2

Numerical Methods for SDE

This chapter covers basic and new material on the subject of simulation of solutions of stochastic differential equations. The chapter reviews results from the well-known reference [142] but also covers new results such as, but not only, exact sampling (see [29]). Other related references mentioned in this chapter are, for example, [156] and [125].

There are two main objectives in the simulation of the trajectory of a process solution of a stochastic differential equation: either interest is in the whole trajectory or in the expected value of some functional of the process (moments, distributions, etc) which usually are not available in explicit analytical form. Variance reduction techniques for stochastic differential equations can be borrowed from standard variance reduction techniques of the Monte Carlo method (see Section 1.4) and clearly only apply when interest is in the functionals of the process.

Simulation methods are usually based on discrete approximations of the continuous solution to a stochastic differential equation. The methods of approximation are classified according to their different properties. Mainly two criteria of optimality are used in the literature: the *strong* and the *weak* (orders of) convergence.

Strong order of convergence

A time-discretized approximation Y_δ of a continuous-time process Y, with δ the maximum time increment of the discretization, is said to be of general *strong* order of convergence γ to Y if for any fixed time horizon T it holds true that

$$\mathbb{E}|Y_\delta(T) - Y(T)| \leq C\delta^\gamma, \quad \forall\, \delta < \delta_0\,,$$

with $\delta_0 > 0$ and C a constant not depending on δ. This kind of criterion is similar to the one used in the approximation of the trajectories of nonstochastic dynamical systems.

S.M. Iacus, *Simulation and Inference for Stochastic Differential Equations*,
doi: 10.1007/978-0-387-75839-8_2,

Weak order of convergence

Along with the strong convergence, the *weak* convergence can be defined. Y_δ is said to converge *weakly* of order β to Y if for any fixed horizon T and any $2(\beta+1)$ continuous differentiable function g of polynomial growth, it holds true that

$$|\mathbb{E}g(Y(T)) - \mathbb{E}g(Y_\delta(T))| \leq C\delta^\beta, \quad \forall\, \delta < \delta_0\,,$$

with $\delta_0 > 0$ and C a constant not depending on δ.

Schemes of approximation of some order that strongly converge usually have a higher order of weak convergence. This is the case with the Euler scheme, which is strongly convergent of order $\gamma = \frac{1}{2}$ and weakly convergent of order $\beta = 1$ (under some smoothness conditions on the coefficients of the stochastic differential equation). While the schemes have their own order of convergence, it is usually the case that, for some actual specifications of the stochastic differential equations, they behave better.

2.1 Euler approximation

One of the most used schemes of approximation is the *Euler* method, originally used to generate solutions to deterministic differential equations. We implicitly used this method in Chapter 1 several times. The idea is the following: given an Itô process $\{X_t, 0 \leq t \leq T\}$ solution of the stochastic differential equation

$$\mathrm{d}X_t = b(t, X_t)\mathrm{d}t + \sigma(t, X_t)\mathrm{d}W_t,$$

with initial deterministic value $X_{t_0} = X_0$ and the discretization $\Pi_N = \Pi_N([0,T])$ of the interval $[0,T]$, $0 = t_0 < t_1 < \cdots < t_N = T$. The *Euler approximation* of X is a continuous stochastic process Y satisfying the iterative scheme

$$Y_{i+1} = Y_i + b(t_i, Y_i)(t_{i+1} - t_i) + \sigma(t_i, Y_i)(W_{i+1} - W_i), \tag{2.1}$$

for $i = 0, 1, \ldots, N-1$, with $Y_0 = X_0$. We have simplified the notation setting $Y(t_i) = Y_i$ and $W(t_i) = W_i$. Usually the time increment $\Delta t = t_{i+1} - t_i$ is taken to be constant (i.e., $\Delta t = 1/N$). In between any two time points t_i and t_{i+1}, the process can be defined differently. One natural approach is to consider linear interpolation so that $Y(t)$ is defined as

$$Y(t) = Y_i + \frac{t - t_i}{t_{i+1} - t_i}(Y_{i+1} - Y_i), \quad t \in [t_i, t_{i+1})\,.$$

From (2.1), one can see that to simulate the process Y one only needs to simulate the increment of the Wiener process, and we already know how to do this from Chapter 1. As mentioned, the Euler scheme has order $\gamma = \frac{1}{2}$ of strong convergence.

2.1.1 A note on code vectorization

Consider for example the Ornstein-Uhlenbeck process (see Section 1.13.1) solution of

$$\mathrm{d}X_t = (\theta_1 - \theta_2 X_t)\mathrm{d}t + \theta_3 \mathrm{d}W_t\,.$$

For this process, $b(t,x) = (\theta_1 - \theta_2 x)$ and $\sigma(t,x) = \theta_3$. Suppose we fix an initial value $X_0 = x$ and the set of parameters $(\theta_1, \theta_2, \theta_3) = (0, 5, 3.5)$. The following algorithm can be used to simulate the trajectory of the process using the Euler algorithm instead of the simulation of the stochastic integral as in Section 1.13.1:

```
> # ex 2.01.R
> set.seed(123)
> N <- 100
> T <- 1
> x <- 10
> theta <- c(0, 5, 3.5)
> Dt <- 1/N
> Y <- numeric(N+1)
> Y[1] <- x
> Z <- rnorm(N)
> for(i in 1:N)
+  Y[i+1] <- Y[i] + (theta[1] - theta[2]*Y[i])*Dt + theta[3]*sqrt(Dt)*Z[i]
> Y <- ts(Y,start=0, deltat=1/N)
> plot(Y)
```

At first glance, this algorithm appears not to be efficient from the R point of view. We can try to optimize this code by replacing the `for` loop with some `*apply` function. Indeed, noticing that

$$Y(2) = Y(1)(1 - \theta_2 \Delta t) + \theta_3 * \sqrt{\Delta t} Z(1)$$

and

$$\begin{aligned} Y(3) &= Y(2)(1 - \theta_2 \Delta t) + \theta_3 \sqrt{\Delta t} Z(2) \\ &= \{Y(1)(1 - \theta_2 \Delta t) + \theta_3 \sqrt{\Delta t} Z(1)\}(1 - \theta_2 \Delta t) + \theta_3 \sqrt{\Delta t} Z(2) \\ &= Y(1)(1 - \theta_2 \Delta t)^2 + \theta_3 \sqrt{\Delta t} Z(1)(1 - \theta_2 \Delta t) + \theta_3 \sqrt{\Delta t} Z(2) \end{aligned}$$

by iterative substitution, we get the general formula for the kth step of the Euler scheme for the Ornstein-Uhlenbeck process,

$$Y(k) = Y(1) \cdot (1 - \theta_2 \Delta t)^{k-1} + \sum_{j=1}^{k-1} (\theta_3 * \sqrt{\Delta t} * Z(j)) * (1 - \theta_2 \Delta t)^{(k-j-1)},$$

which leads to the following algorithm that uses linear algebra instead of `for` loops.

```
> # ex 2.02.R
> set.seed(123)
> theta <- c(0, 5, 3.5)
> N <- 100
> T <- 1
> x <- 10
```

```
> Z <- rnorm(N)
> Dt <- 1/N
> A <- theta[3]*sqrt(Dt)*Z
> P <- (1-theta[2]*Dt)^(0:(N-1))
> X0 <- x
> X <- sapply(2:(N+1), function(x) X0*(1-theta[2]*Dt)^(x-1) +
+     A[1:(x-1)] %*% P[(x-1):1])
> Y <- ts(c(X0,X),start=0, deltat=1/N)
> plot(Y)
```

But, this one-line code is not at all better than the one implying the `for` loop. In fact, it is even worse, as it implies more calculations than needed. To show this, we embed the two codes into two functions `OU`, and `OU.vec`, to measure their performance in terms of CPU time.[1]

```
> # ex 2.03.R
> OU <- function(a,b, x, N=1000){
+   Y <- numeric(N+1)
+   Y[1] <- x
+   Z <- rnorm(N)
+   Dt <- 1/N
+   for(i in 1:N)
+     Y[i+1] <- Y[i] - a*Y[i]*Dt + b*sqrt(Dt)*Z[i]
+    invisible(Y)
+ }

> OU.vec <- function(a, b, x, N=1000){
+   Dt <- 1/N
+   Z <- rnorm(N)
+   A <- b*sqrt(Dt)*Z
+   P <- (1-a*Dt)^(0:(N-1))
+    X0 <- x
+    X <- c(X0, sapply(2:(N+1),
+       function(x) X0*(1-a*Dt)^(x-1) +
+       sum(A[1:(x-1)] * P[(x-1):1])))
+  invisible(X)
+ }
```

Using the `system.time` function, we test the two implementations

```
> set.seed(123)
> system.time(OU(10,5,3.5))
[1] 0.037 0.001 0.044 0.000 0.000

> set.seed(123)
> system.time(OU.vec(10,5,3.5))
[1] 0.198 0.024 0.261 0.000 0.000
```

which shows that the vectorized version is much slower than the naive one. The moral of this example is that `for` loops in R should be replaced by vectorized code *only if the number of calculations does not increase*, as the case above shows. Vectorization can really reduce simulation/estimation time, but only under the conditions above. As vectorization can become a nightmare for the average R programmer, the reader has been warned.

To convince the reader that it is worth using vectorized code when it is appropriate, we report without comments the example from Chapter 1 for generating paths of Brownian motion with two levels of optimization in the R code.

[1] Times may vary on the reader's machine.

```
# ex 2.04.R
> BM.1 <- function(N=10000){ # brutal code
+  W <- NULL
+  for(i in 2:(N+1))
+          W <- c(W, rnorm(1) / sqrt(N))
+ }

> BM.2 <- function(N=10000){ # smarter
+  W <- numeric(N+1)
+  Z <- rnorm(N)
+  for(i in 2:(N+1))
+          W[i] <- W[i-1] + Z[i-1] / sqrt(N)
+ }

> BM.vec <- function(N=10000) # awesome !
+  W <- c(0,cumsum(rnorm(N)/sqrt(N)))

> set.seed(123)
> system.time(BM.1())
[1] 1.354 1.708 3.856 0.000 0.000

> set.seed(123)
> system.time(BM.2())
[1] 0.281 0.011 0.347 0.000 0.000

> set.seed(123)
> system.time(BM.vec())
[1] 0.008 0.001 0.010 0.000 0.000
```

2.2 Milstein scheme

The Milstein scheme[2] [164] makes use of Itô's lemma to increase the accuracy of the approximation by adding the second-order term. Denoting by σ_x the partial derivative of $\sigma(t,x)$ with respect to x, the Milstein approximation looks like

$$\begin{aligned} Y_{i+1} = Y_i &+ b(t_i,Y_i)(t_{i+1}-t_i) + \sigma(t_i,Y_i)(W_{i+1}-W_i) \\ &+ \frac{1}{2}\sigma(t_i,Y_i)\sigma_x(t_i,Y_i)\left\{(W_{i+1}-W_i)^2 - (t_{i+1}-t_i)\right\} \end{aligned} \tag{2.2}$$

or, in more symbolic form,

$$Y_{i+1} = Y_i + b\Delta t + \sigma\Delta W_t + \frac{1}{2}\sigma\sigma_x\left\{(\Delta W_t)^2 - \Delta t\right\}.$$

This scheme has strong and weak orders of convergence equal to one. Let us consider once again the Ornstein-Uhlenbeck process solution of (1.39). For this process, $b(t,x) = \theta_1 - \theta_2 \cdot x$ and $\sigma(t,x) = \theta_3$, and thus $\sigma_x(t,x) = 0$ and the Euler and Milstein schemes coincide. This is one case in which the Euler scheme is of strong order of convergence $\gamma = 1$.

[2] Actually, Milstein proposed two schemes of approximation. The one presented in this section corresponds to the one most commonly known as the "Milstein scheme" and has the simplest form. Another Milstein scheme of higher-order approximation, will be presented later in this chapter.

The geometric Brownian motion

A more interesting case is that of geometric Brownian motion, presented in Section 1.7, solving the stochastic differential equation

$$\mathrm{d}X_t = \theta_1 X_t \mathrm{d}t + \theta_2 X_t \mathrm{d}W_t.$$

For this process, $b(t,x) = \theta_1 \cdot x$, $\sigma(t,x) = \theta_2 \cdot x$, and $\sigma_x(t,x) = \theta_2$. The Euler discretization for this process looks like

$$Y^E_{i+1} = Y^E_i(1 + \theta_1 \cdot \Delta t) + \theta_2 Y^E_i \Delta W_t,$$

and the Milstein scheme reads

$$\begin{aligned} Y^M_{i+1} &= Y^M_i + \theta_1 \cdot Y^M_i \Delta t + \theta_2 Y^M_i \Delta W_t + \frac{1}{2}\theta_2^2 Y^M_i \left\{(\Delta W_t)^2 - \Delta t\right\} \\ &= Y^M_i \left(1 + \left(\theta_1 - \frac{1}{2}\theta_2^2\right)\Delta t\right) + \theta_2 Y^M_i \Delta W_t + \frac{1}{2}\theta_2^2 Y^M_i (\Delta W_t)^2. \end{aligned}$$

Recall that $\Delta W_t \sim \sqrt{\Delta t} Z$ with $Z \sim N(0,1)$. Thus

$$Y^E_{i+1} = Y^E_i(1 + \theta_1 \cdot \Delta t + \theta_2 \sqrt{\Delta t} Z)$$

and

$$\begin{aligned} Y^M_{i+1} &= Y^M_i \left(1 + \left(\theta_1 - \frac{1}{2}\theta_2^2\right)\Delta t\right) + \theta_2 Y^M_i \sqrt{\Delta t} Z + \frac{1}{2}\theta_2^2 Y^M_i \Delta t Z^2 \\ &= Y^M_i \left(1 + \left(\theta_1 + \frac{1}{2}\theta_2^2 (Z^2 - 1)\right)\Delta t + \theta_2 \sqrt{\Delta t} Z\right). \end{aligned}$$

Looking at the exact solution in (1.7), the Milstein scheme makes the expansion exact up to order $O(\Delta t)$. Indeed, formal Taylor expansion leads to

$$\begin{aligned} X_{t+\Delta t} &= X_t \exp\left\{\left(\theta_1 - \frac{\theta_2^2}{2}\right)\Delta t + \theta_2 \sqrt{\Delta t} Z\right\} \\ &= X_t \left\{1 + \left(\theta_1 - \frac{\theta_2^2}{2}\right)\Delta t + \theta_2 \sqrt{\Delta t} Z + \frac{1}{2}\theta_2^2 \Delta t Z^2 + O(\Delta t)\right\} \\ &= Y^M_{i+1}. \end{aligned}$$

