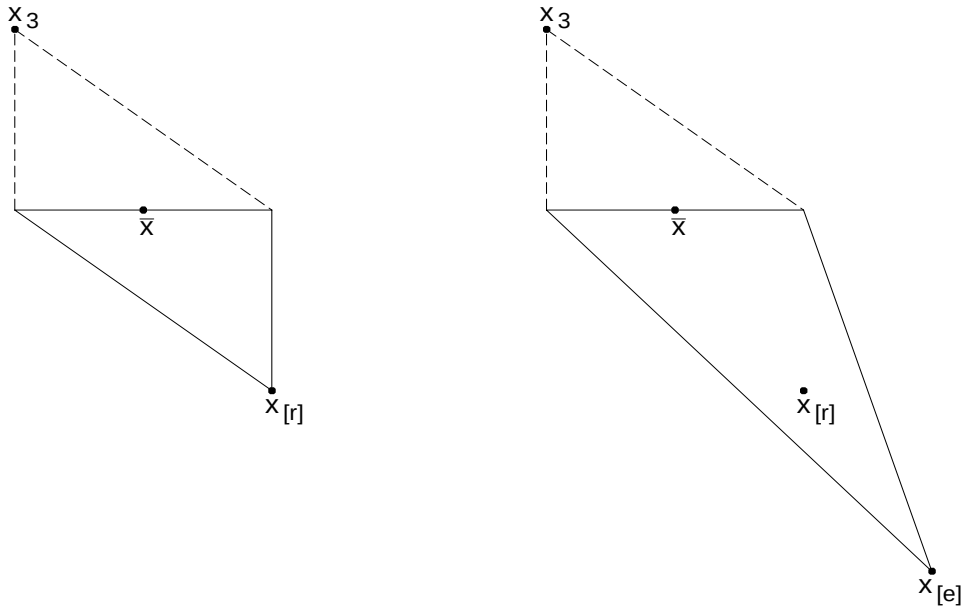

Figure A.1: Nelder-Mead simplices after a reflection (left) and an expansion step (right). The original simplex is indicated by the dashed lines.

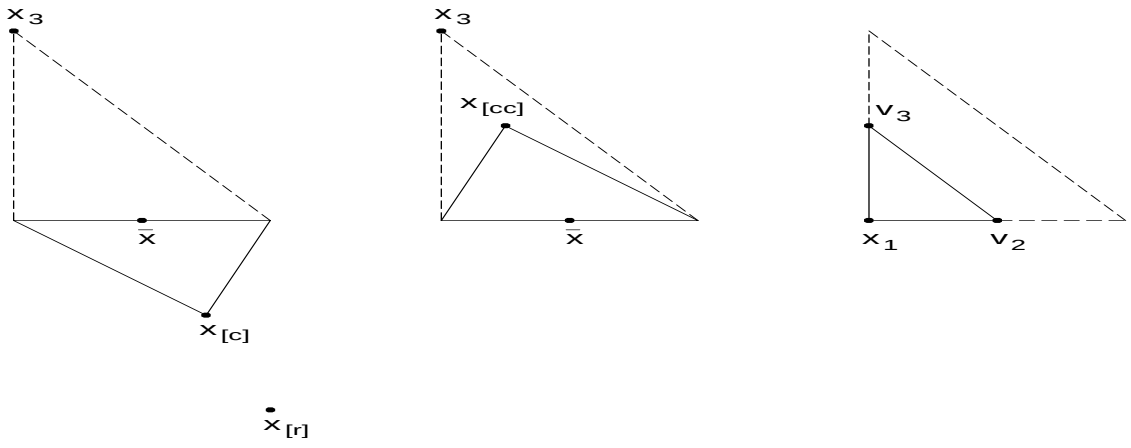

Figure A.2: Nelder-Mead simplices after an outside contraction (left), an inside contraction, and a shrink (right). The original simplex is indicated by the dashed lines.

Shrink ordering rule. In a shrink step the only vertex carried over from iteration k to iteration $k+1$ is $\boldsymbol{x}_1^{(k)}$. For the case that $\boldsymbol{x}_1^{(k)}$ and one or more of the new points are ties as the best point, only one tie-breaking rule is specified. If

$$\min\left\{f(\boldsymbol{v}_2^{(k)}),\ldots,f(\boldsymbol{v}_{n+1}^{(k)})\right\} = f(\boldsymbol{x}_1^{(k)}),$$

then $\boldsymbol{x}_1^{(k+1)} = \boldsymbol{x}_1^{(k)}$.

Finally we note that, to have a stopping criterion, one can compute the difference between the worst and the best vertex, $f_{n+1}^{(k)} - f_1^{(k)}$, and stop the algorithm if this expression is sufficiently small.

For more details on the Nelder-Mead simplex algorithm and variations we refer to Walters et al. (1991).

Appendix B

Implementation of GM-MGMC Sampler for AOP Model

Here we provide the C++ implementation of the GM-MGMC sampler for the AOP model. Before compiling the program, one must adapt some file names and constants to the data. This must be done in the section which is marked by 'Change only this section'. More informations can be seen directly in the program code which contains many detailed comments. At the beginning the two header-files 'randomGM.h' and 'matrixGM.h' are included. These can be found in Appendix D and E, respectively, and must be copied into the same directory as the following program.

```
//-------------------------------------------------------------------------------
//             GM-MGMC SAMPLER FOR AOP MODEL --- by GERNOT MUELLER, 2004
//-------------------------------------------------------------------------------
// Change program only in the marked section below !
//
// Steps needed before executing the program:
// - adapt the program code to your data and files (only in marked section!)
// - save the program (aop.cpp)
// - compile the program (e.g. using: CC -O4 aop.cpp)
// - you will get an executable 'a.out' file
// - type a.out to execute
//
// Maximal number of covariates:    8
// Maximal number of cutpoints :   10
// Maximal length of data      : 6000
//-------------------------------------------------------------------------------

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <iostream.h>

#define SPEC
```

```
#include "randomGM.h"
#include "matrixGM.h"

//-------------------------------------------------------------------------------
//---------------------------- CHANGE ONLY THIS SECTION --------------------------
//-------------------------------------------------------------------------------

#define TTTTTT 2000      // length of data

unsigned int ITER=15000; // do not change ITER (number of iterations in total) and
unsigned int BURN=5000;  // BURN (number of Burnin iterations) without changing the
                         // strategy for result files!

unsigned int NNCP=3;     // number of cutpoints (e.g. 4 data categories => NNCP=3 )
unsigned int NCOV=4;     // number of (covariates including intercept) + 1
                         // (e.g. two covariates => set NCOV = 4)
unsigned int CINT=0;     // must equal 1 if the design file contains column with 1's
                         //      for intercept
                         // must equal 0 else
unsigned int USEGM=1;    // if grouped move steps should be used, set to 1, else to 0

double tau  =0.10000000; // has meaning of tau^{-2}   (hyperparam. for beta-prior)
double sigma=1.00000000; // has meaning of sigma^{-2} (hyperparam. for y_0^\ast-pr.)
double rho  =10.0000000; // has meaning of rho^{-2}   (hyperparam. for phi-prior)

                         // give paths for data source files:
                         // procsource: response file, desgsource: design file
char procsource[]="ibm04decproc.txt";
char desgsource[]="ibm04decdesgTDSIZE.txt";

                         // give paths for parameter result files:
                         // resultGM:   result file, when GM steps are used
                         // resultNOGM: result file, when no GM steps are used
char resultGM[]="TDSIZE_LRARgm.txt";
char resultNOGM[]="TDSIZE_LRARnogm.txt";

                         // give paths for latent variable result files:
                         // resultY5:  result file for iterations  5001- 7000
                         // resultY7:  result file for iterations  7001- 9000
                         // resultY9:  result file for iterations  9001-11000
                         // resultY11: result file for iterations 11001-13000
                         // resultY13: result file for iterations 13001-15000
char resultY5[]="TDSIZE_LRARgmY5.txt";
char resultY7[]="TDSIZE_LRARgmY7.txt";
char resultY9[]="TDSIZE_LRARgmY9.txt";
char resultY11[]="TDSIZE_LRARgmY11.txt";
char resultY13[]="TDSIZE_LRARgmY13.txt";

//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------

// further global variables
Matrix X(TTTTTT,NCOV);
Matrix XtXinv(NCOV,NCOV);
Matrix Tau(NCOV,NCOV);
Matrix Trunc(NCOV,3);
Matrix Start(NCOV,1);
Matrix Mu(NCOV,1);
Matrix F(TTTTTT,1);
```

```
Matrix beta(NCOV,1);

Matrix ESTX(TTTTTT,NCOV);
Matrix ESTXtXinv(NCOV,NCOV);
Matrix ESTMu(NCOV,1);
Matrix ESTF(TTTTTT,1);

unsigned int TTTT=TTTTTT;
unsigned int Y[6004];
double YA_hat[6004];
double c_hat[12];
double beta_hat[12];

//-------------------------------------------------------------------------------
// Gibbs sampler for AOP model
// T         = length of data
// NREG      = number of (covariates including intercept) + 1
// NCP       = number of cutpoints (e.g. 4 data categories => NNCP=3 )
// iter      = number of iterations to do
// burnin    = number of burn-in iterations
// gm_start  = first iteration with grouped move step
// gm_end    = last iteration with grouped move step (gm_end<gm_start: no gm steps!)
// hist_file = if history files should be produced =1, else =0

void gibbs(unsigned int T, unsigned int NREG, unsigned int NCP,
           unsigned long int iter, unsigned int burnin, unsigned int gm_start,
           unsigned int gm_end, unsigned int hist_file=0)
{
   //------------------------------------------------
   // declaration of variables

   unsigned long int i;
   int k,l,t;
   double c_sum[12], c_sum2[12];
   double beta_sum[12],  beta_sum2[12];

   for (i=0; i<=10; i++)
   {
      c_sum[i]=0.00000; c_sum2[i]=0.00000;
      beta_sum[i]=0.00000;  beta_sum2[i]=0.00000;
   }

   double g_sq=0.00000, g=0.00000;

   FILE *stream;
   FILE *streamY5;
   FILE *streamY7;
   FILE *streamY9;
   FILE *streamY11;
   FILE *streamY13;

   double d0=0.0,d1=0.0,d2=0.0,d3=0.0,d4=0.0,d5=0.0,d6=0.0,d7=0.0,d8=0.0,d9=0.0;
   double d10=0.0,d11=0.0,d12=0.0,d13=0.0,d14=0.0,d15=0.0,d16=0.0,d20=0.0;
   double d21=0.0,d22=0.0,d23=0.0,d24=0.0;
   double d100=0.0,d101=0.0,d102=0.0,d103=0.0;
   double impact0=0.0000;

   //------------------------------------------------
   // when result files must be produced, write column names into the files

   if (hist_file==1)
```

```
{
   if (gm_start<=iter && gm_end>=iter) stream=fopen(resultGM,"w");
   if (gm_start>iter)                  stream=fopen(resultNOGM,"w");

   for (k=2; k<=NCP; k++) fprintf(stream, "c%d \t", k);
   for (k=0; k<=NREG-2;   k++) fprintf(stream, "beta%d \t", k);
   fprintf(stream, "beta%d \n", NREG-1);

   streamY5=fopen(resultY5,"w");
   for (k=0; k<=T-1;   k++) fprintf(streamY5, "Yast%d \t", k);
   fprintf(streamY5, "Yast%d \n", T);
   streamY7=fopen(resultY7,"w");
   for (k=0; k<=T-1;   k++) fprintf(streamY7, "Yast%d \t", k);
   fprintf(streamY7, "Yast%d \n", T);
   streamY9=fopen(resultY9,"w");
   for (k=0; k<=T-1;   k++) fprintf(streamY9, "Yast%d \t", k);
   fprintf(streamY9, "Yast%d \n", T);
   streamY11=fopen(resultY11,"w");
   for (k=0; k<=T-1;   k++) fprintf(streamY11, "Yast%d \t", k);
   fprintf(streamY11, "Yast%d \n", T);
   streamY13=fopen(resultY13,"w");
   for (k=0; k<=T-1;   k++) fprintf(streamY13, "Yast%d \t", k);
   fprintf(streamY13, "Yast%d \n", T);
}

//-------------------------------------------------
// starting values for c
for (i=1; i<=NCP; i++) c_hat[i]=2*(i-1.000000);  // 0,2,4,6,8,10,...
                           // if you change anything here, then you must change
                           // also the starting values for the y_t^\ast's !
//-------------------------------------------------
// starting values for beta
for (i=0; i<=NREG-1; i++) beta_hat[i]=0.000000;

//-------------------------------------------------
// write the starting values for c and beta in the file

if (hist_file==1)
{
   for (k=2; k<=NCP; k++)    fprintf(stream, "%f \t", c_hat[k]);
   for (k=0; k<=NREG-2; k++) fprintf(stream, "%f \t", beta_hat[k]);
   fprintf(stream, "%f \n", beta_hat[NREG-1]);
}

//-------------------------------------------------
// starting values for the y_t^\ast's

for (i=1; i<=T; i++)
   for (k=1; k<=NCP+1; k++)
      if (Y[i]==k) YA_hat[i]=uniform(2*k-3.9500,2*k-2.0500);

//-------------------------------------------------
// generate matrix Tau
for (i=1; i<=NREG; i++)
{
   for (k=1; k<=NREG; k++)
   {
      if (i==k) Tau(i,i)=tau;
      else      Tau(i,k)=0.00000;
   }
}
```

```
Tau(NREG,NREG)=rho;

//---------------------------------------------------
// set random generator on a random position
randomize();

//---------------------------------------------------
// iteration loop

for (i=1; i<=iter; i++)
{
   if (i%100==1 && gm_start<=iter && gm_end>=iter)   printf("\n GM \n");
   if (i%100==1 && gm_start>iter)                    printf("\n NO \n");
