CHALMERS | GÖTEBORG UNIVERSITY

MASTER'S THESIS

Stepsize Controlled Schemes for Diffusions exhibiting Volatility Induced Stationarity

RICKARD KJELLIN

Department of Mathematical Statistics
CHALMERS UNIVERSITY OF TECHNOLOGY
GÖTEBORG UNIVERSITY
Göteborg, Sweden 2005

Thesis for the Degree of Master of Science (20 credits)

Stepsize Controlled Schemes for Diffusions exhibiting Volatility Induced Stationarity

Rickard Kjellin

CHALMERS | GÖTEBORG UNIVERSITY

Department of Mathematical Statistics
Chalmers University of Technology and Göteborg University
SE – 412 96 Göteborg, Sweden

Göteborg, September 2005

Stepsize controlled schemes
for diffusions exhibiting volatility induced stationarity

Rickard Kjellin

September 4, 2005

Abstract

This thesis investigates two types of adaptive numerical integration scheme for certain stochastic differential equations exhibiting volatility induced stationarity. One is a simple scheme restricting the step length based on the magnitude of the current process value. The other uses Brownian path interpolation. It is indicated by empirical studies that the stochastic differential equation underlying the CKLS model can be numerically solved using the first type of scheme. The second type of scheme seems not to work for these types of problems.

Acknowledgements

First I thank Professor Patrik Albin for being the best supervisor a student could ever ask for. Without all the help you have generously given, this thesis would not have come to be. I would also like to thank Anders Muszta for helping me and answering questions I have had concerning numerical stochastic calculus and Christian Larsson for much needed help with computer matters and programming difficulties. I also thank Mattias Bengtsson and Johan Tykesson who, together with Prof. Patrik Albin, superbly introduced me to the field of stochastic calculus. In addition I thank my girlfriend, family and friends for the support they have given during the many hours ploughed into this work.

Contents

Chapter 1

Introduction

In this thesis we present three different stochastic differential equations whose solution exhibits a special property denoted volatility induced stationarity. These types of equations are notoriously difficult to numerically integrate with standard methods due to their volatile behaviour. The aim of the thesis is thus to investigate special types of numerical schemes that can handle this behaviour. Two different schemes are presented, both being so called adaptive schemes. The idea is to discretize the stochastic differential equations on an uneven spaced time grid such that the accuracy is improved during periods of increased volatility.

In the first part of the thesis, the necessary background on standard numerical solution methods for stochastic differential equations is introduced together with an overview of the three diffusions the thesis aims to provide appropriate numerical methods for. In the second part two adaptive schemes are introduced and their performance is evaluated.

Chapter 2

Theoretical foundations

2.1 Mathematical preliminaries and notation

Throughout the thesis a filtered probability space $(\Omega, \mathfrak{F}, \{\mathfrak{F}_t\}_{t\in T}, \mathbb{P})$, is assumed. All models are driven by a scalar, standard Brownian motion B_t started at zero. For a wealth of information about Brownian motion processes as well as stochastic differential equations, the reader is refered to [7].

Stochastic differential equations

The goal of the thesis is to find stable numerical schemes for the discretization of certain stochastic differential equations. We work with equations of the type

$$\mathrm{d}X_t = \mu(X_t)\mathrm{d}t + \sigma(X_t)\mathrm{d}B_t. \tag{2.1}$$

Is is important to recall that the notation above is merely a shorthand for the more rigoruous integral notation

$$X_t = \int_0^t \mu(X_s)\mathrm{d}s + \int_0^t \sigma(X_s)\mathrm{d}B_s. \tag{2.2}$$

We start the process at

$$X_0 = \zeta, \quad \zeta \text{ random variable measurable wrt. } \mathfrak{F}_0.$$

2.2 Numerical integration of stochastic differential equations

In order to obtain an approximate solution to a stochastic differential equation one natural way to proceed is to use time stepping schemes similar to those used for discretizing ordinary differential equations. In essence, one makes a Taylor expansion[1] of the diffusion at a first time point and computes the approximate process value at a consecutive, choosen point based on the Taylor expansion. In the next step one repeats the above scheme, but now uses the approximated point as the center for the new Taylor expansion. Continuing recursively, one build up the sample path point by point.

Finally, one ends up with a process, $\widetilde{X}_{t_n}$, defined on the choosen time points, approximating the true solution X_{t_n} at those points. To make analysis of the schemes easier, the process $\widetilde{X}$, defined on the discrete set of points $\{t_n\}_{n\in K\subset\mathbb{Z}_+}$, is modified by an extension of its domain of definition to a subset of the real numbers by linear interpolation, making it a continuous process.

2.2.1 Equidistant first order schemes

The most simple numerical schemes employs a low order Taylor expansion on an evenly spaced grid of discretization time points. The so called *Euler-Maruyama* scheme was first proposed in [9] and is a stochastic version of the simple Euler time stepping method from computational ordinary differential equations. Only terms up the the first order stochastic terms are used.

Consider the case where we want to discretize equation (2.1) on the interval $[0, T]$. We first determine the number of points, N, to calculate and make an equidistant partition, Π_N, of $[0, T]$ in the following way

$$\Pi_N : \quad 0 = t_0 < \ldots < t_k < t_{k+1} < \ldots < t_N = T \tag{2.3}$$

$$\Delta t_k = t_k - t_{k-1} = \frac{T}{N}, \ k \in \{1, \ldots, N\} \tag{2.4}$$

The method uses a first order Taylor expansion and have the following formula

$$\begin{aligned} \mathrm{X}^N_{t_{n+1}} &= \mathrm{X}^N_{t_n} + \mu(\mathrm{X}^N_{t_n})\mathrm{X}^N_{t_n}\Delta t_{k+1} + \sigma(\mathrm{X}^N_{t_n})\Delta B_{t_{n+1}} \\ \Delta B_{t_{n+1}} &= B_{t_{n+1}} - B_{t_n} = \sqrt{\Delta t_{k+1}} \times \mathcal{G}, \ \mathcal{G} \sim \mathrm{N}(0, 1). \end{aligned} \tag{2.5}$$

[1]The Brownian component of the equation calls for a stochastic variant of the usual Taylor expansion called the Itô-Taylor expansion. Although technically different, the intuition behind them in the current context is the same.

Convergence properties

A natural question when employing such a scheme as the Euler-Maruyama is whether the obtained solution is a good approximation to the theoretical solution of the discretized equation. Moreover, it is important to understand how the dicretization error is affected by changes to the setup, such as making the partition finer or coarser. Since the number of computations rises with finer partitions, ideally one would like to be able to balance the demand for computational resources with the error tolerance in an efficient manner.

There exist many ways to measure the discretization error in the literature. One of the more common measures, which allows for an easy analysis of the Euler-Maruyama scheme applied to a certain class of diffusions, is the $\mathcal{L}^2$-error,

$$\mathcal{L}^2\text{-error} = \mathbb{E}\left[\sup_{\tau\in[0,T]} |\widetilde{X}_\tau - X_\tau|^2\right]^{\frac{1}{2}} \tag{2.6}$$

2.2.2 Higher order, explicit schemes

The Euler-Maruyama scheme only utilizes the first terms of the Taylor expansion. Higher order schemes also truncates the expansion, but retains more terms for added accuracy and faster convergence. In the thesis a 1.5 order scheme is used. We used the definition of the 1.5 order scheme from [8]. Note that this version of the scheme requires $\mu(x)$ and $\sigma(x)$ to be differentiable. Letting $\Delta Z_{t_{n+1}}$ denote a double Itô integral, the scheme is of the following form

$$\begin{aligned}
\mathrm{X}^N_{t_{n+1}} &= \mathrm{X}^N_{t_n} + \mu(\mathrm{X}^N_{t_n})\Delta t_{k+1} + \sigma(\mathrm{X}^N_{t_n})\Delta B_{t_{n+1}} + \frac{1}{2}\sigma(\mathrm{X}^N_{t_n})\sigma'(\mathrm{X}^N_{t_n}) \\
&+ \mu'(\mathrm{X}^N_{t_n})\sigma(\mathrm{X}^N_{t_n})\Delta Z_{t_{n+1}} + \frac{1}{2}\left(\mu(\mathrm{X}^N_{t_n})\mu'(\mathrm{X}^N_{t_n}) + \frac{1}{2}\sigma^2(\mathrm{X}^N_{t_n})\mu''(\mathrm{X}^N_{t_n})\right)\Delta t^2_{k+1} \\
&+ \left(\mu(\mathrm{X}^N_{t_n})\sigma'(\mathrm{X}^N_{t_n}) + \frac{1}{2}\sigma^2(\mathrm{X}^N_{t_n})\sigma''(\mathrm{X}^N_{t_n})\right) \times \left(\Delta B_{t_{n+1}}\Delta t_{k+1} - \Delta Z_{t_{n+1}}\right) \\
&+ \frac{1}{2}\sigma(\mathrm{X}^N_{t_n})\left(\sigma(\mathrm{X}^N_{t_n})\sigma''(\mathrm{X}^N_{t_n}) + (\sigma'(\mathrm{X}^N_{t_n}))^2\right) \times \left(\frac{1}{3}(\Delta B_{t_{n+1}})^2 - \Delta t_{k+1}\right)\Delta B_{t_{n+1}},
\end{aligned} \tag{2.7}$$

where

$$\begin{aligned}
\Delta B_{t_{n+1}} &= B_{t_{n+1}} - B_{t_{n+1}} = \sqrt{\Delta t_{k+1}} \times \mathcal{G}_1, \\
\Delta Z_{t_{n+1}} &= \frac{1}{2}\Delta t^{3/2}_{k+1}\left(\mathcal{G}_1 + \frac{1}{\sqrt{3}}\mathcal{G}_2\right).
\end{aligned}$$

Here $\mathcal{G}_1$ and $\mathcal{G}_2$ are independent, standard normal variables.