2.3 Relationship between Milstein and Euler schemes

Following [125], we now show a result on transformations of stochastic differential equations and the two schemes of approximation. Given the generic stochastic differential equation

$$\mathrm{d}X_t = b(t, X_t)\mathrm{d}t + \sigma(t, X_t)\mathrm{d}W_t, \tag{2.3}$$

the Milstein scheme for it is

$$\begin{aligned}\Delta X = X_{i+1} - X_i = &\left(b(t_i, X_i) - \frac{1}{2}\sigma(t_i, X_i)\sigma_x(t_i, X_i)\right)\Delta t \\ &+ \sigma(t_i, X_i)\sqrt{\Delta t}Z + \frac{1}{2}\sigma(t_i, X_i)\sigma_x(t_i, X_i)\Delta t Z^2\end{aligned} \tag{2.4}$$

with $Z \sim N(0,1)$. Consider now the transformation $y = F(x)$ and its inverse $x = G(y)$. Then (2.3) becomes by Itô's lemma

$$\mathrm{d}Y_t = \left(F'(X_t)b(t, X_t) + \frac{1}{2}F''(X_t)\sigma^2(t, X_t)\right)\mathrm{d}t + F'(X_t)\sigma(t, X_t)\mathrm{d}W_t \tag{2.5}$$

with $Y_t = F(X_t)$. Now choose F as the Lamperti transform (1.34) so that

$$F'(x) = \frac{1}{\sigma(t,x)}, \qquad F''(x) = -\frac{\sigma_x(t,x)}{\sigma^2(t,x)}\,.$$

We know from (1.35) that (2.5) becomes

$$\mathrm{d}Y_t = \left(\frac{b(t, X_t)}{\sigma(t, X_t)} - \frac{1}{2}\sigma_x^2(t, X_t)\right)\mathrm{d}t + \mathrm{d}W_t\,.$$

We remark again that the Lamperti transform is such that the multiplicative factor in front of the Wiener process no longer depends on the state of the process. Thus the Euler scheme for $Y_t = F(X_t)$ reads

$$\Delta Y = \left(\frac{b(t_i, X_i)}{\sigma(t_i, X_i)} - \frac{1}{2}\sigma_x(t_i, X_i)\right)\Delta t + \sqrt{\Delta t}Z\,.$$

The next step is to calculate the Taylor expansion of the inverse transformation in order to obtain some comparable expression.

$$G(Y_i + \Delta Y) = G(Y_i) + G'(Y_i)\Delta Y + \frac{1}{2}G''(Y_i)(\Delta Y_i)^2 + O\left(\Delta Y^3\right).$$

Notice that

$$G'(y) = \frac{\mathrm{d}}{\mathrm{d}y}F^{-1}(y) = \frac{1}{F'(G(y))} = \sigma(t, G(y))$$

and

$$G''(y) = G'(y)\sigma_x(t, G(y)) = \sigma(t, G(y))\sigma_x(t, G(y)),$$

which finally leads to

$$\begin{aligned}G(Y_i + \Delta Y) - G(Y_i) = &\left(b(t_i, X_i) - \frac{1}{2}\sigma(t_i, X_i)\sigma_x(t_i, X_i)\right)\Delta t \\ &+ \sigma(t_i, X_i)\sqrt{\Delta t}Z + \frac{1}{2}\sigma(t_i, X_i)\sigma_x(t_i, X_i)\Delta t Z^2 \\ &+ O\left(\Delta t^{\frac{3}{2}}\right).\end{aligned}$$

Thus the Milstein scheme on the original process (2.4) and the Euler scheme on the transformed process are equal up to and including the order Δt. So, in principle, whenever possible, one should use the Euler scheme on the transformed process.

In general, if F (not necessarily the Lamperti transformation) eliminates the interactions between the state of the process and the increments of the Wiener process, this transformation method is probably always welcome because it reduces instability in the simulation process. A detailed account on this matter can be found in [70]. We now show a couple of applications of the transformation method just presented.

2.3.1 Transform of the geometric Brownian motion

The first example is on the geometric Brownian motion. If we use $F(x) = \log(x)$, then from (1.32) and Itô's lemma, we obtain

$$\mathrm{d}\log X_t = \left(\theta_1 - \frac{1}{2}\theta_2^2\right)\mathrm{d}t + \theta_2 \mathrm{d}W_t.$$

Thus the Euler scheme for the transformed process is

$$\Delta \log X = \left(\theta_1 - \frac{1}{2}\theta_2^2\right)\Delta t + \theta_2\sqrt{\Delta t}Z\,.$$

Now, using the Taylor expansion on the inverse transform $G(y) = x^y$, we get the Milstein scheme.

2.3.2 Transform of the Cox-Ingersoll-Ross process

One more interesting application of the Lamperti transform concerns the Cox-Ingersoll-Ross process introduced in Section 1.13.3. The dynamics of the process is

$$\mathrm{d}X_t = (\theta_1 - \theta_2 X_t)\mathrm{d}t + \theta_3\sqrt{X_t}\mathrm{d}W_t, \tag{2.6}$$

and the Milstein scheme for it reads as

$$\Delta X = \left((\theta_1 - \theta_2 X_i) - \frac{1}{4}\theta_3^2\right)\Delta t + \theta_3\sqrt{X_i}\sqrt{\Delta t}Z + \frac{1}{4}\theta_3^2\Delta t Z^2. \tag{2.7}$$

Now, using the transformation $y = \sqrt{x}$, we obtain the transformed stochastic differential equation

$$\mathrm{d}Y_t = \frac{1}{2Y_t}\left((\theta_1 - \theta_2 Y_t^2) - \frac{1}{4}\theta_3^2\right)\mathrm{d}t + \frac{1}{2}\theta_3 \mathrm{d}W_t\,,$$

for which the Euler scheme is

$$\Delta Y = \frac{1}{2Y_i}\left((\theta_1 - \theta_2 Y_i^2) - \frac{1}{4}\theta_3^2\right)\Delta t + \frac{1}{2}\theta_3\sqrt{\Delta t}Z\,.$$

Since $G(y) = x^2$, we obtain

$$\begin{aligned} G(Y_i + \Delta Y) - G(Y_i) &= (Y_i + \Delta Y)^2 - Y_i^2 = (\Delta Y)^2 + 2Y_i\Delta Y \\ &= \frac{1}{4}\theta_3^2\Delta t Z^2 + O(\Delta t^2) \\ &\quad + \left((\theta_1 - \theta_2 Y_i^2) - \frac{1}{4}\theta_3^2\right)\Delta t + Y_i\theta_3\sqrt{\Delta t}Z\,, \end{aligned}$$

which is exactly (2.7) given that $Y_i = \sqrt{X_i}$.

2.4 Implementation of Euler and Milstein schemes: the sde.sim function

We now show generic implementations of both the Euler and Milstein schemes.

```
sde.sim <- function(t0=0, T=1, X0=1, N=100, delta,
         drift, sigma, sigma.x,
         method=c("euler","milstein")){

  if(missing(drift) )
   stop("please specify at least the drift coefficient of the SDE")

  if(missing(sigma))
   sigma <- expression(1)

  if(!is.expression(drift) | !is.expression(sigma))
   stop("coefficients must be expressions in `t' and `x'")

  method <- match.arg(method)
  needs.sx <- FALSE

  if(method=="milstein") needs.sx <- TRUE

   if(needs.sx & missing(sigma.x)){
    cat("sigma.x not provided, attempting symbolic derivation.\n")
    sigma.x <- D(sigma,"x")
  }

  d1 <- function(t,x) eval(drift)
  s1 <- function(t,x) eval(sigma)
  sx <- function(t,x) eval(sigma.x)

  if(t0<0 | T<0)
   stop("please use positive times!")

  if(missing(delta)){
    t <- seq(t0,T, length=N+1)
  } else {
   t <- c(t0,t0+cumsum(rep(delta,N)))
   T <- t[N+1]
   warning("T set to =",T,"\n")
  }

  Z <- rnorm(N)
  X <- numeric(N+1)
```

```
  Dt <- (T-t0)/N
  sDt <- sqrt(Dt)
  X[1] <- X0

  if(method=="euler"){
   for(i in 2:(N+1))
     X[i] <- X[i-1] + d1(t[i-1],X[i-1])*Dt +
             s1(t[i-1],X[i-1])*sDt*Z[i-1]
  }
  if(method=="milstein"){
   for(i in 2:(N+1)){
    X[i] <- X[i-1] + d1(t[i-1],X[i-1])*Dt +
              s1(t[i-1],X[i-1])*sDt*Z[i-1] +
              0.5*s1(t[i-1],X[i-1])* sx(t[i-1],X[i-1]) *
              (Dt*Z[i-1]^2-Dt)
   }
  }
  X <- ts(X,start=t0 ,deltat=Dt)
  invisible(X)
}
```

The `sde.sim` function above can be used to simulate paths of solutions to generic stochastic differential equations. The function can simulate trajectories with either the "`euler`" or "`milstein`" scheme. The function accepts the two coefficients `drift` and `sigma` and eventually the partial derivative of the diffusion coefficient `sigma.x` for the Milstein scheme. If `sigma.x` is not provided by the user, the function tries to provide one itself using the R function `D`. Coefficients must be objects of class `expression` with arguments named `t` and `x`, respectively, interpreted as time and space (i.e., the state of the process). If the diffusion coefficient `sigma` is not specified, it is assumed to be unitary (i.e., identically equal to one). The user can specify the initial value X_0 (defaulted to 1), the interval $[t_0, T]$ (defaulted to $[0, 1]$), the Δ step `delta`, and the number `N` of values of the process to be generated. The function always returns a `ts` (time series) object of length $N + 1$; i.e., the initial value X_0 and the N new simulated values of the trajectory. If Δ is not specified, $\Delta = (T - t_0)/N$. If Δ is specified, then N new observations are generated at time increments of Δ and the time horizon is adjusted accordingly as $T = \Delta \cdot N$. We have added the opportunity to specify the Δ step directly because this will be relevant in Chapter 3.

2.4.1 Example of use

The following examples use the `sde.sim` function for some of the processes introduced earlier. The reader doesn't need to write the code for the `sde.sim` function because it is included in the CRAN package called `sde`. Moreover, this function is going to evolve during this chapter as new methods are introduced.

The Ornstein-Uhlenbeck process

$$\mathrm{d}X_t = (\theta_1 - \theta_2 X_t)\mathrm{d}t + \theta_3 \mathrm{d}W_t, \qquad X_0 = 10\,,$$

with $(\theta_1, \theta_2, \theta_3) = (0, 5, 3.5)$.

```
> # ex 2.05.R
> # Ornstein-Uhlenbeck process
> require(sde)
[1] TRUE
> set.seed(123)
> d <- expression(-5 * x)
> s <- expression(3.5)
> sde.sim(X0=10,drift=d, sigma=s) -> X
sigma.x not provided, attempting symbolic derivation.
> plot(X,main="Ornstein-Uhlenbeck")
```

The Cox-Ingersoll-Ross process

$$\mathrm{d}X_t = (\theta_1 - \theta_2 X_t)\mathrm{d}t + \theta_3\sqrt{X_t}\mathrm{d}W_t, \qquad X_0 = 10\,,$$

with $(\theta_1, \theta_2, \theta_3) = (6, 3, 2)$.

```
> # ex 2.06.R
> # Cox-Ingersoll-Ross (CIR-1)
> set.seed(123)
> d <- expression( 6-3*x )
> s <- expression( 2*sqrt(x) )
> sde.sim(X0=10,drift=d, sigma=s) -> X
sigma.x not provided, attempting symbolic derivation.
> plot(X,main="Cox-Ingersoll-Ross")
```

The Cox-Ingersoll-Ross process with Milstein scheme

We need the partial derivative with respect to variable x of the coefficient $\sigma(\cdot, \cdot)$,

$$\sigma_x(t, x) = \frac{\partial}{\partial x} 2\sqrt{x} = \frac{1}{\sqrt{x}}\,.$$

Therefore,

```
> # ex 2.07.R
> # Cox-Ingersoll-Ross (CIR-2)
> d <- expression( 6-3*x )
> s <- expression( 2*sqrt(x) )
> s.x <- expression( 1/sqrt(x) )
> set.seed(123)
> sde.sim(X0=10, drift=d, sigma=s, sigma.x=s.x,
+     method="milstein") -> X
> plot(X,main="Cox-Ingersoll-Ross")
```

The Cox-Ingersoll-Ross process with Euler scheme on the transformed process $Y_t = \sqrt{X_t}$

$$\mathrm{d}Y_t = \frac{1}{2Y_t}\left(\theta_1 - \theta_2 Y_t^2 - \frac{1}{4}\theta_3^2\right)\mathrm{d}t + \frac{1}{2}\theta_3 \mathrm{d}W_t, \qquad Y_0 = \sqrt{10}\,,$$

with $(\theta_1, \theta_2, \theta_3) = (6, 3, 2)$.

```
> # ex 2.08.R
> # Cox-Ingersoll-Ross (CIR-3)
> set.seed(123)
> d <- expression( (6-3*x^2 - 1)/(2*x) )
> s <- expression( 1 )
> sde.sim(X0=sqrt(10),drift=d, sigma=s) -> Y
> plot(Y^2,main="Cox-Ingersoll-Ross")
```

The reader can verify that the Milstein scheme `CIR-2` returns the same path as `CIR-3` but on a different scale (i.e., $Y^2 = X$).

The geometric Brownian motion process

$$\mathrm{d}X_t = \theta_1 X_t \mathrm{d}t + \theta_2 X_t \mathrm{d}W_t, \qquad X_0 = 10\,,$$

with $(\theta_1, \theta_2) = \left(1, \frac{1}{2}\right)$.

```
> # ex 2.09.R
> # geometric Brownian Motion
> set.seed(123)
> d <- expression( x )
> s <- expression( 0.5*x )
> sde.sim(X0=10,drift=d, sigma=s) -> X
> plot(X,main="geometric Brownian Motion")
```

2.5 The constant elasticity of variance process and strange paths

The constant elasticity of variance (CEV) process introduced in finance in option pricing (see [201] and [25]) is another particular member of the CKLS family of models (see Section 1.13.4) and is the solution of the stochastic differential equation

$$\mathrm{d}X_t = \mu X_t \mathrm{d}t + \sigma X_t^{\gamma} \mathrm{d}W_t, \quad \gamma \geq 0\,. \tag{2.8}$$

This process is quite useful in modeling a skewed implied volatility. In particular, for $\gamma < 1$, the skewness is negative, and for $\gamma > 1$ the skewness is positive. For $\gamma = 1$, the CEV process is a particular version of the geometric Brownian motion. Even if this process is assumed to be positive, the discretized version of it can reach negative values. To understand why this could happen and what to do with the paths that cross zero, one should go back to the theoretical properties of the process. For example, in [38] it is shown that for $\gamma < \frac{1}{2}$ there is a positive probability for this process to be absorbed in zero. Hence, if in one simulation the path crosses the zero line, one should stop the simulation and consider this path as actually absorbed in 0.