   if (i%10==0) printf("%d \n",i);

   //-----------------------------------------------
   // y_0^\ast - update

   impact0=beta_hat[0];
   for (k=1; k<=NREG-2; k++) impact0+=beta_hat[k]*X(1,k+1);
   d0 = beta_hat[NREG-1]*(YA_hat[1]-impact0)/
                         (1+beta_hat[NREG-1]*beta_hat[NREG-1]+sigma);
   YA_hat[0]=normal(d0,sqrt(1.00000/(1+beta_hat[NREG-1]*beta_hat[NREG-1]+sigma)));

   //-----------------------------------------------
   // y_t^\ast - update, t=1,...,T-1
   for (t=1; t<=T-1; t++)
   {
      impact0=beta_hat[0];
      for (k=1; k<=NREG-2; k++) impact0+=beta_hat[k]*X(t+1,k+1);

      d1 = YA_hat[t+1]-impact0;

      impact0=beta_hat[0];
      for (k=1; k<=NREG-2; k++) impact0+=beta_hat[k]*X(t,k+1);

      d2 = -beta_hat[NREG-1]*YA_hat[t-1]-impact0;
      d3 = 1.00000/(1+beta_hat[NREG-1]*beta_hat[NREG-1]);
      d4 = (beta_hat[NREG-1]*d1-d2)*d3;
      if (Y[t]==1)
         YA_hat[t]=rs_trunc_normal(0.000000,d4,sqrt(d3));
      if (Y[t]>1 && Y[t]<NCP+1)
         YA_hat[t]=ds_trunc_normal(c_hat[Y[t]-1],c_hat[Y[t]],
                                   d4,sqrt(d3));
      if (Y[t]==NCP+1)
         YA_hat[t]=ls_trunc_normal(c_hat[NCP],d4,sqrt(d3));
   }

   //-----------------------------------------------
   // y_T^\ast - update

   impact0=beta_hat[0];
   for (k=1; k<=NREG-2; k++) impact0+=beta_hat[k]*X(T,k+1);

   d5 = impact0+beta_hat[NREG-1]*YA_hat[T-1];
   if (Y[T]==1)
      YA_hat[T]=rs_trunc_normal(0.000000,d5,1.0000);
   if (Y[T]>1 && Y[T]<NCP+1)
      YA_hat[T]=ds_trunc_normal(c_hat[Y[T]-1],c_hat[Y[T]],d5,1.0000);
   if (Y[T]==NCP+1)
      YA_hat[T]=ls_trunc_normal(c_hat[NCP],d5,1.0000);
```

```
//-----------------------------------------------
// beta - update
// write last column of matrix X
for (t=1; t<=T; t++) X(t,NREG)=YA_hat[t-1];
XtXinv = (!(X.transp()*X+Tau));
for (t=1; t<=T; t++) F(t,1)=YA_hat[t];
Mu = XtXinv*(X.transp())*F;
beta = mvnormal(Mu,XtXinv);
for (k=1; k<=NREG; k++) beta_hat[k-1]=beta(k,1);

//-----------------------------------------------
// c - update
for (k=2; k<=NCP; k++)
{
   d15=c_hat[k-1];
   for (l=1; l<=T; l++)   if (Y[l]==k   && YA_hat[l]>d15) d15=YA_hat[l];
   if (k<NCP)  d16 = c_hat[k+1];
   else        d16 = 1000.0;
   for (l=1; l<=T; l++)   if (Y[l]==k+1 && YA_hat[l]<d16) d16=YA_hat[l];

   c_hat[k] = uniform(d15,d16);
}

if (i>=gm_start && i<=gm_end)
{
   // grouped move step
   d20=0.000000;
   for (t=1; t<=T; t++)
   {
      impact0=beta_hat[0];
      for (k=1; k<=NREG-2; k++) impact0+=beta_hat[k]*X(t,k+1);
      d20=d20 + ( YA_hat[t] - impact0 - beta_hat[NREG-1]*YA_hat[t-1] )*
                ( YA_hat[t] - impact0 - beta_hat[NREG-1]*YA_hat[t-1] );
   }
   d20=d20 + sigma*YA_hat[0]*YA_hat[0];
   for (k=0; k<=NREG-2; k++) d20 += tau*beta_hat[k]*beta_hat[k];
   d20=d20/2.000000;

   // draw gamma^2 from truncated Gamma-distribution
   do {
      g_sq  = rand_gamma(0.50000*(1.00000*T+1.00000*NREG+1.00000*NCP),d20);
   } while (g_sq>100/(c_hat[NCP]*c_hat[NCP]));
   g     = sqrt(g_sq);

   for (t=0; t<=T; t++)      YA_hat[t]    = g*YA_hat[t];
   for (k=0; k<=NREG-2; k++) beta_hat[k]  = g*beta_hat[k]; // not phi!
   for (k=2; k<=NCP; k++)    c_hat[k] = g*c_hat[k];

}

//-----------------------------------------------
// write c, beta, and the YA's in the result file
if (hist_file==1)
{
   for (k=2; k<=NCP; k++)    fprintf(stream, "%f \t", c_hat[k]);
   for (k=0; k<=NREG-2; k++) fprintf(stream, "%f \t", beta_hat[k]);
   fprintf(stream, "%f \n", beta_hat[NREG-1]);
}
if (hist_file==1 && i>5000 && i<=7000)
{
```

```
         for (k=0; k<=T-1; k++)    fprintf(streamY5, "%f \t", YA_hat[k]);
         fprintf(streamY5, "%f \n", YA_hat[T]);
      }
      if (hist_file==1 && i>7000 && i<=9000)
      {
         for (k=0; k<=T-1; k++)   fprintf(streamY7, "%f \t", YA_hat[k]);
         fprintf(streamY7, "%f \n", YA_hat[T]);
      }
      if (hist_file==1 && i>9000 && i<=11000)
      {
         for (k=0; k<=T-1; k++)   fprintf(streamY9, "%f \t", YA_hat[k]);
         fprintf(streamY9, "%f \n", YA_hat[T]);
      }
      if (hist_file==1 && i>11000 && i<=13000)
      {
         for (k=0; k<=T-1; k++)   fprintf(streamY11, "%f \t", YA_hat[k]);
         fprintf(streamY11, "%f \n", YA_hat[T]);
      }
      if (hist_file==1 && i>13000 && i<=15000)
      {
         for (k=0; k<=T-1; k++)   fprintf(streamY13, "%f \t", YA_hat[k]);
         fprintf(streamY13, "%f \n", YA_hat[T]);
      }

      //-----------------------------------------------
      // produce posterior mean estimates

      if (i>burnin)
      {
         for (k=2; k<=NCP; k++)    c_sum[k] = c_sum[k] + c_hat[k];
         for (k=0; k<=NREG-1; k++) beta_sum[k]  = beta_sum[k]  + beta_hat[k];
      }

   }

   //-----------------------------------------------
   // print posterior mean estimates on screen
   for (k=2; k<=NCP; k++)    c_sum[k] = c_sum[k]/(iter-burnin);
   for (k=0; k<=NREG-1; k++) beta_sum[k]  = beta_sum[k]/(iter-burnin);
   for (k=2; k<=NCP; k++)    printf("c%d:\t%f\n",k,c_sum[k]);
   for (k=0; k<=NREG-2; k++) printf("beta%d:\t%f\n",k,beta_sum[k]);
   printf("phi:\t%f\n",beta_sum[NREG-1]);

   //-----------------------------------------------
   // close result files
   if (hist_file==1)
   {
      fclose(stream);   fclose(streamY5);  fclose(streamY7);
      fclose(streamY9); fclose(streamY11); fclose(streamY13);
   }
}

//-----------------------------------------------------------------------------------
// main program

int main(void)
{
   //-----------------------------------------------
   // set random generator on a random position
   randomize();
```

```
//-------------------------------------------------
// declaration of variables
int i;
float f1,f2,f3,f4,f5,f6,f7,f8,f9;
FILE *sourcestream1;
FILE *sourcestream2;

sourcestream1 = fopen(procsource,"r");
sourcestream2 = fopen(desgsource,"r");

//-------------------------------------------------
// read process data
for (i=1; i<=TTTT; i++) fscanf(sourcestream1,"%d\n",&Y[i]);
fclose(sourcestream1);

//-------------------------------------------------
// read covariate data
if (CINT==1)  // File with covariates contains 1's for intercept
{
   for (i=1; i<=TTTT; i++)
   {
      switch(NCOV)
      {
         case 4:  fscanf(sourcestream2,"%f %f %f \n",&f1,&f2,&f3);
                  X(i,1)=1.000000; X(i,2)=f2; X(i,3)=f3; break;
         case 5:  fscanf(sourcestream2,"%f %f %f %f \n",&f1,&f2,&f3,&f4);
                  X(i,1)=1.000000; X(i,2)=f2; X(i,3)=f3; X(i,4)=f4; break;
         case 6:  fscanf(sourcestream2,"%f %f %f %f %f \n",&f1,&f2,&f3,&f4,&f5);
                  X(i,1)=1.000000; X(i,2)=f2; X(i,3)=f3; X(i,4)=f4; X(i,5)=f5;
                  break;
         case 7:  fscanf(sourcestream2,"%f %f %f %f %f %f \n",&f1,&f2,&f3,&f4,&f5,
                                                              &f6);
                  X(i,1)=1.000000; X(i,2)=f2; X(i,3)=f3; X(i,4)=f4; X(i,5)=f5;
                  X(i,6)=f6; break;
         case 8:  fscanf(sourcestream2,"%f %f %f %f %f %f %f \n",&f1,&f2,&f3,&f4,
                                                                 &f5,&f6,&f7);
                  X(i,1)=1.000000; X(i,2)=f2; X(i,3)=f3; X(i,4)=f4; X(i,5)=f5;
                  X(i,6)=f6; X(i,7)=f7; break;
         case 9:  fscanf(sourcestream2,"%f %f %f %f %f %f %f %f \n",&f1,&f2,&f3,
                                                                   &f4,&f5,&f6,&f7,&f8);
                  X(i,1)=1.000000; X(i,2)=f2; X(i,3)=f3; X(i,4)=f4; X(i,5)=f5;
                  X(i,6)=f6; X(i,7)=f7; X(i,8)=f8; break;
         case 10: fscanf(sourcestream2,"%f %f %f %f %f %f %f %f %f \n",&f1,&f2,
                                                                     &f3,&f4,&f5,&f6,&f7,&f8,&f9);
                  X(i,1)=1.000000; X(i,2)=f2; X(i,3)=f3; X(i,4)=f4; X(i,5)=f5;
                  X(i,6)=f6; X(i,7)=f7; X(i,8)=f8; X(i,9)=f9; break;
      }
   }
   fclose(sourcestream2);
}
else          // File with covariates does not contain 1's for intercept
{
   for (i=1; i<=TTTT; i++)
   {
      switch(NCOV)
      {
         case 4:  fscanf(sourcestream2,"%f %f \n",&f2,&f3);
                  X(i,1)=1.000000; X(i,2)=f2; X(i,3)=f3; break;
         case 5:  fscanf(sourcestream2,"%f %f %f \n",&f2,&f3,&f4);
                  X(i,1)=1.000000; X(i,2)=f2; X(i,3)=f3; X(i,4)=f4; break;
         case 6:  fscanf(sourcestream2,"%f %f %f %f \n",&f2,&f3,&f4,&f5);
```

```
                    X(i,1)=1.000000; X(i,2)=f2; X(i,3)=f3; X(i,4)=f4; X(i,5)=f5;
                    break;
            case 7:  fscanf(sourcestream2,"%f %f %f %f %f \n",&f2,&f3,&f4,&f5,&f6);
                    X(i,1)=1.000000; X(i,2)=f2; X(i,3)=f3; X(i,4)=f4; X(i,5)=f5;
                    X(i,6)=f6; break;
            case 8:  fscanf(sourcestream2,"%f %f %f %f %f %f \n",&f2,&f3,&f4,&f5,&f6,
                                                                &f7);
                    X(i,1)=1.000000; X(i,2)=f2; X(i,3)=f3; X(i,4)=f4; X(i,5)=f5;
                    X(i,6)=f6; X(i,7)=f7; break;
            case 9:  fscanf(sourcestream2,"%f %f %f %f %f %f %f \n",&f2,&f3,&f4,&f5,
                                                                   &f6,&f7,&f8);
                    X(i,1)=1.000000; X(i,2)=f2; X(i,3)=f3; X(i,4)=f4; X(i,5)=f5;
                    X(i,6)=f6; X(i,7)=f7; X(i,8)=f8; break;
            case 10: fscanf(sourcestream2,"%f %f %f %f %f %f %f %f \n",&f2,&f3,&f4,
                                                                  &f5,&f6,&f7,&f8,&f9);
                    X(i,1)=1.000000; X(i,2)=f2; X(i,3)=f3; X(i,4)=f4; X(i,5)=f5;
                    X(i,6)=f6; X(i,7)=f7; X(i,8)=f8; X(i,9)=f9; break;
         }
      }
      fclose(sourcestream2);
   }

   // GM steps should be used
   if (USEGM==1) gibbs(TTTT,NCOV,NNCP,ITER,BURN,1,ITER,1);
   // no GM steps should be used
   else          gibbs(TTTT,NCOV,NNCP,ITER,BURN,ITER+1,ITER,1);

   return(1);
}
```