2.2.3 Implicit first order schemes

With implicit schemes, one does not simply compute the Itô-Taylor expansion to extract the process value for the next point. Instead of using the left interval endpoint in the approximation of the stochastic integral (which renders the Itô integral in the limit), the right endpoint is used. To ensure convergence to the Itô solution correction terms must be embedded into the scheme. Using the same settings as for the explicit Euler scheme, the expression is of the form

$$\begin{aligned} \mathrm{X}^N_{t_{n+1}} &= \mathrm{X}^N_{t_n} + (\mu(\mathrm{X}^N_{t_{n+1}}) - \sigma(\mathrm{X}^N_{t_{n+1}})\sigma'(\mathrm{X}^N_{t_{n+1}}))\Delta t_{k+1} + \sigma(\mathrm{X}^N_{t_{n+1}})\Delta B_{t_{n+1}} \\ \Delta B_{t_{n+1}} &= B_{t_{n+1}} - B_{t_{n+1}} = \sqrt{\Delta t_{k+1}} \times \mathcal{G},\ \mathcal{G} \sim \mathrm{N}(0,1). \end{aligned} \tag{2.8}$$

Apparent from the formula, at each iteration a possibly nonlinear algebraic equation must be solved. This is one disadvantage of the implicit types of schemes. The implicit schemes are stable for a much larger class of stochastic differential equations and for greater step lengths.

2.2.4 Stepsize controlled schemes

The schemes discussed so far have all been equidistant schemes, meaning that the time interval have been partitioned evenly. This is a restriction that can be relaxed. Allowing the step length to vary over the time interval can for example reduce the discretization error and improve the convergence properties. Regulating the step length may be done by using an adaptive scheme. In [8], the family of adaptive schemes shares the common trait that the step length, Δt_{n+1}, is determined based on the information in the filtration $\mathfrak{F}_{t_n}$.

2.3 Volatility Induced Stationarity

A number of stochastic processes has a property denoted *volatility induced stationarity*, or VIS. This concerns the nature of the stochastic movements of the trajectory and gives the processes an entirely different behaviour from stochastic differential equations without the VIS property, which in some sense behaves like ordinary first order differential equations under stochastic perturbations. Processes exhibiting VIS has a dispersion term, $\sigma(x)$, that dominates the behaviour of the dynamics and actually forces the process into stationarity under certain conditions. This is in contrast to other stationary diffusions, where it is the drift term, $\mu(x)$, that gives the process its mean-reverting property. The notion of volatility induced stationarity was introduced in [4]

2.3.1 Numerical integration of VIS diffusions

The special properties of the VIS diffusions makes numerical integration of the underlying differential equations an especially tricky matter. It is evident (see [10]) that special tricks has to be used to successfully make accurate discretizations of the sample paths.

The main problem with VIS models is that ordinary equidistant, explicit methods such as scheme (2.5) fails, and is in fact transient, with positive probability no matter how small step length is used. For a proof of the transcience of scheme (2.5), see [10].

Implicit schemes

One remedy for the instability issues is to employ an implicit numerical scheme. This has been done with positive results in [10]. A problem with the implicit schemes is that they seem to underestimate some of the characteristic properties of the models.

2.3.2 The CKLS model

One model exhibiting excessive VIS behaviour is the short rate-model proposed in [2] by Chan, Koralyi, Longstaff and Sanders. Is is a generalization of the CIR model proposed by Cox, Ingersoll and Ross in [3]. In differential notation the model obeys the following stochastic differential equation

$$\mathrm{d}X_t = (\alpha - \beta X_t)\mathrm{d}t + \sigma X_t^{\gamma}\mathrm{d}B_t \tag{2.9}$$

with

$$X_0 = \xi, \quad \mathbb{E}\left[|\xi|\right] < \infty.$$

See [10] for restrictions on the parameters. A typical trajectory started at $X_0 = 1$ (with $\alpha = (-\beta) = \sigma = 1$ and $\gamma = 3$) is shown in Figure 2.1. Depending on the parameter γ, the trajectories are more or less spikey. The model is mean reverting and, as indicated by the figure, has a stationary distribution. The mean-reversion stems both from the VIS property and, depending on the parameters, the drift term. The stationary distribution is proportional to speed measure given by (see [10])

$$\frac{m(x)}{\mathrm{d}x} = 2x^{2\gamma}\exp\left(\frac{2\alpha}{\sigma^2(1-2\gamma)}x^{1-2\gamma} + \frac{2\beta}{\sigma^2(2-2\gamma)}x^{2-2\gamma}\right) \tag{2.10}$$

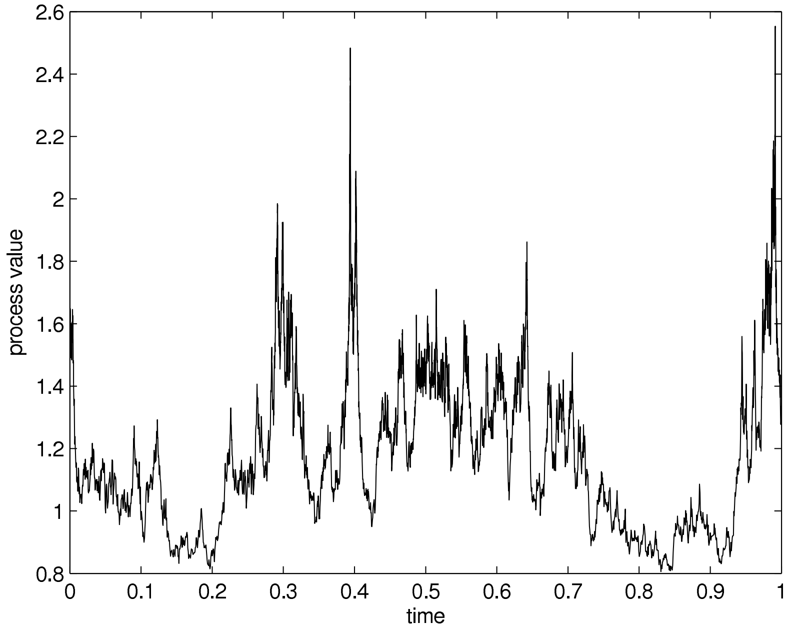

Figure 2.1: CKLS trajectory

Electricity price time series

In the econometric modeling of time series from the electricity markets, certain peculiarities are present. Because of the seasonal demand, a cyclical trend is often notised. There also seem to be a strong reversion to some mean level. It is also a fact that the price trajectories often exhibit a spikey behaviour. For more of the characteristics of electricity prices, see [6]. It is proposed that such time series, void of their cyclical trends, could be modeled by the CKLS model using a high value for γ.

2.3.3 The Hyperbolic model

The Hyperbolic diffusion model is discussed in [1] as a model proposed for stock prices. The underlying stochastic differential equation is of the form

$$\mathrm{d}X_t = \sigma \exp\left\{ \frac{1}{2} \left(\alpha \sqrt{\delta^2 + (X_t - \mu)^2} - \beta (X_t - \mu) \right) \right\} \mathrm{d}B_t \tag{2.11}$$

with

$$X_0 = \xi$$

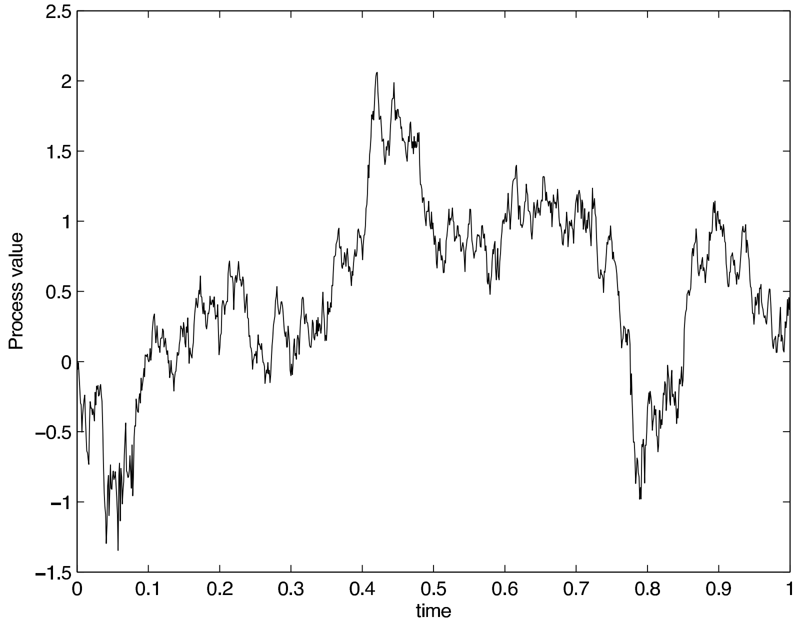

Figure 2.2: Hyperbolic trajectory

and

$$\alpha > |\beta| \geq 0, \ \ \delta, \sigma > 0, \ \ \mu \in \mathbb{R}. \tag{2.12}$$

A typical trajectory is shown in Figure 2.2. It lacks much of the spikey characteristics of the high-γ CKLS trajectories. The stationary density for the model is, as the name suggests, hyperbolic and has the following distribution function,

$$\mathbb{P}(X_t \in A) = \int_A \sigma^2 \exp\left\{-\left(\alpha\sqrt{\delta^2 + (x-\mu)^2} - \beta(x-\mu)\right)\right\} \mathrm{d}x, \tag{2.13}$$

assuming that ξ also follows the same distribution.

