Topics in the Numerical Simulation of Pathwise Solutions to Stochastic Differential Equations

Jessica Gabrielle Gaines

Presented for the Degree of Doctor of Philosophy in Mathematics
University of Edinburgh

1995

Abstract

This work contains several developments in the area of numerical simulation of pathwise solutions to stochastic differential equations (SDE's). In the first chapter we define and motivate pathwise solutions and give a brief survey of numerical methods for approximating them.

The main key to enlarging the scope of numerical methods for SDE's is a good representation of Brownian paths. A binary tree structure is an essential tool in Chapter Two, which presents a general method for solution of SDE's using variable time steps.

In the case of a general SDE, improvement of the order of convergence compared with standard methods, demands generation of the Lévy area integrals. Chapter Three presents a method of random generation of the Lévy area for a Brownian path in $I\!R^2$. The method is based on Marsaglia's rectangle-wedge-tail method for fast generation of normally distributed deviates.

Since the solution of an SDE generally depends on an infinite sequence of iterated integrals of the driving noise, it makes sense to examine these integrals and the algebraic relations between them. In Chapter Four, it is shown how known facts about shuffle algebras can be used to get a better understanding of stochastic iterated integrals of Ito and Stratonovich type and obtain practical algebraic bases for these two sets. We use the algebra to calculate moments of stochastic integrals, needed when calculating moments of errors during numerical solution of SDE's.

The work on the generation of area integrals, described in Chapter Three, gives rise to general questions about the generation of random deviates. some of which are addressed in the last two chapters. In Chapter Five, we present a polynomial-time algorithm for finding the partition, into rectangles or triangles, of certain types of region in $I\!R^2$, that has the lowest entropy. When the area under a density is divided into simple pieces, the entropy of the partition provides a lower bound on the time taken to choose one piece randomly with probability proportionate to the area.

Chapter Six explores the idea that Marsaglia's method for generation of random deviates could lead to efficient "black box" algorithms. The main criticism of the rectangle-wedge-tail approach is that for each density, tables of constants have to be calculated, leading to hard work for the programmer and lengthy and complicated code. Our idea is to write a program that, given any density from a wide class, can generate its own tables of constants dynamically, while simultaneously generating random deviates.

Declaration

This thesis is submitted for the degree of Doctor of Philosophy at the University of Edinburgh. None of the material has previously been submitted for any other degree or qualification. The thesis has been composed by myself and the work is my own except where I have explicitly stated otherwise.

Aknowledgements

I would like to thank my supervisor, T.J. Lyons, for his time, encouragement and helpful ideas, to which much of the work in this thesis owes its existence.

I would also like to thank my husband, Sandy Davie, for his support, both moral and mathematical.

Table of Contents

Chapter 1

Numerical Approximation of Pathwise Solutions to SDE's

1.1 Pathwise solutions: definition and motivation

We are interested in pathwise (or strong) solutions to a stochastic differential equation (SDE) where the driving noise $w(t)$ is a standard d-dimensional Brownian motion. In Itô form, such an equation is given by

$$\begin{cases} dx_t &= g_0(x,t)dt + \sum_{i=1}^{d} g_i(x,t)\, dw_t^i \\ x(0) &= x_0 \end{cases} \tag{1.1}$$

with $x \in I\!R^n$ and $g_i (i = 0, \ldots, d) : I\!R^n \to I\!R^n$. The equation may also be written in Stratonovich form as

$$\begin{cases} dx_t &= f_0(x,t)dt + \sum_{i=1}^{d} f_i(x,t) \circ dw_t^i \\ x(0) &= x_0 \end{cases} \tag{1.2}$$

where

$$f_0^i(x,t) = g_0^i(x,t) - \frac{1}{2}\sum_{j=1}^{d}\sum_{k=1}^{n} \frac{\partial g_j^i}{\partial x^k} g_j^k$$

and $f_i(x,t) = g_i(x,t), \forall i = 1, \ldots, d$.

Superscripts, such as x^k, denote components of vectors. We will presume that the vector-valued functions g_i, $(i = 0, \ldots, d)$ are sufficiently smooth to guarantee existence and uniqueness of solutions to (1.1) (see Section 1.3 for details). We are mainly concerned with cases where the initial value, x_0, is deterministic, but it could also be random as long as it is independent of $(w(t))$ and observable.

When we choose or are given **one** particular Brownian path $w(t)$, $0 \leq t \leq T$, then by pathwise solution we mean the path $x(t)$, $0 \leq t \leq T$, unique solution of (1.1) for that realisation of $w(t)$. This definition is in opposition to that of a weak solution, which is the expected value of the solution or of a function of the solution over all possible Brownian paths. We will only be concerned with numerical methods used to obtain approximations to pathwise solutions.