2.6 Predictor-corrector method

Both schemes of discretization consider the coefficients b and σ as not varying during the time interval Δt, which is of course untrue for a generic stochastic differential equation, as b and σ can depend on both the time t and the state of the process X_t. One way to recover the varying nature of these coefficients is to average their values in some way. Since the coefficients depend on X_t and we are simulating X_t, the method we present here just tries to approximate the states of the process first. This method is of weak convergence order 1. The predictor-corrector algorithm is as follows. First consider the simple approximation (the *predictor*)

$$\tilde{Y}_{i+1} = Y_i + b(t_i, Y_i)\Delta t + \sigma(t_i, Y_i)\sqrt{\Delta t}Z.$$

Then choose two weighting coefficients α and η in $[0, 1]$, and calculate the *corrector* as

$$\begin{aligned} Y_{i+1} = Y_i &+ \left(\alpha\tilde{b}(t_{i+1}, \tilde{Y}_{i+1}) + (1-\alpha)\tilde{b}(t_i, Y_i)\right)\Delta t \\ &+ \left(\eta\sigma(t_{i+1}, \tilde{Y}_{i+1}) + (1-\eta)\sigma(t_i, Y_i)\right)\sqrt{\Delta t}Z \end{aligned}$$

with

$$\tilde{b}(t_i, Y_i) = b(t_i, Y_i) - \eta\sigma(t_i, Y_i)\sigma_x(t, Y_i)\,.$$

The next code shows an implementation of the predictor-corrector method. With a (small) loss of efficiency, the new `sde.sim` function can replace the old one. Note that the predictor-corrector method falls back to the standard Euler method for $\alpha = \eta = 0$. The function by default implements the predictor-corrector method with $\alpha = \eta = \frac{1}{2}$. We only report here the modification to previous code. As usual, the complete version of `sde.sim` can be found in the `sde` package.

```
sde.sim <- function(t0=0, T=1, X0=1, N=100, delta,
         drift, sigma, sigma.x,
         method=c("euler","milstein"),
         alpha=0.5, eta=0.5, pred.corr=T){

# (...)

  if(pred.corr==F){
   alpha <- 0
   eta <- 0
   sigma.x <- NULL
  }

# (...)

  if(method=="milstein" | (method=="euler" & pred.corr==T))
   needs.sx <- TRUE

  d1.t <- function(t,x) d1(t,x) - eta * s1(t,x) * sx(t,x)

  if(method=="euler"){
   for(i in 2:(N+1)){
     Y[i] <- X[i-1] + d1(t[i-1],X[i-1])*Dt +
             s1(t[i-1],X[i-1])*sDt*Z[i-1]
     if(pred.corr==T)
      X[i] <- X[i-1] + (alpha*d1.t(t[i],Y[i]) +
              (1-alpha)* d1.t(t[i],X[i-1]))*Dt +
              (eta * s1(t[i],Y[i]) +
              (1-eta)*s1(t[i-1],Y[i-1]))*sDt*Z[i-1]
     else
      X[i] <- Y[i]
   }
  }
# (...)
}
```

There are different predictor-corrector methods available that can be applied to discretization schemes of order greater than 1. The reader should look, for example, at Section 5.3 in [142].

2.7 Strong convergence for Euler and Milstein schemes

To show in practice how strong convergence takes place in discretization schemes, we reproduce here nice empirical evidence from [125]. The objective of this section is to show how the Milstein scheme outperforms the Euler scheme in convergence in the simple case of geometric Brownian motion. As already mentioned, the theory says that convergence is in the limit of the discretization step as $\Delta t \to 0$. This experiment proceeds as follows:

1. We first simulate trajectories of the Brownian motion with an increased level of refinement (i.e., with a decreasing value of Δt). This is done iteratively using the Brownian bridge.
2. We then construct the trajectory of the geometric Brownian motion with both Euler and Milstein schemes using the path of the Wiener process available.
3. We then compare the values of the process X at time T in the two cases.

Figure 2.1 is the result of step 1, and we now explain the rationale behind it. On the top-left corner is depicted the trajectory of a Brownian bridge starting from 1 at time $t_0 = 0$ and ending at 1 at time $T = 1$ using $N = 2$ intervals, and indeed we have three points of the trajectory at times 0, $\frac{1}{2}$, and 1. The top-right picture has been generated using two Brownian bridges. The first Brownian bridge starts at 1 at time 0 and ends at $B(\frac{1}{2})$ at time $t = \frac{1}{2}$. The second Brownian bridge starts at $B(\frac{1}{2})$ at time $t = \frac{1}{2}$ and ends up at 1 at time 1. This trajectory has five points at times 0, $\frac{1}{4}$, $\frac{1}{2}$, $\frac{3}{4}$, 1. In the next step, the procedure is iterated splitting each interval $[0, \frac{1}{4}], \ldots, [\frac{3}{4}, 1]$ into two parts up to the final bottom-right picture, consisting of $2^{14} + 1$ points of the trajectory of the Wiener process. The figure is generated with the following code, which we show because this dyadic algorithm is also a constructive way of building a process with continuous but nowhere differentiable path (i.e., the Wiener process).

```
> # ex 2.10.R
> set.seed(123)
> W <- vector(14,mode="list")
> W[[1]] <- BBridge(1,1,0,1,N=2)
>
> for(i in 1:13){
+  cat(paste(i,"\n"))
+  n <- length(W[[i]])
+  t <- time(W[[i]])
+  w <- as.numeric(W[[i]])
+  tmp <- w[1]
+  for(j in 1:(n-1)){
+    tmp.BB <- BBridge(w[j],w[j+1],t[j],t[j+1],N=2)
+    tmp <- c(tmp, as.numeric(tmp.BB[2:3]))
+   }
+   W[[i+1]] <- ts(tmp,start=0,deltat=1/(2^(i+1)))
+ }
> min.w <- min(unlist(W))-0.5
> max.w <- max(unlist(W))+0.5
> opar <- par(no.readonly = TRUE)
> par(mfrow=c(7,2),mar=c(3,0,0,0))
> for(i in 1:14){
```

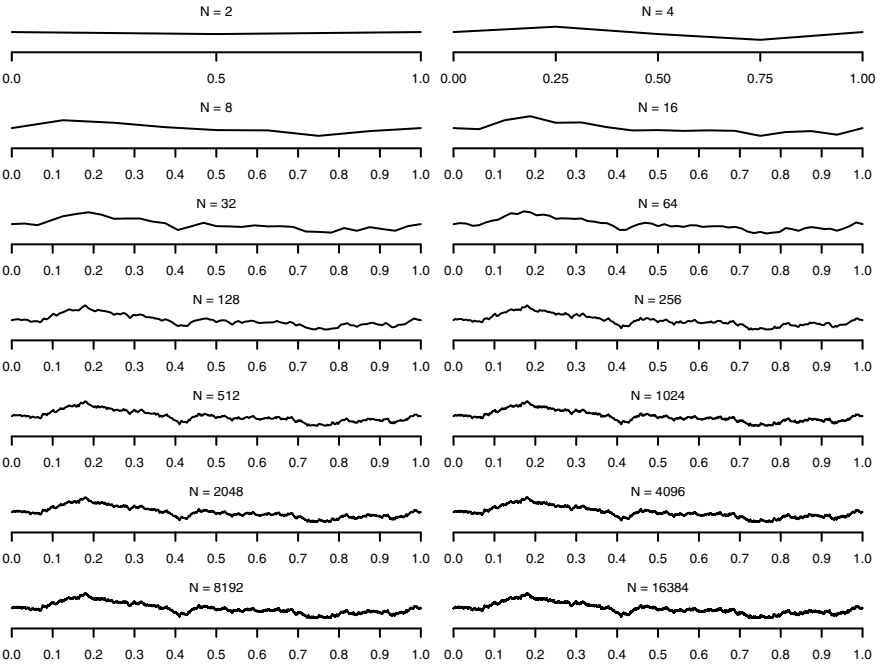

Fig. 2.1. A simulated path of the Wiener process for increased levels of discretization, N being the number of subintervals of [0,1].

```
+  plot(W[[i]], ylim=c(min.w, max.w),axes=F)
+  if(i==1)
+   axis(1,c(0,0.5,1))
+  if(i==2)
+  axis(1,c(0,0.25,0.5,0.75,1))
+  if(i>2)
+   axis(1,c(0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1))
+  text(0.5,2.2,sprintf("N = %d",2^i))
+ }
> par(opar)
```

In the R code above, we make use of the `BBridge` function of the `sde` package (see also Listing 1.2).

The next step is to simulate the trajectory of the geometric Brownian motion using both the Euler and Milstein schemes and calculate the value $X(T)$ with the two schemes. The following script does the calculations and plots both values against the true value $X(T)$ for the given Wiener process path (i.e., the Wiener process ending in 1 at time 1, which is a sort of Brownian bridge) $X(1) = \exp\{\theta_1 - \frac{1}{2}\theta_2^2\}$. We have chosen $\theta_1 = 1$ and $\theta_2 = \frac{1}{2}$.

```
> # ex 2.11.R
> S0 <- 1
> theta <- c(1, 0.5)
> euler <- NULL
> milstein <- NULL
> for(i in 1:14){
```

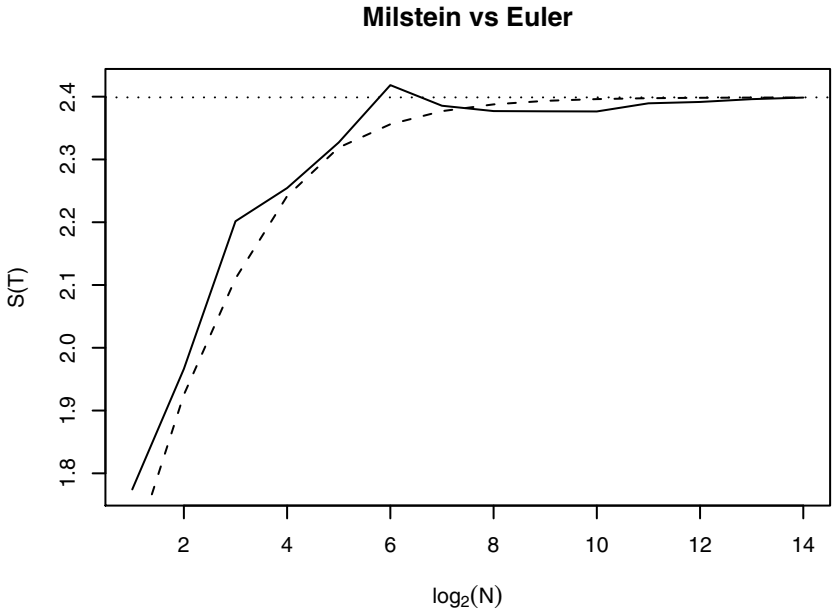

Fig. 2.2. Speed of convergence of Euler and Milstein schemes (Euler = solid line, Milstein = dashed line) to the true value (dotted line) as a function of the discretization step $\Delta t = 1/N$.

```
+   n <- length(W[[i]])
+   Dt <- 1/n
+   sDt <- sqrt(Dt)
+   E <- numeric(n)
+   E[1] <- S0
+   M <- numeric(n)
+   M[1] <- S0
+   for(j in 2:n){
+       Z <- W[[i]][j]-W[[i]][j-1]
+       E[j] <-  E[j-1] * (1 + theta[1] * Dt + theta[2] * Z)
+       M[j] <-  M[j-1] * (1 + (theta[1] - 0.5* theta[2]^2) * Dt +
+        theta[2] * Z + 0.5 * theta[2]^2 * Z^2)
+   }
+   cat(paste(E[n],M[n],"\n"))
+   euler <- c(euler, E[n])
+   milstein <- c(milstein, M[n])
+ }

> plot(1:14,euler,type="l",main="Milstein vs Euler",
+  xlab=expression(log[2](N)), ylab="S(T)")
> lines(1:14,milstein,lty=2)
> abline(h=exp(theta[1]-0.5*theta[2]^2),lty=3)
```

Figure 2.2 shows the speed of convergence of both schemes (Euler = solid line, Milstein = dashed line) to the true value (dotted line) as a function of $\Delta t = 1/N$.

2.8 KPS method of strong order $\gamma = 1.5$

By adding more terms to the Itô-Taylor expansion, one can achieve a strong order γ higher than 1 (for a detailed review and implementation, see [142]). In particular, the following scheme[3] (see, e.g., [143]) has strong order $\gamma = 1.5$:

$$\begin{aligned} Y_{i+1} = Y_i &+ b\Delta t + \sigma \Delta W_t + \frac{1}{2}\sigma\sigma_x \left\{(\Delta W_t)^2 - \Delta t\right\} \\ &+ \sigma b_x \Delta U_t + \frac{1}{2}\left\{bb_x + \frac{1}{2}\sigma^2 b_{xx}\right\}\Delta t^2 \\ &+ \left\{b\sigma_x + \frac{1}{2}\sigma^2\sigma_{xx}\right\}\{\Delta W_t \Delta t - \Delta U_t\} \\ &+ \frac{1}{2}\sigma(\sigma\sigma_x)_x \left\{\frac{1}{3}(\Delta W_t)^2 - \Delta t\right\}\Delta W_t \,, \end{aligned}$$

where

$$\Delta U_t = \int_{t_0}^{t_{i+1}} \int_{t_i}^{s} \mathrm{d}W_u \mathrm{d}s$$

is a Gaussian random variable with zero mean and variance $\frac{1}{3}\Delta t^3$ and such that $\mathbb{E}(\Delta U_t \Delta W_t) = \frac{1}{2}\Delta t^2$. All the pairs $(\Delta W_t, \Delta U_t)$ are mutually independent for all t_i's. To implement this scheme, additional partial derivatives of the drift and diffusion coefficient are required and the algorithm in `sde.sim` that generates Gaussian variates must be changed to allow for bivariate Gaussian variates for the pairs $(\Delta W_t, \Delta U_t)$. With this aim, we use the function `mvrnorm` in the `MASS` package, which implements the algorithm described in [195].

Note that the Euler scheme is not of strong order $\gamma = 1.5$ for the Ornstein-Uhlenbeck process, as there is the additional term $\sigma b_x \Delta U_t$ in the expansion. The next code implements the strong order scheme above and, as usual, we just report the additional part of the `sde.sim` function.

```
sde.sim <- function(t0=0, T=1, X0=1, N=100, delta,
          drift, sigma, drift.x, sigma.x, drift.xx, sigma.xx,
          method=c("euler","milstein","KPS"),
          alpha=0.5, eta=0.5, pred.corr=T){

# (...)

  if(method == "KPS") {
   needs.sx <- TRUE
   needs.dx <- TRUE
   needs.sxx <- TRUE
   needs.dxx <- TRUE
  }

# (...)

  if(method=="euler")
   X <- sde.sim.euler(X0, t0, Dt, N, d1, s1, s1.x, alpha, eta, pred.corr)
```

[3] We name this scheme KPS after the authors who proposed it.

```
  if(method=="milstein")
   X <- sde.sim.milstein(X0,  t0, Dt, N, d1, s1, s1.x)

  if(method=="KPS"){
    require(MASS)
    Sigma <- matrix(c(Dt, 0.5*Dt^2, 0.5*Dt^2, 1/3*Dt^3),2,2)
    tmp <- mvrnorm(N, c(0,0), Sigma)
    Z <- tmp[,1]
    U <- tmp[,2]
    X <- sde.sim.KPS(X0,  t0, Dt, N, d1, d1.x, d1.xx,
                s1, s1.x, s1.xx, Z, U)
  }

# (...)

}
```

Listing 2.1. Simulation of paths of processes governed by stochastic differential equations.