Appendix C

Implementation of GM-MGMC Sampler for OSV and OSVt Model

Here we provide the C++ implementation of the GM-MGMC sampler for both the OSV and the OSVt model. Before compiling the program, one must adapt some file names and constants to the data. This must be done in the section which is marked by 'Change only this section'. More informations can be seen directly in the program code which contains many detailed comments. At the beginning the two header-files 'randomGM.h' and 'matrixGM.h' are included. These can be found in Appendix D and E, respectively, and must be copied into the same directory as the following program.

```
//-------------------------------------------------------------------------------
//        GM-MGMC SAMPLER FOR OSV AND OSVt MODEL --- by GERNOT MUELLER, 2004
//-------------------------------------------------------------------------------
// Change program only in the marked section below !
//
// Steps needed before executing the program:
// - adapt the program code to your data and files (only in marked section!)
// - save the program (osvt.cpp)
// - compile the program (e.g. using: CC -O4 osvt.cpp)
// - you will get an executable 'a.out' file
// - type a.out to execute
//
// Maximal number of X-covariates:   10 (including intercept)
// Maximal number of Z-covariates:    6
// Maximal number of cutpoints:       6
// Maximal length of data:        60000
//-------------------------------------------------------------------------------

#include<stdio.h>
#include<stdlib.h>
#include<iostream.h>
#include<math.h>
#include"matrixGM.h"
```

```
#include"randomGM.h"

//-------------------------------------------------------------------------------//
// function declarations

double func(double x1=0.00000, double x2=0.00000, double x3=0.00000,
            double x4=0.00000, double x5=0.00000, double x6=0.00000,
            double x7=0.00000, double x8=0.00000, double x9=0.00000,
            double x10=0.00000);
void neldermead(unsigned int n, double xstart1=0.00000, double xstart2=0.00000,
                double xstart3=0.00000, double xstart4=0.00000,
                double xstart5=0.00000, double xstart6=0.00000,
                double xstart7=0.00000, double xstart8=0.00000,
                double xstart9=0.00000, double xstart10=0.00000,
                unsigned int steps=100, double precision=0.0000001,
                unsigned int viewsteps=0);
void simu(double mu, double phi, double sigma1);
double impactX(void);
void draw_CP(void);
void draw_Y(void);
void draw_s(void);
void draw_nu(void);
void draw_LAMBDA(void);
void draw_H(void);
void draw_BETA(void);
void Hesse(unsigned int n, double griddist=0.0001, double x1=0.00000,
           double x2=0.00000, double x3=0.00000, double x4=0.00000,
           double x5=0.00000, double x6=0.00000, double x7=0.00000,
           double x8=0.00000, double x9=0.00000, double x10=0.00000);
double minimum(double x, double y);

//-------------------------------------------------------------------------------//
//-------------------------------------------------------------------------------//
//                                                                               //
//               C H A N G E   O N L Y   T H I S   S E C T I O N                 //
//                                                                               //
//-------------------------------------------------------------------------------//
//-------------------------------------------------------------------------------//
//                             DATA AND MODEL PARAMETERS                         //
//-------------------------------------------------- ----------------------------//
#define TTTT 22689         // length of data                                      //
#define NNNN 4             // number of parameters to estimate in log-vola-equation//
                           // length of theta = (alpha_1,...,alpha_NCOVZ,phi,sigma)//
                           // = NCOVZ + 2                                         //
#define NCOVX 2            // length of covariate vector X (including intercept)  //
#define NCOVZ 2            // length of covariate vector Z (Z never contains an   //
                           // intercept!)                                         //
#define NMATRIX 2          // set to 1 if NCOVZ=0, else set to NCOVZ              //
#define NNCP 6             // number of response categories minus 1               //
#define ITER 4000          // number of iterations to do                          //
#define BURNIN 1000        // number of burnin-steps                              //
#define TAKENORMALERRORS 1 // if 1, normal errors, if 0 t errors are assumed      //
#define USESIMULATION 0    // if real data is to be estimated, type 0, otherwise 1 //
//-------------------------------------------------------------------------------//
//                             PARAMETERS FOR SIMULATION                         //
//-------------------------------------------------------------------------------//
#define SIMUPHI      0.90  //                                                     //
#define SIMUSIGMA    0.20  //                                                     //
                           //                                                     //
#define SIMUBETA0    3.50  //                                                     //
#define SIMUBETA1   -0.30  //                                                     //
```

```
#define SIMUBETA2   -1.60  //                                                  //
#define SIMUBETA3    0.50  //                                                  //
#define SIMUBETA4    0.00  //                                                  //
#define SIMUBETA5    0.00  //                                                  //
#define SIMUBETA6    0.00  //                                                  //
#define SIMUBETA7    0.00  //                                                  //
#define SIMUBETA8    0.00  //                                                  //
#define SIMUBETA9    0.00  //                                                  //
                           //                                                  //
#define SIMUALPHA1   0.25  //                                                  //
#define SIMUALPHA2   0.15  //                                                  //
#define SIMUALPHA3   0.00  //                                                  //
#define SIMUALPHA4   0.00  //                                                  //
#define SIMUALPHA5   0.00  //                                                  //
#define SIMUALPHA6   0.00  //                                                  //
                           //                                                  //
#define SIMUCUTP1    0.00  //                                                  //
#define SIMUCUTP2    0.90  //                                                  //
#define SIMUCUTP3    1.80  //                                                  //
#define SIMUCUTP4    2.75  //                                                  //
#define SIMUCUTP5    3.65  //                                                  //
#define SIMUCUTP6    4.50  //                                                  //
                           // covariates which are to be used in simuation must be //
                           // defined in function simu(...)!                   //
//-------------------------------------------------------------------------------//
//                               STARTING VALUES                                 //
//-------------------------------------------------------------------------------//
#define PHISTART     0.85  //                                                  //
#define SIGMASTART   0.30  //                                                  //
#define NUSTART      10    //                                                  //
                           //                                                  //
#define ALPHA1START  0.00  //                                                  //
#define ALPHA2START  0.00  //                                                  //
#define ALPHA3START  0.00  //                                                  //
#define ALPHA4START  0.00  //                                                  //
#define ALPHA5START  0.00  //                                                  //
#define ALPHA6START  0.00  //                                                  //
                           //                                                  //
#define CP2START     1.00  //                                                  //
#define CP3START     2.00  //                                                  //
#define CP4START     3.00  //                                                  //
#define CP5START     4.00  //                                                  //
#define CP6START     5.00  //                                                  //
//-------------------------------------------------------------------------------//
//                            HYPERPARAMETERS FOR PRIORS                         //
//-------------------------------------------------------------------------------//
#define BETAVARIANCE 10.0000 // each beta_j  has prior N(0,BETAVARIANCE)          //
#define ALPHABOUND   100.000 // each alpha_j has pr. Unif(-ALPHABOUND,ALPHABOUND) //
#define SIGMABOUND   10.0000 // sigma has prior Unif(0,SIGMABOUND)               //
//-------------------------------------------------------------------------------//
//                              INPUT AND OUTPUT FILES                           //
//-------------------------------------------------------------------------------//
char procsource[]="ibm9daysproc.txt";   // input file for response               //
char desgX[]="ibm9daysxdesg.txt";       // input file for design X (tab-seperated) //
char desgZ[]="ibm9dayszdesgCENT.txt";   // input file for design Z (tab-seperated) //
                                        //                                      //
char paraOUT[]="osvtPARA.dat";          // output file for parameter estimates  //
char volasOUT[]="osvtVOLAS.dat";        // output file for est. volatilities    //
char logvolasOUT[]="osvtLOGVOLAS.dat"; // output file for est. log-volatilities //
                                        //                                      //
char procSIMlatent[]="osvtY.dat";       // output file for lat. variables in sim. //
```

```
char procSIM[]="osvtYOBS.dat";          // output file for simulated process       //
//-------------------------------------------------------------------------------//
//-------------------------------------------------------------------------------//
//                                ADVANCED OPTIONS                               //
//-------------------------------------------------------------------------------//
#define MUFIX -0.60         // model parameter mu must be fixed for identifiab.    //
#define GMSTEPS 1           // if set to 1, program uses GM-steps; if 0, not       //
#define GMSTART 0           // GM-steps will be used only after iteration GMSTART  //
#define NUDRAWSTART 50      // nu will be drawn only after iteration NUDRAWSTART   //
#define PHIDRAWSTART 0      // phi will be drawn only after iteration PHIDRAWSTART //
#define USEPSEUDOAUTOREGR 1 // see function SIMU() !                               //
#define USECHANGEDHTEQUATION 1 // if 1, covariates Z only have impact on present   //
#define DFDF 15             // degr. of freedom of t-distr. (only for simulation)  //
#define PRECISION 0.0000001 // for minimization-algorithm                          //
#define MAXMINSTEPS 200     // for minimization-algorithm                          //
#define GRID 0.001          // grid distance for computation of the Hessian matrix //
#define CONTROLONSCREEN 0   // if 1, current control info is printed on screen     //
#define SCREENUPDATEINTERVAL 2 // used only if CONTROLONSCREEN equals 0            //
//-------------------------------------------------------------------------------//
//-------------------------------------------------------------------------------//
//-------------------------------------------------------------------------------//


//-------------------------------------------------------------------------------
// Declaration of variables and matrices

Matrix Z(TTTT,NMATRIX);
Matrix ALPHAest(NMATRIX,1);

double min[21];
double minhist[21][ITER+2];
double alphahist[11][ITER+2];
double avg[21];
double avgalpha[11];
double h00=0.00000, p00=0.00000;
double H[TTTT+2];
double LAMBDA[TTTT+2];
double U[TTTT+2];
double Y[TTTT+2];
double Yest[TTTT+2];
unsigned int YOBS[TTTT+2];
double CP[10];
double CPest[10];
Matrix X(TTTT,NCOVX);
Matrix BETA(12,1);
Matrix BETAest(NCOVX,1);
Matrix UNIT(NCOVX,NCOVX);
Matrix XT(NCOVX,1);
Matrix INTER1(NCOVX,NCOVX);
Matrix INTER2(NCOVX,NCOVX);
Matrix INTER3(NCOVX,NCOVX);
Matrix INTER3INV(NCOVX,NCOVX);
Matrix INTER4(NCOVX,1);
Matrix INTER5(NCOVX,1);
Matrix ALPHA(12,1);
double YAST[TTTT+2];
double NORMAL[TTTT+2];
double HESSE[11][11];
unsigned int sest[TTTT+2];
double Hest[TTTT+2];
double LogVolaEST[TTTT+2];
```

```
double VolaEST[TTTT+2];
double LAMBDAest[TTTT+2];
unsigned int nuest=NUSTART;
double phiest=0.90000, sigma1est=0.20000, accnu=0.00000;
double mufix=MUFIX;
double htt[TTTT+2], httm1[TTTT+2], kt[TTTT+2], fttm1[TTTT+2], pttm1[TTTT+2];
double ptt[TTTT+2], et[TTTT+2],nt[TTTT+2],rt[TTTT+2],ut[TTTT+2],dt[TTTT+2];
double ct[TTTT+2], zetat[TTTT+2],bt[TTTT+2];

Matrix HESSEM(NNNN,NNNN);
Matrix HESSEINV(NNNN,NNNN);
Matrix MU(NNNN,1);
Matrix MVN(NNNN,1);
Matrix THETA(NNNN,1);

unsigned int ACTUALITER=0;
double PI   = 3.14159265358979;

//--------------------------------------------------------------------------------
// constants for the seven-component mixture approximation

double qmix[9]     ={ 0.00000,  0.00730, 0.10556, 0.00002, 0.04395, 0.34001, 0.24566,
                      0.25750,  0.00000};
double qmixacc[9]  ={ 0.00000,  0.00730, 0.11286, 0.11288, 0.15683, 0.49684, 0.74250,
                      1.00000,  0.00000};
double mumix[9]    ={ 0.00000,-11.40039,-5.24321,-9.83726, 1.50746,-0.65098, 0.52478,
                     -2.35859,  0.00000};
double sigma2mix[9]={ 0.00000,  5.79596, 2.61369, 5.17950, 0.16735, 0.64009, 0.34023,
                      1.26261,  0.00000};
double sigmamix[9] ={ 0.00000,  2.40748, 1.61669, 2.27585, 0.40908, 0.80006, 0.58329,
                      1.12366, 0.00000};

//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------

int main(void)
{
   // Declaration of variables
   int i,j,k,l,ncovz;
   ncovz=NCOVZ;
   double frac1=0.00000, frac2=0.00000, acceptprob=0.50000;
   double interc;
   double dgm1=0.00000, dgm2=0.00000, dgm3=0.00000, dgmgamma=0.00000;
   FILE *stream;
   FILE *procstream;
   FILE *proc;
   FILE *xdesg;
   FILE *zdesg;
   unsigned int ui1;
   double f1,f2,f3,f4;
   for (i=1; i<=TTTT; i++) LogVolaEST[i]=0.000000;
   for (i=1; i<=TTTT; i++) VolaEST[i]=0.000000;
   FILE *LogVolas;
   FILE *Volas;

   // for fast evaluation of student-t density -> see random.h
   savegammafactors();

   // generate unit matrix
   for (i=1; i<=NCOVX; i++) for (k=1; k<=NCOVX; k++) UNIT(i,k)=0.00000;
```

```
for (i=1; i<=NCOVX; i++) UNIT(i,i)=1.00000;

// open output file and generate head line
stream = fopen(paraOUT,"w+");
if (TAKENORMALERRORS==1) fprintf(stream,"phi \t sigma \t ");
else                     fprintf(stream,"phi \t sigma \t nu \t ");

for (i=0; i<NCOVX-1; i++) fprintf(stream,"beta%d \t ",i);
if (NCOVZ==0 && NNCP<2)
{
   if (GMSTEPS==1)
   {
      fprintf(stream,"beta%d \t",NCOVX-1);
      fprintf(stream,"gamma \n");
   }
   else
      fprintf(stream,"beta%d \n",NCOVX-1);
}
else
{
   fprintf(stream,"beta%d \t",NCOVX-1);
   if (NCOVZ==0)
   {
      for (i=2; i<NNCP; i++) fprintf(stream,"cutpoint%d \t",i);
      if (GMSTEPS==1)
      {
         fprintf(stream,"cutpoint%d \t",NNCP);
         fprintf(stream,"gamma \n");
      }
      else
         fprintf(stream,"cutpoint%d \n",NNCP);
   }
   if (NNCP<2)
   {
      for (i=1; i<NCOVZ; i++) fprintf(stream,"alpha%d \t",i);
      if (GMSTEPS==1)
      {
         fprintf(stream,"alpha%d \t",NCOVZ);
         fprintf(stream,"gamma \n");
      }
      else
         fprintf(stream,"alpha%d \n",NCOVZ);
   }
   if (NCOVZ>0 && NNCP>=2)
   {
      for (i=1; i<=NCOVZ; i++) fprintf(stream,"alpha%d \t ",i);
      for (i=2; i<NNCP; i++)   fprintf(stream,"cutpoint%d \t",i);
      if (GMSTEPS==1)
      {
         fprintf(stream,"cutpoint%d \t",NNCP);
         fprintf(stream,"gamma \n");
      }
      else
         fprintf(stream,"cutpoint%d \n",NNCP);
   }
}

// set random number generator on a random position
randomize();

if (USESIMULATION==1)
```

```
{