There are several reasons why people should wish to obtain pathwise solutions to SDE's. Firstly, they may wish to examine the dependence of the solution on the initial condition or on the value of one or more parameters that appear in the functions defining the SDE. In this case one Brownian path is generated and then used repeatedly to obtain a set of solutions with varying initial conditions or parameter values. Secondly, there are situations, such as filtering problems, where the Brownian path or a noisy signal supposed to be a function of a Brownian path, is not generated by a computer, but obtained as a reading of a measuring instrument. In this case, the recorded signal is used to solve an SDE just once, with the aim of getting more information about the process generating the signal. It is the first kind of application that will interest us the most. Although the second kind is very important, the fact that the driving noise is not generated 'at will' by the person performing the numerical solution puts limitations on the methods available.

1.2 Notation

We will assume that the probability space underlying the SDE's (1.1) and (1.2) is the canonical filtered probability space $(\Omega, \mathcal{F}_T, (\mathcal{F}_t), P)$, where $\Omega = C([0,T], \mathbb{R}^d)$ is the set of continuous functions from $[0,T]$ to $\mathbb{R}^d$, $\mathcal{F}_t$ is the $\sigma-$field generated by $(w(s), 0 \leq s \leq t)$ and P is $d-$dimensional Wiener measure.

Definition 1.2.1 *The n-tuple*

$$J = (j_1, j_2, \ldots, j_n) \quad \textit{with } j_i \in \{0, 1, \ldots, d\}; \quad \forall i = 1, \ldots, n$$

is called a multi-index. *The length of J, written $|J|$, is the number of indices in J and the order of J, denoted $||J||$, is equal to the number of non-zero indices plus twice the number of zero indices.*

Let $w(t) = (w_1(t), w_2(t), \ldots, w_d(t))$ be a d-dimensional Brownian motion. We will use the convention that

$$w_0(t) = t$$

Using this convention, the SDE's (1.1) and (1.2) can be written as

$$\begin{cases} dx_t &= \sum_{i=0}^{d} g_i(x,t)\, dw_t^i \\ x(0) &= x_0 \end{cases}$$

and

$$\begin{cases} dx_t &= \sum_{i=0}^{d} f_i(x,t) \circ dw_t^i \\ x(0) &= x_0 \end{cases}$$

We will use Δw in general to refer to increments of the Brownian path, using either the notation

$$\Delta w(s,t) \doteq w(t) - w(s)$$

or else using subscripts, such as

$$\Delta w_{k+1} = w(t_{k+1}) - w(t_k)$$

whenever a sequence of times has been defined.

Definition 1.2.2 *We define iterated stochastic Itô and Stratonovich integrals, $I_J(s,t)$ and $S_J(s,t)$ by*

$$I_J(s,t) = I_{j_1 j_2 \ldots j_n}(s,t) = \int_{\Sigma_n(s,t)} dw^{j_1}(t_1) dw^{j_2}(t_2) \ldots dw^{j_n}(t_n) \tag{1.3}$$

and

$$S_J(s,t) = S_{j_1 j_2 \ldots j_n}(s,t) = \int_{\Sigma_n(s,t)} \circ dw^{j_1}(t_1) \circ dw^{j_2}(t_2) \ldots \circ dw^{j_n}(t_n) \tag{1.4}$$

where $\Sigma_n(s,t)$ is the n-simplex in R_+^n

$$\Sigma_n(s,t) = \{s < t_1 < t_2 \ldots < t_n < t\}$$

We will write $I_J(t)$ for $I_J(0,t)$ and similarly for Stratonovich integrals.

Definition 1.2.3 *For use with Itô calculus, we define the operators $\mathcal{L}^j$, $j = 0, \ldots, d$ by*

$$\mathcal{L}^0 \doteq \frac{\partial}{\partial t} + \sum_{k=1}^{n} g_0^k \frac{\partial}{\partial x^k} + \frac{1}{2} \sum_{k,l=1}^{n} \sum_{j=1}^{d} g_j^k g_j^l \frac{\partial^2}{\partial x^k \partial x^l}$$

and

$$\mathcal{L}^j \doteq \sum_{k=1}^{n} g_j^k \frac{\partial}{\partial x^k} \quad j = 1, \ldots, d$$

The corresponding operators for Stratonovich calculus are

$$\tilde{\mathcal{L}}^0 \doteq \frac{\partial}{\partial t} + \sum_{k=1}^{n} f_0^k \frac{\partial}{\partial x^k}$$

and

$$\tilde{\mathcal{L}}^j \doteq \mathcal{L}^j \quad j = 1, \ldots, d$$

1.3 Existence and uniqueness of pathwise solutions

K. Itô first proved ([23]) that under certain assumptions the SDE (1.1) admits a solution and that this solution is unique.

Theorem 1.3.1 *(Ito) Suppose that the coefficients g_i, $i = 0, \ldots, d$ in (1.1) satisfy the Lipschitz condition*

$$|g_i(x,t) - g_i(y,t)| \leq K|x-y| \quad (i = 0, \ldots, d) \tag{1.5}$$

for some constant $K < \infty$, $\forall t \geq 0$ and $\forall x, y \in I\!R^n$ and suppose that there is some C_T such that

$$\sum_{i=0}^{d} |g_i(x,t)| \leq C_T$$

for all $t \leq T$. Furthermore, let the initial value, x_0, be such that

$$E(|x_0|^2) < \infty$$

Then there is one and only one semi-martingale x satisfying equation (1.1).