In the code of Listing 2.1, we have separated into three functions the different schemes of simulation: `sde.sim.euler`, `sde.sim.milstein`, and `sde.sim.KPS`. The reason for this approach is twofold. On the one hand, the `sde.sim` became just an interface for different schemes, hence allowing even for generalization toward the implementation of new schemes. On the other hand, separating functions allows for an implementation in `C` code to speed up the execution (this will be appreciated in Monte Carlo experiments). The R code corresponding to the functions above can be found in Listings 2.2, 2.3, and 2.4. Of their `C` counterparts we only present the Milstein scheme, in Listing 2.5, while the complete source code can be found as part of the R package `sde` available through the CRAN repository. The `C` code will be called in R using the following lines.

```
sde.sim.milstein <- function(X0,  t0, Dt, N, d1, s1, s1.x){
   return( .Call("sde_sim_milstein",  X0,  t0, Dt, as.integer(N), d1,
               s1, s1.x, .GlobalEnv) )
}
```

What is interesting from the reading of `C` code is the use of the `feval()` function. This function, defined in Listing 2.6, evaluates an R function directly from the `C` code and uses the result in the internal loops, speeding up the whole simulation scheme. We found that these `C` routines are about two times faster than their R counterparts but no more. Indeed, we are paying the cost of flexibility by allowing the `sde.sim` function to accept any sort of specification of drift and diffusion coefficients. Of course, for intensive simulation studies on a specific model, writing the complete code in `C` might be worth trying.

```
sde.sim.euler <- function(X0, t0, Dt, N, d1, s1, s1.x,
   alpha, eta, pred.corr){
  X <- numeric(N+1)
  Y <- numeric(N+1)
  sDt <- sqrt(Dt)
  Z <- rnorm(N, sd=sDt)
  X[1] <- X0
  Y[1] <- X0
```

```
 d1.t <- function(t,x)  d1(t,x) - eta * s1(t,x) * s1.x(t,x)

  if(pred.corr==TRUE){
   for(i in 2:(N+1)){
     Y[i] <- X[i-1] + d1(t[i-1],X[i-1])*Dt + s1(t[i-1],X[i-1])*Z[i-1]
     X[i] <- X[i-1] + (alpha*d1.t(t[i],Y[i]) +
               (1-alpha)* d1.t(t[i],X[i-1]))*Dt +
               (eta * s1(t[i],Y[i]) +
               (1-eta)*s1(t[i-1],Y[i-1]))*Z[i-1]
   }
  } else {
   for(i in 2:(N+1))
    X[i] <- X[i-1] + d1(t[i-1],X[i-1])*Dt + s1(t[i-1],X[i-1])*Z[i-1]
  }
  return(X)
}
```

Listing 2.2. R code for Euler simulation scheme.

```
sde.sim.milstein <- function(X0,  t0, Dt, N, d1, s1, s1.x){
  X <- numeric(N+1)
  Y <- numeric(N+1)
  sDt <- sqrt(Dt)
  Z <- rnorm(N, sd=sDt)

  X[1] <- X0
  Y[1] <- X0

  for(i in 2:(N+1)){
    X[i] <- X[i-1] + d1(t[i-1],X[i-1])*Dt +
              s1(t[i-1],X[i-1])*Z[i-1] +
              0.5*s1(t[i-1],X[i-1])* s1.x(t[i-1],X[i-1]) *
              (Z[i-1]^2-Dt)
  }
 return(X)
}
```

Listing 2.3. R code for Milstein simulation scheme.

```
sde.sim.KPS <- function(X0,  t0, Dt, N, d1, d1.x, d1.xx,
   s1, s1.x, s1.xx, Z, U){
  X <- numeric(N+1)
  Y <- numeric(N+1)
  Dt <- (T-t0)/N
  sDt <- sqrt(Dt)
  X[1] <- X0
  Y[1] <- X0

   for(i in 2:(N+1)){
    D1 <- d1(t[i-1],X[i-1])
    D1.x <- d1.x(t[i-1],X[i-1])
    D1.xx <- d1.xx(t[i-1],X[i-1])
    S1 <- s1(t[i-1],X[i-1])
    S1.x <- s1.x(t[i-1],X[i-1])
    S1.xx <- s1.xx(t[i-1],X[i-1])

    X[i] <- X[i-1] + D1 * Dt + S1 * Z[i-1] +
         0.5 * S1 * S1.x * (Z[i-1]^2-Dt) +
    S1 * D1.x * U[i-1] +
    0.5 * (D1 * D1.x + 0.5 * S1^2 * D1.xx) * Dt^2 +
    (D1 * S1.x + 0.5 * S1^2 * S1.xx) * (Z[i-1] * Dt - U[i-1]) +
     0.5 * S1 * (S1.x^2 + S1*S1.xx) * (1/3*Z[i-1]^2 - Dt) * Z[i-1]
   }
  return(X)
```

```
}
```

Listing 2.4. `R` code for KPS simulation scheme.

```
SEXP sde_sim_milstein(SEXP x0, SEXP t0, SEXP delta, SEXP N,
                      SEXP d, SEXP s, SEXP sx, SEXP rho)
{
  double T, DELTA;
  double sdt, Z, tmp, D, S, Sx;
  int i, n;
  SEXP X;

  if(!isNumeric(x0)) error("`x0' must be numeric");
  if(!isNumeric(t0)) error("`t0' must be numeric");
  if(!isNumeric(delta)) error("`delta' must be numeric");
  if(!isInteger(N)) error("`N' must be integer");
  if(!isFunction(d)) error("`d' must be a function");
  if(!isFunction(s)) error("`s' must be a function");
  if(!isFunction(sx)) error("`sx' must be a function");
  if(!isEnvironment(rho)) error("`rho' must be an environment");

  PROTECT(x0 = AS_NUMERIC(x0));
  PROTECT(delta = AS_NUMERIC(delta));
  PROTECT(t0 = AS_NUMERIC(t0));
  PROTECT(N = AS_INTEGER(N));

  T = *NUMERIC_POINTER(t0);
  n = *INTEGER_POINTER(N);
  DELTA = *NUMERIC_POINTER(delta);

  PROTECT(X = NEW_NUMERIC(n+1));
  REAL(X)[0] = *NUMERIC_POINTER(x0);
  sdt = sqrt(DELTA);

  GetRNGstate();
  for(i=1; i<= n+1; i++){
   Z = rnorm(0,sdt);
   T = T + DELTA;
   tmp = REAL(X)[i-1];
   D = feval(T,tmp,d,rho);
   S = feval(T,tmp,s,rho);
   Sx = feval(T,tmp,sx,rho);
   REAL(X)[i] = tmp + D*DELTA + S*Z + 0.5*S*Sx*(Z*Z-DELTA);
  }
  PutRNGstate();

  UNPROTECT(5);
  return(X);
}
```

Listing 2.5. `C` code for Milstein simulation scheme.

```
/*
   t : time variable
   x : space variable
   f : a SEXP to a R function
   rho : the environment `f' is going to be evaluated

   on return:   the value of f(t,x)
*/
double feval(double t, double x, SEXP f, SEXP rho)
{
 double val= 0.0;
 SEXP R_fcall, tpar, xpar;
```

```
  PROTECT(tpar = allocVector(REALSXP, 1));
  PROTECT(xpar = allocVector(REALSXP, 1));
  REAL(tpar)[0] = t;
  REAL(xpar)[0] = x;

  PROTECT(R_fcall = allocList(3));
  SETCAR(R_fcall, f);
  SET_TYPEOF(R_fcall, LANGSXP);
  SETCADR(R_fcall, tpar);
  SETCADDR(R_fcall, xpar);

  val = *NUMERIC_POINTER(eval(R_fcall, rho));
  UNPROTECT(3);
  return(val);
}
```

Listing 2.6. `C` code for the `feval` function, which allows for the calculation of `R` functions directly in the `C` code.

2.9 Second Milstein scheme

In [164], Milstein proposed both (2.2) and the approximation

$$\begin{aligned} Y_{i+1} = Y_i &+ \left(b - \frac{1}{2}\sigma\sigma_x\right)\Delta t + \sigma Z\sqrt{\Delta t} + \frac{1}{2}\sigma\sigma_x \Delta t Z^2 \\ &+ \Delta t^{\frac{3}{2}}\left(\frac{1}{2}b\sigma_x + \frac{1}{2}b_x\sigma + \frac{1}{4}\sigma^2\sigma_{xx}\right)Z \\ &+ \Delta t^2\left(\frac{1}{2}bb_x + \frac{1}{4}b_{xx}\sigma^2\right). \end{aligned} \tag{2.9}$$

This method has weak second-order convergence in contrast to the weak first-order convergence of the Euler scheme. This method requires partial (first and second) derivatives of both drift and diffusion coefficients. Listing 2.7 contains the code corresponding to the approximation (2.9). The function `sde.sim` needs to be modified as follows.

```
  if(method == "KPS" | method == "milstein2") { # added "milstein2" method
   needs.sx <- TRUE
   needs.dx <- TRUE
   needs.sxx <- TRUE
   needs.dxx <- TRUE
  }

# (...)

  if(method=="milstein2") # added a call to the sde.sim.milstein2
   X <- sde.sim.milstein2(X0,  t0, Dt, N, d1, d1.x, d1.xx, s1, s1.x, s1.xx)
```

```
sde.sim.milstein2 <- function(X0,  t0, Dt, N, d1, d1.x, d1.xx,
      s1, s1.x, s1.xx){
  X <- numeric(N+1)
  Y <- numeric(N+1)
  sDt <- sqrt(Dt)
```

```
  Z <- rnorm(N, sd=sDt)
  X[1] <- X0
  Y[1] <- X0

  for(i in 2:(N+1)){
    X[i] <- X[i-1] + d1(t[i-1],X[i-1])*Dt +
              s1(t[i-1],X[i-1])*Z[i-1] +
              0.5*s1(t[i-1],X[i-1])* s1.x(t[i-1],X[i-1]) *(Z[i-1]^2-Dt) +
         Dt^(3/2)*(0.5*d1(t[i-1],X[i-1])*s1.x(t[i-1],X[i-1]) +
         0.5*d1.x(t[i-1],X[i-1])*s1(t[i-1],X[i-1])+
         0.25*s1(t[i-1],X[i-1])^2 * s1.xx(t[i-1],X[i-1]))*Z[i-1] +
         Dt^2*(0.5* d1(t[i-1],X[i-1])*d1.x(t[i-1],X[i-1])+
         0.25*d1.xx(t[i-1],X[i-1])*s1(t[i-1],X[i-1])^2)
  }
 return(X)
}
```

Listing 2.7. `R` code for the second Milstein simulation scheme.

2.10 Drawing from the transition density

All the methods presented so far are based on the discretized version of the stochastic differential equation. In the case where a transition density of X_t given some previous value X_s, $s < t$, is known in explicit form, direct simulation from this can be done. Unfortunately, the transition density is known for very few processes, and these cases are the ones for which exact likelihood inference can be done, as will be discussed in Chapter 3. In these fortunate cases, the algorithm for simulating processes is very easy to implement. We suppose that a random number generator is available for the transition density for the process $p_\theta(\Delta, y|x) = Pr(X_{t+\Delta} \in \mathrm{d}y|X_t = x)$. If this generator is not available one can always use one of the standard methods to draw from known densities, such as the rejection method. We are not going to discuss this approach here and assume this random number generator exists in the form of an R function that accepts the number `n` of pseudo random numbers to draw, a vector of length `n` of values `x` (this will play the role of the X_t's), the time lags `Dt`, and a vector of parameters `theta`. The fact that we allow for `n` random numbers to be generated will be useful whenever one wants to simulate multiple trajectories of the same process in a way we discuss at the end of the chapter. The next function generates a complete path of a stochastic process for which the random number generator `rcdist` is known. We assume that the corresponding model is parametrized through a vector of parameters `theta`.

```
sde.sim.cdist <- function(X0=1, t0=0, Dt=0.1, N, rcdist=NULL, theta=NULL){
   X <- numeric(N+1)
   X[1] <- X0
   for(i in 2:(N+1)){
    X[i] <- rcdist(1, Dt, X[i-1], theta)
   }
 ts(X, start=t0, deltat=Dt)
}
```

The function `sde.sim.cdist` just iterates calls to the random number generator `rcdist`, assigning to `X[i]` the pseudo random number generated from the law of $X_{t+\Delta t}|X_t =$ `X[i-1]`. We now write the random number generators for the processes for which the conditional distribution is known.

2.10.1 The Ornstein-Uhlenbeck or Vasicek process

Recall that the Ornstein-Uhlenbeck or Vasicek process solution to

$$\mathrm{d}X_t = (\theta_1 - \theta_2 X_t)\mathrm{d}t + \theta_3 \mathrm{d}W_t, \qquad X_0 = x_0,$$

has a known Gaussian transition density $p_\theta(\Delta, y|X_t = x)$ with mean and variance as in (1.41) and (1.42), respectively. Hence we can just make use of the `rnorm` function to build our random number generator.

```
rcOU <- function(n=1, t, x0, theta){
  Ex <- theta[1]/theta[2]+(x0-theta[1]/theta[2])*exp(-theta[2]*t)
  Vx <- theta[3]^2*sqrt((1-exp(-2*theta[2]*t))/(2*theta[2]))
  rnorm(n, mean=Ex, sd = sqrt(Vx))
}
```

The functions `[rpdq]cOU` in the `sde` package provide interfaces to random number generation, cumulative distribution function, density function, and quantile calculations, respectively, for the *conditional* law of the Ornstein-Uhlenbeck process. Similarly, the functions `[rpdq]sOU` provide the same functionalities for the *stationary* law of the process.

2.10.2 The Black and Scholes process

The Black and Scholes or geometric Brownian motion process solution of

$$\mathrm{d}X_t = \theta_1 X_t \mathrm{d}t + \theta_2 X_t \mathrm{d}W_t$$

has a log-normal transition density $p_\theta(\Delta, y|x)$, where the log-mean and log-variance are given in (1.43). The following code implements the random number generator from the conditional law.

```
rcBS <- function(n=1, Dt, x0, theta){
   lmean <- log(x0) + (theta[1]-0.5*theta[2]^2)*Dt
   lsd <- sqrt(Dt)*theta[2]
   rlnorm(n, meanlog = lmean, sdlog = lsd)
}
```

The package `sde` provides the functions `[rpdq]cBS` for random number generation, cumulative distribution function, density function, and quantile calculations of the conditional law of $X_{t+\Delta}|X_t$.

2.10.3 The CIR process

The conditional density of $X_{t+\Delta}|X_t = x$ for the Cox-Ingersoll-Ross process solution of

$$\mathrm{d}X_t = (\theta_1 - \theta_2 X_t)\mathrm{d}t + \theta_3\sqrt{X_t}\mathrm{d}W_t$$

is a noncentral chi-squared distribution (see Section 1.13.3). In particular, we have shown in (1.48) that $p_\theta(\Delta, y|x)$ can be rewritten in terms of the transition density of $Y_t = 2cX_t$, which has a chi-squared distribution with $\nu = 4\theta_1/\theta_3^2$ degrees of freedom and noncentrality parameter $Y_s e^{-\theta_2 t}$, where $c = 2\theta_2/(\theta_3^2(1-e^{-\theta_2 t}))$. So we just need to simulate a value of y from $Y_t|Y_s = 2cx_s$ and return $y/(2c)$. The following code does the job.

```
rcCIR <- function(n=1, Dt, x0, theta){
   c <- 2*theta[2]/((1-exp(-theta[2]*t))*theta[3]^2)
   ncp <- 2*c*x0*exp(-theta[2]*Dt)
   df <- 4*theta[1]/theta[3]^2
   rchisq(n, df=df, ncp=ncp)/(2*c)
}
```

Also, for the Cox-Ingersoll-Ross process, the package `sde` provides the functions `[rpdq]cCIR` and `[rpdq]sCIR` for random number generation, cumulative distribution function, density function, and quantile calculations, respectively, for the conditional and stationary laws.