On 1.5 order simulations

In [1], it is claimed that the hyperbolic diffusions have been successfully discretized using higher order explicit schemes. However, we have experienced instabilities using such a scheme, even for very short step lengths. See the appendix for a Matlab routine showing such behaviour.

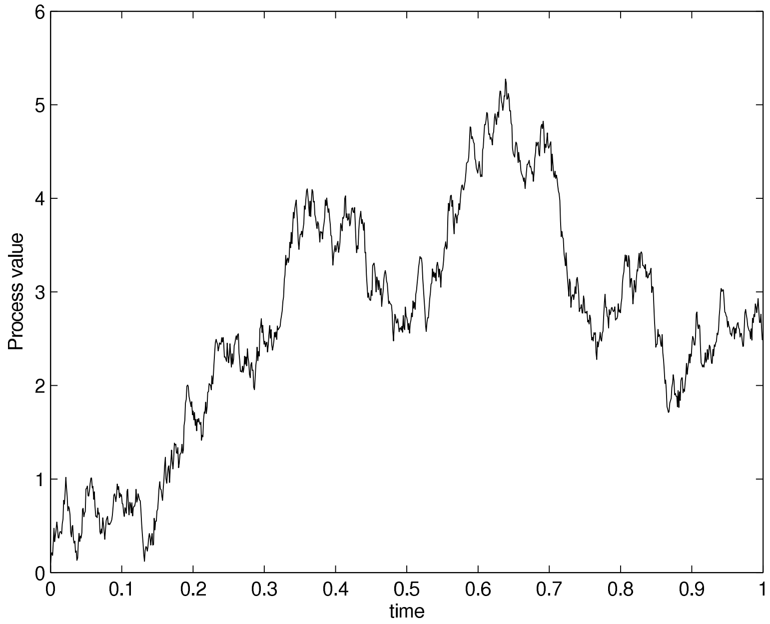

Figure 2.3: Heavy-tailed trajectory

2.3.4 Heavy-tailed diffusion

A class of diffusions discussed in [10] is the Heavy-tailed diffusions. They obey the following stochastic differential equation

$$\mathrm{d}X_t = 3X_t^a \mathrm{d}t + 3X_t^{2/3}\mathrm{d}B_t \tag{2.14}$$

with

$$X_0 = \xi, \quad \xi \text{ stricty positive and } \alpha < \frac{1}{3}.$$

The process lives on the positive halfline. Inspection of the trajectory in figure 2.3 reaveals the characteristic aversion from the zero level. The trajectory spends considerably more time at large values than typical high-γ CKLS trajectories.

The stationary density is proportional to the speed measure given by

$$\frac{m(x)}{\mathrm{d}x} = \frac{2}{9}x^{-4/3}\exp\left(\frac{2}{3a-1}x^{a-1/3}\right). \tag{2.15}$$

Chapter 3

Stepsize controlled schemes

This chapter describes two numerical schemes and gives an empirical analysis of their application to the three diffusion models introduced in the last chapter.

3.1 Simple scheme

The first of the proposed step size controlled schemes is a simple modification of the equidistant Euler-Maruyama scheme. Since the CKLS model exhibits wild fluctuations when the process inhabits higher values the obvious step size adaptation is to decrease the step size proportional to the magnitude of the process. Therefore, we suggest the following scheme

$$\mathrm{X}^N_{t_{n+1}} = \mathrm{X}^N_{t_n} + \mu(\mathrm{X}^N_{t_n}, t_n)\mathrm{X}^N_{t_n}\widetilde{\Delta} t_{k+1} + \sigma(\mathrm{X}^N_{t_n}, t_n)\widetilde{\Delta} B_{t_{n+1}} \tag{3.1}$$

$$\widetilde{\Delta} t_{k+1} = \frac{\Delta}{1 + f(|\mathrm{X}^N_{t_n}|)}, \quad f(\cdot) \text{ monotonous and increasing} \tag{3.2}$$

$$\widetilde{\Delta} B_{t_{n+1}} = B_{t_{n+1}} - B_{t_n}. \tag{3.3}$$

In general one would add the term K_{realmin} to the right hand side of (3.2), where K_{realmin} is some small constant corresponding to the minimum step length. This is to avoid the algorithm from halting should the discretized process reach too large values and thereby force the step length to zero because of truncation. On most computer systems, one may extract the smallest floating point number the system can represent and use this as minimum step length.

3.1.1 The CKLS model

The scheme tends to work fairly well for the CKLS model. For moderate parameter values the method is both fast and stable. For $\alpha = (-\beta) = \sigma = 1$ and $\gamma = 3$ and $f(x) = x^p$, $p \approx 2\gamma$, in equation 3.2, over 100000 runs, using a base step length of $\Delta t = 2^{-8}$, has been successfully made without any instability problems.

For more extreme values of γ, it seems that the scheme also works well. With $\gamma = 40$, $\alpha = \beta = \sigma = 1$ and $f(x) = x^{80}$, over 500 consecutive runs were successfully made without instability problems. A step length of $\Delta t = 2^{-10}$ were used. When the process leaves the center of its stationary distribution, it rises very fast to high levels, but the instantaneous decimation of the step length greatly lowers the probability for instability. The high volatility quickly forces the process down towards the stationary level, increasing the step length. This makes the scheme relatively fast.

The largest problems seems to be for $\gamma \in [15, 25]$. Here the process stays at high levels during longer periods of time, forcing down the step length to very small levels. This gives a major speed hit, and the discretizations can go into an almost halted state, due to the many calculations.

3.1.2 The Hyperbolic model

The scheme tends to not work well for the hyperbolic model. Using a monomial in equation 3.2 like for the CKLS model is not working. Instabilities then occur almost for every trajectory. Taking into account the need for a steeper reduction in step length for large, absolute process values, suitable functions include the exponential function. Using $f(x) = \exp\{x^p\}$, the step length will be reduced in a very aggressive manner. This, however, did not prove to be sufficient to ensure stability. Using moderate step lengths, like $\Delta t = 2^{-10}$, still generated a large proportion of unstable discretizations. Apart from the risk of the discretization exploding in finite time, the scheme also displayed problem with large jumps from positive to negative values and vice versa. Increasing the parameter p to values larger than 1.5 made the scheme virtually unusable due to the small step lengths[1].

Decreasing the base step length drastically, to values around $\Delta t = 2^{-20}$ and smaller, seems to suppress most of these problems. There are however other problems, like computation speed, that arises for such small step lengths.

[1]The memory demands of such discretizations on the unit interval were in several cases over 1 Gigabyte

3.1.3 Heavy-tailed diffusion

Integration of the heavy-tailed diffusion imposes two numerical problems. First there is a risk for instability when the process is at high values. This trait is shared with both the CKLS model and the hyperbolic model. Furthermore, when the schemes close in near zero, there is a risk of instability if the scheme hits or crosses the zero boundary. In order to cope with this type of instability, the scheme has to be somewhat modified. Ignoring the risk of explosion, equation 3.2 is rewritten as

$$\widetilde{\Delta} t_{k+1} = \Delta t_{k+1} f(|\mathrm{X}^N_{t_n}|), \tag{3.4}$$

$f(\cdot)$ monotonous and increasing with $f(0) = 0$ for $\mathrm{X}^N_{t_n} < 1$, and

$$\widetilde{\Delta} t_{k+1} = \Delta \text{ otherwise.} \tag{3.5}$$

This seems to be the trickiest model to discretize. Using moderate step lengths, $\Delta = 2^{-N}$, $N \in \{8, \dots, 16\}$, instability is highly probable if the scheme drops below 0.5.

Using smaller step lengths is a partial remedy. For the parameter a negative and close to zero, even a very small step length, 2^{-24} can not ensure stability. Starting the process at a small value, for example $X_0 = 0.1$, will lead to instability for a large proportion of the discretizations.