   // simulate process
   simu(MUFIX,SIMUPHI,SIMUSIGMA);
   // write simulated process in corresponding file
   procstream = fopen(procSIM,"w+");
   for (i=1; i<=TTTT; i++) fprintf(procstream,"%d \n",YOBS[i]);
   fclose(procstream);
}
else
{
   // read input file which contains the observations
   proc  = fopen(procsource,"r");
   for (i=1; i<=TTTT; i++)
   {
      fscanf(proc,"%d\n",&ui1);
      YOBS[i]=ui1;
   }
   fclose(proc);

   // read input file which contains x-design
   xdesg = fopen(desgX,"r");
   for (i=1; i<=TTTT; i++)
   {
      for (k=1; k<NCOVX; k++)
      {
         fscanf(xdesg,"%lf\t",&f1);
         if (k==1) X(i,k)=1.000000;
         else      X(i,k)=f1;
      }
      fscanf(xdesg,"%lf\n",&f1);
      X(i,NCOVX)=f1;
   }
   fclose(xdesg);

   // read input file which contains z-design
   zdesg = fopen(desgZ,"r");
   for (i=1; i<=TTTT; i++)
   {
      for (k=1; k<NCOVZ; k++)
      {
         fscanf(zdesg,"%lf\t",&f3);
         Z(i,k)=f3;
      }
      fscanf(zdesg,"%lf\n",&f3);
      Z(i,NCOVZ)=f3;
   }
   fclose(zdesg);
}
//-----------------------------------------------------
// initialize parameters and variables

// initialize LAMBDAest
if (TAKENORMALERRORS==0)
   for (i=1; i<=TTTT; i++)
      LAMBDAest[i]=rand_gamma(0.50000*NUSTART,0.50000*NUSTART);
else
   for (i=1; i<=TTTT; i++)
      LAMBDAest[i]=1.00000;

// initialize Hest
Hest[0]=mufix;
```

```
for (i=1; i<=TTTT; i++)
{
   interc=0.00000;
   if (NCOVZ>0) interc += ALPHA1START*Z(i,1);
   if (NCOVZ>1) interc += ALPHA2START*Z(i,2);
   if (NCOVZ>2) interc += ALPHA3START*Z(i,3);
   if (NCOVZ>3) interc += ALPHA4START*Z(i,4);
   if (NCOVZ>4) interc += ALPHA5START*Z(i,5);
   if (NCOVZ>5) interc += ALPHA6START*Z(i,6);
   Hest[i]=mufix+interc+PHISTART*(Hest[i-1]-mufix)+
                        SIGMASTART*normal(0.00000,1.00000);
}

// initialize CPest
CPest[1]=0.000000; // this cutpoint is fixed for reasons of identifiability!
CPest[2]=CP2START;
CPest[3]=CP3START;
CPest[4]=CP4START;
CPest[5]=CP5START;
CPest[6]=CP6START;

// initialize Yest
for (i=1; i<=TTTT; i++)
{
   switch (YOBS[i])
   {
      case 1: Yest[i] = -0.500000; break;
      case 2: Yest[i] = 0.500000*(CPest[1]+CPest[2]); break;
      case 3: Yest[i] = 0.500000*(CPest[2]+CPest[3]); break;
      case 4: Yest[i] = 0.500000*(CPest[3]+CPest[4]); break;
      case 5: Yest[i] = 0.500000*(CPest[4]+CPest[5]); break;
      case 6: Yest[i] = 0.500000*(CPest[5]+CPest[6]); break;
      case 7: Yest[i] = CPest[6]+0.500000; break;
   }
}

//-----------------------------------------------------
// Gibbs sampler loop

for (i=1; i<=ITER; i++)
{
   ACTUALITER=i;
   if (CONTROLONSCREEN==1) printf("\n");
   if (CONTROLONSCREEN==1) printf("iteration: %d \n",i);
   else
      if (ACTUALITER%SCREENUPDATEINTERVAL==0) printf("iteration: %d \n",i);
   //-------------------------------------------
   // beta-update

   if (CONTROLONSCREEN==1) printf("draw_BETA\n");
   draw_BETA();

   //-------------------------------------------
   // y_t^\ast-update

   if (CONTROLONSCREEN==1) printf("draw_Y\n");
   draw_Y();

   //-------------------------------------------
   // cupoint-update
```

```
if (CONTROLONSCREEN==1) printf("draw_CP\n");
draw_CP();

//-------------------------------------------
// GM-STEP

dgmgamma = 1.000000;
if (GMSTEPS==1 && ACTUALITER>GMSTART)
{
   dgm1=0.50000*(TTTT+NNCP+NCOVX);   // NNCP+1 = K; NCOVX-1 = p,
                                     // therefore K+p = NNCP+NCOVX;  now dgm1=a

   dgm2=0.00000;
   for (k=1; k<=NCOVX; k++)
      dgm2 += BETAest(k,1)*BETAest(k,1)*(1.00000/BETAVARIANCE);

   for (k=1; k<=TTTT; k++)
   {
      dgm3=0.00000;
      for (j=1; j<=NCOVX; j++)  dgm3 += BETAest(j,1)*X(k,j); //   dgm3=x_k'beta
      dgm2 += (Yest[k]-dgm3)*(Yest[k]-dgm3)*LAMBDAest[k]/exp(Hest[k]);
   }

   dgm2 = 0.50000*dgm2; // now dgm2=b;

   dgmgamma=rand_gamma(dgm1,dgm2);
   dgmgamma=sqrt(dgmgamma);
   if (dgmgamma>1.050000) dgmgamma=1.050000;
   if (dgmgamma<0.952381) dgmgamma=0.952381;

   if (CONTROLONSCREEN==1) printf("GAMMA: %f\n",dgmgamma);

   for (k=1; k<=NCOVX; k++)  BETAest(k,1)= dgmgamma*BETAest(k,1);
   for (k=2; k<=NNCP;  k++)  CPest[k]    = dgmgamma*CPest[k];
   for (k=1; k<=TTTT;  k++)  Yest[k]     = dgmgamma*Yest[k];

}

//-------------------------------------------
// mixture index update

if (CONTROLONSCREEN==1) printf("draw_s\n");
draw_s();

//-------------------------------------------
// MH-step for update of phi, sigma, alpha_j, j=1,...,q

if (CONTROLONSCREEN==1) printf("draw_THETA\n");
// Search the argument, which maximizes log(g(theta)),
// that is, which minimizes -log(g(theta))
if (i==1)
{
   switch (ncovz)
   {
      case 0: neldermead(NNNN,PHISTART,SIGMASTART); break;
      case 1: neldermead(NNNN,PHISTART,SIGMASTART,ALPHA1START); break;
      case 2: neldermead(NNNN,PHISTART,SIGMASTART,ALPHA1START,ALPHA2START);
              break;
      case 3: neldermead(NNNN,PHISTART,SIGMASTART,ALPHA1START,ALPHA2START,
                         ALPHA3START); break;
      case 4: neldermead(NNNN,PHISTART,SIGMASTART,ALPHA1START,ALPHA2START,
```

```
                        ALPHA3START,ALPHA4START); break;
      case 5: neldermead(NNNN,PHISTART,SIGMASTART,ALPHA1START,ALPHA2START,
                        ALPHA3START,ALPHA4START,ALPHA5START); break;
      case 6: neldermead(NNNN,PHISTART,SIGMASTART,ALPHA1START,ALPHA2START,
                        ALPHA3START,ALPHA4START,ALPHA5START,ALPHA6START);
              break;
   }
}
else
{
   switch (ncovz)
   {
      case 0: neldermead(NNNN,THETA(1,1),THETA(2,1)); break;
      case 1: neldermead(NNNN,THETA(1,1),THETA(2,1),THETA(3,1)); break;
      case 2: neldermead(NNNN,THETA(1,1),THETA(2,1),THETA(3,1),THETA(4,1));
              break;
      case 3: neldermead(NNNN,THETA(1,1),THETA(2,1),THETA(3,1),THETA(4,1),
                        THETA(5,1)); break;
      case 4: neldermead(NNNN,THETA(1,1),THETA(2,1),THETA(3,1),THETA(4,1),
                        THETA(5,1),THETA(6,1)); break;
      case 5: neldermead(NNNN,THETA(1,1),THETA(2,1),THETA(3,1),THETA(4,1),
                        THETA(5,1),THETA(6,1),THETA(7,1)); break;
      case 6: neldermead(NNNN,THETA(1,1),THETA(2,1),THETA(3,1),THETA(4,1),
                        THETA(5,1),THETA(6,1),THETA(7,1),THETA(8,1)); break;
   }
}

for (k=1; k<=2; k++)      minhist[k][i]  = min[k];
for (k=1; k<=NCOVZ; k++)  alphahist[k][i]= min[2+k];

// Compute the Hessian at the minimum
switch (ncovz)
{
   case 0: Hesse(NNNN,GRID,min[1],min[2]); break;
   case 1: Hesse(NNNN,GRID,min[1],min[2],min[3]); break;
   case 2: Hesse(NNNN,GRID,min[1],min[2],min[3],min[4]); break;
   case 3: Hesse(NNNN,GRID,min[1],min[2],min[3],min[4],min[5]); break;
   case 4: Hesse(NNNN,GRID,min[1],min[2],min[3],min[4],min[5],min[6]); break;
   case 5: Hesse(NNNN,GRID,min[1],min[2],min[3],min[4],min[5],min[6],min[7]);
           break;
   case 6: Hesse(NNNN,GRID,min[1],min[2],min[3],min[4],min[5],min[6],min[7],
                 min[8]); break;
}

// Copy the arrays into matrices
for (k=1; k<=NNNN; k++) MU(k,1)=min[k];
for (k=1; k<=NNNN; k++) for (l=1; l<=NNNN; l++) HESSEM(k,l)=HESSE[k][l];
if (CONTROLONSCREEN==1)
   printf("Hessian matrix pos.def.: %d \n",HESSEM.posdef());

// Draw a proposal MVN from the multivariate normal distribution:
HESSEINV=HESSEM.invertPosDef();
MVN=mvnormal(MU,HESSEINV);
if (CONTROLONSCREEN==1)
{
   printf("min phi  : %f \n",MU(1,1));
   printf("min sigma: %f \n",MU(2,1));
   if (NCOVZ>0) printf("min alpha1: %f \n",MU(3,1));
   if (NCOVZ>1) printf("min alpha2: %f \n",MU(4,1));
   if (NCOVZ>2) printf("min alpha3: %f \n",MU(5,1));
   if (NCOVZ>3) printf("min alpha4: %f \n",MU(6,1));
```

```
   if (NCOVZ>4) printf("min alpha5: %f \n",MU(7,1));
   if (NCOVZ>5) printf("min alpha6: %f \n",MU(8,1));
}

// We need theta_0 for computing the acceptance probability
if (i==1)  for (k=1; k<=NNNN; k++) THETA(k,1)=min[k];

// Compute the acceptance probability
switch (ncovz)
{
   case 0: frac1 = exp(-func(MVN(1,1),MVN(2,1))
                       +func(THETA(1,1),THETA(2,1))); break;
   case 1: frac1 = exp(-func(MVN(1,1),MVN(2,1),MVN(3,1))
                       +func(THETA(1,1),THETA(2,1),THETA(3,1))); break;
   case 2: frac1 = exp(-func(MVN(1,1),MVN(2,1),MVN(3,1),MVN(4,1))
                       +func(THETA(1,1),THETA(2,1),THETA(3,1),THETA(4,1)));
           break;
   case 3: frac1 = exp(-func(MVN(1,1),MVN(2,1),MVN(3,1),MVN(4,1),MVN(5,1))
                       +func(THETA(1,1),THETA(2,1),THETA(3,1),THETA(4,1),
                             THETA(5,1)));
           break;
   case 4: frac1 = exp(-func(MVN(1,1),MVN(2,1),MVN(3,1),MVN(4,1),MVN(5,1),
                             MVN(6,1))
                       +func(THETA(1,1),THETA(2,1),THETA(3,1),THETA(4,1),
                             THETA(5,1),THETA(6,1)));
           break;
   case 5: frac1 = exp(-func(MVN(1,1),MVN(2,1),MVN(3,1),MVN(4,1),MVN(5,1),
                             MVN(6,1),MVN(7,1))
                       +func(THETA(1,1),THETA(2,1),THETA(3,1),THETA(4,1),
                             THETA(5,1),THETA(6,1),THETA(7,1)));
           break;
   case 6: frac1 = exp(-func(MVN(1,1),MVN(2,1),MVN(3,1),MVN(4,1),MVN(5,1),
                             MVN(6,1),MVN(7,1),MVN(8,1))
                       +func(THETA(1,1),THETA(2,1),THETA(3,1),THETA(4,1),
                             THETA(5,1),THETA(6,1),THETA(7,1),THETA(8,1)));
           break;
}
frac2 = undmvnormal(THETA,MU,HESSEINV)/undmvnormal(MVN,MU,HESSEINV);

acceptprob=minimum(1.00000,frac1*frac2);
if (CONTROLONSCREEN==1) printf("ACCEPT-PROBABILITY: %f \n",acceptprob);
minhist[4][i]=acceptprob;

// if we accept the proposal MVN, theta changes
if (uniform(0,1)<=acceptprob)  for (k=1; k<=NNNN; k++) THETA(k,1)=MVN(k,1);

if (ACTUALITER<=PHIDRAWSTART) THETA(1,1)=PHISTART;

//--------------------------------------------
// update of log-volatilities

if (CONTROLONSCREEN==1) printf("draw_H\n");
draw_H();

// estimate the log-volatilities and write output-file at iteration ITER
if (i>BURNIN)
{
   for (k=1; k<=TTTT; k++) LogVolaEST[k] += Hest[k];
}
if (i==ITER)
{
```

```
   for (k=1; k<=TTTT; k++) LogVolaEST[k]=LogVolaEST[k]/(1.00000*(ITER-BURNIN));
   LogVolas = fopen(logvolasOUT,"w+");
   fprintf(LogVolas,"logVolatilities \n");
   for (k=1; k<=TTTT; k++) fprintf(LogVolas,"%f \n",LogVolaEST[k]);
   fclose(LogVolas);
}

// estimate the volatilities and write output-file at iteration ITER
if (i>BURNIN)
{
   for (k=1; k<=TTTT; k++) VolaEST[k] += exp(Hest[k]+0.60000);
}
if (i==ITER)
{
   for (k=1; k<=TTTT; k++) VolaEST[k] = VolaEST[k]/(1.00000*(ITER-BURNIN));
   Volas = fopen(volasOUT,"w+");
   fprintf(Volas,"NormalizedVolatilities \n");
   for (k=1; k<=TTTT; k++) fprintf(Volas,"%f \n",VolaEST[k]);
   fclose(Volas);
}

//--------------------------------------------
// nu-update (only for OSVt-model)

if (ACTUALITER<=NUDRAWSTART || TAKENORMALERRORS==1) nuest=NUSTART;
else
{
   if (CONTROLONSCREEN==1) printf("draw_nu\n");
   draw_nu();
}
minhist[5][ACTUALITER]=(double) nuest;
minhist[6][ACTUALITER]=accnu;

//--------------------------------------------
// lamda-update  (only for OSVt-model)

if (TAKENORMALERRORS==0)
{
   if (CONTROLONSCREEN==1) printf("draw_LAMBDA\n");
   draw_LAMBDA();
}
else
{
   for (k=1; k<=TTTT; k++) LAMBDAest[k]=1.00000;
}

//--------------------------------------------
// write output-file which contains the parameter estimates

if (TAKENORMALERRORS==1)
   fprintf(stream,"%f \t %f \t ",THETA(1,1),THETA(2,1));
else
   fprintf(stream,"%f \t %f \t %d \t ",THETA(1,1),THETA(2,1),nuest);
for (k=1; k<NCOVX; k++) fprintf(stream,"%f \t ",BETAest(k,1));
if (NCOVZ==0 && NNCP<2)
{
   if (GMSTEPS==1)
   {
      fprintf(stream,"%f \t", BETAest(NCOVX,1));
      fprintf(stream,"%f \n", dgmgamma);
   }
```

```
         else
            fprintf(stream,"%f \n", BETAest(NCOVX,1));
      }
      else
      {
         fprintf(stream,"%f \t",BETAest(NCOVX,1));
         if (NCOVZ==0)
         {
            for (k=2; k<NNCP; k++) fprintf(stream,"%f \t",CPest[k]);
            if (GMSTEPS==1)
            {
               fprintf(stream,"%f \t",CPest[NNCP]);
               fprintf(stream,"%f \n",dgmgamma);
            }
            else
               fprintf(stream,"%f \n",CPest[NNCP]);
         }
         if (NNCP<2)
         {
            for (k=1; k<NCOVZ; k++) fprintf(stream,"%f \t ",THETA(2+k,1));
            if (GMSTEPS==1)
            {
               fprintf(stream,"%f \t",THETA(2+NCOVZ,1));
               fprintf(stream,"%f \n",dgmgamma);
            }
            else
               fprintf(stream,"%f \n",THETA(2+NCOVZ,1));
         }
         if (NCOVZ>0 && NNCP>=2)
         {
            for (k=1; k<=NCOVZ; k++) fprintf(stream,"%f \t ",THETA(2+k,1));
            for (k=2; k<NNCP; k++)   fprintf(stream,"%f \t",CPest[k]);
            if (GMSTEPS==1)
            {
               fprintf(stream,"%f \t",CPest[NNCP]);
               fprintf(stream,"%f \n",dgmgamma);
            }
            else
               fprintf(stream,"%f \n",CPest[NNCP]);
         }
      }
   }

   // write results on screen
   for (i=BURNIN+1; i<=ITER; i++)
      for (k=1; k<=6+NCOVX; k++)    avg[k]      += minhist[k][i];
   for (i=BURNIN+1; i<=ITER; i++)
      for (k=1; k<=NCOVZ; k++)      avgalpha[k] += alphahist[k][i];
   printf("\n\nPosterior mean estimates:\n");
   printf("phi   :\t%f\n",avg[1]/(ITER-BURNIN));
   printf("sigma :\t%f\n",avg[2]/(ITER-BURNIN));
   if (TAKENORMALERRORS==0) printf("nu    :\t%f\n",avg[5]/(ITER-BURNIN));
   for (i=1; i<=NCOVX; i++) printf("beta%d :\t%f\n",i-1,avg[6+i]/(ITER-BURNIN));
   for (i=1; i<=NCOVZ; i++) printf("alpha%d:\t%f\n",i,avgalpha[i]/(ITER-BURNIN));
   printf("\nAverage of acceptance probabilities:\n");
   printf("theta:\t%f\n",avg[4]/(ITER-BURNIN));
   if (TAKENORMALERRORS==0) printf("nu   :\t%f\n",avg[6]/(ITER-BURNIN));
   printf("\n");

   fclose(stream);
}
```

```
//------------------------------------------------------------------------------

void Hesse(unsigned int n, double griddist,
           double x1, double x2, double x3, double x4, double x5,
           double x6, double x7, double x8, double x9, double x10)
{
   int i,j,k;
   double x[11], xl[11], xr[11], xlo[11], xro[11], xlu[11], xru[11];
   double xx, xxr, xxl, xxro, xxlo, xxru, xxlu;
   x[1]=x1; x[2]=x2; x[3]=x3; x[4]=x4; x[5]=x5;
   x[6]=x6; x[7]=x7; x[8]=x8; x[9]=x9; x[10]=x10;

   for (i=1; i<=n; i++) for (k=1; k<=n; k++) HESSE[i][k]=0.00000;

   xx = func(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10);

   // Compute diagonal of matrix
   for (i=1; i<=n; i++)
   {
      for (k=1; k<=n; k++) xl[k] = x[k];
      for (k=1; k<=n; k++) xr[k] = x[k];
      xl[i] = x[i] - griddist;
      xr[i] = x[i] + griddist;
      xxl   = func(xl[1],xl[2],xl[3],xl[4],xl[5],xl[6],xl[7],xl[8],xl[9],xl[10]);
      xxr   = func(xr[1],xr[2],xr[3],xr[4],xr[5],xr[6],xr[7],xr[8],xr[9],xr[10]);
      HESSE[i][i] = (xxr-2*xx+xxl)/(griddist*griddist);
   }
   for (i=1; i<=n; i++)
   {
      for (j=1; j<=n; j++)
      {
         if (j!=i)
         {
            for (k=1; k<=n; k++) xlo[k] = x[k];
            for (k=1; k<=n; k++) xro[k] = x[k];
            for (k=1; k<=n; k++) xlu[k] = x[k];
            for (k=1; k<=n; k++) xru[k] = x[k];
            xlo[i] = xlo[i]-griddist;
            xlo[j] = xlo[j]+griddist;
            xro[i] = xro[i]+griddist;
            xro[j] = xro[j]+griddist;
            xlu[i] = xlu[i]-griddist;
            xlu[j] = xlu[j]-griddist;
            xru[i] = xru[i]+griddist;
            xru[j] = xru[j]-griddist;
            xxro   = func(xro[1], xro[2], xro[3], xro[4], xro[5],
                          xro[6], xro[7], xro[8], xro[9], xro[10]);
            xxlo   = func(xlo[1], xlo[2], xlo[3], xlo[4], xlo[5],
                          xlo[6], xlo[7], xlo[8], xlo[9], xlo[10]);
            xxru   = func(xru[1], xru[2], xru[3], xru[4], xru[5],
                          xru[6], xru[7], xru[8], xru[9], xru[10]);
            xxlu   = func(xlu[1], xlu[2], xlu[3], xlu[4], xlu[5],
                          xlu[6], xlu[7], xlu[8], xlu[9], xlu[10]);
            HESSE[i][j] = (xxro-xxlo-xxru+xxlu)/(4*griddist*griddist);
            HESSE[j][i] = HESSE[i][j];
         }
      }
   }
}
```

```
//--------------------------------------------------------------------------------

double func(double x1, double x2, double x3, double x4, double x5,
            double x6, double x7, double x8, double x9, double x10) // MinusLogG
{
   int k,t;
   double result  = 0.000000;
   double intert  = 0.000000;
   double intertm1= 0.000000;
   int alphajoutofbounds=0;

   double muF     = mufix;
   double phiF    = x1;
   double sigma1F = x2;

   double alpha[10];
   if (NCOVZ>0) alpha[1] = x3;
   if (NCOVZ>1) alpha[2] = x4;
   if (NCOVZ>2) alpha[3] = x5;
   if (NCOVZ>3) alpha[4] = x6;
   if (NCOVZ>4) alpha[5] = x7;
   if (NCOVZ>5) alpha[6] = x8;

   for (k=1; k<=NCOVZ; k++)