2.10.4 Drawing from one model of the previous classes

Given that for the Cox-Ingersoll-Ross, Ornstein-Uhlenbeck, and geometric Brownian motion processes the transition densities are known in explicit form, we can add a more flexible interface in `sde.sim` to simulate these models. In fact, we add the switch `model`, which can be one between "`OU`," "`BS`," and "`CIR`" (with the obvious meanings) and a vector of parameter `theta`. So, for example,

```
sde.sim(model="CIR", theta=c(3,2,1))
```

simulates a path of the Cox-Ingersoll-Ross process solution of $\mathrm{d}X_t = (3 - 2X_t)\mathrm{d}t + \sqrt{X_t}\mathrm{d}W_t$ with initial value $X_0 = 1$ (the default value in `sde.sim`). Below we show the relevant code change to the `sde.sim` function.

```
sde.sim <- function (t0 = 0, T = 1, X0 = 1, N = 100, delta, drift, sigma,
    drift.x, sigma.x, drift.xx, sigma.xx, drift.t, method = c("euler",
        "milstein", "KPS", "milstein2", "cdist"),
    alpha = 0.5, eta = 0.5, pred.corr = T, rcdist = NULL, theta = NULL,
    model = c("CIR", "VAS", "OU", "BS"))
{
    method <- match.arg(method)
    if(!missing(model)){
     model <- match.arg(model)
     method <- "model"
        }

    if (missing(drift)){
     if (method == "cdist" || !missing(model))
      drift <- expression(NULL)
     else
        stop("please specify al least the drift coefficient of the SDE")
    }

# (...)
```

```
     if(method == "model"){
      if(is.null(theta))
         stop("please provide a vector of parameters for the model")
     if(model == "CIR")
        X <- sde.sim.cdist(X0, t0, Dt, N, rcCIR, theta)
     if(model == "OU")
        X <- sde.sim.cdist(X0, t0, Dt, N, rcOU, theta)
     if(model == "BS")
        X <- sde.sim.cdist(X0, t0, Dt, N, rcBS, theta)
      }

# (...)
}
```

2.11 Local linearization method

The local linearization method consists in approximating locally the drift of the stochastic differential equation with a linear function. The main idea behind this technique is that a linear approximation is better than the simple constant approximation made by the Euler method (see, e.g., [24], [13]). The method has been proposed in the context of stochastic differential equations by Ozaki and developed by him and his co-authors (see [173], [174], [175], [204], [206], [207]).

2.11.1 The Ozaki method

The first approach we present is the Ozaki method, and it works for homogeneous stochastic differential equations. Consider the stochastic differential equation

$$\mathrm{d}X_t = b(X_t)\mathrm{d}t + \sigma \mathrm{d}W_t\,, \tag{2.10}$$

where σ is supposed to be constant. The construction of the method starts from the corresponding deterministic dynamical system

$$\frac{\mathrm{d}x_t}{\mathrm{d}t} = b(x_t)\,,$$

where x_t has to be a smooth function of t in the sense that it is two times differentiable with respect to t. Then, with a little abuse of notation, we have

$$\frac{\mathrm{d}^2 x_t}{\mathrm{d}t^2} = b_x(x_t)\frac{\mathrm{d}x_t}{\mathrm{d}t}\,.$$

Suppose now that $b_x(x)$ is constant in the interval $[t, t+\Delta t)$, and hence by iterated integration of both sides of the equation above, first from t to $u \in [t, t+\Delta t)$ and then from t to $t+\Delta t$, we obtain the difference equation

$$x_{t+\Delta t} = x_t + \frac{b(x_t)}{b_x(x_t)}\left(e^{b_x(x_t)\Delta t} - 1\right)\,. \tag{2.11}$$

Now we translate the result above back to the stochastic dynamical system in (2.10). So, suppose $b(x)$ is approximated by the linear function $K_t x$, where K_t is constant in the interval[4] $[t, t+\Delta t)$. The solution to the stochastic differential equation is

$$X_{t+\Delta t} = X_t e^{K_t \Delta t} + \sigma \int_t^{t+\Delta t} e^{K_t(t+\Delta t-u)} \mathrm{d}W_u \,.$$

Now what remains to be done is to determine the constant K_t. The main assumption is that the conditional expectation of $X_{t+\Delta t}$ given X_t,

$$\mathbb{E}(X_{t+\Delta t}|X_t) = X_t e^{K_t \Delta t},$$

coincides with the state of the linearized dynamical system (2.11) at time $t + \Delta t$, which means that we ask for the following equality to hold:

$$X_t e^{K_t \Delta t} = X_t + \frac{b(X_t)}{b_x(X_t)} \left(e^{b_x(X_t)\Delta t} - 1 \right).$$

From the above, we obtain the constant K_t very easily:

$$K_t = \frac{1}{\Delta t} \log \left(1 + \frac{b(X_t)}{X_t b_x(X_t)} \left(e^{b_x(X_t)\Delta t} - 1 \right) \right).$$

Notice that K_t depends on t only through the state of the process $X_t = x$. Hence we denote this constant by K_x to make the notation more consistent throughout the book. Given that the stochastic integral is a Gaussian random variable, it is clear that the transition density for $X_{t+\Delta t}$ given X_t is Gaussian as well. In particular, we have that $X_{t+\Delta t}|X_t = x \sim N(E_x, V_x)$, where

$$E_x = x + \frac{b(x)}{b_x(x)} \left(e^{b_x(x)\Delta t} - 1 \right), \tag{2.12}$$

$$V_x = \sigma^2 \frac{e^{2K_x \Delta t} - 1}{2K_x}, \tag{2.13}$$

with

$$K_x = \frac{1}{\Delta t} \log \left(1 + \frac{b(x)}{x b_x(x)} \left(e^{b_x(x)\Delta t} - 1 \right) \right).$$

So it is possible to use the method of drawing from the conditional law to simulate the increments of the process as in Section 2.10, simulating `X[i+1]` according to $N(E_x, V_x)$, where $x =$ `X[i]`. It is easy to implement this simulation scheme, and it is presented in Listing 2.8. The function `sde.sim.ozaki` will be called by `sde.sim` when `method` is equal to "`ozaki`." This function assumes that the diffusion coefficient is a constant and that the drift function depends only on the state variable `x`. These assumptions are checked inside the `sde.sim` interface as follows.

[4] Hence the name "local linearization method."

```
if (method == "ozaki"){
      vd <- all.vars(drift)
      vs <- all.vars(sigma)
      if(length(vd)!=1 || length(vs)>0)
        stop("drift must depend on `x' and volatility must be constant")
      if((length(vd) == 1) && (vd != "x"))
        stop("drift must depend on `x'")
     X <- sde.sim.ozaki(X0, t0, Dt, N, d1, d1.x, s1)
}
```

Please note that a constant drift is not admissible since K_x and hence V_x are not well defined.

```
"sde.sim.ozaki" <-
function(X0, t0, Dt, N, d1, d1.x, s1){
   X <- numeric(N+1)
   B <- function(x) d1(1,x)
   Bx <- function(x) d1.x(1,x)
   S <- s1(1,1)
   X[1] <- X0
   for(i in 2:(N+1)){
    x <- X[i-1]
    Kx <- log(1+B(x)*(exp(Bx(x)*Dt)-1)/(x*Bx(x)))/Dt
    Ex <- x + B(x)/Bx(x)*(exp(Bx(x)*Dt)-1)
    Vx <- S^2 * (exp(2*Kx*Dt) -1)/(2*Kx)
    X[i] <- rnorm(1, mean=Ex, sd=sqrt(Vx))
   }
 X
}
```

Listing 2.8. R code for the Ozaki simulation scheme.

Of course, the Ozaki method coincides with the Euler method if the drift is linear. The reader can try the following lines of code.

```
> # ex 2.12.R
> set.seed(123)
> X <- sde.sim(drift=expression(-3*x), method="ozaki")
> set.seed(123)
> Y <- sde.sim(drift=expression(-3*x))
> plot(X)
> lines(as.numeric(time(Y)), Y, col="red")
```

2.11.2 The Shoji-Ozaki method

An extension of the previous method to the more general case in which the drift is allowed to depend on the time variable also and the diffusion coefficient can vary is the Shoji-Ozaki method (see [204], [205], and [206]). Consider the stochastic differential equation

$$\mathrm{d}X_t = b(t, X_t)\mathrm{d}t + \sigma(X_t)\mathrm{d}W_t\,,$$

where b is two times continuously differentiable in x and continuously differentiable in t and σ is continuously differentiable in x. We already know that it is always possible to transform this equation into one with a constant diffusion coefficient using the Lamperti transform of Section 1.11.4. So one can start by considering the nonhomogeneous stochastic differential equation

$$\mathrm{d}X_t = b(t, X_t)\mathrm{d}t + \sigma \mathrm{d}W_t\,,$$

which is different from (2.10) in that the drift function also depends on variable t. Now the local linearization method is developed by studying the behavior of b locally. We skip all the details, which can be found in the original works [207] and [206], but the main point is that the equation above is approximated locally on $[s, s+\Delta s)$ with

$$\mathrm{d}X_t = (L_s X_t + tM_s + N_s)\mathrm{d}t + \sigma \mathrm{d}W_t, \qquad t \geq s,$$

where

$$L_s = b_x(s, X_s), \qquad M_s = \frac{\sigma^2}{2} b_{xx}(s, X_s) + b_t(s, X_s)\,,$$
$$N_s = b(s, X_s) - X_s b_x(s, X_s) - sM_s\,.$$

The next step is to consider the transformed process $Y_t = e^{-L_s t} X_t$, which has the explicit solution

$$Y_t = Y_s + \int_s^t (M_s u + N_s) e^{-L_s u} \mathrm{d}u + \sigma \int_s^t e^{-L_s u} \mathrm{d}W_u,$$

from which the discretization of X_t can be easily obtained and reads as

$$X_{s+\Delta s} = A(X_s) X_s + B(X_s) Z,$$

where

$$A(X_s) = 1 + \frac{b(s, X_s)}{X_s L_s}\left(e^{L_s \Delta s} - 1\right) + \frac{M_s}{X_s L_s^2}\left(e^{L_s \Delta s} - 1 - L_s \Delta s\right), \quad (2.14)$$

$$B(X_s) = \sigma \sqrt{\frac{e^{2L_s \Delta s} - 1}{2L_s}}, \quad (2.15)$$

and $Z \sim N(0,1)$. From the above, it follows that

$$X_{s+\Delta s} | X_s = x \sim N(A(x)x, B^2(x))\,.$$

This method is also quite easy to implement and is just a modification of the previous method. For the sake of simplicity, we will call this the "`shoji`" method to distinguish it in the R code. The only difference with respect to all previous methods is that we also need to specify the partial derivative of the drift coefficient with respect to variable t. This will be an argument of `sde.sim` called `drift.t` and eventually calculated using R symbolic differentiation. We skip the corresponding code and present just Listing 2.9, which implements the simulation part.

```
sde.sim.shoji <- function(X0, t0, Dt, N, d1, d1.x, d1.xx, d1.t, s1){
   X <- numeric(N+1)
   S <- s1(1,1)
   X[1] <- X0
```

```
   for(i in 2:(N+1)){
    x <- X[i-1]
    Lx <- d1.x(Dt,x)
    Mx <- S^2 * d1.xx(Dt,x)/2 + d1.t(Dt,x)
    Ex <- (x + d1(Dt,x)*(exp(Lx*Dt)-1)/Lx +
           Mx*(exp(Lx*Dt) -1 -Lx*Dt)/Lx^2)
    Vx <- S^2*(exp(2*Lx*Dt)-1)/(2*Lx)
    X[i] <- rnorm(1, mean=Ex, sd=sqrt(Vx))
   }
 X
}
```

Listing 2.9. `R` code for the Shoji-Ozaki simulation scheme.

Please note that in this case as well, a drift function not depending on x is not admissible since $A(X_s)$ is not well-defined. One more thing to note is that the Ozaki, Shoji-Ozaki, and Euler methods draw increments from a Gaussian law with mean E_x and variance V_x that in the case of the Euler scheme are $E_x = x + b(x,t)\mathrm{d}t$ and $V_x = V = \sigma^2 \mathrm{d}t$. For the Euler method, the variance V_x is independent from the previous state of the process $X_t = x$, and this property is inherited from the independence of the increments of the Brownian motion. On the contrary, V_x for the Ozaki and Shoji-Ozaki methods depend on the previous state of the process and differ from the value of the constants K_x and L_x, respectively. Even in the linear case, the Shoji-Ozaki method performs differently from the Euler and Ozaki methods. The difference is in the fact that the Shoji-Ozaki method also takes into account the stochastic behavior of the discretization because of the Itô formula. Of course, in the linear homogeneous case the Euler, Shoji-Ozaki, and Ozaki methods coincide. One added value in using the Shoji-Ozaki method over the Ozaki and Euler methods is that it is more stable if the time Δ is large. In fact, not surprisingly, the Euler scheme tends to explode in non-linear cases when Δ is large enough. The following example shows some empirical evidence of this fact. We simulate the solution of $\mathrm{d}X_t = (5 - 11X_t + 6X_t^2 - X_t^3)\mathrm{d}t + \mathrm{d}W_t$, $X_0 = 5$ for $\Delta = 0.1$ and $\Delta = 0.25$. For small values of Δ, all three methods gave similar results, but this is not the case for $\Delta = 0.25$, as can be seen in Figure 2.3, produced with the following code.

```
> # ex 2.13.R
> bX <- expression((5 - 11 * x + 6 * x^2 - x^3))
> x0 <- 5
> DT <- 0.1
> par(mfrow=c(2,3))
> set.seed(123)
> X <- sde.sim(drift=bX, delta=DT,X0=x0)
> plot(X,main="Euler")
> set.seed(123)
> Y <- sde.sim(drift=bX, method="ozaki",delta=DT,X0=x0)
> plot(Y,main="Ozaki")
> set.seed(123)
> Z <- sde.sim(drift=bX, method="shoji",delta=DT,X0=x0)
> plot(Z,main="Shoji-Ozaki")
>
> DT <- 0.25
> set.seed(123)
> X <- sde.sim(drift=bX, delta=DT,X0=x0)
> plot(X, main="Euler")
```

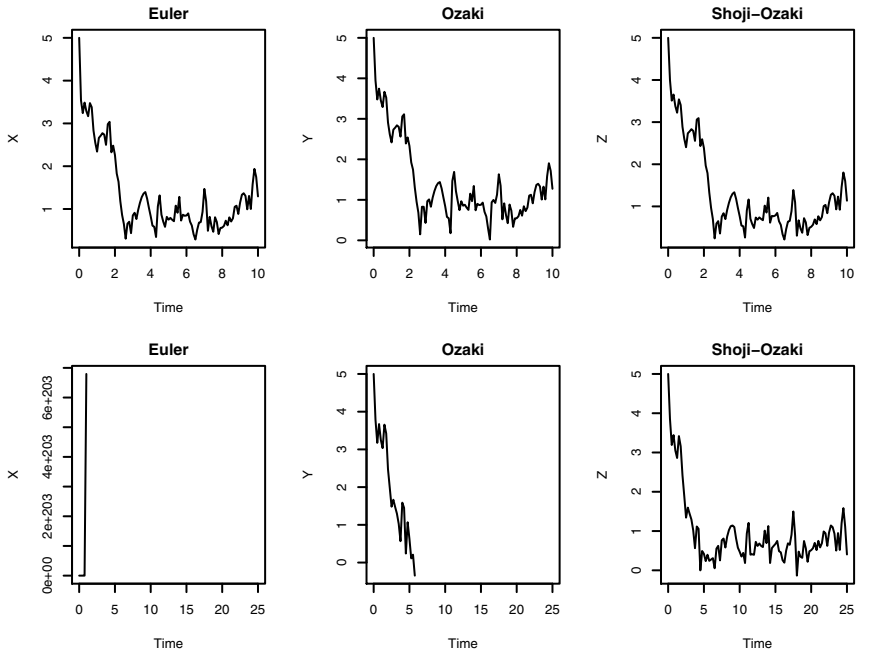

Fig. 2.3. Different performance of the Euler, Ozaki, and Shoji-Ozaki methods for different values of Δ (top: 0.1; bottom: 0.25). The Euler scheme explodes for high values of Δ.

```
> set.seed(123)
> Y <- sde.sim(drift=bX, method="ozaki",delta=DT,X0=x0)
> plot(Y,main="Ozaki")
> set.seed(123)
> Z <- sde.sim(drift=bX, method="shoji",delta=DT,X0=x0)
> plot(Z,main="Shoji-Ozaki")
```

Further properties of the method

As for the properties of this method, the authors show that the Shoji-Ozaki discretization performs well in terms of one-step-ahead error in mean absolute and mean square values. In particular, the mean absolute one-step-ahead error is of order $O(\Delta t^2)$ and the mean square one-step-ahead error is of order $O(\Delta t^3)$ as $\Delta t \to 0$. These errors are measured in terms of the distance between the true trajectory and the approximated trajectory. In particular, the mean square error attains the optimum rate in the sense of Rümelin [198].