However, for $a = -10$ and $X_0 = 1$, over 50 consecutive trajectories has been simulated without instability. This is probably due to the large upward force from the drift term when the scheme drops below 1. It is indicated by repeated simulation that discretizations using large negative values of a, $a \approx -10$, are less prone to instability than smaller negative values in the approximate range $a \in [-4, 0]$.

3.1.4 Empirical test of convergence rates

In order to assess the quality of the scheme, its convergence properties are investigated by applying the scheme to an analytically solvable stochastic differential equation. The simple example of Geometric Brownian Motion is chosen. It has the following form

$$\mathrm{d}X_t = \mu X_t \mathrm{d}t + \sigma X_t \mathrm{d}B_t \tag{3.6}$$

where

$$\mu, \sigma \in \mathbb{R}.$$

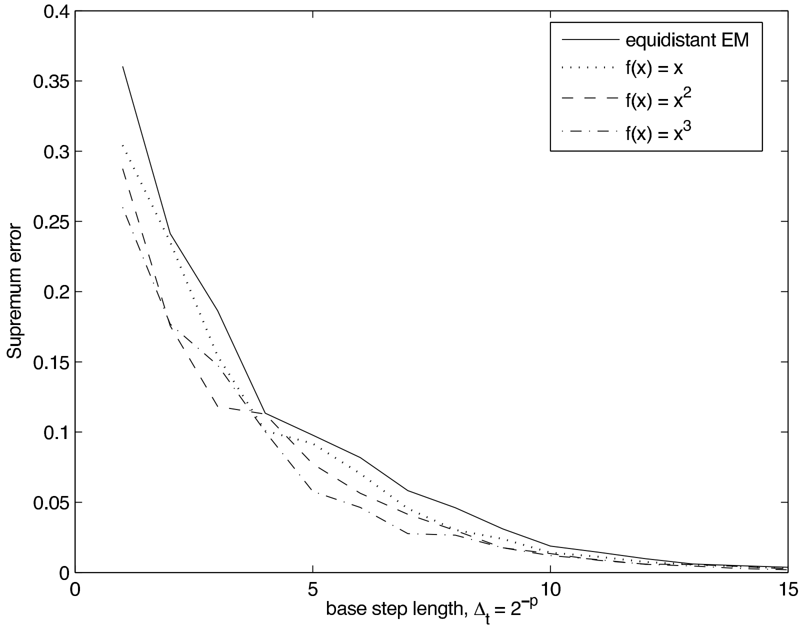

Figure 3.1: Convergence rates

This equation has an analytic solution which can be derived using Itô's formula. For a fixed initial value, $X_0 = \chi$, the solution is

$$X_t = X(0)\exp\left\{(\mu - \frac{\sigma^2}{2})t + \sigma B_t\right\},\ B_0 = 0 \tag{3.7}$$

Now, as long as the Brownian motion is retained from the discretizations, the exact solution can be computed and used as a reference. To this end, we discretized the equation 3.6 with $\mathrm{X}_0 = \mu = \sigma = 1$. The discretizations were performed on the unit interval with a decreasing sequence of base step lengths, $2^{-4}, 2^{-5}, \ldots, 2^{-15}$. For each stepsize, the absolute supremum error, $\sup_{s\in[0,1]} |\mathrm{X}_s - \mathrm{X}_s^N|$, is calculated for 100 trajectories and the mean of these errors is shown in Figure 3.1. Starting with the unmodified Euler-Maruyama scheme, the square of the error decreases linearly as predicted by the theory.

The adaptive scheme was employed using a monomial of increasing order in equation 3.2, $f(x) = x^p$ with $p \in \{1, 2, 3\}$. We see that the error decreases approximately as for the unmodified scheme. Also, the higher the order of the monomial, the lower the error. The conclusion drawn from this test is that the scheme seems to converge to the true solution, at least for easily discretized equations like the Geometric Brownian motion.

It should be noted though that since the solution on average behaves like an expontial, the error of the adaptive scheme depends on the trend, $\alpha = (\mu - \frac{\sigma^2}{2})$ since the step length is reduced for larger values. This makes it hard to make conclusions concerning the possible error reductions from the adaptiveness in the case of general diffusions.

3.1.5 Empirical test for stationarity

In order to examine the stationary behaviour of the discretized solutions, the empirical distribution functions of the discretized trajectories will be compared to the theoretical stationary densities of the models. The test will be based on the *Kolmogorov-Smirnoff distance*. For more information on the empirical distribution function and the Kolmogorov-Smirnoff distance, see for instance [11]. The implementations were done by normalizing the speed measures in *Mathematica* and then using numerical quadrature and the function *ecdf()* in Matlab.

For the CKLS model, the absolute distance between the empirical and theoretical is shown in Figure 3.2. The parameters used are $\alpha = \beta = \sigma = 1$ and $\gamma = 3$. The trajectories were started at $X_0 = 1$. The empirical distribution function was calculated using 100000 trajectories. The Kolmogorov-Smirnoff distance is in this setup

$$\text{K-S distance}_{\text{CKLS}} = 0.0065. \tag{3.8}$$

We see that the difference between the theoretical and empirical stationary distribution is at its largest for value around the stationary level. This seems to be systematic for various parameters, although the cause is unknown.

For the hyperbolic model, the result is disappointing. Because of the demand for very short step lengths, it unpractical to simulate more than 1000 trajectories. Even then, around 15 percent of the trajectories had to be discarded due to instabilities. The parameters used are $\mu = 0$, $\sigma = \delta = 1$, $\beta = -1.5$ and $\alpha = 2$. The absolute value of the difference between theoretical and empirical stationary distribution is shown in Figure 3.3. It is apparent that the scheme display properties far from those of the theoretical model. The reason for this is unknown. The Kolmogorov-Smirnoff distance is

$$\text{K-S distance}_{\text{Hyperbolic}} = 0.1844. \tag{3.9}$$

Equally disappointing is the result for the heavy-tailed model. The analysis is done using the parameter value $a = -6$. The process is started at $X_0 = 1$ and 1000 trajectories were discretized. There is no apparent connection between the theoretical and empirical cumulative distribution functions as indicated by Figure 3.4. The Kolmogorov-Smirnoff distance is

$$\text{K-S distance}_{\text{Heavy-tailed}} = 0.4432. \tag{3.10}$$

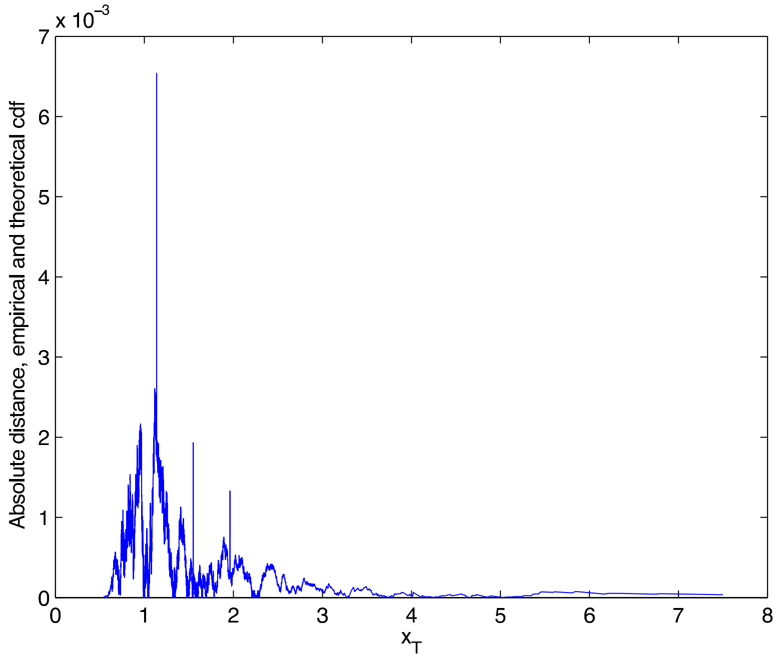

Figure 3.2: Comparison between theoretical and empirical CDF for the CKLS model

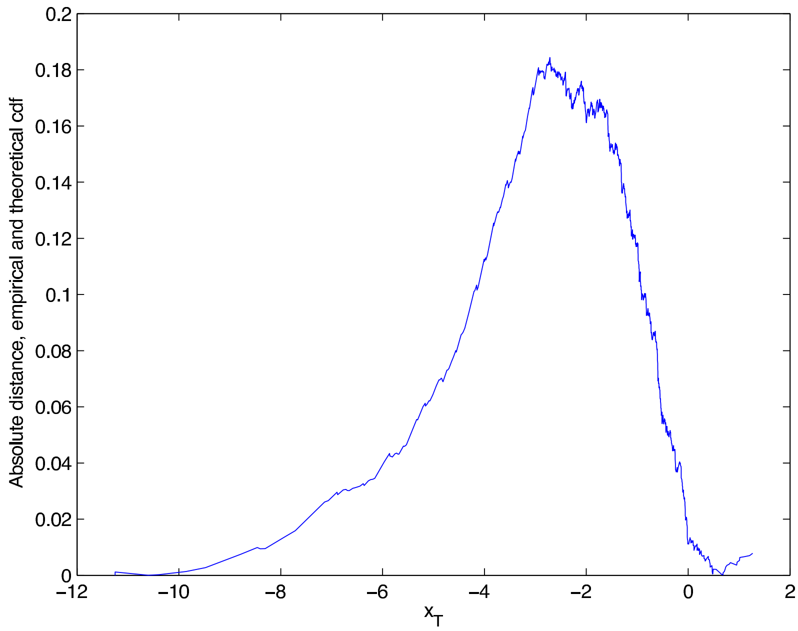

Figure 3.3: Comparison between theoretical and empirical CDF for the Hyperbolic model