The constraint on the functions $g_i(x,t)$ in Theorem 1.3.1, can be weakened quite easily, but is not too prohibitive in the context of the current work. The global Lipschitz condition is prohibitive, but it can be replaced with the local Lipschitz condition

$$|g_i(x,t) - g_i(y,t)| \leq K_N|x-y|$$

whenever $|x| \vee |y| \leq N$ and $0 \leq t \leq N$.

Uniqueness and existence proofs have been given by Doléans-Dade [13] and Protter [49] for SDE's driven by a general semi-martingale.

1.4 Numerical methods for pathwise solutions

The standard method of obtaining numerical approximations to pathwise solutions of (1.1) is to discretise the equation. The approximate solution $\bar{x}$ will be calculated at a set of times $t_0 = 0, t_1 = t_0 + h_1, \ldots, t_{k+1} = t_k + h_{k+1}, \ldots, t_N = T$ using a scheme of the type

$$\begin{aligned} \bar{x}(t_0) &= x(t_0) \\ \bar{x}(t_{k+1}) &= \phi\left(t_k, \bar{x}(t_k), h_{k+1}, \Delta w_{k+1}\right), \quad k = 0, \ldots, N-1 \end{aligned} \tag{1.6}$$

where

$$\begin{aligned} h_{k+1} &\doteq t_{k+1} - t_k \\ \Delta w_{k+1} &\doteq w(t_{k+1}) - w(t_k) \end{aligned}$$

Definition 1.4.1 *An approximate solution, $\bar{x}$, obtained using a discretisation scheme as in (1.6), is said to* converge pathwise *(or converge strongly) to x, the solution of the SDE (1.1), if*

$$\lim_{\delta \to 0} \sup_{0 \le t \le T} |x(t) - \bar{x}(t)| = 0 \ a.s.$$

where $\delta = \max_k h_k$.

The easiest method of obtaining discretisation schemes is to truncate the stochastic Taylor series expansion of the solution to the SDE (1.1), keeping only terms up to a chosen order. This method is due to Platen and Wagner [48] and Kloeden and Platen [25]. Azencott [1] showed the size of the remainder term and Ben Arous [2] examined the convergence of the stochastic Taylor series.

Theorem 1.4.2 *(Platen and Wagner, Azencott) Let τ be the lifetime of the stochastic process x_t, solution of (1.1), and let $g_i \in C^{p+1}$, $\forall i = 0, \ldots, d$. For all $0 < t < \tau$, define $R_{p+1}(t)$ by*

$$x(t) = x_0 + \sum_{m=1}^{p} \sum_{\|J\|=m} I_J(t)\, g_J(x_0) + t^{(p+1)/2} R_{p+1}(t) \tag{1.7}$$

where for $|J| > 1$, the function g_J is defined recursively by:

$$g_{j_1 j_2 \ldots j_m} = \mathcal{L}^{j_1} g_{j_2 \ldots j_m}$$

Then

$$\lim_{t->0,r->\infty} P(|R_{p+1}(t)| \geq r) = 0$$

Theorem 1.4.3 *(Kloeden and Platen, Ben Arous) Let τ be the lifetime of the stochastic process x_t, solution of (1.2), and let $f_i \in C^{p+1}$, $\forall i = 0, \ldots, d$. For all $0 < t < \tau$, define $R_{p+1}(t)$ by*

$$x(t) = x_0 + \sum_{m=1}^{p} \sum_{||J||=m} S_J(t)\ f_J(x_0) + t^{(p+1)/2} R_{p+1}(t) \tag{1.8}$$

where for $|J| > 1$, the function f_J is defined recursively by:

$$f_{j_1 j_2 \ldots j_m} = \tilde{\mathcal{L}}^{j_1} f_{j_2 \ldots j_m}$$

Then

$$\lim_{t->0,r->\infty} P(|R_{p+1}(t)| \geq r) = 0$$

Using these two theorems, it is possible to expand the solution using either Itô or Stratonovich calculus. We will use the Itô stochastic Taylor series as an illustration. The Stratonovich series is simpler to use. Initially we will take a Brownian path of dimension one. To avoid lengthy formulae, we will generally omit direct reference to the time variable from now and refer to the autonomous SDE

$$\begin{cases} dx_t &= g_0(x)dt + \sum_{i=1}^{d} g_i(x)\ dw_t^i \\ x(0) &= x_0 \end{cases} \tag{1.9}$$

In many places no generality is actually lost. It is always possible to replace a non-autonomous SDE with an autonomous SDE, by including time in the state vector, thereby increasing the dimension of the system.