Nonconstant diffusion coefficient

If the original stochastic differential equation X_t does not have a constant diffusion coefficient, it is always possible to apply the Lamperti transform of

Section 1.11.4 to obtain a new process Y_t that has a unitary diffusion coefficient. So one can simulate the path of $Y_t = F(X_t)$ and then use the inverse transform F^{-1} to get X_t's path. So, for example, following [207], consider the stochastic differential equation

$$\mathrm{d}X_t = (\alpha_0 + \alpha_1 X_t + \alpha_2 X_t^2 + \alpha_3 X_t^3)\mathrm{d}t + \sigma X_t^\gamma \mathrm{d}W_t\,,$$

where $(\alpha_0 = 6, \alpha_1 = -11, \alpha_2 = 6, \alpha_3 = -1, \gamma = 0.5, \sigma = 1)$. The transformation

$$F(x) = \frac{1}{\sigma}\int_0^x \frac{1}{u^\gamma}\mathrm{d}u = \frac{1}{\sigma}\frac{x^{1-\gamma}}{1-\gamma}$$

and its inverse

$$F^{-1}(y) = (\sigma y(1-\gamma))^{\frac{1}{1-\gamma}}$$

can be used to apply the Shoji-Ozaki method to the stochastic differential equation above. The drift function of process Y_t is then

$$b_y(t,x) = \frac{b(t,x)}{\sigma x^\gamma} - \frac{\gamma\sigma}{2}x^{\gamma-1},$$

which has to be calculated in $F^{-1}(y)$. For the particular choice of σ and γ, we have that $F^{-1}(y) = (y/2)^2$. Hence the process Y_t satisfies

$$\mathrm{d}Y_t = \frac{23 - 11Y_t^2 + \frac{3}{2}Y_t^4 - \frac{1}{2^4}Y_t^6}{2Y_t}\mathrm{d}t + \mathrm{d}W_t, \quad Y_0 = 2\sqrt{X_0}\,.$$

Once we have the trajectory of Y_t, we can get a trajectory of X_t by means of the transformation $X_t = (Y_t/2)^2$. The following lines of code show how to proceed.

```
> # ex 2.14.R
> bY <- expression( (23-11*x^2+1.5*x^4-(x^6)/(2^4))/(2*x) )
> bX <- expression( (6-11*x+6*x^2-x^3) )
> sX <- expression( sqrt(x) )
>
> set.seed(123)
> X <- sde.sim(drift=bX, sigma=sX)
> plot(X)
> set.seed(123)
> Y <- sde.sim(drift=bY, X0 = 2, method="shoji")
> plot((Y/2)^2)
```

2.12 Exact sampling

Very recently there appeared a new proposal for an exact sampling algorithm that, when feasible, is also easy to implement (see [27] and [29]). This method is a rejection sampling algorithm (see, e.g., [227], [68]) for diffusion processes. The rejection sampling algorithm for finite-dimensional random variables is as follows. Suppose f and g are two densities with respect to some measure in $\mathbb{R}^d$ and such that $f(x) \leq \epsilon g(x)$ for some $\epsilon > 0$. If one wants to simulate pseudorandom numbers from f and knows how to simulate from g, then one can use the following algorithm:

1. Sample y from g, $y \sim g$.
2. Sample u from the uniform law, $u \sim U(0,1)$.
3. If $u < \epsilon f(y)/g(y)$, retain y; otherwise iterate from 1.

Then y will be f-distributed. This algorithm needs some modifications if it is to be applied to continuous-time processes such as diffusions because in principle there is the need to generate a whole continuous path of the diffusion[5] before accepting/rejecting it and not just a simple number. Luckily, such an exact sampling algorithm relies on the fact that the rejection rule can be made equivalent to the realization of some event related to point processes and hence simpler to handle. We skip here all the details, but these and other considerations constitute the core of the algorithm proposed in [27]. The algorithm is given for a diffusion with unit diffusion coefficient

$$\mathrm{d}X_t = b(X_t)\mathrm{d}t + \mathrm{d}W_t, \qquad 0 \le t \le T, \quad X_0 = x\,. \tag{2.16}$$

Again, if this is not the case, the Lamperti transform in Section 1.11.4 can be used. From now on, we discuss the algorithm for the exact simulation of the random variable X_Δ for some $\Delta > 0$ and initial value $X_0 = x$. We assume that $b(\cdot)$ satisfies the usual conditions for the existence of the stochastic differential equation (2.16) and also the following assumption.

Assumption 2.1

(i) The derivative b_x of b exists.
(ii) There exist k_1 and k_2 such that $k_1 \le \frac{1}{2}b^2(x) + \frac{1}{2}b_x(x) \le k_2$ for any $x \in \mathbb{R}$.

Let us denote $\phi(x) = \frac{1}{2}b^2(x) + \frac{1}{2}b_x(x) - k_1$. A further requirement is that $0 \le \phi(x) \le M$ for any $x \in \mathbb{R}$, with $M = (k_2 - k_1)$, which implies also that $\Delta \le 1/M$ for identifiability. Now set

$$A(z) = \int_0^z b(u)\mathrm{d}u$$

and

$$h(z) = \exp\left\{A(z) - \frac{(z-x)^2}{2\Delta}\right\}, \quad K = \int_{-\infty}^{\infty} h(u)\mathrm{d}u\,.$$

The function $\tilde{h}(x) = h(x)/K$ is a density function on $\mathbb{R}$ whenever $K < \infty$. The algorithm requires the ability to generate[6] pseudo random numbers from the density $\tilde{h}$. We present here a version of the exact algorithm (EA) in the simplified form as described in [56].

1. Simulate $Y_\Delta = y$ according to distribution $\tilde{h}$.

[5] The ratio $f(y)/g(y)$ of the algorithm should be a likelihood ratio in the case of a diffusion, as given by the Girsanov theorem.

[6] In this case, it is possible to draw from h using a reject sampling algorithm with Gaussian proposals.

2. Simulate $\tau = k$ from the Poisson distribution with intensity $\lambda = \Delta M$.
3. Draw $(T_i, V_i) = (t_i, v_i)$ according to $U[(0, \Delta) \times (0, M)]$, $i = 1, \ldots, k$.
4. Generate a Brownian bridge starting at x at time 0 and ending in y at time Δ at time instants t_i; i.e., generate $Y_{t_i} = y_i$, $i = 1, \ldots, k$.
5. Compute the indicator function

$$I = \prod_{i=1}^{k} \mathbf{1}_{\{\phi(y_i) \leq v_i\}}.$$

6. If $I = 1$, the trajectory $(x, Y_{t_1} = y_1, \ldots, Y_{t_k} = y_k, Y_\Delta = y)$ is accepted and Y_Δ is an exact draw of X_Δ. Otherwise, restart from step 1.

Two approaches are possible if one wants to simulate a process up to an arbitrary time T: either set $\Delta = T$ or set $\Delta = T/N$ and iterate the algorithm N times. As in [56], we suggest keeping only the last value of X_Δ and simulating the next one $X_{2\Delta}$ (conditionally on $X_\Delta = y$) up to the final time T. The advantage of this approach is that we get a path of the process on a regular grid, which makes this path compatible with the other schemes presented in this book. The EA algorithm is implemented in Listing 2.10, and the relevant changes to the `sde.sim` function are given below.

```
sde.sim <- function (t0 = 0, T = 1, X0 = 1, N = 100, delta, drift, sigma,
    drift.x, sigma.x, drift.xx, sigma.xx, drift.t, method = c("euler",
        "milstein", "KPS", "milstein2", "cdist","ozaki","shoji","EA"),
    alpha = 0.5, eta = 0.5, pred.corr = T, rcdist = NULL, theta = NULL,
    model = c("CIR", "VAS", "OU", "BS"),
    k1, k2, phi, max.psi = 1000, rh, A){

# (...)

    if(method == "ozaki" || method == "shoji" || method == "EA")
        needs.dx <- TRUE

# (...)

     if (method == "EA")
      X <- sde.sim.ea(X0, t0, Dt, N, d1, d1.x, k1, k2, phi, max.psi, rh, A)

# (...)
}
```

Remarks on the method

The hypothesis of boundedness of ϕ can be too restrictive, and some relaxation is possible as described in [27] with some modifications of the algorithm (known as EA2 and EA3 schemes, in contrast with the EA1 algorithm of Listing 2.10).

It is important to mention that the probability of the event $I = 1$ in the algorithm above, which is the probability of accepting a simulated path, exponentially decreases to zero as $\Delta \to 0$ and is at least e^{-1}, which justifies using this rejection algorithm from the point of view of efficiency.

One more important remark is that approximation schemes usually affect the statistical procedures in Monte Carlo experiments. On the contrary, the EA method does not influence the estimators (see [27]).

```
"sde.sim.ea" <-
function(X0, t0, Dt, N, d1, d1.x, k1, k2, phi, psi, max.psi, rh, A){

   psi <- function(x) 0.5*d1(1,x)^2 + 0.5*d1.x(1,x)

   if(missing(k1)){
    cat("k1 missing, trying numerical minimization...")
    k1 <- optimize(psi, c(0, max.psi))$obj
    cat(sprintf("(k1=%5.3f)\n",k1))
   }
   if(missing(k2)){
    cat("k2 missing, trying numerical maximization...")
    k2 <- optimize(psi, c(0, max.psi),max=TRUE)$obj
    cat(sprintf("(k2=%5.3f)\n",k2))
  }

  if(missing(phi))
    phi <- function(x) 0.5*d1(1,x) + 0.5*d1.x(1,x) - k1
  else
   phi <- function(x) eval(phi)

  M <- k2-k1
  if(M==0)
   stop("`k1' = `k2' probably due to numerical maximization")

  if(Dt>1/M)
    stop(sprintf("discretization step greater than 1/(k2_k1)"))

   if(missing(A))
     A <- function(x) integrate(d1, 0, x)

  if(missing(rh)){
   rh <- function(){
    h <- function(x) exp(A(x) - x^2/(2*Dt))
    f <- function(x) h(x)/dnorm(x,sqrt(Dt))
    maxF <- optimize(f,c(-3*Dt, 3*Dt),max=TRUE)$obj
    while(1){
     y <- rnorm(1)
     if( runif(1) < f(y)/maxF )
      return(y)
   }
  }
 }

  x0 <- X0
  X <- numeric(N)
  X[1] <- X0
  rej <- 0
  j <- 1
  while(j <= N){
   y <- x0+rh()
   k <- rpois(1,M*Dt)
   if(k>0){
    t <- runif(k)*Dt
    v <- runif(k)*M
    idx <- order(t)
    t <- c(0, t[idx], Dt)
    v <- v[idx]

    DT <- t[2:(k+2)] - t[1:(k+1)]
    W <- c(0,cumsum(sqrt(DT) * rnorm(k+1)))
    Y <- x0 + W -(W[k+2] -y+x0)*t/Dt
```

```
    if( prod(phi(Y[2:(k+1)]) <= v) == 1){
     j <- j+1
     x0 <- Y[k+2]
     X[j] <-  Y[k+2]
    } else {
     rej <- rej +1
   }
  }
 }
 cat(sprintf("rejection rate: %5.3f\n",rej/(N+rej)))
 X
}
```

Listing 2.10. R code for the EA1 simulation algorithm.