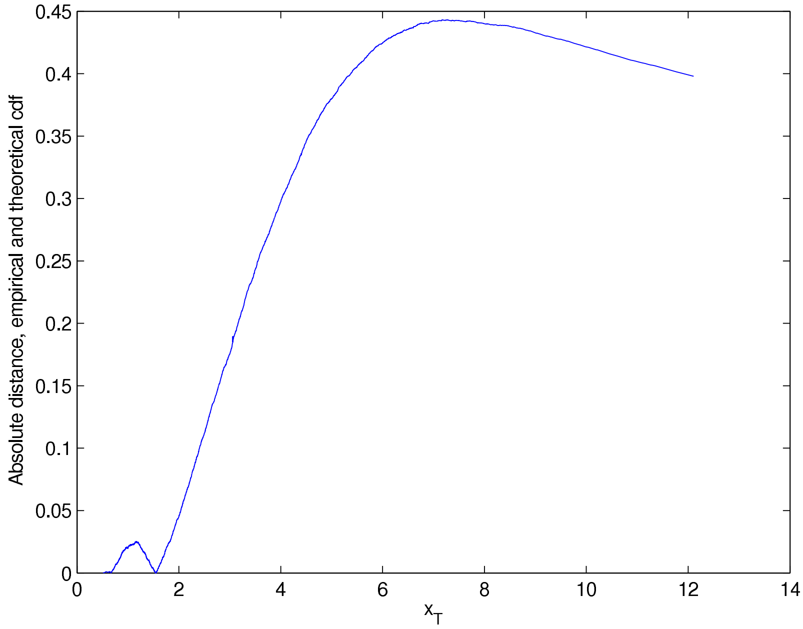

Figure 3.4: Comparison between theoretical and empirical CDF for the Heavy-tailed model

The results in this section is for the most part disappointing. The scheme seems to work very well for the CKLS model, but not for the other two models. In the case of the hyperbolic model, this might stem from the fact that the scheme is unable to resolve the instabilities. Around 15 percent of the trajectories are unstable, which is an improvement from the Euler-Maruyama scheme, but still not acceptable. For the heavy-tailed model, the cause of the behaviour is unknown. No instabilities were noted during the simulations for the analysis.

3.2 Brownian refinement scheme

The other adaptive scheme evaluated is slightly more advanced. It is based on Levy's construction of Brownian motion and utilizes a Brownian interpolation step to refine the time interval partition.

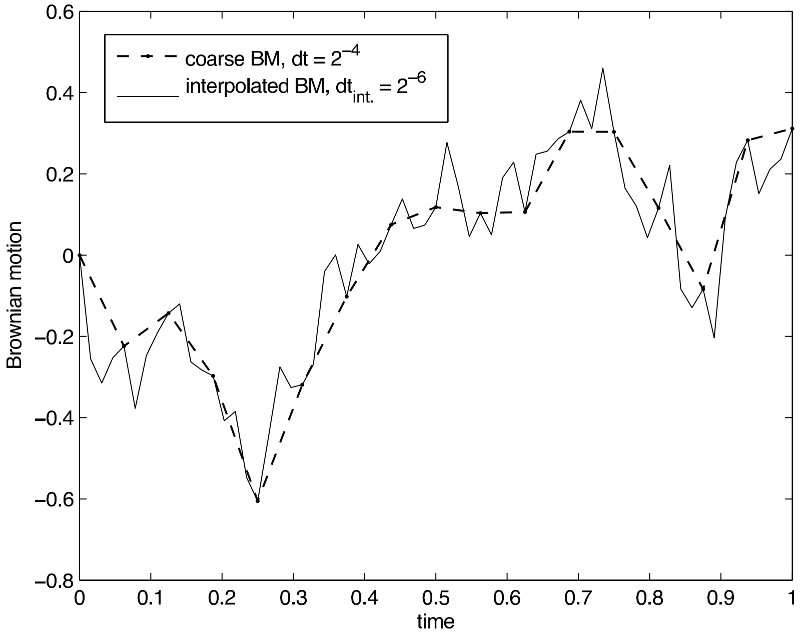

Figure 3.5: Interpolated Brownian motion

3.2.1 Interpolation of Brownian motion

Given a Brownian motion on the line with values on a discrete set of points, K_0, it is easy to interpolate Brownian values for timepoints on a finer grid K_1, $K_0 \subset K_1$. Consider the case where we have a two time points, $-\infty < s < t < \infty$, and a Brownian motion with values B_s and B_t on those time points. Suppose now that we want to interpolate a Brownian value on the time point $\theta = \frac{t-s}{2}$. Then, conditioning B_θ on B_s and B_t, B_θ is normally distributed with mean and variance given by

$$\mu = \frac{B_s + B_t}{2} \tag{3.11}$$

and

$$\sigma^2 = \frac{t-s}{4}. \tag{3.12}$$

An example of such a refined trajectory is shown in Figure 3.5. Notice that the refined Brownian motion coincides with the coarser trajectory on the coarser grid. For a proof of the above, see [7]. A short program making such global refinements is found in the appendix for added illustration.

3.2.2 Implementation

The idea of our proposed scheme is based upon the work of Gaines and Lyons in [5].

1. Simulate a Brownian motion of coarse resolution on the interval and store it a linked list. Place the current position of the scheme at the beginning of the list, t_0.
2. Starting from the current position, t_n, calculate the approximated process value for the next time point, t_{n+1}, using the Euler-Maruyama scheme and the precomputed Brownian value.
   - If the dispersion is smaller than some predetermined threshold, store the result in the linked list and traverse the current position to that time point. Repeat from step (2) in the algorithm to move forward down the list.
   - If the dispersion is larger than the threshold, insert a new time point halfway in between the current position, t_n and the next. Then use Brownian interpolation to interpolate a new Brownian value for this intermediate time point, conditioned on the Brownian values for the two surrounding time points. Then start over from step (2).

This way the scheme will behave like the ordinary Euler-Maruyama scheme when the process remains in a neighborhood of the stationary level. During bursts of volatility though, the scheme will start to cut the step lengths, trying to avoid instability. Understanding of the scheme is greatly enhanced by inspecting the code in the appendix.

One drawback with the proposed algorithm is that the steplength for time point t_{n+1} is not included in the filtration, $\mathfrak{F}_n$, at time t_n. Therefor the scheme falls outside the definition of adaptive schemes found in [8]. This is also unfortunate from the view of financial applications, since it implies some degree of anticipative ability of making correct, short term predictions of market data. This is probably not consistent with real world trading situations.

3.2.3 Performance

The algorithm was implemented in C++. It seems like the scheme is unable to resolve the difficulties connected with volatility induced stationarity. The scheme is useless for all but a small fraction of the simulation runs as the scheme goes into endless loops, interpolating ad infinitum. This behaviour was noted for all of the models. One possible remedy would be to restrict the number of consecutive interpolation steps. This, however, led to instabilies for all models. Balancing the threshold for the dispersion term was also impossible, giving instabilities for large values and endless loops for smaller.

The conclusion is that this type of scheme does not work in the current context.

Chapter 4

Concluding discussion

For the CKLS model we may conclude that the simple adaptive scheme works fairly well. Even taking into account speed considerations, the scheme is great improvement over the equidistant, nonadaptive variants since rahter long step length may be used. The scheme performs well for a wide range of parameter values, including very extreme values for the parameter γ. This property, we believe, makes the scheme a possible candidate when modeling, for example, electricity spot rates by the CKLS model.

However, the proposed scheme is nowhere as suitable for the other two types of VIS diffusion models, the hyperbolic diffusion model and the heavy-tailed diffusion model. The algorithm is unable to resolve the instability issue without reducing the step length too much. The analysis of the empirical stationary distributions of the models also revealed that the discretizations had vastly different statistical properties than that of the theoretical models.

The second type of algorithm, while beautiful in idea, did not seem to work at all for us. This is in contrast to the results presented in [5], where a similar scheme does work for diffusions with drift and dispersion coefficients that satisfy various Lipschitz and Hölder conditions.

Possible extensions of the work made in this thesis is among other things a solid theoretical analysis of the stability of the simple scheme applied to the CKLS model. An answer to what is the minimum growth rate of the step length-reducing function in order to ensue stability would make it possible to further optimize the execution speed.