      if (alpha[k]<-ALPHABOUND || alpha[k]>ALPHABOUND) alphajoutofbounds=1;

   double sigma2F = sigma1F*sigma1F;
   double phi2F   = phiF*phiF;

   if (sigma1F<0.001 || sigma1F>SIGMABOUND ||
       phiF>=0.999 || phiF<=-0.999 || alphajoutofbounds==1)
      return (1000000000);
   else
   {
      htt[0]= muF;
      ptt[0]= sigma2F/(1.00000-phi2F);

      for (t=1; t<=TTTT; t++)
      {
         intert   = 0.00000;
         for (k=1; k<=NCOVZ; k++)  intert   += alpha[k]*Z(t,k);
         intertm1 = 0.00000;
         if (USECHANGEDHTEQUATION==1)
         {
            if (t>1) for (k=1; k<=NCOVZ; k++)  intertm1 += alpha[k]*Z(t-1,k);
         }

         httm1[t] = muF + intert + phiF*(htt[t-1]-muF-intertm1);
         pttm1[t] = phi2F*ptt[t-1] + sigma2F;

         fttm1[t] = pttm1[t] + sigma2mix[sest[t]];
         kt[t]    = pttm1[t]/fttm1[t];
         htt[t]   = httm1[t] + kt[t]*(YAST[t]-mumix[sest[t]]-httm1[t]);
         ptt[t]   = (1-kt[t])*pttm1[t];

         result   = result - log(fttm1[t]) - (YAST[t]-mumix[sest[t]]-httm1[t])*
                                             (YAST[t]-mumix[sest[t]]-httm1[t])/fttm1[t];
      }

      return (-0.50000*result);
   }
```

```
}

//-------------------------------------------------------------------------------

void simu(double mu, double phi, double sigma1)
{
   int t,i,k;
   double x0       = 0.00000;
   double z0       = 0.00000;
   double inter    = 0.00000;
   double interz   = 0.00000;
   double interzm1 = 0.00000;
   FILE *streamY;
   streamY = fopen(procSIMlatent,"w+");


   FILE *xde;
   FILE *zde;
   double f11,f12,f13,f14;

   // read given (simulated or real) files with covariates
   xde = fopen(desgX,"r");
   for (i=1; i<=TTTT; i++)
   {
      for (k=1; k<NCOVX; k++)
      {
         fscanf(xde,"%lf\t",&f11);
         if (k==1) X(i,k)=1.000000;
         else      X(i,k)=f11;
      }
      fscanf(xde,"%lf\n",&f11);
      X(i,NCOVX)=f11;
   }
   fclose(xde);

   zde = fopen(desgZ,"r");
   for (i=1; i<=TTTT; i++)
   {
      for (k=1; k<NCOVZ; k++)
      {
         fscanf(zde,"%lf\t",&f13);
         Z(i,k)=f13;
      }
      fscanf(zde,"%lf\n",&f13);
      Z(i,NCOVZ)=f13;
   }

   fclose(zde);

   // copy the true beta's in vector BETA
   BETA(1,1)=SIMUBETA0; BETA(2,1)=SIMUBETA1; BETA(3,1)=SIMUBETA2;
   BETA(4,1)=SIMUBETA3; BETA(5,1)=SIMUBETA4; BETA(6,1)=SIMUBETA5;
   BETA(7,1)=SIMUBETA6; BETA(8,1)=SIMUBETA7; BETA(9,1)=SIMUBETA8;
   BETA(10,1)=SIMUBETA9;

   // copy the true alpha's in vector ALPHA
   ALPHA(1,1)=SIMUALPHA1; ALPHA(2,1)=SIMUALPHA2; ALPHA(3,1)=SIMUALPHA3;
   ALPHA(4,1)=SIMUALPHA4; ALPHA(5,1)=SIMUALPHA5; ALPHA(6,1)=SIMUALPHA6;

   // copy the true cutpoints in vector ALPHA
   CP[1]=SIMUCUTP1; CP[2]=SIMUCUTP2; CP[3]=SIMUCUTP3;
```

```
   CP[4]=SIMUCUTP4; CP[5]=SIMUCUTP5; CP[6]=SIMUCUTP6;

   // generate the needed random variables
   for (t=1; t<=TTTT; t++)  NORMAL[t]=normal(0.00000,sigma1);
   for (t=1; t<=TTTT; t++)  U[t]     =normal(0,1);
   for (t=1; t<=TTTT; t++)  LAMBDA[t]=rand_gamma(DFDF/2,DFDF/2);
   if (TAKENORMALERRORS==1) for (t=1; t<=TTTT; t++)  LAMBDA[t]=1.00000;

   // generate process
   H[0]=mu;
   for (t=1; t<=TTTT; t++)
   {
      // compute the covariates' impact at time t
      inter=0.00000;
      if (USEPSEUDOAUTOREGR==1)
      {
         if (t==1) X(t,2)=4.00000;
         else      X(t,2)=(double) YOBS[t-1];
      }
      for (i=1; i<=NCOVX; i++)           inter    += BETA(i,1) *X(t,i);
      interz=0.00000;
      for (i=1; i<=NCOVZ; i++)           interz   += ALPHA(i,1)*Z(t,i);
      interzm1 = 0.00000;
      if (USECHANGEDHTEQUATION==1)
      {
         if (t>1) for (i=1; i<=NCOVZ; i++)  interzm1 += ALPHA(i,1)*Z(t-1,i);
      }

      // compute H[t] and Y[t] at time t
      H[t] = mu + interz + phi*(H[t-1]-mu-interzm1) + NORMAL[t];
      Y[t] = inter + exp(0.5000*H[t]) * U[t] * sqrt(LAMBDA[t]);

      if (Y[t]<CP[1]) YOBS[t]=1;
      else
         if (Y[t]<CP[2] || NNCP==1) YOBS[t]=2;
         else
            if (Y[t]<CP[3] || NNCP==2) YOBS[t]=3;
            else
               if (Y[t]<CP[4] || NNCP==3) YOBS[t]=4;
               else
                  if (Y[t]<CP[5] || NNCP==4) YOBS[t]=5;
                  else
                     if (Y[t]<CP[6] || NNCP==5) YOBS[t]=6;
                     else YOBS[t]=7;

      if (t<10 && CONTROLONSCREEN==1)
         printf("H[%d]: %f \t impact of alpha: %f \n",t,H[t],interz);
      fprintf(streamY,"%f \t %d \n",Y[t],YOBS[t]);
   }

   fclose(streamY);
}

//--------------------------------------------------------------------------------

double impactX(unsigned int t)
{
   int i;
   double inter=0.000000;
   if (NCOVX==0) return 0.000000;
   else
```

```
   {
      for (i=1; i<=NCOVX; i++) inter+=BETAest(i,1)*X(t,i);
      return inter;
   }
}

//-------------------------------------------------------------------------------

void draw_CP(void)
{
   int i,t;
   double left=0.000000, right=0.000000;
   for (i=2; i<=NNCP; i++)
   {
      left = CPest[i-1];
      if (i<NNCP) right = CPest[i+1]; else right=1000.0;
      for (t=1; t<=TTTT; t++)
      {
         if (YOBS[t]==i)   if (Yest[t]>left)  left  = Yest[t];
         if (YOBS[t]==i+1) if (Yest[t]<right) right = Yest[t];
      }
      CPest[i]=uniform(left,right);
      if (CONTROLONSCREEN==1)
         printf("CPest[%d] = %f \t left = %f \t right = %f \n",
                i,CPest[i],left,right);
   }
}

//-------------------------------------------------------------------------------

void draw_Y(void)
{
   unsigned int k,t;
   double interb=0.000000;
   for (t=1; t<=TTTT; t++)
   {
      interb=0.000000;
      for (k=1; k<=NCOVX; k++) interb += BETAest(k,1)*X(t,k);

      switch (YOBS[t])
      {
         case 1: Yest[t] = rs_trunc_normal(         0.00000, interb,
                                           sqrt(exp(Hest[t])/LAMBDAest[t])); break;
         case 2: Yest[t] = ds_trunc_normal(0.00000, CPest[2],interb,
                                           sqrt(exp(Hest[t])/LAMBDAest[t])); break;
         case 3: Yest[t] = ds_trunc_normal(CPest[2],CPest[3],interb,
                                           sqrt(exp(Hest[t])/LAMBDAest[t])); break;
         case 4: Yest[t] = ds_trunc_normal(CPest[3],CPest[4],interb,
                                           sqrt(exp(Hest[t])/LAMBDAest[t])); break;
         case 5: Yest[t] = ds_trunc_normal(CPest[4],CPest[5],interb,
                                           sqrt(exp(Hest[t])/LAMBDAest[t])); break;
         case 6: Yest[t] = ds_trunc_normal(CPest[5],CPest[6],interb,
                                           sqrt(exp(Hest[t])/LAMBDAest[t])); break;
         case 7: Yest[t] = ls_trunc_normal(CPest[6],         interb,
                                           sqrt(exp(Hest[t])/LAMBDAest[t])); break;
      }
   }
}

//-------------------------------------------------------------------------------
```

```
void draw_BETA(void)
{
   unsigned int i,k,t;
   double d1=0.00000,d2=0.00000;

   for (i=1; i<=NCOVX; i++) for (k=1; k<=NCOVX; k++) INTER1(i,k)=0.00000;
   for (i=1; i<=NCOVX; i++) for (k=1; k<=NCOVX; k++) INTER2(i,k)=0.00000;
   for (i=1; i<=NCOVX; i++) for (k=1; k<=NCOVX; k++) INTER3(i,k)=0.00000;
   for (i=1; i<=NCOVX; i++) INTER4(i,1)=0.00000;

   for (t=1; t<=TTTT; t++)
   {
      XT     = (X.tmat2(t,1,t,NCOVX)).transp();
      INTER2 = XT*XT.transp();
      d1     = LAMBDAest[t]/exp(Hest[t]);
      INTER3 = INTER3 + d1*INTER2;

      d2     = Yest[t]*d1;
      INTER4 = INTER4 + d2*XT;
   }
   INTER3 = INTER3 + (1.00000/BETAVARIANCE)*UNIT;
   // INTER 3 is just the matrix B now

   INTER3INV = INTER3.invertPosDef();
   INTER5    = INTER3INV*INTER4;

   BETAest   = mvnormal(INTER5,INTER3INV);

   for (i=1; i<=NCOVX; i++) minhist[6+i][ACTUALITER]=BETAest(i,1);
}

//-------------------------------------------------------------------------------

void draw_nu(void)
{
   unsigned int nnu,nustep,k,t;
   double dens[130];
   unsigned int expo[130];
   unsigned int maxfound=0;
   unsigned int proposal=150;
   double aa,bb,vv;
   double ffrac1=0.000000;
   double ffrac2=0.000000;
   int ediff=0;

   double result=1.00000;
   double resultold=1.00000;
   double resultnew=1.00000;



   // find approximate argmax for target density
   nnu=64;
   nustep=32;

   result=1.000000;
   dens[nnu]=0.000000;
   expo[nnu]=0;

   for (t=1; t<=TTTT; t++)
   {
```

```
   result*=dstudent(Yest[t],nnu,impactX(t),exp(0.500000*Hest[t]));
   // avoiding too small or too big results
   if (result<0.0100000) { result*=100.00000; expo[nnu]+=2; }
   if (result>100.00000) { result/=100.00000; expo[nnu]-=2; }
}
dens[nnu]=result;

while (maxfound==0)
{
   result=1.000000;
   dens[nnu-nustep]=0.000000;
   expo[nnu-nustep]=0;

   for (t=1; t<=TTTT; t++)
   {
      result*=dstudent(Yest[t],nnu-nustep,impactX(t),exp(0.500000*Hest[t]));
      // avoiding too small or too big results
      if (result<0.0100000) { result*=100.00000; expo[nnu-nustep]+=2; }
      if (result>100.00000) { result/=100.00000; expo[nnu-nustep]-=2; }
   }
   dens[nnu-nustep]=result;

   result=1.000000;
   dens[nnu+nustep]=0.000000;
   expo[nnu+nustep]=0;

   for (t=1; t<=TTTT; t++)
   {
      result*=dstudent(Yest[t],nnu+nustep,impactX(t),exp(0.500000*Hest[t]));
      // avoiding too small or too big results
      if (result<0.0100000) { result*=100.00000; expo[nnu+nustep]+=2; }
      if (result>100.00000) { result/=100.00000; expo[nnu+nustep]-=2; }
   }
   dens[nnu+nustep]=result;

   if (expo[nnu+nustep]<expo[nnu] ||
      (expo[nnu+nustep]==expo[nnu] && dens[nnu+nustep]>dens[nnu])) nnu+=nustep;
   else
      if (expo[nnu-nustep]<expo[nnu] ||
         (expo[nnu-nustep]==expo[nnu] && dens[nnu-nustep]>dens[nnu])) nnu-=nustep;

   if (nustep==1) maxfound=nnu;
   else nustep/=2;
}
if (CONTROLONSCREEN==1) printf("maxfound=%d\n",maxfound);

// use discretized gamma distr. with truncation to 0.5 - 127.5 as proposal density

while (proposal>127 || proposal<1)
{
   vv=0.2*maxfound*maxfound;
   if (maxfound<30) vv=0.10*maxfound*maxfound;
   if (maxfound<10) vv=0.05*maxfound*maxfound;
   bb=maxfound/(2*vv)+sqrt(maxfound*maxfound+4*vv)/2/vv;
   aa=bb*bb*vv;
   // mode of this gamma distribution is now maxfound
   proposal = (int) floor(rand_gamma(aa,bb)+0.50000);
}

// accept or not?
// compute posterior(proposal)/posterior(nuest)
```

```
   expo[nuest]=0;
   resultold=1.000000;

   for (t=1; t<=TTTT; t++)
   {
      resultold*=dstudent(Yest[t],nuest,impactX(t),exp(0.500000*Hest[t]));
      // avoiding too small or too big results
      if (resultold<0.0100000) { resultold*=100.00000; expo[nuest]+=2; }
      if (resultold>100.00000) { resultold/=100.00000; expo[nuest]-=2; }
   }

   expo[proposal]=0;
   resultnew=1.000000;

   for (t=1; t<=TTTT; t++)
   {
      resultnew*=dstudent(Yest[t],proposal,impactX(t),exp(0.500000*Hest[t]));
      // avoiding too small or too big results
      if (resultnew<0.0100000) { resultnew*=100.00000; expo[proposal]+=2; }
      if (resultnew>100.00000) { resultnew/=100.00000; expo[proposal]-=2; }
   }

   ediff=expo[nuest]-expo[proposal];
   ffrac1=resultnew/resultold;
   if (ediff<0) for (k=0; k<-ediff; k+=2) ffrac1*=0.010000;
   if (ediff>0) for (k=0; k<ediff;  k+=2) ffrac1*=100.0000;

   // compute proposal-density(nuest)/proposal-density(proposal)
   // via Newton-Cotes-formula
   ffrac2=unddisgamma(nuest,aa,bb)/unddisgamma(proposal,aa,bb);
   // via linear approximation
   // ffrac2=pow(1.00000*nuest/(1.00000*proposal),aa-1)*
   //        exp(bb*(-1.00000*nuest+1.00000*proposal));

   // compute acceptance probability
   accnu=minimum(1.00000,ffrac1*ffrac2);

   if (CONTROLONSCREEN==1)
   {
      printf("resultold=%f; expo=%d; nuest   =%d\n",resultold,expo[nuest],nuest);
      printf("resultnew=%f; expo=%d; proposal=%d\n",
             resultnew,expo[proposal],proposal);
      printf("aa=%f\n",aa);
      printf("bb=%f\n",bb);
      printf("ffrac1=%f\n",ffrac1);
      printf("ffrac2=%f\n",ffrac2);
   }

   if (uniform(0,1)<=accnu) nuest=proposal;
   else                     nuest=nuest;
   if (CONTROLONSCREEN==1)
      printf("accept proposal with prob=%f; nuest=%d\n",accnu,nuest);

   if (ACTUALITER<=NUDRAWSTART) nuest=NUSTART;
}


//--------------------------------------------------------------------------------

void draw_LAMBDA(void)
```

```
{
   unsigned int t;

   for (t=1; t<=TTTT; t++)
      LAMBDAest[t]=rand_gamma(0.50000*(nuest+1.00000),
                             0.50000*(nuest+(Yest[t]-impactX(t))*
                                            (Yest[t]-impactX(t))/exp(Hest[t])));
}

//--------------------------------------------------------------------------------

void draw_H(void)
{
   int k,t;
   double intert  = 0.000000;
   double intertm1= 0.000000;

   double muF     = mufix;
   double phiF    = THETA(1,1);
   double sigma1F = THETA(2,1);

   unsigned int counter=0;

   double alpha[10];
   if (NCOVZ>0) alpha[1] = THETA(3,1);
   if (NCOVZ>1) alpha[2] = THETA(4,1);
   if (NCOVZ>2) alpha[3] = THETA(5,1);
   if (NCOVZ>3) alpha[4] = THETA(6,1);
   if (NCOVZ>4) alpha[5] = THETA(7,1);
   if (NCOVZ>5) alpha[6] = THETA(8,1);

   double sigma2F = sigma1F*sigma1F;
   double phi2F   = phiF*phiF;

   htt[0]= muF;
   ptt[0]= sigma2F/(1.00000-phi2F);

   for (t=1; t<=TTTT; t++)
   {
      intert   = 0.00000;
      for (k=1; k<=NCOVZ; k++)  intert += alpha[k]*Z(t,k);
      intertm1 = 0.00000;
      if (USECHANGEDHTEQUATION==1)
      {
         if (t>1) for (k=1; k<=NCOVZ; k++)  intertm1 += alpha[k]*Z(t-1,k);
      }

      httm1[t] = muF + intert + phiF*(htt[t-1]-muF-intertm1);
      pttm1[t] = phi2F*ptt[t-1] + sigma2F;

      fttm1[t] = pttm1[t]+sigma2mix[sest[t]];
      kt[t]    = pttm1[t]/fttm1[t];
      htt[t]   = httm1[t]+kt[t]*(YAST[t]-mumix[sest[t]]-httm1[t]);
      ptt[t]   = (1-kt[t])*pttm1[t];

      et[t]    = YAST[t]-mumix[sest[t]]-httm1[t];
   }
   // now all e_t's, fttm1_t's, and k_t's are known

   rt[TTTT]=0.00000;
   ut[TTTT]=0.00000;
```

```
   counter=0;

   for (t=TTTT; t>=1; t--)
   {
      nt[t]    = 1.00000/fttm1[t] + phiF*phiF*kt[t]*kt[t]*ut[t];
      dt[t]    = et[t]/fttm1[t] - rt[t]*phiF*kt[t];

      ct[t]    = sigma2mix[sest[t]] - sigma2mix[sest[t]]*sigma2mix[sest[t]]*nt[t];

      if (ct[t]>=0.00000) zetat[t] = normal(0.00000,sqrt(ct[t]));
      else
      {
         zetat[t] = 0.00000;
         counter++;
      }

      bt[t]    = sigma2mix[sest[t]]*( nt[t] - phiF*phiF*kt[t]*ut[t]);

      rt[t-1]  = et[t]/fttm1[t]   + (phiF - phiF*kt[t])*rt[t] - bt[t]*zetat[t]/ct[t];
      ut[t-1]  = 1.00000/fttm1[t] + (phiF - phiF*kt[t])*(phiF - phiF*kt[t])*ut[t] +
                 bt[t]*bt[t]/ct[t];

      Hest[t]  = YAST[t] - mumix[sest[t]] - sigma2mix[sest[t]]*dt[t] - zetat[t];

      if (Hest[t]<-20)
      {
         Hest[t]=-20;
         if (CONTROLONSCREEN==1) printf("H[%d] set to -20\n",t);
      }
      if (Hest[t]>  5)
      {
         Hest[t]=5;
         if (CONTROLONSCREEN==1) printf("H[%d] set to 5\n",t);
      }
   }
   if (CONTROLONSCREEN==1) printf("zeta[t]-corrections: %d \n",counter);
}

//-------------------------------------------------------------------------------

void draw_s(void)
{
   int t;
   double unifrandoms[TTTT+2];
   double punnorm1, punnorm2, punnorm3, punnorm4, punnorm5, punnorm6, punnorm7, ss;

   for (t=1; t<=TTTT; t++)
      YAST[t] = log((Yest[t]-impactX(t))*(Yest[t]-impactX(t))) + log(LAMBDAest[t]);

   for (t=1; t<=TTTT; t++) unifrandoms[t]=uniform(0.0000,1.0000);

   for (t=1; t<=TTTT; t++)
   {
      punnorm1 = 0.00730*dnorm(YAST[t],Hest[t]-11.40039,2.40748);
      punnorm2 = 0.10556*dnorm(YAST[t],Hest[t]- 5.24321,1.61669);
      punnorm3 = 0.00002*dnorm(YAST[t],Hest[t]- 9.83726,2.27585);
      punnorm4 = 0.04395*dnorm(YAST[t],Hest[t]+ 1.50746,0.40908);
      punnorm5 = 0.34001*dnorm(YAST[t],Hest[t]- 0.65098,0.80006);
      punnorm6 = 0.24566*dnorm(YAST[t],Hest[t]+ 0.52478,0.58329);
      punnorm7 = 0.25750*dnorm(YAST[t],Hest[t]- 2.35859,1.12366);
```

```
      ss = punnorm1+punnorm2+punnorm3+punnorm4+punnorm5+punnorm6+punnorm7;

      if (unifrandoms[t]>(punnorm1+punnorm2+punnorm3+punnorm4+punnorm5+punnorm6)/ss)
         sest[t]=7;
      else
         if (unifrandoms[t]>(punnorm1+punnorm2+punnorm3+punnorm4+punnorm5)/ss)
            sest[t]=6;
         else
            if (unifrandoms[t]>(punnorm1+punnorm2+punnorm3+punnorm4)/ss) sest[t]=5;
            else
               if (unifrandoms[t]>(punnorm1+punnorm2+punnorm3)/ss) sest[t]=4;
               else
                  if (unifrandoms[t]>(punnorm1+punnorm2)/ss) sest[t]=3;
                  else
                     if (unifrandoms[t]>punnorm1/ss) sest[t]=2;
                     else sest[t]=1;
   }
}

//-------------------------------------------------------------------------------

void neldermead(unsigned int n, double xstart1, double xstart2, double xstart3,
                double xstart4, double xstart5, double xstart6, double xstart7,