For simplicity of notation, we write $x_k \doteq x(t_k)$ and $\overline{x}_k \doteq \overline{x}(t_k)$. The solution to (1.9) at time t_{k+1} can be written as

$$x_{k+1} = x_k + \int_{t_k}^{t_{k+1}} g_0(x_t)\ dt + \int_{t_k}^{t_{k+1}} g_1(x_t)\ dw_t \tag{1.10}$$

Using the Itô formula

$$a(x_t) = a(x_k) + \int_{t_k}^{t} \mathcal{L}^0 a(x_s) ds + \int_{t_k}^{t} \mathcal{L}^1 a(x_s)\ dw_s \tag{1.11}$$

in equation (1.10), to expand both $g_0(x_t)$ and $g_1(x_t)$ around t_k, we obtain

$$\begin{aligned} x_{k+1} = \; & x_k + g_0(x_k)h_{k+1} + g_1(x_k)\Delta w_{k+1} \\ & + \int_{t_k}^{t_{k+1}} \int_{t_k}^{t} \mathcal{L}_0 g_0(x_s)\, ds\, dt + \int_{t_k}^{t_{k+1}} \int_{t_k}^{t} g_0'(x_s) g_1(x_s)\, dw_s\, dt \\ & + \int_{t_k}^{t_{k+1}} \int_{t_k}^{t} \mathcal{L}_0 g_1(x_s)\, ds\, dw_t + \int_{t_k}^{t_{k+1}} \int_{t_k}^{t} g_1'(x_s) g_1(x_s)\, dw_s\, dw_t \end{aligned} \tag{1.12}$$

If the variance of any term in such a stochastic Taylor series is Ch^α, we will say the term is of order $O(h^\alpha)$. Discarding the double integrals in (1.12), each of which is of order $O(h^\alpha)$ with $\alpha \geq 1$, we obtain the Euler-Maruyama scheme

$$\bar{x}_{k+1} = \bar{x}_k + g_0(\bar{x}_k)h_{k+1} + g_1(\bar{x}_k)\Delta w_{k+1} \tag{1.13}$$

This scheme, the simplest discretisation scheme for SDE's, is named after Maruyama, [39], one of the first to examine it. The multi-dimensional version is

$$\bar{x}_{k+1} = \bar{x}_k + g_0(\bar{x}_k)h_{k+1} + \sum_{i=1}^{d} g_i(\bar{x}_k)\Delta w_{k+1}^i \tag{1.14}$$

If we use (1.11) to expand $g_1'(x_s)g_1(x_s)$ in (1.12) and discard all terms of order $O(h^\alpha)$ for $\alpha \geq 3/2$, we obtain

$$\overline{x}_{k+1} = \overline{x}_k + g_0(\overline{x}_k)h_{k+1} + g_1(\overline{x}_k)\Delta w_{k+1} + g_1'(\overline{x}_k)g_1(\overline{x}_k) \int_{t_k}^{t_{k+1}} \int_{t_k}^{t} dw_s\, dw_t$$

or

$$\overline{x}_{k+1} = \overline{x}_k + g_0(\overline{x}_k)h_{k+1} + g_1(\overline{x}_k)\Delta w_{k+1} + \frac{1}{2} g_1'(\overline{x}_k)g_1(\overline{x}_k)\left((\Delta w_{k+1})^2 - h_{k+1}\right) \tag{1.15}$$

which is the Milshtein scheme, first proposed by Milshtein [43] for weak solutions.

Most often we take $h_k = h = T/N$, $(k = 1, \ldots, N)$ in the above schemes, but they will both converge to the true solution for any set t_k of stopping times. With a constant time step $h = T/N$, a discretisation scheme is said to have order of pathwise convergence α if

$$h^{-\alpha} \sup_k |x_k - \overline{x}_k^h| \to 0 \; a.s.$$

Newton ([44], [46]) and Faure ([14], [15]) proved that with a constant time step the Euler-Maruyama and the Milshtein schemes have pathwise order of convergence

$1/2 - \epsilon$ and $1 - \epsilon$ respectively, $\forall \epsilon > 0$, in the one-dimensional case and that the Euler-Maruyama scheme has the same order of pathwise convergence in the multi-dimensional case.

If one tries to improve on the order of accuracy of the scheme, by keeping more terms from the Taylor series, then in general one will need to include multiple stochastic integrals in the scheme as well as increments of the Brownian path. For example, for a one-dimensional SDE, inclusion of terms of order $O(h^{3/2})$ will in general mean inclusion of the integrals

$$\int_{t_k}^{t_{k+1}} \int_{t_k}^{t} dw_s \, dt$$

These integrals have a normal distribution and it is therefore possible to generate them jointly with the increments Δw_{k+1}.