Bibliography

[1] Bibby, B., Sorensen, M.,
'A hyperbolic diffusion model for stock prices',
Finance and Stochastics, **1**, 25–41, 1997

[2] Chan, K., Koralyi, G., Longstaff, F. & Sanders, A.,
'An empirical comparison of alternative models of the short-term interest rate',
Journal of Finance, **47**(3), 1209-1227

[3] Cox, J., Ingersoll, J. & Ross, S.,
'A theory if the term strutcure of interest rates',
Econometrica, **53**(3), 385-408

[4] Conley, T. G., Hansen, L. P., Luttmer, E. G. and Scheinkman, J. A.,
'Short-term interest rates as subordinated diffusions',
The review of Financial Studies, **10**, 525-577, 1997

[5] Gaines, J. G. and Lyons, T. J.,
'Variable step size control in the numerical solution of stochastic differential equations',
SIAM Journal of Applied Mathematics, **57**(5), 1455-1484, 1997

[6] Geman, H.,
'Towards a European Market of Electricity: Spot and Derivatives Trading',
`http://www.iea.org/` (retrieved Jan 18, 2005)

[7] Karatzas, I., Shreve, S.,
'Brownian motion and Stochastic Calculus',
Springer, Berlin, 1991

[8] Kloeden, P., Platen, E.,
'Numerical Solution of Stochastic Differential Equations',
Springer, Berlin, 2000

[9] Maruyama, G.,
'Continuous Markov processes and stochastic equations',
Rend. Circolo Math. Palermo, **4**, 48 – 90, 1955

[10] MUSZTA, A., RICHTER, M., ALBIN, J. M. P. AND ASTRUP JENSEN, B, *'On volatility induced stationarity for stochastic differential equations'*, Applied Probability Trust, 2005

[11] RÅDE, L., WESTERGREN, B., *'Mathematics Handbook for Science and Engineering BETA'*, Studentlitteratur, Lund, 1995

Appendix A

Code

A.1 Matlab routines

Below are some Matlab routines used in the thesis.

A.1.1 Explicit 1.5 order scheme

Here is Matlab routine showing instability for the Hyperbolic diffusion model using an explicit 1.5 order scheme. Changing the seed for the random number generator may give stable results.

explicit15.m

```
% Explicit strong 1.5 order scheme for hyperbolic SDE
% by Rickard Kjellin 2005
% uses the definition 10.4.1 of strong 1.5 order scheme from
% Kloeden&Platen, Springer Verlag

% example seed which exhibits instability with sigma=beta=delta=mu=1,
% alpha=2: 100234433
randn('state',10056463)
sigma = 1; alpha = 2; delta = 1; mu = 1; beta = 1;
Xzero = 1; % problem parameters
T = 1; N = 2^8; dt = 1/N;
U1 = randn(1,N);
U2 = randn(1,N);
dW = sqrt(dt)*U1; % Brownian increments
dZ = 0.5*(dt^(3/2))*(U1 + (1/sqrt(dt))*U2); % multiple Ito integral
X = zeros(1,N); % preallocate for efficiency
Xtemp = Xzero;
```

```
for j = 1:N
% calculate various derivatives of the dispersion function
b = sigma*exp(0.5*(alpha*sqrt(delta^2 + (Xtemp-mu)^2) - beta*(Xtemp-mu)));
bprim = b*(alpha*(2*Xtemp-2*mu)/(4*sqrt(delta^2+(Xtemp-mu)^2))-0.5*beta);
bbiss = b*(bprim^2 + alpha/(2*(alpha*sqrt(delta^2 + (Xtemp-mu)^2)) - ...
        alpha*(2*Xtemp-2*mu)^2/(8*(sqrt(delta^2 + (Xtemp-mu)^2))^3)));
% compute the 1.5 order difference step
Xtemp = Xtemp + b*dW(j) + 0.5*b*bprim*(dW(j)^2-dt) + ... Euler terms
        0.5*b^2*bbiss*(dW(j)*dt-dZ(j)) + ...             High order terms
        0.5*b*(b*bbiss + bprim^2)*(1/3*dW(j)^2-dt)*dW(j);
X(j) = Xtemp;
end
plot(0:dt:(1-dt),X);
```

A.1.2 Global interpolation of Brownian motion

This short Matlab function takes an $n * 2$ array, $\mathbf{W}_n$ as input argument, where the first column is an increasing, equidistant sequence of time points and the second column is a Brownian motion on those time points. The function returns an $(2n-1) * 2$ array, $\mathbf{W}_{n+1}$ consisting of the interpolated Brownian motion and a corresponding refined grid of time points.

interpolate.m

```
% brownian interpolation
% by Rickard Kjellin, 2005

function Wint = interpolate(W)
 dt = W(2,1) - W(1,1);
 W = W(:,2);

 % compute the conditional
 % variance of the interpolating
 % Brownian points
 Varn = 0.25*dt;

 % compute the conditional mean
 % of the interpolating
 % Brownian points
 mun = 0.5*(W + circshift(W,1));
 mun = mun(2:end);

 % create the interpolating Brownian
 % points
 Wn = mun + sqrt(Varn)*randn(length(W)-1,1);
```

```

  % stretch out the original BM and
  % insert the interpolating points
  Wtemp = zeros(2*length(W)-1,1);
  Wtemp(1:2:end) = W;
  Wtemp(2:2:end-1) = Wn;

  % create a new time grid
  dtn = 1/(length(Wtemp)-1);
  tn = (0:dtn:1)';

  % return the interpolated
  % Brownian motion
  Wint = [tn Wtemp];
end
```

A.2 C++ routines

All C++ programs were compiled using GCC/G++ 3.3 under both Linux and Apple OS X. The random number generator used is a high quality open source generator found at `http://www.agner.org/random/`.

The programs all read parameters from a textfile named `config`. The structure of the config-file following form

config

```
% configuration file for CKLS VIS-simulation
alpha = 1
beta = 1
mu = 1
gamma = 3
initialval = 1
power = 6
MinStep = 2
T = 100
N = 10
seed = 0
```

To fit the code on the page some line breaks have been inserted. This is mostly in the function headers. It is apparent from the syntax where the line breaks are.

A.2.1 Simple scheme

Below is the code for the simple scheme applied to the different diffusions. The overhead code is approximately the same for the differenet schemes, so it will only be included for the CKLS model.