                double xstart8, double xstart9, double xstart10,
                unsigned int steps, double precision, unsigned int viewsteps)
{
   double rho=1.00000, chi=2.00000, gamma=0.50000, sigma=0.50000;

   double x[12], xx[12][13], xorder[12][13], xbar[12];
   double xreflect[12], xexpand[12], xc[12], xcc[12];
   double f[12], freflect, fexpand, fc, fcc, fchange;
   unsigned int shrink=0,i,k,j,ochange;
   unsigned int order[12];
   unsigned int iter=0;
   double ninv;
   switch (n)
   {
      case  1: ninv=1.0000000; break;
      case  2: ninv=0.5000000; break;
      case  3: ninv=0.3333333; break;
      case  4: ninv=0.2500000; break;
      case  5: ninv=0.2000000; break;
      case  6: ninv=0.1666667; break;
      case  7: ninv=0.1428571; break;
      case  8: ninv=0.1250000; break;
      case  9: ninv=0.1111111; break;
      case 10: ninv=0.1000000; break;
   }


   // start vertices
   for (i=1; i<=n+1; i++)
   {
      switch (n)
      {
         case 10: xorder[10][i]=xstart10;
         case  9: xorder[ 9][i]=xstart9;
         case  8: xorder[ 8][i]=xstart8;
         case  7: xorder[ 7][i]=xstart7;
         case  6: xorder[ 6][i]=xstart6;
```

```
      case  5: xorder[ 5][i]=xstart5;
      case  4: xorder[ 4][i]=xstart4;
      case  3: xorder[ 3][i]=xstart3;
      case  2: xorder[ 2][i]=xstart2;
      case  1: xorder[ 1][i]=xstart1;
   }
}

for (i=1; i<=n; i++) xorder[i][i]=xorder[i][i]+0.1;

switch (n)
{
   case  1: for (k=1; k<=2; k++)
               f[k]=func(xorder[1][k]);
            break;
   case  2: for (k=1; k<=3; k++)
               f[k]=func(xorder[1][k],xorder[2][k]);
            break;
   case  3: for (k=1; k<=4; k++)
               f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k]);
            break;
   case  4: for (k=1; k<=5; k++)
               f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],xorder[4][k]);
            break;
   case  5: for (k=1; k<=6; k++)
               f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],xorder[4][k],
                         xorder[5][k]);
            break;
   case  6: for (k=1; k<=7; k++)
               f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],xorder[4][k],
                         xorder[5][k],xorder[6][k]);
            break;
   case  7: for (k=1; k<=8; k++)
               f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],xorder[4][k],
                         xorder[5][k],xorder[6][k],xorder[7][k]);
            break;
   case  8: for (k=1; k<=9; k++)
               f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],xorder[4][k],
                         xorder[5][k],xorder[6][k],xorder[7][k],xorder[8][k]);
            break;
   case  9: for (k=1; k<=10; k++)
               f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],xorder[4][k],
                         xorder[5][k],xorder[6][k],xorder[7][k],xorder[8][k],
                         xorder[9][k]);
            break;
   case 10: for (k=1; k<=11; k++)
               f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],xorder[4][k],
                         xorder[5][k],xorder[6][k],xorder[7][k],xorder[8][k],
                         xorder[9][k],xorder[10][k]);
            break;
}

if (viewsteps==1)
{
   printf("Opti-Step: %d \t Values of f: ", iter);
   for (k=1; k<=n+1; k++) printf("%f ", f[k]);
   printf("\n");
}

for (i=1; i<=n+1; i++) order[i]=i;
for (k=1; k<=n+1; k++)
```

```
{
   for (j=1; j<=n; j++)
   {
      if (f[j+1]<f[j])
      {
         fchange=f[j+1];     f[j+1]=f[j];         f[j]=fchange;
         ochange=order[j+1]; order[j+1]=order[j]; order[j]=ochange;
      }
   }
}
for (k=1; k<=n+1; k++) for (i=1; i<=n; i++)  xx[i][k]=xorder[i][order[k]];
for (k=1; k<=n+1; k++) for (i=1; i<=n; i++)  xorder[i][k]=xx[i][k];

if (viewsteps==1)
{
   for (k=1; k<=n; k++)
   {
      printf("XORDER: ");
      for (i=1; i<=n; i++) printf("%f ",xorder[k][i]);
      printf("\n");
   }
}

while (iter<steps && fabs((f[n+1]-f[1])/f[n+1])>precision)
{
   iter++;
   if (viewsteps==1)
   {
      printf("Opti-Step: %d \t Values of f: ", iter);
      for (k=1; k<=n+1; k++) printf("%f ", f[k]);
      printf("\n");
   }

   // REFLECT
   for (i=1; i<=n; i++)
   {
      xbar[i]=0.0000000;
      for (k=1; k<=n; k++)  xbar[i]=xbar[i]+xorder[i][k];
      xbar[i]=ninv*xbar[i];
   }

   //printf("Values of xbar: %f %f %f \n", xbar[1], xbar[2], xbar[3]);

   for (i=1; i<=n; i++)  xreflect[i]=(1.00000+rho)*xbar[i]-rho*xorder[i][n+1];

   switch (n)
   {
      case  1: freflect=func(xreflect[1]); break;
      case  2: freflect=func(xreflect[1],xreflect[2]); break;
      case  3: freflect=func(xreflect[1],xreflect[2],xreflect[3]); break;
      case  4: freflect=func(xreflect[1],xreflect[2],xreflect[3],xreflect[4]);
               break;
      case  5: freflect=func(xreflect[1],xreflect[2],xreflect[3],xreflect[4],
                             xreflect[5]);
               break;
      case  6: freflect=func(xreflect[1],xreflect[2],xreflect[3],xreflect[4],
                             xreflect[5],xreflect[6]);
               break;
      case  7: freflect=func(xreflect[1],xreflect[2],xreflect[3],xreflect[4],
                             xreflect[5],xreflect[6],xreflect[7]);
               break;
```

```
   case  8: freflect=func(xreflect[1],xreflect[2],xreflect[3],xreflect[4],
                          xreflect[5],xreflect[6],xreflect[7],xreflect[8]);
            break;
   case  9: freflect=func(xreflect[1],xreflect[2],xreflect[3],xreflect[4],
                          xreflect[5],xreflect[6],xreflect[7],xreflect[8],
                          xreflect[9]);
            break;
   case 10: freflect=func(xreflect[1],xreflect[2],xreflect[3],xreflect[4],
                          xreflect[5],xreflect[6],xreflect[7],xreflect[8],
                          xreflect[9],xreflect[10]);
            break;
}

//printf("freflect: %f \n",freflect);


if (f[1]<=freflect && freflect<f[n])
{
   //accept
   for (i=1; i<=n; i++) { xorder[i][n+1]=xreflect[i]; f[n+1]=freflect; }
   if (viewsteps==1)  printf("REFLECT 1\n");
}
else
{
   if (freflect<f[1])   // EXPAND
   {
      for (i=1; i<=n; i++)
         xexpand[i]=(1.00000+rho*chi)*xbar[i]-rho*chi*xorder[i][n+1];

      switch (n)
      {
         case  1: fexpand=func(xexpand[1]); break;
         case  2: fexpand=func(xexpand[1],xexpand[2]); break;
         case  3: fexpand=func(xexpand[1],xexpand[2],xexpand[3]); break;
         case  4: fexpand=func(xexpand[1],xexpand[2],xexpand[3],xexpand[4]);
                  break;
         case  5: fexpand=func(xexpand[1],xexpand[2],xexpand[3],xexpand[4],
                               xexpand[5]);
                  break;
         case  6: fexpand=func(xexpand[1],xexpand[2],xexpand[3],xexpand[4],
                               xexpand[5],xexpand[6]);
                  break;
         case  7: fexpand=func(xexpand[1],xexpand[2],xexpand[3],xexpand[4],
                               xexpand[5],xexpand[6],xexpand[7]);
                  break;
         case  8: fexpand=func(xexpand[1],xexpand[2],xexpand[3],xexpand[4],
                               xexpand[5],xexpand[6],xexpand[7],xexpand[8]);
                  break;
         case  9: fexpand=func(xexpand[1],xexpand[2],xexpand[3],xexpand[4],
                               xexpand[5],xexpand[6],xexpand[7],xexpand[8],
                               xexpand[9]);
                  break;
         case 10: fexpand=func(xexpand[1],xexpand[2],xexpand[3],xexpand[4],
                               xexpand[5],xexpand[6],xexpand[7],xexpand[8],
                               xexpand[9],xexpand[10]);
                  break;
      }

      if (fexpand<freflect)
      {
         //accept xexpand
```

```
      for (i=1; i<=n; i++)  { xorder[i][n+1]=xexpand[i];  f[n+1]=fexpand;  }
      if (viewsteps==1) printf("EXPAND\n");
   }
   else
   {
      //accept xreflect
      for (i=1; i<=n; i++)  { xorder[i][n+1]=xreflect[i]; f[n+1]=freflect; }
      if (viewsteps==1) printf("REFLECT 2\n");
   }
}
else   // (freflect>=f[n]) // OUTSIDE CONTRACTION
{
   if (freflect<f[n+1])
   {
      for (i=1; i<=n; i++)
         xc[i]=(1.00000+rho*gamma)*xbar[i]-rho*gamma*xorder[i][n+1];

      switch (n)
      {
         case  1: fc=func(xc[1]); break;
         case  2: fc=func(xc[1],xc[2]); break;
         case  3: fc=func(xc[1],xc[2],xc[3]); break;
         case  4: fc=func(xc[1],xc[2],xc[3],xc[4]); break;
         case  5: fc=func(xc[1],xc[2],xc[3],xc[4],xc[5]); break;
         case  6: fc=func(xc[1],xc[2],xc[3],xc[4],xc[5],xc[6]); break;
         case  7: fc=func(xc[1],xc[2],xc[3],xc[4],xc[5],xc[6],xc[7]); break;
         case  8: fc=func(xc[1],xc[2],xc[3],xc[4],xc[5],xc[6],xc[7],xc[8]);
                  break;
         case  9: fc=func(xc[1],xc[2],xc[3],xc[4],xc[5],xc[6],xc[7],xc[8],
                           xc[9]);
                  break;
         case 10: fc=func(xc[1],xc[2],xc[3],xc[4],xc[5],xc[6],xc[7],xc[8],
                  xc[9],xc[10]); break;
      }

      if (fc<=freflect)
      {
         //accept xc
         for (i=1; i<=n; i++)  { xorder[i][n+1]=xc[i]; f[n+1]=fc; }
         if (viewsteps==1) printf("OUT CONTRACTION\n");
      }
      else  shrink=1;
   }
   else   // INSIDE CONTRACTION
   {
      for (i=1; i<=n; i++)
         xcc[i]=(1.00000-gamma)*xbar[i]+gamma*xorder[i][n+1];

      switch (n)
      {
         case  1: fcc=func(xcc[1]); break;
         case  2: fcc=func(xcc[1],xcc[2]); break;
         case  3: fcc=func(xcc[1],xcc[2],xcc[3]); break;
         case  4: fcc=func(xcc[1],xcc[2],xcc[3],xcc[4]); break;
         case  5: fcc=func(xcc[1],xcc[2],xcc[3],xcc[4],xcc[5]); break;
         case  6: fcc=func(xcc[1],xcc[2],xcc[3],xcc[4],xcc[5],xcc[6]);
                  break;
         case  7: fcc=func(xcc[1],xcc[2],xcc[3],xcc[4],xcc[5],xcc[6],
                           xcc[7]);
                  break;
         case  8: fcc=func(xcc[1],xcc[2],xcc[3],xcc[4],xcc[5],xcc[6],
```

```
                                xcc[7],xcc[8]);
                        break;
                case  9: fcc=func(xcc[1],xcc[2],xcc[3],xcc[4],xcc[5],xcc[6],
                                xcc[7],xcc[8],xcc[9]);
                        break;
                case 10: fcc=func(xcc[1],xcc[2],xcc[3],xcc[4],xcc[5],xcc[6],
                                xcc[7],xcc[8],xcc[9],xcc[10]);
                        break;
            }

            if (fcc<f[n+1])
            {
               //accept xcc
               for (i=1; i<=n; i++)  { xorder[i][n+1]=xcc[i]; f[n+1]=fcc; }
               if (viewsteps==1) printf("INS CONTRACTION\n");
            }
            else  shrink=1;
         }
      }
   }
   if (shrink==1)   // SHRINK STEP
   {
      shrink=0;
      if (viewsteps==1) printf("SHRINK\n");

      for (k=2; k<=n+1; k++)
         for (i=1; i<=n; i++)
            xorder[i][k]=(1.00000-sigma)*xorder[i][1]+sigma*xorder[i][k];
      // ORDER
      switch (n)
      {
         case  1: for (k=2; k<=2; k++)
                     f[k]=func(xorder[1][k]);
                  break;
         case  2: for (k=2; k<=3; k++)
                     f[k]=func(xorder[1][k],xorder[2][k]);
                  break;
         case  3: for (k=2; k<=4; k++)
                     f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k]);
                  break;
         case  4: for (k=2; k<=5; k++)
                     f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],
                               xorder[4][k]);
                  break;
         case  5: for (k=2; k<=6; k++)
                     f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],
                               xorder[4][k],xorder[5][k]);
                  break;
         case  6: for (k=2; k<=7; k++)
                     f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],
                               xorder[4][k],xorder[5][k],xorder[6][k]);
                  break;
         case  7: for (k=2; k<=8; k++)
                     f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],
                               xorder[4][k],xorder[5][k],xorder[6][k],
                               xorder[7][k]);
                  break;
         case  8: for (k=2; k<=9; k++)
                     f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],
                               xorder[4][k],xorder[5][k],xorder[6][k],
                               xorder[7][k],xorder[8][k]);
```

```
                break;
      case  9: for (k=2; k<=10; k++)
                  f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],
                            xorder[4][k],xorder[5][k],xorder[6][k],
                            xorder[7][k],xorder[8][k],xorder[9][k]);
                break;
      case 10: for (k=2; k<=11; k++)
                  f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],
                            xorder[4][k],xorder[5][k],xorder[6][k],
                            xorder[7][k],xorder[8][k],xorder[9][k],
                            xorder[10][k]);
                break;
   }

   for (i=1; i<=n+1; i++) order[i]=i;
   for (k=1; k<=n+1; k++)
   {
      for (j=1; j<=n; j++)
      {
         if (f[j+1]<f[j])
         {
            fchange=f[j+1];     f[j+1]=f[j];         f[j]=fchange;
            ochange=order[j+1]; order[j+1]=order[j]; order[j]=ochange;
         }
      }
   }
   for (k=1; k<=n+1; k++) for (i=1; i<=n; i++)  xx[i][k]=xorder[i][order[k]];
   for (k=1; k<=n+1; k++) for (i=1; i<=n; i++)  xorder[i][k]=xx[i][k];
}
else
{
   // ORDER
   switch (n)
   {
      case  1: for (k=2; k<=2; k++)
                  f[k]=func(xorder[1][k]);
                break;
      case  2: for (k=2; k<=3; k++)
                  f[k]=func(xorder[1][k],xorder[2][k]);
                break;
      case  3: for (k=2; k<=4; k++)
                  f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k]);
                break;
      case  4: for (k=2; k<=5; k++)
                  f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],
                            xorder[4][k]);
                break;
      case  5: for (k=2; k<=6; k++)
                  f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],
                            xorder[4][k],xorder[5][k]);
                break;
      case  6: for (k=2; k<=7; k++)
                  f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],
                            xorder[4][k],xorder[5][k],xorder[6][k]);
                break;
      case  7: for (k=2; k<=8; k++)
                  f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],
                            xorder[4][k],xorder[5][k],xorder[6][k],
                            xorder[7][k]);
                break;
      case  8: for (k=2; k<=9; k++)
```

```
                        f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],
                                  xorder[4][k],xorder[5][k],xorder[6][k],
                                  xorder[7][k],xorder[8][k]);
                     break;
            case  9: for (k=2; k<=10; k++)
                        f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],
                                  xorder[4][k],xorder[5][k],xorder[6][k],
                                  xorder[7][k],xorder[8][k],xorder[9][k]);
                     break;
            case 10: for (k=2; k<=11; k++)
                        f[k]=func(xorder[1][k],xorder[2][k],xorder[3][k],
                                  xorder[4][k],xorder[5][k],xorder[6][k],
                                  xorder[7][k],xorder[8][k],xorder[9][k],
                                  xorder[10][k]);
                     break;
         }

         for (i=1; i<=n+1; i++) order[i]=i;
         for (k=1; k<=n+1; k++)
         {
            for (j=1; j<=n; j++)
            {
               if (f[j+1]<f[j])
               {
                  fchange=f[j+1];     f[j+1]=f[j];         f[j]=fchange;
                  ochange=order[j+1]; order[j+1]=order[j]; order[j]=ochange;
               }
            }
         }
         for (k=1; k<=n+1; k++) for (i=1; i<=n; i++)  xx[i][k]=xorder[i][order[k]];
         for (k=1; k<=n+1; k++) for (i=1; i<=n; i++)  xorder[i][k]=xx[i][k];
         //-------------------------------------------------------------------------
      }

      if (viewsteps==1)
      {
         printf("actual precision: %f\n",(f[n+1]-f[1])/f[n+1]);
         printf("based on the values:\n");
         printf("Opti-Step: %d \t Values of f: ", iter);
         for (k=1; k<=n+1; k++) printf("%f ", f[k]);
         printf("\n");
      }
   }

   for (k=1; k<=n; k++) min[k] = xorder[k][1];

}

//---------------------------------------------------------------------------------

double minimum(double x, double y)
{
   if (x<=y) return x;
   else      return y;
}

//---------------------------------------------------------------------------------
```