Periodic drift example (SINE process)

Consider the following example from [27]. We have a process satisfying the stochastic differential equation

$$\mathrm{d}\xi_t = \sin(\xi_t)\mathrm{d}t + \mathrm{d}W_t, \quad \xi_0 = 0\,. \tag{2.17}$$

The drift $b(x) = \sin(x)$ satisfies the usual hypotheses, and is differentiable and such that

$$-\frac{1}{2} = k_1 \leq \frac{1}{2}b(u)^2 + \frac{1}{2}b_x(u) = \frac{1}{2}\sin(u)^2 + \frac{1}{2}\cos(u) \leq k_2 = \frac{5}{8};$$

hence $M = k_2 - k_1 = 9/8$ and $A(u) = 1 - \cos(u)$. The following code simulates an exact path of the SINE process.

```
> # ex 2.15.R
> set.seed(123)
> d <- expression(sin(x))
> d.x <- expression(cos(x))
> A <- function(x) 1-cos(x)
> sde.sim(method="EA", delta=1/20, X0=0, N=500, drift=d,
+       drift.x = d.x, A=A) -> X
k1 missing, trying numerical minimization...(k1=-0.500)
k2 missing, trying numerical maximization...(k2=0.625)
rejection rate: 0.215
> plot(X, main="Periodic drift")
```

A more complicated example: the hyperbolic process

Another example taken from [56] is the following. Consider the modified Cox-Ingersoll-Ross process (see Section 1.13.6) solution to

$$\mathrm{d}X_t = -\theta_1 X_t \mathrm{d}t + \theta_2\sqrt{1 + X_t^2}\mathrm{d}W_t$$

with $\theta_1 + \theta_2^2/2 > 0$. Using the Lamperti transform, we get the new process $Y_t = F(X_t)$, which satisfies the stochastic differential equation

$$\mathrm{d}Y_t = -(\theta_1/\theta_2 + \theta_2/2)\tanh(\theta_2 Y_t)\mathrm{d}t + \mathrm{d}W_t$$

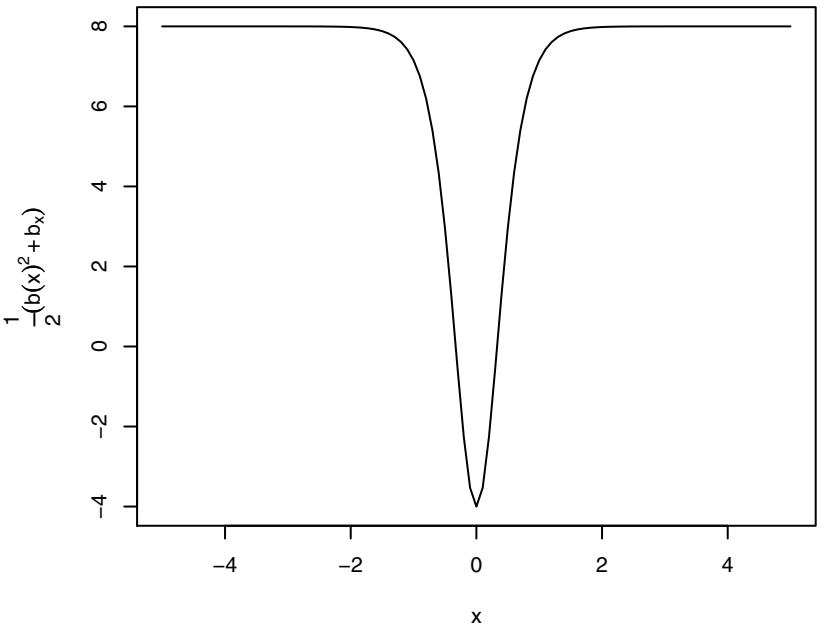

Fig. 2.4. Graph of the curve $\frac{1}{2}(b(x)^2 + b_x(x))$.

with $Y_0 = F(X_0)$ (see Section 1.13.6 for details). Choose $\theta_1 = 6$ and $\theta_2 = 2$. In this case,

$$\frac{1}{2}(b^2(x) + b_x(x)) = \frac{-8 + 16(\sinh(2x))^2}{2(\cosh(2x))^2},$$

and it is easy to show that

$$-4 = k_1 \leq \frac{1}{2}(b(x)^2 + b_x(x)) \leq k_2 = 8$$

(see also Figure 2.4) and

$$A(x) = \int_0^x -4\tanh(2u)\mathrm{d}u = -2\log(\cosh(2x)).$$

Hence $M = k_2 - k_1 = 12$, $0 \leq \phi(x) \leq 1/M$, and the constant $K = \int_{\mathbb{R}} e^{A(x) - \frac{x^2}{2T}}$ can be found numerically.

Once we have simulated a path of Y_t using the exact algorithm, we can obtain a trajectory of X_t using the inverse of F; i.e., $X_t = F^{-1}(Y_t) = \sinh(\theta_2 Y_t)$. The next R code performs the simulation, and both the original and the transformed paths of the process are shown in Figure 2.5.

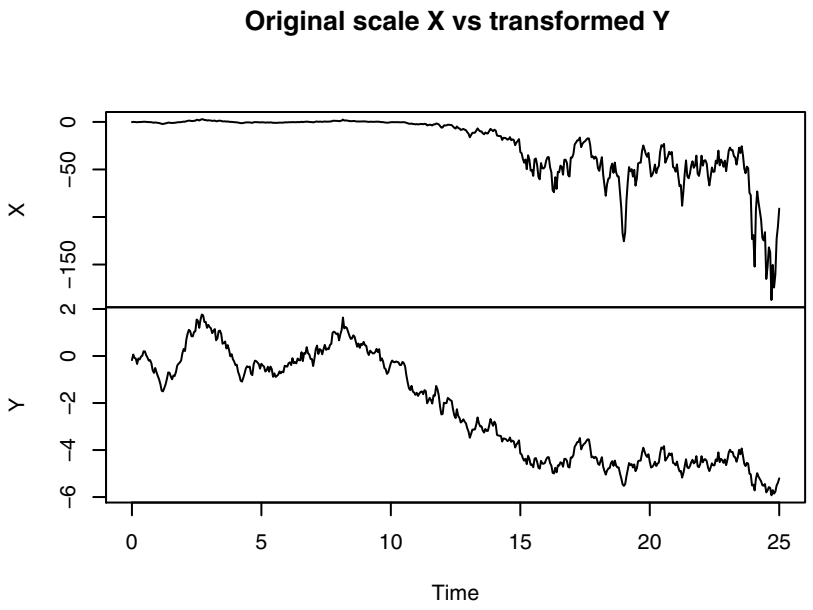

Fig. 2.5. Simulation of the process Y_t solution of (1.54) and its transformed version $X_t = \sinh(2Y_t)$ using the EA1 algorithm.

```
> # ex 2.16.R
> set.seed(123)
> d <- expression(-4*tanh(2*x))
> d.x <- expression(-(4 * (2/cosh(2 * x)^2)))
> A <- function(x) -(0.5+6/4)*log(cosh(2*x))
> X0 <- rt(1, df=4)/2
> F <- function(x) log(x + sqrt(1+x^2))/2
> Y0 <- F(X0)
> sde.sim(method="EA", delta=1/20, X0=Y0, N=500, drift=d,
+       drift.x=d.x, A=A, k1=-4,k2=8) -> Y
rejection rate: 0.474
> X <- sinh(Y)
> ts(cbind(X,Y),start=0,delta=1/20) -> XY
> plot(XY,main="Original scale X vs transformed Y")
```

The Cox-Ingersoll-Ross and Ornstein-Uhlenbeck processes and EA algorithm

Consider the Cox-Ingersoll-Ross process

$$\mathrm{d}X_t = (\theta_1 - \theta_2 X_t)\mathrm{d}t + \theta_3\sqrt{X_t}\mathrm{d}W_t, \qquad X_0 = 10\,,$$

for which the Lamperti transform

$$F(x) = \int_0^x \frac{1}{\theta_3\sqrt{u}}\mathrm{d}u = \frac{2\sqrt{x}}{\theta_3}$$

gives $Y_t = F(X_t)$, satisfying

$$\mathrm{d}Y_t = \frac{4\theta_1 - \theta_3^2}{2\theta_3^2 Y_t}\mathrm{d}t - \frac{\theta_2}{2}Y_t\mathrm{d}t + \mathrm{d}W_t\,.$$

The transformed drift $b(x) = \frac{4\theta_1-\theta_3^2}{2\theta_3^2 x} - \frac{\theta_2}{2}x$ is not bounded in zero from above or below and hence the EA algorithm is not applicable to this process. The same situation occurs for the Ornstein-Uhlenbeck process of equation (1.39).

2.13 Simulation of diffusion bridges

As we will see in the next chapter, exact likelihood inference for discretely observed diffusion processes is not always possible because the likelihood is not available in many cases. Section 3.3.2 describes the *simulated likelihood method*, which consists in estimating the transition density between two consecutive observations using the Monte Carlo approach. In this situation, the ability to simulate paths between two observations is essential. To this end an MCMC-algorithm was proposed in [196], and the exact method of Section 2.12 can also be applied. A new simple method that applies to ergodic diffusion processes has recently been introduced in [37]. This method relies on the *time-reversibility* property of the ergodic diffusion process and essentially consists in the simulation of two paths of a diffusion process, one moving forward in time and another one moving backward in time. If the two trajectories intersect, then the combined path is a realization of the bridge. Let (l, r) with $-\infty \leq l \leq r \leq +\infty$ be the state space of the diffusion process X solution to

$$\mathrm{d}X_t = b(X_t)\mathrm{d}t + \sigma(X_t)\mathrm{d}W_t$$

and take a and b as two points in the state space of X. A solution of the previous equation in the interval $[t_1, t_2]$ such that $X_{t_1} = a$ and $X_{t_2} = b$ is called a (t_1, x_1, t_2, x_2)-diffusion bridge. Time reversibility of an ergodic diffusion is assured by a mild set of conditions (see, e.g., [135]).

Let $m(x)$ and $s(x)$ be, respectively, the speed measure (1.22) and the scale measure (1.21) of the diffusion X. Under Assumption 1.5, we know that the diffusion X is also ergodic with invariant density proportional to the speed measure up to a normalizing constant. Our interest is in the simulation of a $(0, a, 1, b)$-diffusion bridge. Let W^1 and W^2 be two independent Wiener processes and define X^1 and X^2 as solutions to

$$\mathrm{d}X_t^i = b(X_t^i)\mathrm{d}t + \sigma(X_t^i)\mathrm{d}W_t^i$$

with $X_0^1 = a$ and $X_0^2 = b$.

Fact 2.1 (Theorem 1 in [37]) *Let* $\tau = \inf\{0 \leq t \leq 1 | X_t^1 = X_{1-t}^2\}$, *where the* inf *over the empty set is taken to be* ∞. *Define*

$$Z_t = \begin{cases} X^1_t & \text{if} \quad 0 \le t \le \tau, \\ X^2_{1-t} & \text{if} \quad \tau < t \le 1. \end{cases}$$

Then, the distribution of $\{Z_t\}_{0\le t\le 1}$ *conditional on the event* $\{\tau \le 1\}$ *is equal to the conditional distribution of* $\{X_t\}_{0\le t\le 1}$ *given* $X_0 = a$ *and* $X_1 = b$*; i.e.,* Z *is a* $(0, a, 1, b)$*-diffusion bridge.*

2.13.1 The algorithm

We now assume that our interest is in the simulation of a $(0, a, \Delta, b)$-diffusion bridge; i.e., a bridge on the generic interval $[0, \Delta]$. The algorithm consists in simulating two independent diffusion processes X^1 and X^2 using one of the previous methods (e.g., the Euler or Milstein scheme) on the time interval $[0, \Delta]$ with discretization step $\delta = \Delta/N$ and applying a rejection sampling procedure. Let $Y^1_{i\delta}$ and $Y^2_{i\delta}$, $i = 0, 1, \ldots, N$, be independent simulations of X^1 and X^2. If either $Y^1_{i\delta} \ge Y^2_{(N-i)\delta}$ and $Y^1_{(i+1)\delta} \le Y^2_{(N-(i+1))\delta}$ or $Y^1_{i\delta} \le Y^2_{(N-i)\delta}$ and $Y^1_{(i+1)\delta} \ge Y^2_{(N-(i+1))\delta}$, a crossing has been realized. Hence, let $\nu = \min\{i \in (1, \ldots, N) | Y^1_{i\delta} \le Y^2_{(N-i)\delta}\}$ if $Y^1_0 \ge Y^2_\Delta$ and $\nu = \min\{i \in (1, \ldots, N) | Y^1_{i\delta} \ge Y^2_{(N-i)\delta}\}$ if $Y^1_0 \le Y^2_\Delta$, and define

$$B_{i\delta} = \begin{cases} Y^1_{i\delta} & \text{for} \quad i = 0, 1, \ldots, \nu - 1, \\ Y^2_{(N-i)\delta} & \text{for} \quad i = \nu, \ldots, N. \end{cases}$$

Then B is a simulation of a $(0, a, \Delta, b)$-diffusion bridge. If no crossing happened, start again by simulating $Y^1_{i\delta}$ and $Y^2_{i\delta}$ and iterate until a crossing of the two trajectories is realized.

The nice feature of this method is that this algorithm produces trajectories with the same order (weak or strong) of approximation as the method used to simulate Y^1 and Y^2.

As for the exact algorithm, it is also interesting to evaluate the rejection rate of the method. The rejection probability (i.e., the probability of no crossings) depends on the drift and diffusion coefficients as well as the points a and b and the length of the time interval Δ. Simulation experiments (see [37]) show that if a and b are not too distant and Δ is relatively small, which usually occurs in inference for discretely observed diffusion processes, the rejection rate is acceptable. Usually, when the exact algorithm is feasible, it is more efficient. The nice property of the present approach is that the algorithm is relatively simple and works for the class of generic time-homogeneous ergodic diffusion processes.[7]

The following code illustrates how to implement the algorithm above. Although a `DBridge` function exists in the `sde` package which we discuss later,

[7] We recall again that version 3 of the exact sampling algorithm exists, which does not require bounds on the coefficients but is mathematically more involved [28].

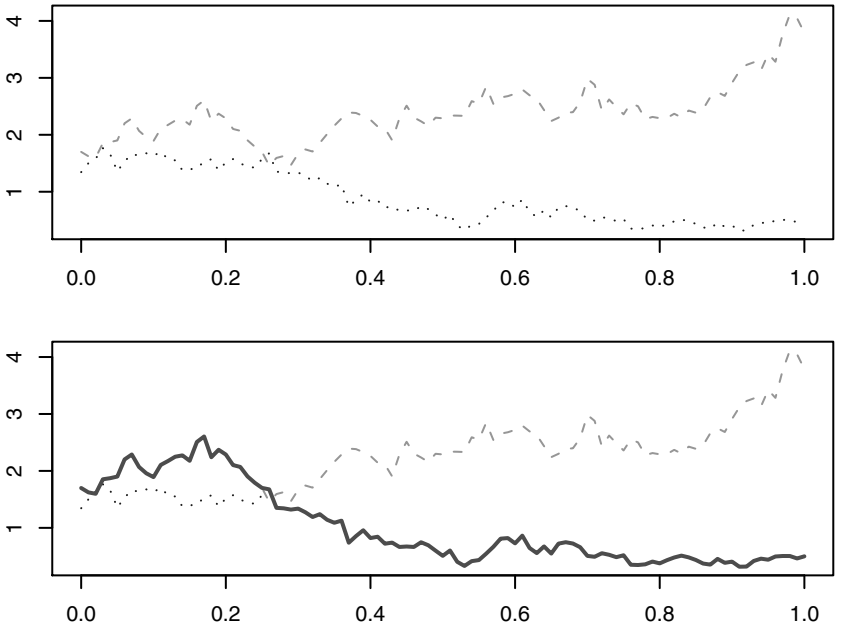

Fig. 2.6. A simulated path of a diffusion bridge (continuous line). The bridge is obtained by merging two paths of diffusion processes (dotted and dashed lines) at the first crossing.

we present the algorithm in an intuitive version in what follows. The following code creates two trajectories, `Y1` and `Y2`, the first starting at $a = 1.7$ and the second starting at $b = 0.5$.

```
> # ex2.17.R
> drift <- expression((3-x))
> sigma <- expression(1.2*sqrt(x))
> a <- 1.7
> b <- 0.5
> set.seed(123)
> Y1 <- sde.sim(X0=a, drift=drift, sigma=sigma, T=1, delta=0.01)
> Y2 <- sde.sim(X0=b, drift=drift, sigma=sigma, T=1, delta=0.01)
> Y3 <- ts(rev(Y2), start=start(Y2), end=end(Y2),deltat=deltat(Y2))
```

The second trajectory is then time-reversed into `Y3`.

```
> id1 <- Inf
> if(Y1[1]>=Y3[1]){
+  if(!all(Y1>Y3))
+   min(which(Y1 <= Y3))-1 -> id1
+ } else {
+  if(!all(Y1<Y3))
+  min(which(Y1 >= Y3))-1 -> id1
+ }
> if(id1==0 || id1==length(Y1)) id1 <- Inf
```