Simple scheme for CKLS

Here is the code for the CKLS model discretization

simpleckls.cpp

```
// file and string streams, eg i/o
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>

// linked lists
#include <list>
#include <stdlib.h>

// standard math functions
#include <math.h>

// needed to extract the machine precision for double
#include <limits.h>
#include <float.h>

// include for measuring execution speeds
#include <sys/time.h>

// uniform random
#include "randomc.h"
//#include "mersenne.cpp"

// nonuniform random
#include "stocc.h"
//#include "stoc1.cpp"

using namespace std;

//Declare Classes
class ProcessData
{
```

```
   friend ostream &operator<<(ostream &, const ProcessData &);

   public:
      double x;
      double y;

      ProcessData();
      ProcessData(const ProcessData &);
      ~ProcessData(){};
      ProcessData &operator=(const ProcessData &rhs);
      int operator==(const ProcessData &rhs) const;
      int operator<(const ProcessData &rhs) const;
};

//Declare data structures
struct parameters
{
 double alpha;
 double beta;
 double mu;
 double gamma;
 double initialval;
 double dt;

 int power;
 int MinStep;
 int T; //length of simulation interval
 int N; //defines maximum steplength by dt = T/(2^N)

 int seed; //Seed for random number generators
};

//////////////////////////////////////////////////////
//Declare function
bool init(parameters &param);

bool simulate(parameters &param, StochasticLib1 &stochgen,
 list<ProcessData> &Process);

bool difference(parameters &param, StochasticLib1 &stochgen,
 ProcessData &NewPoint, ProcessData &OldPoint);

bool cleanup(parameters &param, list<ProcessData> &Process);

//////////////////////////////////////////////////////
```

```
bool init(parameters &param){
 char peek;  //parameter to read

 ifstream file("config"); //Open a filestream to config-file

 string configline; //Declare a string for linereading from config-file
 istringstream instream; //Create a string stream for reading from string

 while(getline(file, configline))
 {
  instream.clear(); //clears the string stream
  instream.str(configline); //use configline string as input

  peek = instream.peek(); //Check the first character of the line
  instream.ignore(15, ’=’); //skip to after equality sign

  switch(peek) //set the parameter variables
  {
   case ’%’: //skip commenting lines
    break;

   case ’a’:
    instream >> param.alpha;
    break;

   case ’b’:
    instream >> param.beta;
    break;

   case ’m’:
    instream >> param.mu;
    break;

   case ’g’:
    instream >> param.gamma;
    break;

   case ’i’:
    instream >> param.initialval;
    break;

   case ’T’:
    instream >> param.T;
    break;
```

```
   case ’p’:
    instream >> param.power;
    break;

   case ’M’:
    instream >> param.MinStep;
    break;

   case ’N’:
    instream >> param.N;
    break;

   case ’s’:
    instream >> param.seed;
    break;

   default:
    cout << "Error parsing config file. Peek found: " << peek << endl;
    return(false);
  }
  configline.clear();
 }

 file.close();

 if(param.seed == 0){
  param.seed = time(0);
  cout << "using random seed" << endl;
 }

 //calculate base steplength
 param.dt = pow(static_cast<double> (2),static_cast<double> (-param.N));
 cout << "base steplength: " << param.dt << endl;
 return(true);
}

bool simulate(parameters &param, StochasticLib1 &stochgen, list<ProcessData> &Process){
 double minStep = pow(2.0,-param.MinStep);
 ProcessData NewCurrentPoint; //create object for storing temporary points of process
 ProcessData OldCurrentPoint;
  NewCurrentPoint.x = 0;    //set X(0) = initialvalue
  NewCurrentPoint.y = param.initialval;

 Process.push_back(NewCurrentPoint); //add first coordinates to process
```

```
 OldCurrentPoint = NewCurrentPoint;
 double baseX = OldCurrentPoint.x;

 double raknaUpp = double(param.T)/50.0;
 double n = raknaUpp;
 for (int i = 1; i <= 50; i++){
  cout << "*";
 }
 cout << endl;

 while (OldCurrentPoint.x < param.T){
  while (OldCurrentPoint.x < n){
   if(!difference(param, stochgen, NewCurrentPoint, OldCurrentPoint)){
    cout << "Strul med differensmotorn!" << endl;
    return(false);
   }
   OldCurrentPoint = NewCurrentPoint;
   if(NewCurrentPoint.x - baseX >= minStep){
     Process.push_back(NewCurrentPoint);
     baseX = OldCurrentPoint.x;
   }
   if(NewCurrentPoint.y != NewCurrentPoint.y){
     cout << endl;
     cout << "sorry, instability occured at " << OldCurrentPoint.x << endl;
     exit(1);
   }
  }
  cout << "*" << flush;
  n = n + raknaUpp;
 }
 cout << endl;
 return(true);
}

bool difference(parameters &param, StochasticLib1 &stochgen,
 ProcessData &NewPoint, ProcessData &OldPoint){
 double stepsize = param.dt/(1+pow(OldPoint.y,param.power))
 + DBL_MIN; //determine local stepsize
 double dB = sqrt(stepsize)*stochgen.Normal(0,1); //create the brownian increment

 // Perform the finite difference calculation
 NewPoint.y = OldPoint.y + stepsize*(param.alpha+param.beta*OldPoint.y)
 + param.mu*pow(OldPoint.y, param.gamma)*dB;
 NewPoint.x = OldPoint.x + stepsize;
 return(true);
```

```
}

bool cleanup(parameters &param, list<ProcessData> &Process){
 list<ProcessData>::iterator i; //create iterator for traversing list
 ofstream fileout("process.dat"); // open textfile for writing

  fileout << "# Discretization of CKLS model" << endl;
  fileout << "# Parameters used are "   << endl;
  fileout << "# alpha\t=\t"   << param.alpha << endl;
  fileout << "# beta\t=\t"   << param.beta << endl;
  fileout << "# mu\t=\t"    << param.mu  << endl;
  fileout << "# gamma\t=\t"   << param.gamma << endl;
  fileout << "# initialvalue\t=\t" << param.initialval << endl;
  fileout << "# power\t=\t"   << param.power << endl;
  fileout << "# mesh, N\t=\t"   << param.N  << endl;
  fileout << "# seed\t=\t"   << param.seed << endl;

 i = Process.begin();
 double x, y;
 fileout.precision(30);
 for(i=Process.begin(); i != Process.end(); ++i){
  x = (*i).x;
  y = (*i).y;
  fileout << x << "\t" << y << endl; // print data to file step by step
 }

 fileout.close();
 Process.clear();
 return(true);
}

/////////////////////////////////////////////////////////////////
// Define Class Members
ProcessData::ProcessData()   // Constructor
{
   x = 0;
   y = 0;
}

ProcessData::ProcessData(const ProcessData &copyin){
   x = copyin.x;
   y = copyin.y;
}

ostream &operator<<(ostream &output, const ProcessData &processdata)
```

```
{
   output << processdata.x << ' ' << processdata.y << endl;
   return output;
}

ProcessData& ProcessData::operator=(const ProcessData &rhs)
{
   this->x = rhs.x;
   this->y = rhs.y;
   return *this;
}

int ProcessData::operator==(const ProcessData &rhs) const
{
   if( this->x != rhs.x) return 0;
   if( this->y != rhs.y) return 0;
   return 1;
}

int ProcessData::operator<(const ProcessData &rhs) const
{
   if( this->x == rhs.x && this->y < rhs.y) return 1;
   if( this->x < rhs.x ) return 1;
   return 0;
}

list<ProcessData> sortIt( list<ProcessData>& L)
{
   L.sort();
   return L;
}


///////////////////////////////////////////////////

int main (int argc, char *argv[]){
 parameters param; //make instance of parameters structure
 if(!init(param))
 {
  cout << "init failed..." << endl;
  return(0);
 }

 cout << "alpha\t\t\t=\t" << param.alpha << endl;
 cout << "beta\t\t\t=\t" << param.beta << endl;
```

```
 cout << "mu\t\t\t=\t" << param.mu << endl;
 cout << "gamma\t\t\t=\t" << param.gamma << endl;
 cout << "initialvalue\t\t=\t" << param.initialval << endl;
 cout << "N\t\t\t=\t" << param.N << endl;
 cout << "seed\t\t\t=\t" << param.seed << endl;

 StochasticLib1 stochgen(param.seed); //start random number generator
 list<ProcessData> Process; //create doubly linked list

 if(!simulate(param, stochgen, Process)){
  cout << "Fel vid simulering" << endl;
 }

 if(!cleanup(param, Process)){
  cout << "fel vid cleanup!" << endl;
 }
    return 0;
}

```

Simple scheme for hyperbolic diffusion

Only the difference engine is shown for this program since the overall structure is similar to the CKLS implementation.

simplehyper.cpp

```
// Hyperbolic diffusion

bool difference(parameters &param, StochasticLib1 &stochgen,
 ProcessData &NewPoint, ProcessData &OldPoint){
 //determine local stepsize
 double stepsize = param.dt/(exp(pow(fabs(OldPoint.y),param.power))) + DBL_MIN;
 double dB = sqrt(stepsize)*stochgen.Normal(0,1); //create the brownian increment

 // Perform the finite difference calculation
 double kvadratRot = sqrt(pow(param.delta,2) + pow(OldPoint.y - param.mu,2));
 double exponent = param.alpha*kvadratRot - param.beta*(OldPoint.y - param.mu);
 NewPoint.y = OldPoint.y + param.sigma*exp(0.5*exponent)*dB;
 NewPoint.x = OldPoint.x + stepsize;
 return(true);
}
```

Simple scheme for heavy-tailed diffusion

Only the difference engine is shown for this program since the overall structure is similar to the CKLS implementation.

simpleheavy.cpp

```
// Heavytailed diffusion

bool difference(parameters &param, StochasticLib1 &stochgen,
 ProcessData &NewPoint, ProcessData &OldPoint){
 double stepsize;
 double gamma = 2/3;
 if(OldPoint.y > 1){
  stepsize = param.dt; //determine local stepsize
 }
 else{
  stepsize = param.dt*pow(OldPoint.y,param.power) + DBL_MIN; //determine local stepsize
 }

 double dB = sqrt(stepsize)*stochgen.Normal(0,1); //create the brownian increment

 // Perform the finite difference calculation
 NewPoint.y = OldPoint.y + 3*stepsize*pow(OldPoint.y,param.alpha)
 + 3*pow(OldPoint.y,gamma)*dB;
 NewPoint.x = OldPoint.x + stepsize;
 return(true);
}
```

A.2.2 Brownian refinement scheme

Below is the code for the interpolating scheme applied to the CKLS model.