Appendix D

Implementation of Procedures for Random Variables

The following C++ code is the header-file 'randomGM.h' which is used by the programs in Appendices B and C. It contains functions for random number generation and density evaluations and must be copied into the same directory as the calling program.

```
//-----------------------------------------------------------------------------
//         FILE FOR GENERATION OF RANDOM NUMBERS AND EVALUATION OF DENSITIES
//                                by GERNOT MUELLER, 2004
//-----------------------------------------------------------------------------

#ifndef __RANDOM_H
#define __RANDOM_H

#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <time.h>
#include "matrixGM.h"

//-----------------------------------------------------------------------------
//                     INITIALIZATION OF THE RANDOM NUMBER GENERATOR
//-----------------------------------------------------------------------------
// set the pseudo random number generator on a random starting position

void randomize(void)
{
   time_t t;
   srand((unsigned) time(&t));
}

//-----------------------------------------------------------------------------
//               GENERATION OF RANDOM NUMBERS FROM UNIVARIATE DISTRIBUTIONS
//-----------------------------------------------------------------------------
// draw a random number from the Unif(a,b) distribution
```

```
inline double uniform(double a=0.00000, double b=1.00000)
{
   int zufall = 0;
   while ((zufall == 0) || (zufall == RAND_MAX)) zufall = rand();
   return (double(zufall)/double(RAND_MAX))*(b-a)+a;
}

//------------------------------------------------------------------------------
// draw a random number from the N(mu,sigma) distribution
// sigma = standard deviation!

inline double normal(const double & mu=0, const double & sigma=1)
{
   double u1 = uniform();
   double u2 = uniform();
   return mu + sigma*(sqrt(-2*log(u1))*sin(6.2831853*u2));
}

//------------------------------------------------------------------------------
// draw a random number from the Exp(lambda) distribution

inline double rand_expo(double lambda)
{
   return (- 1/lambda)*log(uniform());
}

//------------------------------------------------------------------------------
// draw a random number from the Gamma(a,b) distribution

double rand_gamma(double a,double b)
{
   if (a > 1)
   {
      double h1 = a-1;          // h1 equals b in Devroye (1986)
      double h2 = 3*a-0.75;     // h2 equals c in Devroye (1986)
      double U,V,W,Y,X,Z;
      int accept = 0;
      do
      {
         U = uniform();
         V = uniform();
         W = U*(1-U);
         Y = sqrt(h2/W)*(U-0.5);
         X = h1 + Y;
         if (X > 0)
         {
            Z = 64*W*W*W*V*V;
            if (Z <= (1-(2*Y*Y)/X)) accept = 1;
            else
            {
               if (((X/h1) > 0) && (log(Z) <= ( 2*(h1*log(X/h1) - Y)))) accept = 1;
            }
         }
      }
      while (accept == 0);
      return X/b;
   }
   else
   {
      if (a == 1) return rand_expo(b);
      else
```

```
      {
         double X = rand_gamma(a+1,1)*pow(uniform(),1/a);
         return X/b;
      }
   }
}

//-----------------------------------------------------------------------------
// draw a random number double-side truncated normal distribution

inline double ds_trunc_normal(const double & left, const double & right,
                              const double & mu=0, const double & sigma=1)
{
   // default value if support of truncated distribution is too far away
   // from mean of non-truncated distribution
   if (left-mu>3*sigma)  return 0.98*left+0.02*right;
   if (mu-right>3*sigma) return 0.02*left+0.98*right;
   unsigned accept = 0;
   double rand;
   while (!accept)
   {
      rand = normal(mu,sigma);
      if ((rand <= right) && (rand >= left)) accept = 1;
   }
   return rand;
}

//-----------------------------------------------------------------------------
// draw a random number left-side truncated normal distribution

inline double ls_trunc_normal(const double & left,
                              const double & mu=0, const double & sigma=1)
{
   // default value if support of truncated distribution is too far away
   // from mean of non-truncated distribution
   if (left-mu>3*sigma)  return left+0.02;
   unsigned accept = 0;
   double rand;
   while (!accept)
   {
      rand = normal(mu,sigma);
      if (rand >= left) accept = 1;
   }
   return rand;
}

//-----------------------------------------------------------------------------
// draw a random number from right-side truncated normal distribution

inline double rs_trunc_normal(const double & right,
                              const double & mu=0, const double & sigma=1)
{
   // default value if support of truncated distribution is too far away
   // from mean of non-truncated distribution
   if (mu-right>3*sigma) return right-0.02;
   unsigned accept = 0;
   double rand;
   while (!accept)
   {
      rand = normal(mu,sigma);
      if (rand <= right) accept = 1;
```

```
   }
   return rand;
}

//-------------------------------------------------------------------------------
//                         DENSITIES FROM UNIVARIATE DISTRIBUTIONS
//-------------------------------------------------------------------------------
// density of normal distribution

inline double dnorm(double x, double mu, double sigma)
{
   return exp(-0.5000000000*(x-mu)*(x-mu)/(sigma*sigma))/
          (sqrt(2.0000000000*3.14159265359)*sigma);
}

// unnormalized density of gamma distribution

inline double undgamma(double y, double alpha, double beta)
{
   return pow(y,alpha-1.000000)*exp(-beta*y);
}

//-------------------------------------------------------------------------------
// density of student t-distribution
// definition see Gelman, Carlin, Stern, and Rubin: Bayesian Data Analysis
// nu: degress of freedom, mean: location parameter, sigma: scale parameter

double gammafactors[302];

// compute Gamma((nu+1)/2)/Gamma(nu/2)
inline double gammafactorsforstudent(unsigned int nu)
{
   unsigned int t=0;
   double result=1.00000000;

   for (t=nu; t>=3; t-=2)
      result=result*(1.00000*(t-1))/(1.00000*(t-2));

   if (t==2) result=result*0.886227;
   if (t==1) result=result/1.772454;

   return result;
}

// store the factors Gamma((nu+1)/2)/Gamma(nu/2)
void savegammafactors(void)
{
   unsigned int i;
   for (i=2; i<=300; i++) gammafactors[i]=gammafactorsforstudent(i);
}

// density of student t-distribution at y
inline double dstudent(double y, unsigned int nu, double mean=0.0000000000,
                       double sigma=1.0000000000)
{
   double result = 0.000000;
   double nud    = 1.000000*nu;
   double PI     = 3.14159265358979;
   double inter  = 0.000000;

   inter  = (y-mean)/sigma;
```

```
   result = pow(1.000000+inter*inter/nud,-0.500000*nud-0.500000);
   result = gammafactors[nu]*result/(sqrt(nud*PI)*sigma);

   return result;
}

//------------------------------------------------------------------------------
// unnormalized density of discretized gamma distribution truncated at 1.5 and 128.5
// using Newton-Cotes with 5 nodes, evaluated at y

inline double unddisgamma(unsigned int y, double alpha, double beta)
{
   double z=1.000000*y;
   return (7*undgamma(z-0.500000,alpha,beta)+32*undgamma(z-0.250000,alpha,beta)
           +12*undgamma(z,alpha,beta)+32*undgamma(z+0.250000,alpha,beta)
           +7*undgamma(z+0.500000,alpha,beta))/90.000000;
}

//------------------------------------------------------------------------------
//                              MULTIVARIATE DISTRIBUTIONS
//------------------------------------------------------------------------------
// random numbers from multivariate normal-distribution

Matrix mvnormal(Matrix & Mu, Matrix & Sigma)
{
   unsigned int i, d=Sigma.m;
   Matrix B(d,d);
   B=Sigma.Cholesky();
   Matrix X(d,1);
   Matrix Y(d,1);
   for (i=1; i<=d; i++) X(i,1)=normal(0,1);
   Y = Mu + B*X;
   return Y;
}

//------------------------------------------------------------------------------
// unnormalized density of multivariate normal distribution, evaluated at X

double undmvnormal(Matrix & X, Matrix & Mu, Matrix & Sigma)
{
   unsigned int d=Sigma.m;
   Matrix M(1,1);
   Matrix XX(d,1);
   Matrix XXX(d,1);

   XX  = X - Mu;
   XXX = Sigma.invertPosDef()*XX;
   M   = XX.transp()*XXX;
   return exp(-0.500000*M(1,1));
}

#endif
```

Appendix E

Implementation of used Matrix Class

The following C++ code is the header-file 'matrixGM.h' which is used by the programs in Appendices B, C, and D. It contains a matrix class with important matrix functions and must be copied into the same directory as the calling program.

```
//--------------------------------------------------------------------------------
//                     MATRIX CLASS --- by GERNOT MUELLER, 2004
//--------------------------------------------------------------------------------
// most important available operators: ! (inversion), *, +, -
// most important available functions:
// FUNCTION NAME              VALUE
// transp                     transposed matrix
// Cholesky                   Cholesky matrix
// invCholesky                inverse of Cholesky matrix (fast implementation!)
// invertPosDef               inverse of pos. definit matrix (fast implementation!)
// tmat2                      submatrix
// rang                       rank of matrix
// det                        determinant of matrix
// posdef                     1, if matrix is pos. definit, 0 else
//--------------------------------------------------------------------------------

#ifndef __MATRIXGM_H
#define __MATRIXGM_H


#include <iostream.h>
#include <assert.h>
#include <math.h>

//#define SPEC huge
#define SPEC
int max(int i, int k);

//--------------------------------------------------------------------------------
// class declaration
```

```
class Matrix
{
public:
   // constructor
   Matrix(unsigned zeilen, unsigned spalten)
   {
      m = zeilen;
      n = spalten;
      M = new SPEC double[m*n];
      for (int x=0; x<m; x++) for (int y=0; y<n; y++) *(M+x*n+y)=0.0;
   }

   // copy constructor
   Matrix(Matrix &toCopy);

   // destructor
   ~Matrix() { delete[] M; }

   // return element, e.g. A(1,2);
   double & operator ()(unsigned zeile, unsigned spalte)
   {
   assert(zeile && zeile<=m && spalte && spalte<=n);
   return *(M+(zeile-1)*n+spalte-1);
   }

   // assignment operator
   Matrix& operator =(const Matrix& toAssign);

   // unary plus
   Matrix operator +()
   {
      return *this;
   }

   // unary minus
   Matrix operator -()
   {
      Matrix Q(m,n);
      for (int i=1; i<=m; i++) {
         for (int k=1; k<=n; k++) Q(i,k) = (-(*(M+(i-1)*n+k-1)));
      }
      return Q;
   }

   // addition
   friend Matrix operator +(const Matrix& A, const Matrix& B);

   // subtraction
   friend Matrix operator -(const Matrix& A, const Matrix& B);

   // multiplication by matrix
   friend Matrix operator *(const Matrix& A, const Matrix& B);

   // multiplication by scalar
   friend Matrix operator *(const double f, const Matrix& B)
   {
      Matrix Q(B.m,B.n);
      for (int i=1; i<=B.m; i++) {
         for (int k=1; k<=B.n; k++)
            Q(i,k) = f*(*(B.M+(i-1)*B.n+k-1));      //Q(i,k) = f*B(i,k);
```

```
   }
   return Q;
}

// transpose matrix
Matrix transp(void);

// invert matrix
friend Matrix operator !(const Matrix& A);

// Cholesky-decomposition
Matrix Cholesky(void);

// invert Cholesky-matrix
Matrix invCholesky(void);

// invert positive definite matrix
Matrix invertPosDef(void);

// change rows i and k
Matrix change_rows(unsigned i, unsigned k);

// change columns i und k
Matrix change_cols(unsigned i, unsigned k);

// multiply row z by double f
Matrix mult_r(double f, unsigned z)
{
   assert(z>0 && z<=m);
   for (int i=1; i<=n; i++) (*(M+(z-1)*n+i-1))=f*(*(M+(z-1)*n+i-1));
   return *this;
}

// multiply column s by double f
Matrix mult_c(double f, unsigned s)
{
   assert(s>0 && s<=n);
   for (int i=1; i<=m; i++) (*(M+(i-1)*n+s-1))=f*(*(M+(i-1)*n+s-1));
   return *this;
}

// add row i to row k
Matrix add_r(unsigned i, unsigned k)
{
   assert(i>0 && i<=m && k>0 && k<=m);
   for (int x=1; x<=n; x++)
      (*(M+(k-1)*n+x-1))=(*(M+(k-1)*n+x-1))+(*(M+(i-1)*n+x-1));
   return *this;
}

// left top ixk-submatrix
Matrix tmat(unsigned i, unsigned k)
{
   assert(i<=m && k<=n);
   Matrix Q(i,k);
   for (int x=1; x<=i; x++) for (int y=1; y<=k; y++) Q(x,y)=*(M+(x-1)*n+y-1);
   return Q;
}

// submatrix (i,k) to (j,l)
Matrix tmat2(unsigned i, unsigned k, unsigned j, unsigned l)
```

```
{
   assert(i<=m && j<=m && k<=n && l<=n);
   Matrix Q(j-i+1,l-k+1);
   for (int x=i; x<=j; x++)
      for (int y=k; y<=l; y++) Q(x-i+1,y-k+1)=*(M+(x-1)*n+y-1);
   return Q;
}

// compute rank
int rang(void)
{
   int x,y;
   Matrix Q(m,n);
   for (x=1; x<=m; x++) for (y=1; y<=n; y++) Q(x,y)=(*(M+(x-1)*n+y-1));

   int i=1,k,q,pivz,pivs,test=0,t=0;
   double max,f=0.;
   if (m>n) q=n; else q=m;
   while (i<=q && test==0) {
      // choose pivot
      max=0.;
      for (x=i; x<=m; x++) {
         for (y=i; y<=n; y++) {
            if (fabs(Q(x,y))>max) { pivz=x; pivs=y; max=fabs(Q(x,y)); }
         }
      }
      if (i==1 && max==0) return(0);
      if (max!=0) {
         Q=Q.change_rows(i,pivz);
         Q=Q.change_cols(i,pivs);
         for (k=i+1; k<=m; k++) {
            if (Q(k,i)!=0) {
               assert(Q(i,i)!=0);
               f=-(Q(k,i)/Q(i,i));
               Q=Q.mult_r(f,i);
               Q=Q.add_r(i,k);
               Q=Q.mult_r(1./f,i);
            }
         }
      }

      for (x=i+1; x<=m; x++) {
         for (y=i+1; y<=n; y++) {
            if ((floor(Q(x,y)*100000+0.5)/100000)!=0.) t=1;
         }
      }
      if (t==1) t=0; else test=1;
      i++;
   }
   return (--i);
}

// compute determinant (Gauss-elimination)
double det(void)
{
   assert(n==m);
   int x,y;
   Matrix Q(n,n);
   for (x=1; x<=n; x++) for (y=1; y<=n; y++) Q(x,y)=(*(M+(x-1)*n+y-1));
   if (Q.rang()<n) return(0.);
   int i=1,k,pivz,pivs,numb_c_changes=0;
```

```
      double max,f=0.,det=0.;
      while (i<=n) {
         // choose pivot in i-th column
         max=0.;
         for (x=i; x<=n; x++) {
            if (fabs(Q(x,i))>max) { pivz=x; max=fabs(Q(x,i)); }
         }

         if (i!=pivz) { Q=Q.change_rows(i,pivz); numb_c_changes++; }
         for (k=i+1; k<=n; k++) {
            if (Q(k,i)!=0) {
               assert(Q(i,i)!=0);
               f=-(Q(k,i)/Q(i,i));
               Q=Q.mult_r(f,i);
               Q=Q.add_r(i,k);
               Q=Q.mult_r(1./f,i);
            }
         }
         i++;
      }

      // compute determinant of upper triangular matrix
      det = 1.;
      for (x=1; x<=n; x++) { det=det*Q(x,x); }
      if (numb_c_changes%2==1) det=-det;
      return(det);
   }

   // test whether matrix is positive definite (0=no, 1=yes)
   int posdef(void)
   {
      assert(m==n);
      int x,y;
      Matrix Q(n,n);
      for (x=1; x<=n; x++) for (y=1; y<=n; y++) Q(x,y)=(*(M+(x-1)*n+y-1));
      for (x=1; x<=n; x++) if (((Q.tmat(x,x)).det())<=0.) return(0);
      return(1);
   }

public:
   unsigned m,n;
   SPEC double *M;
};

//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
// copy constructor
Matrix::Matrix(Matrix &toCopy)
{
   m = toCopy.m;
   n = toCopy.n;
   M = new SPEC double[m*n];
   for (int i=0; i<=m-1; i++) for (int k=0; k<=n-1; k++) *(M+i*n+k)=toCopy(i+1,k+1);
}

//--------------------------------------------------------------------------------
// assignment operator
Matrix& Matrix::operator =(const Matrix& toAssign)
{
   delete M;
```

```
   m = toAssign.m;
   n = toAssign.n;
   M = new SPEC double[m*n];
   for (int i=0; i<=m-1; i++) {
      for ( int k=0; k<=n-1; k++) {
         *(M+i*n+k)=*(toAssign.M+i*toAssign.n+k);    // *(M+i*n+k)=toAssign(i+1,k+1);
      }
   }
   return *this;
}

//-------------------------------------------------------------------------------
// addition
Matrix operator +(const Matrix& A, const Matrix& B)
{
   assert((A.m==B.m) && (A.n==B.n));
   Matrix X(A.m,A.n);
   for (int i=1; i<=A.m; i++)
      for (int k=1; k<=A.n; k++)
         X(i,k)=(*(A.M+(i-1)*A.n+k-1))+(*(B.M+(i-1)*B.n+k-1));//X(i,k)=A(i,k)+B(i,k);
   return X;
}

//-------------------------------------------------------------------------------
// subtraction
Matrix operator -(const Matrix& A, const Matrix& B)
{
   assert((A.m==B.m) && (A.n==B.n));
   Matrix X(A.m,A.n);
   for (int i=1; i<=A.m; i++)
      for (int k=1; k<=A.n; k++)
         X(i,k)=(*(A.M+(i-1)*A.n+k-1))-(*(B.M+(i-1)*B.n+k-1));//X(i,k)=A(i,k)-B(i,k);
   return X;
}

//-------------------------------------------------------------------------------
// multiplication by matrix
Matrix operator *(const Matrix& A, const Matrix& B)
{
   assert(A.n==B.m);
   Matrix X(A.m,B.n);
   for (int i=1; i<=A.m; i++) {
      for (int k=1; k<=B.n; k++) {
         X(i,k)=0.;
         for (int l=1; l<=A.n; l++)
            X(i,k)=X(i,k)+(*(A.M+(i-1)*A.n+l-1))*(*(B.M+(l-1)*B.n+k-1));
                                                              // +A(i,l)*B(l,k);
      }
   }
   return X;
}

//-------------------------------------------------------------------------------
// transpose matrix
Matrix Matrix::transp(void)
{
   Matrix Q(n,m);
   for (int i=1; i<=m; i++) for (int k=1; k<=n; k++) Q(k,i)=*(M+(i-1)*n+k-1);
   return Q;
}
```

```
//---------------------------------------------------------------------------
// invert matrix
Matrix operator !(const Matrix& A)
{
   int a,b,c,d,pivz=0,pivs=0,n=A.n,t=0;
   double max=0.;
   assert(A.m==A.n);
   Matrix Q(n,n);
   Matrix v(n,1), w(n,1);
   for (a=1; a<=n; a++) for (b=1; b<=n; b++) Q(a,b)=(*(A.M+(a-1)*A.n+b-1));
                                                                  // Q(a,b)=A(a,b);
   for (a=1; a<=n; a++) {
      // choose pivot
      max=0.;
      for (b=1; b<=n; b++) {
         for (c=1; c<=n; c++) {
            t=0;
            for (d=1; d<=n; d++) if (v(d,1)==b) t=1;
            if (t==0 && v(c,1)==0) {
               if (fabs(Q(b,c))>max) { pivz=b; pivs=c; max=fabs(Q(b,c)); }
            }
         }
      }
      // now Q(pivz,pivs) is the pivot
      v(pivs,1)=pivz;

      // elements which are not in pivot row or pivot column
      for (b=1; b<=n; b++) {
         if (b!=pivz) {
            for (c=1; c<=n; c++) {
               if (c!=pivs) Q(b,c)=Q(b,c)-(Q(b,pivs)*Q(pivz,c)/Q(pivz,pivs));
            }
         }
      }

      // elements of pivot column
      for (b=1; b<=n; b++) if (b!=pivz) Q(b,pivs)=Q(b,pivs)/Q(pivz,pivs);

      // elements of pivot row
      for (c=1; c<=n; c++) if (c!=pivs) Q(pivz,c)=-(Q(pivz,c)/Q(pivz,pivs));

      // pivot
      Q(pivz,pivs)=1/Q(pivz,pivs);
   }

   w=v;

   for (a=1; a<n; a++) {
      t=0; c=0;
      while (t==0 && c<n) {
         c++;
         if (v(c,1)==a) t=1;
      }
      Q.change_cols(a,c);
      v(c,1)=v(a,1); v(a,1)=a;
   }

   for (b=1; b<n; b++) {
      t=0; d=0;
      while (t==0 && d<n) {
         d++;
```

```
         if (w(d,1)==b) t=1;
      }
      Q.change_rows(int(w(b,1)),b);
      w(d,1)=w(b,1); w(b,1)=b;
   }

   return Q;
}

//--------------------------------------------------------------------------------
// Cholesky-decomposition
Matrix Matrix::Cholesky(void)
{
   Matrix Q(n,n);
   int i,k,j;
   double x;
   for (i=1; i<=n; i++) {
      for (k=i; k<=n; k++) {
         x = (*(M+(i-1)*n+k-1));
         for (j=i-1; j>=1; j--) x = x - (*(M+(k-1)*n+j-1))*(*(M+(i-1)*n+j-1));
         if (i==k)  Q(i,i) = sqrt(x);
         else { Q(k,i) = x / Q(i,i); (*(M+(k-1)*n+i-1)) = x / Q(i,i); }
      }
   }
   for (k=1; k<=n; k++) for (i=1; i<=k-1; i++) (*(M+(k-1)*n+i-1))=(*(M+(i-1)*n+k-1));
   return Q;
}

//--------------------------------------------------------------------------------
// invert Cholesky-matrix
Matrix Matrix::invCholesky(void)
{
   Matrix Q(n,n);
   int i,j,k;
   double sum;
   for (k=1; k<=n; k++) {
      for (i=k; i<=n; i++) {
         if (i==k) sum=1.0000000;
         else      sum=0.0000000;
         for (j=1; j<=i-k; j++) sum -= (*(M+(i-1)*n+k+j-2))*(*(Q.M+(k+j-2)*n+k-1));
         sum /= (*(M+(i-1)*n+i-1));
         Q(i,k)=sum;
      }
   }
   return Q;
}

//--------------------------------------------------------------------------------
// invert positive definite matrix
Matrix Matrix::invertPosDef(void)
{
   Matrix Q(n,n);
   Matrix Inverse(n,n);
   Q = ((*this).Cholesky()).invCholesky();
   for (int i=1; i<=n; i++)
      for (int k=1; k<=n; k++)
         for (int j=max(i,k); j<=n; j++)
            Inverse(i,k) += Q(j,i)*Q(j,k);
   return Inverse;
}
```

```
//-----------------------------------------------------------------------------
// change rows i and k
Matrix Matrix::change_rows(unsigned i, unsigned k)
{
   assert(i && k && i<=m && k<=m);
   double t;
   if (i!=k) {
      for (int p=1; p<=n; p++) {
         t=*(M+(i-1)*n+p-1);
         *(M+(i-1)*n+p-1)=*(M+(k-1)*n+p-1);
         *(M+(k-1)*n+p-1)=t;
      }
   }
   return *this;
}

//-----------------------------------------------------------------------------
// change columns i and k
Matrix Matrix::change_cols(unsigned i, unsigned k)
{
   assert(i && k && i<=n && k<=n);
   double t;
   if (i!=k) {
      for (int p=1; p<=m; p++) {
         t=*(M+(p-1)*n+i-1);
         *(M+(p-1)*n+i-1)=*(M+(p-1)*n+k-1);
         *(M+(p-1)*n+k-1)=t;
      }
   }
   return *this;
}

//-----------------------------------------------------------------------------
int max(int i, int k) { if (i<k) return(k); else return(i); }

#endif
```