In higher dimensions the situation is even worse. In general the order of accuracy of the Euler-Maruyama scheme is the best order obtainable without the generation of multiple integrals. The stochastic Taylor series development of x_{k+1}^i in terms of x_k, for $x \in I\!R^n$, $w(t) \in I\!R^d$ is

$$x_{k+1}^i = x_k^i + g_0^i(x_k)h_{k+1} + \sum_{j=1}^{d} g_j^i(x_k)\Delta w_{k+1}^j + \sum_{j=1}^{n}\sum_{p,q=1}^{d} \frac{\partial g_p^i}{\partial x^j} g_q^j(x_k) I_{pq}(k, k+1) + R \tag{1.16}$$

where

$$I_{pq}(k, k+1) = \int_{t_k}^{t_{k+1}} \int_{t_k}^{t} dw_s^p \, dw_t^q$$

and R consists of terms of order $O(h^\alpha)$ for $\alpha > 1$. If $1 \leq p, q \leq d$, $p \neq q$, then integration by parts gives

$$I_{pq}(k, k+1) + I_{qp}(k, k+1) = \Delta w_{k+1}^p \Delta w_{k+1}^q \doteq B_{pq}(k, k+1)$$

Define

$$A_{pq}(k, k+1) \doteq \frac{1}{2}\left(I_{pq}(k, k+1) - I_{qp}(k, k+1)\right) \tag{1.17}$$

In (1.16) we can substitute for the I_{pq} in terms of B_{pq} and A_{pq}, to obtain

$$\begin{aligned} x^i_{k+1} = \; & x^i_k + g^i_0(x_k)h_{k+1} + \sum_p g^i_p(x_k)\Delta w^p_{k+1} \\ & + \frac{1}{2}\sum_{j=1}^{n}\sum_{p=1}^{d}\frac{\partial g^i_p}{\partial x^j}g^j_p(x_k)\left((\Delta w^p_{k+1})^2 - h_{k+1}\right) \\ & + \sum_{j=1}^{n}\sum_{0<p<q\le d}\frac{1}{2}\left(\frac{\partial g^i_q}{\partial x^j}g^j_p + \frac{\partial g^i_p}{\partial x^j}g^j_q\right)(x_k)B_{pq}(k,k+1) \\ & + \sum_{j=1}^{n}\sum_{0<p<q\le d}\left(\frac{\partial g^i_q}{\partial x^j}g^j_p - \frac{\partial g^i_p}{\partial x^j}g^j_q\right)(x_k)A_{pq}(k,k+1) + R \end{aligned} \tag{1.18}$$

If $\forall i = 1,\ldots,n$, $\forall p,q = 1,\ldots,d$

$$\sum_{j=1}^{n}\left(\frac{\partial g^i_q}{\partial x^j}g^j_p - \frac{\partial g^i_p}{\partial x^j}g^j_q\right) = 0 \tag{1.19}$$

then the A_{pq} terms drop out of equation (1.18). Condition (1.19) is called the commutativity condition, since it can also be written as

$$[g_p, g_q] = 0$$

and is equivalent to saying that the vector fields (g_p, $p = 1,\ldots,d$) commute. When the commutativity condition is not satisfied, then the terms A_{pq}, known as the *Lévy areas*, need to be included in any discretisation scheme that is to attain an order of convergence greater than 1/2. For the proof see Clark and Cameron [9]. Clearly, by dropping the remainder term in (1.18), we obtain a discretisation scheme for general SDE's, of higher order than the Euler-Maruyama scheme, that involves generation of the Lévy areas.

Apart from discretisation schemes that can be obtained directly by truncation of a stochastic Taylor series expansion, there are Runge-Kutta type schemes, that avoid explicit calculation of derivatives. One very simple, and therefore very useful, Runge-Kutta type scheme is the Heun scheme,

$$\begin{aligned} \overline{x}_{k+1} &= \overline{x}_k + \frac{1}{2}(f_0(\overline{x}_k) + f_0(\hat{x}_k))h_{k+1} + \frac{1}{2}\sum_{i=1}^{d}(f_i(\overline{x}_k) + f_i(\hat{x}_k))\Delta w^i_{k+1} \\ \hat{x}_k &= \overline{x}_k + f_0(\overline{x}_k)h_{k+1} + \sum_{i=1}^{d} f_i(\overline{x}_k)\Delta w^i_{k+1} \end{aligned} \tag{1.20}$$

which converges to the solution of the Stratonovich SDE (1.2) at the same rate as the Milshtein scheme converges to the solution of the Ito SDE (1.1). Details

of how to choose Runge-Kutta type schemes that converge to solutions of (1.1) or (1.2) can be found in Rümelin [54].

Although, using only increments of the Brownian path, we cannot obtain a better order of convergence than $O(h)$ for an SDE driven by a one-dimensional Brownian path or satisfying the commutativity condition or $O(\sqrt{h})$ in the general case, it is possible to choose a method giving a lower variance for the errors than the Euler method. Among $\mathcal{P}_N$-measurable approximations, where $\mathcal{P}_N$ is the σ-field generated by (w_{ih}^j) for $i = 1, \ldots, N$, and $j = 1, \ldots, d$, there is a class of numerical schemes with optimal order of pathwise convergence that have the property of being asymptotically efficient in the L_2-sense, as defined by Clark in [7]. An equivalent definition is the following:

Definition 1.4.4 *An approximate solution, $\overline{x}$, to the SDE (1.1) is asymptotically efficient of order α if*

$$\frac{E\left[\left\langle c, \frac{1}{h^\alpha}(x_T - \overline{x}_N)\right\rangle^2 \middle| \mathcal{P}_N\right] + 1}{E\left[\left\langle c, \frac{1}{h^\alpha}(x_T - E[x_T|\mathcal{P}_N])\right\rangle^2 \middle| \mathcal{P}_N\right] + 1} \stackrel{a.s}{\to} 1 \tag{1.21}$$

for all $c \in \mathbb{R}^n$, (where $\langle y, z\rangle$ denotes the scalar product of two vectors).