refinementscheme.cpp

```
// file and string streams, eg i/o
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>

// linked lists
#include <list>
#include <stdlib.h>

// standard math functions
```

```
#include <math.h>

// needed to extract the machine precision for double
#include <limits.h>
#include <float.h>

// include for measuring execution speeds
#include <sys/time.h>

// uniform random
#include "randomc.h"
#include "mersenne.cpp"

// nonuniform random
#include "stocc.h"
#include "stoc1.cpp"

using namespace std;

//Declare Classes
class ProcessData
{
   friend ostream &operator<<(ostream &, const ProcessData &);

   public:
      double x;
      double y;
   double dB;

      ProcessData();
      ProcessData(const ProcessData &);
      ~ProcessData(){};
      ProcessData &operator=(const ProcessData &rhs);
      int operator==(const ProcessData &rhs) const;
      int operator<(const ProcessData &rhs) const;
};

//Declare data structures
struct parameters
{
 double alpha;
 double beta;
 double mu;
 double gamma;
 double initialval;
```

```
 double dt;

 int power;
 int limit; //limit for abs(volatility) before stepsize reduction
 int minStep;
 int T;  //length of simulation interval
 int N;  //defines maximum steplength by dt = T/(2^N)

 int seed;  //Seed for random number generators
};

//////////////////////////////////////////////////////
//Declare function
bool init(parameters &param);

bool simulate(parameters &param, StochasticLib1 &stochgen,
 list<ProcessData> &Process);

bool difference(parameters &param, StochasticLib1 &stochgen,
 ProcessData &basePoint, ProcessData &nextPoint, ProcessData &newPoint);

bool interpolate(StochasticLib1 &stochgen, ProcessData &basePoint,
 ProcessData &nextPoint, ProcessData &newPoint);

bool cleanup(parameters &param, list<ProcessData> &Process);

//////////////////////////////////////////////////////

bool init(parameters &param){
 char peek; //parameter to read

 ifstream file("config"); //Open a filestream to config-file

 string configline; //Declare a string for linereading from config-file
 istringstream instream; //Create a string stream for reading from string

 while(getline(file, configline))
 {
  instream.clear(); //clears the string stream
  instream.str(configline); //use configline string as input

  peek = instream.peek(); //Check the first character of the line
  instream.ignore(15, '='); //skip to after equality sign

  switch(peek) //set the parameter variables
```

```
  {
   case '%': //skip commenting lines
    break;

   case 'a':
    instream >> param.alpha;
    break;

   case 'b':
    instream >> param.beta;
    break;

   case 'm':
    instream >> param.mu;
    break;

   case 'g':
    instream >> param.gamma;
    break;

   case 'i':
    instream >> param.initialval;
    break;

   case 'p':
    instream >> param.power;
    break;

   case 'l':
    instream >> param.limit;
    break;

   case 'M':
    instream >> param.minStep;
    break;

   case 'T':
    instream >> param.T;
    break;

   case 'N':
    instream >> param.N;
    break;

   case 's':
```

```
    instream >> param.seed;
    break;

   default:
    cout << "Error parsing config file. Peek found: " << peek << endl;
    return(false);
  }
  configline.clear();
 }

 file.close();

 if(param.seed == 0){
  param.seed = time(0);
  cout << "using random seed" << endl;
 }

 //calculate base steplength
 param.dt = pow(static_cast<double> (2),static_cast<double> (-param.N));
 cout << "base steplength: " << param.dt << endl;
 return(true);
}

bool simulate(parameters &param, StochasticLib1 &stochgen, list<ProcessData> &Process){
  ProcessData currentPoint;
   currentPoint.x = 0;
   currentPoint.y = param.initialval;
   currentPoint.dB = 0;
   Process.push_back(currentPoint);

  // calculate Brownian path of coarsest resolution
  for (int refineIteration = 1; refineIteration < pow(2.0,param.N); refineIteration++) {
   currentPoint.y = 0;
   currentPoint.x = currentPoint.x + param.dt;
   currentPoint.dB = sqrt(param.dt)*stochgen.Normal(0,1) + currentPoint.dB;
   Process.push_back(currentPoint);
   //cout << currentPoint.dB << " " << flush;
  }
  cout << "last x = " << currentPoint.x << endl << flush;

 // create list iterator
 list<ProcessData>::iterator location;
 location = Process.begin();

 ProcessData nextPoint;
```

```
 ProcessData newPoint;
 ProcessData basePoint = *location;
 //cout << basePoint << endl;

 // take a step forward
 location++;
 nextPoint = *location;
 // cout << nextPoint << endl;

 // main computing loop
 int k = 0;
 int n = 0;
 while (nextPoint.x < param.T){
  if(!difference(param, stochgen, basePoint, nextPoint, newPoint)){
   // create interpolated point
   interpolate(stochgen, basePoint, nextPoint, newPoint);
   // add the point to list
   location = Process.insert(location, newPoint);
   // take a step back
   nextPoint = *location;
   k++;
   n++;
  }
  else {
   location = Process.erase(location);  // erase old value and
   Process.insert(location, newPoint);  // replace with new calculation
   //cout << basePoint.dB << " " << flush;

   nextPoint = *location;   // move forward
   basePoint = newPoint;    // move base forward
   if(basePoint.y != basePoint.y){
    cout << "max consecutive interpol: " << k << endl;
    cout << "nr steps taken: " << n << endl;
    cout << "we're at x = " << basePoint.x << endl;
    cout << "and y is: " << basePoint.y << endl;
    cout << "instability!!" << endl;
    exit(1);
   }
   n++;
  }
 }
 cout << k << endl;
 return(true);
}
```

```
bool difference(parameters &param, StochasticLib1 &stochgen,
 ProcessData &basePoint, ProcessData &nextPoint, ProcessData &newPoint){
 // estimate size of diffusion term for next step
 double diffusion = param.mu*pow(basePoint.y, param.gamma)*(nextPoint.dB-basePoint.dB);
 double step = nextPoint.x - basePoint.x;
 // cout << " " << nextPoint.x << flush;
 double minStep = pow(2.0,-param.minStep);
 if (abs(diffusion) > param.limit && (step > minStep)) //bail out if too large
 {
  return(false);
 }

 // Perform the finite difference calculation
 newPoint.y = basePoint.y + step*(param.alpha+param.beta*basePoint.y) + diffusion;
 newPoint.x = nextPoint.x;
 newPoint.dB = nextPoint.dB;
 return(true);
}

bool interpolate(StochasticLib1 &stochgen, ProcessData &basePoint,
 ProcessData &nextPoint, ProcessData &newPoint){
 double mean = 0.5*(basePoint.dB + nextPoint.dB);
 double variance = 0.25*(nextPoint.x - basePoint.x);
 newPoint.x = basePoint.x + 0.5*(nextPoint.x - basePoint.x);
 newPoint.dB = mean + sqrt(variance)*stochgen.Normal(0,1);
 return(true);
}

bool cleanup(parameters &param, list<ProcessData> &Process){
 list<ProcessData>::iterator i; //create iterator for traversing list
 ofstream fileout("process.dat"); // open textfile for writing

  fileout << "# Discretization of CKLS model" << endl;
  fileout << "# Parameters used are "   << endl;
  fileout << "# alpha\t=\t"   << param.alpha << endl;
  fileout << "# beta\t=\t"   << param.beta << endl;
  fileout << "# mu\t=\t"    << param.mu  << endl;
  fileout << "# gamma\t=\t"   << param.gamma << endl;
  fileout << "# initialvalue\t=\t" << param.initialval << endl;
  fileout << "# power\t=\t"   << param.power << endl;
  fileout << "# limit\t=\t"   << param.limit << endl;
  fileout << "# mesh, N\t=\t"   << param.N  << endl;
  fileout << "# seed\t=\t"   << param.seed << endl;

 i = Process.begin();
```

```
 double x, y;
 fileout.precision(30);
 for(i=Process.begin(); i != Process.end(); ++i){
  x = (*i).x;
  y = (*i).y;
  fileout << x << "\t" << y << endl; // print data to file step by step
 }

 fileout.close();
 Process.clear();
}

/////////////////////////////////////////////////////////////////
// Define Class Members
ProcessData::ProcessData()   // Constructor
{
   x = 0;
   y = 0;
   dB = 0;
}

ProcessData::ProcessData(const ProcessData &copyin)
{
   x = copyin.x;
   y = copyin.y;
   dB = copyin.dB;
}

ostream &operator<<(ostream &output, const ProcessData &processdata)
{
   output << processdata.x << ' ' << processdata.y << ' ' << processdata.dB << endl;
   return output;
}

ProcessData& ProcessData::operator=(const ProcessData &rhs)
{
   this->x = rhs.x;
   this->y = rhs.y;
   this->dB = rhs.dB;
   return *this;
}

int ProcessData::operator==(const ProcessData &rhs) const
{
   if( this->x != rhs.x) return 0;
```

```
   if( this->y != rhs.y) return 0;
   if( this->dB != rhs.dB) return 0;
   return 1;
}

int ProcessData::operator<(const ProcessData &rhs) const
{
   if( this->x == rhs.x && this->y < rhs.y) return 1;
   if( this->x < rhs.x ) return 1;
   return 0;
}

list<ProcessData> sortIt( list<ProcessData>& L)
{
   L.sort();                                               // Sort list
   return L;
}

///////////////////////////////////////////////////

int main (int argc, char *argv[]){
 parameters param; //make instance of parameters structure
 if(!init(param))
 {
  cout << "init failed<< endl;
  return(0);
 }

 cout << "alpha\t\t\t=\t" << param.alpha << endl;
 cout << "beta\t\t\t=\t" << param.beta << endl;
 cout << "mu\t\t\t=\t" << param.mu << endl;
 cout << "gamma\t\t\t=\t" << param.gamma << endl;
 cout << "initialvalue\t\t=\t" << param.initialval << endl;
 cout << "N\t\t\t=\t" << param.N << endl;
 cout << "seed\t\t\t=\t" << param.seed << endl;

 StochasticLib1 stochgen(param.seed); //start random number generator
 list<ProcessData> Process; //create doubly linked list

 if(!simulate(param, stochgen, Process)){
  cout << "Fel vid simulering" << endl;
 }

 if(!cleanup(param, Process)){
  cout << "fel vid cleanup!" << endl;
```

```
 }
    return 0;
}
```