Symbols and Abbreviations

Abbreviations

AOP	autoregressive ordered probit
GM	grouped move
GM-MGMC	grouped move multi-grid Monte Carlo
i.i.d.	independent and identically distributed
MCMC	Markov chain Monte Carlo
MH	Metropolis-Hastings
OSV	Ordinal-response Stochastic Volatility
OSVt	Ordinal-response Stochastic Volatility with t-distributed errors

Functions

$f(x)\|_{x=v}$	$f(x)$ evaluated at $x = v$
$\mathbb{1}_I(x)$	indicator function for interval or set I, evaluated at x (=1, if $x \in I$, else =0)
$\mathbb{1}_{\{condition(x_1,\ldots,x_n)\}}$	indicator function, evaluated at $(x_1, \ldots, x_n)$ (=1, if $condition(x_1, \ldots, x_n)$ is true, else =0)

Special symbols, vectors, and matrices

$(\cdot)'$	transposition
$\equiv$	equal for all indices (e.g. $Z_t \equiv 1$ abbreviates $Z_t = 1 \; \forall \, t$)
$\mathbf{0}$	vector where all components equal zero, $\mathbf{0} := (0, \ldots, 0)'$
$\text{diag}(d_1, \ldots, d_n)$	diagonal matrix with diagonal elements $d_1, \ldots, d_n$
$\mathcal{F}_t$	vector of observations until time t, $\mathcal{F}_t := (y_1, \ldots, y_t)$

Symbols for random variables and distributions

$\text{Bin}(n, p)$	Binomial distribution, success with probability p, n trials

χ^2_ν	Chi-Squared distribution with ν degress of freedom
Dirac(x)	Dirac measure for x, i.e. $X \sim \text{Dirac}(x) \Rightarrow P(X = x) = 1$
$\Gamma(a, b)$	Gamma distribution with parameters a and b
NegBin(μ, α)	Negative Binomial distribution with parameters μ and α
$N(\mu, \sigma^2)$	normal distribution with mean μ and variance σ^2
$N(x\|\mu, \sigma^2)$	density of normal distribution with mean μ and variance σ^2, evaluated at x
$N_{[a,b]}(\mu, \sigma^2)$	normal distribution with mean μ and variance σ^2 truncated to the interval $[a, b]$
$N_p(\boldsymbol{\mu}, \Sigma)$	p-variate normal distribution with mean $\boldsymbol{\mu}$ and covariance matrix Σ
$N_p(\boldsymbol{x}\|\boldsymbol{\mu}, \Sigma)$	density of p-variate normal distribution with mean $\boldsymbol{\mu}$ and covariance matrix Σ, evaluated at $\boldsymbol{x}$
$t_\nu(a, b^2)$	Student-t distribution with ν degrees of freedom and location and scale parameters a and b, respectively
$t_\nu(x\|a, b^2)$	density of Student-t distribution with ν degrees of freedom and location and scale parameters a and b, respectively, evaluated at x
Unif(a, b)	Uniform distribution with boundaries a and b

Symbols for sets

$\mathbb{R}$	set of all real-valued numbers
$\mathbb{N}$	set of all natural numbers, $\{1, 2, 3, \ldots\}$
$\mathbb{N}_0$	set of all natural numbers including zero, $\{0, 1, 2, 3, \ldots\}$

Bibliography

Albert, J. H. and S. Chib (1993). Bayesian Analysis of Binary and Polychotomous Response Data. *Journal of the American Statistical Association 88*, 669–679.

Bauwens, L. and P. Giot (2000). The Logarithmic ACD model: an application to the bid-ask quote process of three NYSE stocks. *Annales d'Economie et de Statistique 60*, 117–149.

Bauwens, L. and P. Giot (2001). *Econometric Modelling of Stock Market Intraday Activity*. Boston: Kluwer Academic Publishers.

Bauwens, L. and D. Veredas (2004). The stochastic conditional duration model: a latent variable model for the analysis of financial durations. *Journal of Econometrics 119*, 381–412.

Breiman, L. (1992). *Probability*. Philadelphia: SIAM.

Chen, M.-H., Q.-M. Shao, and J. G. Ibrahim (2000). *Monte Carlo Methods in Bayesian Computation*. New York: Springer.

Chib, S. (1995). Marginal Likelihood from the Gibbs Output. *Journal of the American Statistical Association 90*, 1313–1321.

Chib, S. and E. Greenberg (1994). Bayes inference for regression models with ARMA(p,q)-errors. *Journal of Econometrics 64*, 183–206.

Chib, S. and I. Jeliazkov (2001). Marginal Likelihood from the Metropolis-Hastings Output. *Journal of the American Statistical Association 96*, 270–281.

Chib, S., F. Nardari, and N. Shephard (2002). Markov chain Monte Carlo methods for stochastic volatility models. *Journal of Econometrics 108*, 281–316.

Cowles, M. K. (1996). Accelerating Monte Carlo Markov chain convergence for cumulative-link generalized linear models. *Statistics and Computing 6*, 101–111.

Dacorogna, M. M., R. Gençay, U. A. Müller, R. B. Olsen, and O. V. Pictet (2001). *An Introduction to High Frequency Finance*. San Diego: Academic Press.

De Jong, P. and N. Shephard (1995). The simulation smoother for time series models. *Biometrika 82*, 339–350.

Diamond, D. W. and R. E. Verrecchia (1987). Constraints on short-selling and asset price adjustment to private information. *Journal of Financial Economics 18*, 277–311.

Easley, D. and M. O'Hara (1987). Price, trade size, and information in security markets. *Journal of Financial Economics 19*, 113–138.

Elerian, O., S. Chib, and N. Shephard (2001). Likelihood inference for discretely observed non-linear diffusions. *Econometrica 69*, 959–994.

Engle, R. F. and J. R. Russell (1998). Autoregressive conditional duration; a new model for irregularly spaced transaction data. *Econometrica 66*, 1127–1162.

Eraker, B. (2001). MCMC analysis of diffusion models with applications to finance. *Journal of Business and Economic Statistics 19*, 177–191.

Gelman, A., J. B. Carlin, H. S. Stern, and D. B. Rubin (1995). *Bayesian Data Analysis*. London: Chapman and Hall.

Geweke, J. (1991). Efficient Simulation from the Multivariate Normal and Student-t Distribution Subject to Linear Constraints and the Evaluation of Constraint Probabilities. *Computing Science and Statistics: Proceedings of the Twenty-Third Symposium on the Interface 23*, 571–578.

Gilks, W. R., S. Richardson, and D. J. Spiegelhalter (1996). *Markov Chain Monte Carlo in Practice*. London: Chapman and Hall.

Gordon, N. J., D. J. Salmond, and A. F. M. Smith (1993). A Novel Approach to Non-Linear and Non-Gaussian Bayesian State Estimation. *IEEE Proceedings F 140*, 107–113.

Harvey, A. C. (1989). *Forecasting, structural time series models and the Kalman filter*. Cambridge: Cambridge University Press.

Hausman, J. A., A. W. Lo, and A. C. MacKinlay (1992). An ordered probit analysis of transaction stock prices. *Journal of Financial Economics 31*, 319–379.

Jacquier, E., N. G. Polson, and P. E. Rossi (1994). Bayesian analysis of stochastic volatility models. *Journal of Economic and Business Statistics 12*, 371–389.

Jasiak, J. (1998). Persistence in intertrade durations. *Finance 19*, 166–195.

Jeffreys, H. (1961). *Theory of Probability* (3rd ed.). Oxford: Clarendon Press.

Kim, S., N. Shephard, and S. Chib (1998). Stochastic volatility: likelihood inference and comparison with ARCH models. *Review of Economic Studies 65*, 361–393.

Kitagawa, G. (1996). Monte Carlo Filter and Smoother for Non-Gaussian Non-Linear State Space Models. *Journal of Computational and Graphical Statistics 5*, 1–25.

Lagarias, J. C., J. A. Reeds, M. H. Wright, and P. E. Wright (1998). Convergence properties of the Nelder-Mead simplex method in low dimensions. *SIAM Journal on Optimization 9*, 112–147.

Liu, J. S. and C. Sabatti (2000). Generalized Gibbs sampler and multigrid Monte Carlo for Bayesian computation. *Biometrika 87*, 353–369.

Nelder, J. A. and R. Mead (1965). A simplex method for function minimization. *Computer Journal 7*, 308–313.

O'Hara, M. (1995). *Market Microstructure Theory*. Malden: Blackwell Publishers.

Pitt, M. K. and N. Shephard (1999). Filtering via Simulation: Auxiliary Particle Filters. *Journal of the American Statistical Association 94*, 590–599.

Rao, M. M. (1987). *Measure Theory and Integration*. New York: Wiley.

Ritter, C. and M. A. Tanner (1992). Facilitating the Gibbs sampler: The Gibbs Stopper and the Griddy-Gibbs sampler. *Journal of the American Statistical Association 87*, 861–868.

Robert, C. P. (1995). Simulation of truncated normal variables. *Statistics and Computing 5*, 121–125.

Robert, C. P. and G. Casella (2000). *Monte Carlo Statistical Methods*. New York: Springer.

Roberts, G. O. and R. L. Tweedie (1996). Geometric convergence and Central Limit Theorems for multidimensional Hastings and Metropolis algorithms. *Biometrika 83*, 95–110.

Rydberg, T. H. and N. Shephard (2003). Dynamics of trade-by-trade price movements: decomposition and models. *Journal of Financial Econometrics 1*, 2–25.

So, M. K. P., K. Lam, and W. K. Lee (1998). A stochastic volatility model with Markov switching. *Journal of Business and Economic Statistics 16*, 244–253.

Spiegelhalter, D. J., N. G. Best, B. P. Carlin, and A. van der Linde (2002). Bayesian measures of model complexity and fit. *Journal of the Royal Statistical Society B 64*, 583–639.

Tauchen, G. E. and M. Pitts (1983). The price variability-volume relationship on speculative markets. *Econometrica 51*, 485–505.

Taylor, S. J. (1986). *Modelling Financial Time Series*. Chichester: John Wiley.

Tierney, L. (1994). Markov Chains for Exploring Posterior Distributions. *The Annals of Statistics 22*, 1701–1728.

Walters, F. H., L. R. Parker, S. L. Morgan, and S. N. Deming (1991). *Sequential Simplex Optimization*. Boca Raton: CRC Press.

Zeger, S. L. and M. R. Karim (1991). Generalized linear models with random effects: a Gibbs sampling approach. *Journal of the American Statistical Association 86*, 79–86.

Index