Newton ([44] and [46]) has derived practical first order asymptotically efficient schemes. Both the Heun and Milshtein methods are asymptotically efficient in the case of an SDE driven by multi-dimensional Brownian motion and not satisfying the commutativity condition.

As well as the stochastic Taylor series approach mentioned above, there is another way of approximating the solution to an SDE locally, that can also lead to numerical methods. Ben Arous, Castell and Hu ([2],[4],[21]) have shown that the solution can be expressed locally as an exponential Lie series and can therefore be approximated by the flow of an ODE. Let

$$\zeta^p(t) = \sum_{m=1}^{p-1} \sum_{||J||=m} c^J(t)\, X^J, \tag{1.22}$$

where

- $$c^J(t) = \sum_{\sigma \in \mathcal{O}_m} \frac{(-1)^{e(\sigma)}}{m^2 \begin{pmatrix} m-1 \\ e(\sigma) \end{pmatrix}} S_{J \circ \sigma^{-1}}(t),$$

- X^J denotes the Lie bracket $[f_{j_1}[\ldots[f_{j_{m-1}} f_{j_m}]\ldots]$,

- $\mathcal{O}_m$ is the set of all permutations of $\{1, 2, \ldots, m\}$,

- $e(\sigma)$ is the number of errors in the ordering $\sigma(1), \ldots, \sigma(m)$, that is the cardinality of the set $\{j \in \{1, \ldots, m-1\}/\sigma(j) > \sigma(j+1)\}$,

- $J \circ \sigma = (j_{\sigma(1)}, \ldots, j_{\sigma(m)})$.

Then

$$x(t) = \exp(\zeta^p(t))(x_0) + t^{\frac{p}{2}} \overline{R}_p(t) \tag{1.23}$$

and $\overline{R}_p$ is bounded in probability when t tends to 0. Here $\exp(\zeta^p(t))(x_0)$ denotes the solution at time 1 of the ordinary differential equation given by $\zeta^p(t)$, ie $\exp(\zeta^p(t))(x) = u(1)$ where u is solution to

$$\begin{cases} \dfrac{du}{ds} &= \zeta^p(t)(u(s)) \\ u(0) &= x_0 \end{cases} \tag{1.24}$$

Numerical methods can be obtained by choosing a value for p, (depending on the number of iterated integrals that can be generated), and solving the ODE (1.24) numerically using a suitable ODE solver. In [5], Castell and Gaines show that this exponential Lie series approach can also provide asymptotically efficient approximation schemes.

Detailed surveys, that describe numerical methods for both pathwise and weak solutions to SDE's, can be found in [8], [19], [24], [26], [47], [56] .

1.5 Representation of Brownian paths

The stochastic Taylor expansions and resulting discretisation schemes shown in Section 1.4 suggest at least three possible ways that one might wish to represent a Brownian path when using it to approximate a pathwise solution to an SDE. The first and simplest representation is as a set of increments

$$\{\Delta w_{k+1} = w(t_{k+1}) - w(t_k),\ k = 1, \ldots, N\}$$

Such a set can be used with the Euler-Maruyama scheme, say, in any number of dimensions.

More complicated ways of viewing and generating Brownian paths consist of generating not only increments but also sets of integrals of the path. When the Brownian path is of dimension one, or when the vector fields defining the diffusion of the SDE commute, then it makes sense to generate the Brownian path as a set of increments and integrals against time,

$$\{\Delta w_{k+1}, A_{i0}(k, k+1),\ k = 1, \ldots, N\}$$

where

$$A_{i0}(k, k+1) = \frac{1}{2}\left(I_{i0}(k, k+1) - I_{0i}(k, k+1)\right) = \int_{t_k}^{t_{k+1}} \int_{t_k}^{t} dw_s^i\, dt - \int_{t_k}^{t_{k+1}} \int_{t_k}^{t} ds\, dw_t^i$$

Since Δw_{k+1}^i and $A_{i,0}(k, k+1)$ have a joint normal distribution, this is perfectly feasible and practical.

A third possibility, which one would like to realise in the case of a multi-dimensional Brownian path, when the commutativity condition is not satisfied, is to generate at the same time increments of the Brownian path and Lévy area integrals, $A_{ij}(k, k+1)$, as defined in (1.17) above. In Chapter 3 we show how this can be done, in the case of a two-dimensional Brownian path at least. It can also sometimes make sense to approximate Lévy areas by adding up increments generated over smaller time steps, as will be the case in Chapter 2. To get the same order of accuracy over bigger steps using approximate Lévy areas as over

smaller steps using only increments of the Brownian path, one has to generate all the increments over the smaller steps any way, but the saving is in function evaluations, since the various functions of x needed in the discretisation scheme need only be evaluated at the beginning of each large time step.