The code then calculates the index `id1` of the first crossing time of the two trajectories and merges `Y1` and `Y3` appropriately if this time (the time of crossing) is finite.

```
> par(mfrow=c(2,1))
> plot(Y1, ylim=c(min(Y1,Y2), max(Y1,Y2)),col="green",lty=2)
> lines(Y3,col="blue",lty=3)
>
> if(id1==Inf ){
+  cat("no crossing")
+ } else {
+  plot(Y1, ylim=c(min(Y1,Y2), max(Y1,Y2)),col="green",lty=2)
+ lines(Y3,col="blue",lty=3)
+ B <- ts(c(Y1[1:id1], Y3[-(1:id1)]), start=start(Y1),end=end(Y1),
+       frequency=frequency(Y1))
+ lines(B,col="red",lwd=2)
+ }
```

Figure 2.6 shows the trajectory of `Y1` and `Y3` (top) and the simulated path of the diffusion bridge (bottom). The following code uses the `DBridge` function as an interface to the algorithm above. The function has the following interface similar to `BBridge` for the simulation of the Brownian bridge:

```
DBridge(x = 0, y = 0, t0 = 0, T = 1, delta, drift, sigma, ...)
```

The variable arguments `...` are passed directly to the `sde.sim` function, which is called internally. This allows selection of any simulation scheme for the diffusion, the default being the default of the `sde.sim` function. The code for `DBridge` simulates a (t_0, x, T, y)-diffusion bridge. The next code provides an example of how it is used and the output is given in Figure 2.7.

```
> # ex2.17.R (cont.)
> d <- expression((3-x))
> s <- expression(1.2*sqrt(x))
> par(mar=c(3,3,1,1))
> par(mfrow=c(2,1))
> set.seed(123)
> X <- DBridge(x=1.7,y=0.5, delta=0.01, drift=d, sigma=s)
> plot(X)
> X <- DBridge(x=1,y=5, delta=0.01, drift=d, sigma=s)

no crossing, trying again...
> plot(X)
```

2.14 Numerical considerations about the Euler scheme

Consider for example the geometric Brownian motion X_t in (1.5). If $X_0 > 0$, this process is always positive, being an exponential functional of the Brownian motion. The application of the Euler approximation can lead to unexpected results. Indeed, the Euler scheme for X_t reads as

$$Y_{i+1} = Y_i(1 + \theta_1 \cdot \Delta t + \theta_2 \sqrt{\Delta t} Z),$$

and if Δt is too small, it can happen in one or more simulations that a pseudo random number Z is drawn from the Gaussian distribution such that

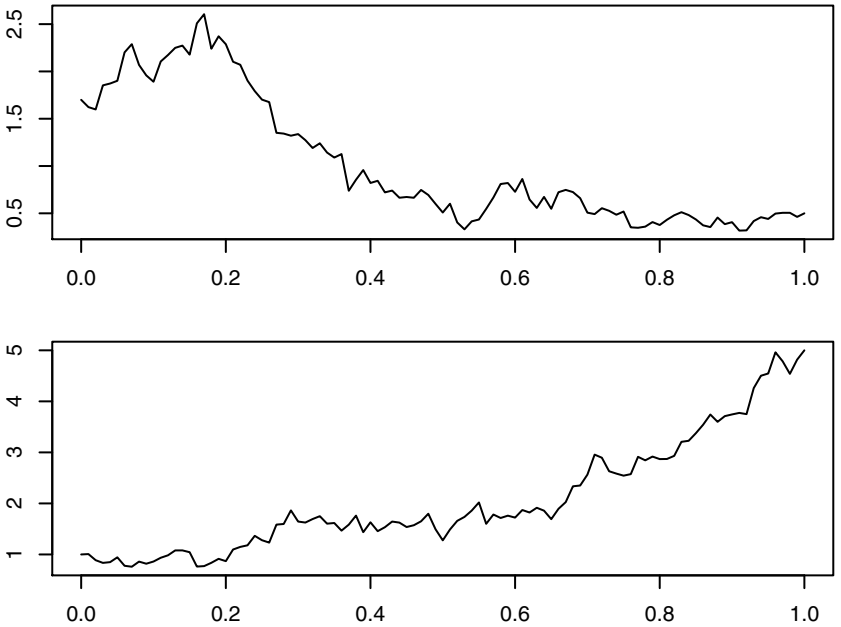

Fig. 2.7. Two simulated paths of diffusion bridges (t_1, x_1, t_2, x_2) using the `DBridge` function for the stochastic differential equation $\mathrm{d}X_t = (3 - X_t)\mathrm{d}t + 1.2\sqrt{X_t}\mathrm{d}W_t$: $(0, 1.7, 1, 0.5)$ (top) and $(0, 1, 1, 5)$ (bottom).

$$Z < -\frac{1 + \theta_1 \Delta t}{\theta_2 \sqrt{\Delta t}}$$

and therefore Y_{i+1} takes negative values. In this case, this is not the same phenomenon of absorption as in the CEV process of Section 2.5 but just a matter of the approximation method used. In fact, the Euler scheme is guaranteed to converge to the mathematical description of the geometric Brownian motion, but simulation by simulation we cannot expect this result to happen every time. An empirical proof of the fact that this is not an absorption phenomenon is that false "absorption" does not occur on the transformed process $\log X_t$.

2.15 Variance reduction techniques

We now adapt the general concepts in Section 1.4 to the framework of stochastic differential equations. In this framework, interest is in the evaluation of the expected value of some functional ψ of the trajectory of the process solution of some stochastic differential equation. Let us denote by Z this expected value,

$$Z = \psi(\{X_t, 0 \le t \le T\}) = \psi(X)\,,$$

with X the solution to the stochastic differential equation

$$\mathrm{d}X_t = b(X_t)\mathrm{d}t + \sigma(X_t)\mathrm{d}W_t, \quad X_0 = x\,. \tag{2.18}$$

The Monte Carlo estimator of $\mathbb{E}Z$ is built as follows. Let $Y_i^j = Y^j(t_i)$, $j = 1, \ldots, N$, $i = 1, \ldots, n$, be the value of the approximating process Y^j at time t_i for the jth simulated path of the process X. Then, the estimator has the form

$$\hat{\mu}_N = \frac{1}{N}\sum_{j=1}^{N} \psi(Y^j)\,.$$

For example, if we are interested in the expected value of X_T (i.e., $\mathbb{E}Z = \mathbb{E}\psi(X) = \mathbb{E}X_T$), the Monte Carlo estimator reads as

$$\hat{\mu}_N = \frac{1}{N}\sum_{j=1}^{N} Y_n^j.$$

From the general Monte Carlo results, we know that $\hat{\mu}_N$ is unbiased and has a variance equal to $\mathrm{Var}\psi(Y_n)/N$, and we also know that the length of the confidence intervals shrinks to 0 at speed $N^{-\frac{1}{2}}$. Once again, if the process itself has large a variance, the confidence interval might be too big to be used to assess the quality of the estimator and hence the need for variance reduction techniques.

2.15.1 Control variables

From Section 1.4.2, we know that in order to reduce the variance of $\mathbb{E}Z$, one possibility is to apply the control variable technique. In particular, we need to rewrite $\mathbb{E}Z$ as $\mathbb{E}(Z-Y)+\mathbb{E}(Y)$, for which $\mathbb{E}Y$ can be calculated explicitly. For obvious reasons, this is easy to implement when $\mathbb{E}Y = 0$. When dealing with stochastic differential equations, a good hint is clearly to build such a random variable Y on top of Brownian motion. Indeed, we know that if $(H(t), 0 \le t \le T)$ is an Itô integrable process, then

$$\mathbb{E}\left(\int_0^T H(s)\mathrm{d}W(s)\right) = 0\,.$$

Thus, in principle, the role of Y might be taken by a properly chosen stochastic integral. In fact, there is a general result that allows us to rewrite any square-integrable random variable, adapted to the natural filtration of the Brownian motion, in terms of its expected value and a stochastic integral of some process H. This theorem, called the *predicted representation* theorem (see [130] or [193]), is rather general, but unfortunately the explicit formula for the process

H is quite difficult to find in general. In [167], one formula based on Malliavin calculus is provided, but it is hard to implement. In general, each case study might provide different ways to obtain control variables. A result descending from the Feynman-Kac theorem relates the construction of such a process to the solution of a partial differential equation. Theorem 5.4.2 in [156] assumes that $u(t,x)$ is a function of class $C^{1,2}$ with bounded derivatives in x and solution of

$$\begin{cases} \left(\frac{\partial u}{\partial t} + \mathcal{L}u\right)(t,x) = f(x) \\ u(T,x) = g(x), \end{cases}$$

where $\mathcal{L}$ is the infinitesimal generator (see (1.28)) of the diffusion process solution of (2.18). Setting

$$Z = g(X_T) - \int_0^T f(X_s)\mathrm{d}s$$

and

$$Y = \int_0^T \frac{\partial u}{\partial x}(s, X_s)\sigma(X_s)\mathrm{d}W_s,$$

then

$$\mathbb{E}Z = Z - Y.$$

This theorem is the key to finding the control variable that are interests us. But still the expression of Y involves partial derivatives of the function u, and hence in practice the approach is to find an approximation $\bar{u}$ of u that is simple to handle and put it in the expression of Y. There are a lot of heuristics behind the application of this method in concrete cases, and we show one from [134]. Suppose we want to calculate the average price of a call option,

$$\mathbb{E}Z = \mathbb{E}\left(e^{-rT}\left(\frac{1}{T}\int_0^T S_u\mathrm{d}u - K\right)_+\right), \tag{2.19}$$

where S is the geometric Brownian motion in (1.5). If $\sigma \simeq 0.5$, $r \simeq 1$, and $T \simeq 1$, then the integral

$$\frac{1}{T}\int_0^T S_u\mathrm{d}u$$

is "close" to

$$\exp\left\{\frac{1}{T}\int_0^T \log(S_u)\mathrm{d}u\right\}.$$

Hence, we can set

$$Y = e^{-rT}\left(e^Z - K\right)_+$$

and

$$Z = \frac{1}{T}\int_0^T \log(S(s))\mathrm{d}s$$

and use Y as the control variable. Moreover, Z is easy to simulate, as it is essentially a Gaussian random variable. This approach successfully reduces the variance of the Monte Carlo estimator. (Details on how (2.19) is related to the previous theorem can be found in Section 5.2.6 of [156].) Here we don't give R code for this case (although it is quite easy to implement). as this example is quite peculiar.

2.16 Summary of the function sde.sim

The package `sde` further generalizes the function `sde.sim`. In particular, it is possible to generate `M` independent trajectories of the same process with one single call of the function by just specifying a value for `M` (which is 1 by default). For `M>=2`, the function `sde.sim` returns an object of class `mts` "multi-dimensional time series." This is quite convenient in order to avoid loops in the case of Monte Carlo replications. The function `sde.sim` has a rather flexible interface, which matches the following definition:

```
sde.sim(t0 = 0, T = 1, X0 = 1, N = 100, delta, drift, sigma,
   drift.x, sigma.x, drift.xx, sigma.xx, drift.t,
   method = c("euler", "milstein", "KPS", "milstein2", "cdist",
   "ozaki","shoji","EA"), alpha = 0.5, eta = 0.5, pred.corr = T,
   rcdist = NULL, theta = NULL,
   model = c("CIR", "VAS", "OU", "BS"),
   k1, k2, phi, max.psi = 1000, rh, A, M=1)
```

A complete description of all of the parameters can be found on the manual page of the `sde` package in the Appendix B of this book. Here we mention that this interface allows us to simulate a stochastic differential equation by specifying the drift and the diffusion coefficient and a simulation scheme, or by specifying a model among them that admits well-known distributional results, by specifying a conditional distribution density. The following code shows an example of such flexibility.

```
> # ex2.18.R
> # Ornstein-Uhlenbeck process
> set.seed(123)
> d <- expression(-5 * x)
> s <- expression(3.5)
> sde.sim(X0=10,drift=d, sigma=s) -> X
> plot(X,main="Ornstein-Uhlenbeck")
>
> # Multiple trajectories of the O-U process
> set.seed(123)
> sde.sim(X0=10,drift=d, sigma=s, M=3) -> X
> plot(X,main="Multiple trajectories of O-U")
>
> # Cox-Ingersoll-Ross process
> # dXt = (6-3*Xt)*dt + 2*sqrt(Xt)*dWt
> set.seed(123)
> d <- expression( 6-3*x )
> s <- expression( 2*sqrt(x) )
> sde.sim(X0=10,drift=d, sigma=s) -> X
```

```
> plot(X,main="Cox-Ingersoll-Ross")
>
> # Cox-Ingersoll-Ross using the conditional distribution "rcCIR"
>
> set.seed(123)
> sde.sim(X0=10, theta=c(6, 3, 2), rcdist=rcCIR, method="cdist") -> X
> plot(X, main="Cox-Ingersoll-Ross")
>
> set.seed(123)
> sde.sim(X0=10, theta=c(6, 3, 2), model="CIR") -> X
> plot(X, main="Cox-Ingersoll-Ross")
```

2.17 Tips and tricks on simulation

We conclude briefly with some general remarks and things to remember before starting Monte Carlo analysis based on simulated paths of the processes seen so far. In general, it is recommended to apply the Lamperti transform[8] to eliminate the dependency of the diffusion coefficient from the state of the process during simulation. We have seen that many methods rely on this transformation without loss of generality (e.g., Ozaki, Shoji, Exact Algorithm). Also, the Euler and Milstein methods may benefit from this preliminary transformation.

If the conditional distribution of the process is known, which is rarely the case, then a simulation method based on this should be used. For example, the simulation of the Cox-Ingersoll-Ross process should be done in this way because simulations based on the discretization of the corresponding stochastic differential equation may lead to unwanted results such as negative values of the process.

In principle, if time is not a major constraint and the model satisfies the right conditions on the drift (which we saw are not satisfied by the Cox-Ingersoll-Ross process), then the exact algorithm must be used. It is worth mentioning that code more efficient than what we present here can be written so time efficiency of EA is not necessarily a concern. This will probably be done in the next version of the `sde` package, and hence this comment only applies to the current implementation.

We also mention that when there is no need to simulate the path of the process on a regular grid of points $t_i = i\Delta/T$, $i = 0, 1, \ldots, N$, $N\Delta = T$ like we did, then the EA algorithm is even faster. In fact, in our approach, we generate different points in between the time instants t_i and t_{i+1} but then keep just the last one and iterate this simulation N times. On the contrary, the algorithm can be used to simulate the path up to time T. In this case, the algorithm generates a random grid of points and simulated values of the process, and then Brownian bridges can be used between the points of the random grid. Of course, the way we use the EA algorithm avoids any dependency of the simulation scheme from the estimation part, as we will note in the following chapters.

[8] Of course, when the transform is well-defined and can be obtained in explicit analytic form and not by numerical integration.

When the interest is in the simulation of diffusion bridges, the algorithm presented in Section 2.13 is a good candidate.

The Ozaki and Shoji-Ozaki methods can be good ways of simulating a path when other methods do not apply (which is the case for unbounded nonlinear drift functions).

If the grid of points is relatively small, we have seen that most discretization methods perform equally well but the Euler method can still be unstable in some particular situations: see the counterexample on the geometric Brownian motion process in Section 2.5. Then a higher order of the approximation is always welcome.

Antithetic sampling and variance reduction techniques might be used when functionals of the processes are of interest. Unfortunately, the control variable approach is always an ad hoc art.

http://www.springer.com/978-0-387-75838-1

Simulation and Inference for Stochastic Differential Equations
With R Examples
Iacus, S.M.
2008, XVIII, 285 p., Hardcover
ISBN: 978-0-387-75838-1