Chapter 2

Variable Step Size Control in the Numerical Solution of SDE's

2.1 Motivation

In this chapter we describe a variable step size method for the numerical approximation of strong solutions to the stochastic differential equation (1.1)

$$\begin{cases} dx_t &= g_0(x,t)dt + \sum_{i=1}^{d} g_i(x,t)\, dw_t^i \\ x(0) &= x_0 \end{cases}$$

or (1.2)

$$\begin{cases} dx_t &= f_0(x,t)dt + \sum_{i=1}^{d} f_i(x,t) \circ dw_t^i \\ x(0) &= x_0 \end{cases}$$

Although variable step size control is widely used in the numerical solution of ordinary differential equations, existing approaches to the solution of SDE's nearly always assume a fixed step size. (For exceptions see K. Bichteler [3] and N.J. Newton [45]) There is, however, a lack of discretisation methods guaranteeing accuracy to high powers of the step size. As mentioned in Chapter 1, in the general case of more than one dimension and without the commutativity condition (1.19), no numerical method, based only on an approximation of a Brownian path by

its values at times separated by an interval h, can guarantee accuracy along the trajectory of a higher order than $O(\sqrt{h})$. Improvements involve the measurement or simulation of area and other iterated integrals. This forces the use of small step sizes, whatever the method used or the SDE to be solved.

In some problems we have examined, the size of the time step giving acceptable accuracy is highly non-homogeneous, varying both along any given trajectory and also with the initial condition. In these cases use of a fixed step size implies the use of an unacceptable amount of both computer memory and computation time. It is this consideration which led us to the implementation of a variable step size method.

Our method is very much dependent on a particular way of viewing and generating a Brownian path. It is standard practice to generate approximations to a Brownian path as a discrete set of points evenly spaced over the required time interval, with points that occur later in time always being generated after points that occur earlier. For our purposes such a method is not satisfactory. We wish to be able to test the accuracy of solutions by solving the same equation repeatedly using decreasing step sizes. Comparability of the consecutive trajectories implies using the same noise, ie. the same Brownian path, each time, but with more points available as the step size decreases. Another reason for requiring a different method of generating Brownian paths, is the fact that a variable step size method demands knowledge of the Brownian path at intermediate points in some subintervals, up to an often unforeseeable number of subdivisions. Generating the path at *all* intermediate points up to a minimum step size would create a very large set of points, removing one of the advantages of using variable step sizes.

We view, and generate, a Brownian path as a tree, to which branches can be added at any time. This allows knowledge of points on the path in as much detail as each particular application requires. Our main tool in the generation of this tree is Lévy's construction [30]. This is a straightforward method that could be extended to other Markovian driving processes, providing that one had explicit forms for the transition densities.

In the present chapter we are interested only in $\mathcal{P}_N$-measurable approximations,

so iterated integrals, when used, will be approximated using increments of the Brownian path.

The outline of the chapter is as follows. First we describe how we generate Brownian paths for use with variable step size methods. Then we describe our general variable step size method, which involves estimation of local errors and of their propagation and choice of which errors are acceptable at which stage. We follow this with proof that our methods converge under the right conditions. In Section 2.5 we illustrate our variable step size methods by applying them to a few particular SDE's. In the final section we give programming details for our method of generation of Brownian paths.

2.2 Generation of Brownian paths

2.2.1 Brownian paths generated as increments

Let us describe the tree structure we have developed for both generation and storage of Brownian paths. The elements stored in the tree are not points on the Brownian path, but differences, Δw, across intervals of particular sizes. In this way, the SDE solving routine can be provided immediately with what it needs after at most one file access and without performing a subtraction. Since the equation (1.1) to be solved may involve a multidimensional Brownian path, the elements Δw are in general vectors.

Let us refer to a Brownian path stored in a tree structure as a Brownian tree. The first level of the Brownian tree is generated by simulating differences across a chosen number, N, of unit time intervals:

$$\Delta w_{k,1} = w(k) - w(k-1), \qquad k = 1, \ldots, N$$

Hence each Δw is simply a normally distributed random number with mean zero and unit variance. On the second and subsequent levels of the tree each previously generated difference can be used to generate two more, the differences across the

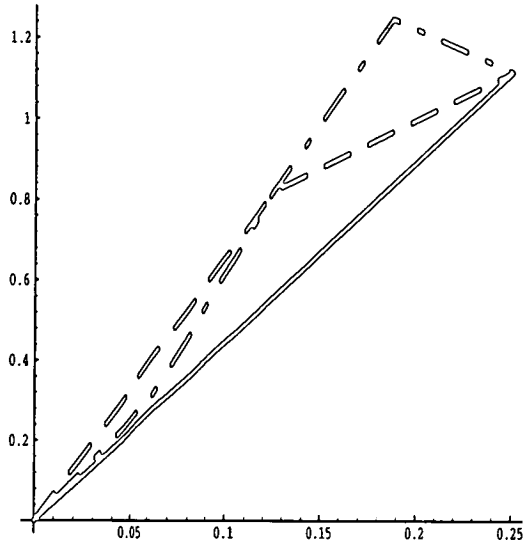


Figure 2–1:

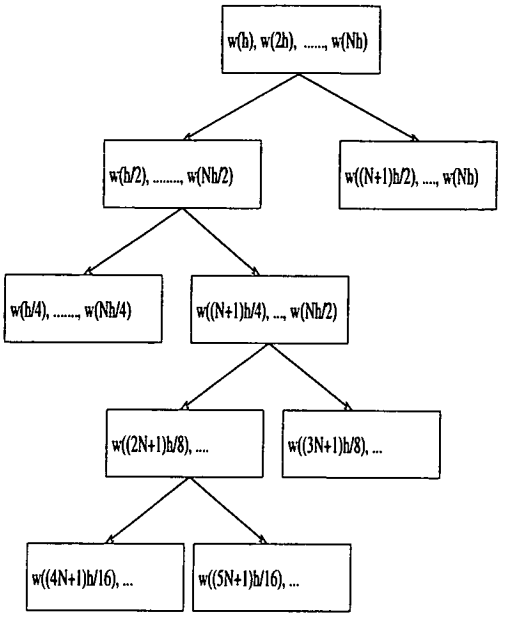


Figure 2–2:

two subintervals created by introducing the midpoint. This method is due to Lévy [30]. Thus on level $j+1$ we have:

$$\begin{aligned} \Delta w_{2k-1,j+1} &= \frac{1}{2}\Delta w_{k,j} + y_{k,j+1} \\ \Delta w_{2k,j+1} &= \frac{1}{2}\Delta w_{k,j} - y_{k,j+1}, \qquad j = 1, 2, \ldots \end{aligned}$$

where $y_{k,j}$ is normally distributed with zero mean and variance 2^{-j}. The tree can continue down to any level, and apart from the initial level no level need be complete. See Figure 2–1 for an illustration of how a Brownian path can be built up using Lévy's construction.

Since, when constructing strong approximations, it is often desired to solve one equation with many different initial conditions or parameter values, using the same noise each time, we save time and effort by storing each Brownian path in a file as it is generated and re-using it. Each record in the file corresponds to a node in the tree. Each node is not a single difference Δw across a single interval, but is in fact a group of M differences across M consecutive intervals, where M is chosen so that any disc access time is comparable with the computational time required to analyse a given section of path. This is for reasons of economy both of

storage space in the file and of file access time when reading from it. Figure 2–2 illustrates how the Brownian path is stored as a tree.

Generation of the Brownian tree and solution of an SDE are performed in parallel. At each step along the trajectory elements of the Brownian path are required. These elements are either already present in the tree and need only be input, or else they belong to a node not yet created. In the latter case, the new node (plus any intermediate branches necessary for arriving at the desired node) is immediately generated from existing ones using new random numbers. All new path elements are stored in the file and the solution proceeds.

The tree may never reach a final state, in that when it is used to solve the same equation from a different initial point or with a different parameter value, the areas where very small steps are needed may be different and therefore the tree will need to be developed further.

More technical details of the data structure and algorithm that we have chosen for the dynamic generation and use of Brownian trees are given below in the section on implementation.

2.2.2 Brownian paths generated as increments and approximate Lévy areas

In the preceding section we have described a system of storage and dynamic generation for a Brownian path composed of a set of increments over steps of varying sizes. It is possible to generalise the data structure to include at each node of the tree not only increments of the Brownian path, but other path statistics, such as integrals against time or Lévy area integrals. The higher order integrals could be generated jointly with the increments (as in Chapter 3). Although in this chapter we are limiting ourselves to representations of Brownian paths that do not include direct generation of higher order integrals, it is still possible to approximate integrals of the Brownian path using increments. In Section 2.4, we will see that to guarantee convergence for variable step size methods in the case of a general SDE (with multi-dimensional noise and not satisfying the commutativity condition), it

is necessary to include in the discretisation scheme approximate Lévy areas as well as increments of the Brownian path. We therefore need to generalise the Brownian tree presented in Section 2.2.1, to allow inclusion of approximate area integrals.

The basic idea is as follows. At each node of the tree at level j, we will store both increments of the Brownian path over time steps of length 2^{-j} and approximate area integrals obtained using increments on a lower level, $j+k$ say. As before, the tree will grow as necessary. When new increments and areas are required at a certain level, the level of the numerical solution, new increments have to also be generated at lower levels and then used to approximate the required areas.

Let a time interval $[t, t+h]$ be subdivided into 2^k equal subintervals and increments of the Brownian path Δw_p, $p = 1, \ldots, 2^k$ be generated over the subintervals using Lévy's construction, as described in Section 2.2.1. Then $\tilde{A}_{ij}$, an approximation to the Lévy area $A_{ij}(t, t+h)$ (as defined in (1.17)) can be obtained using

$$\tilde{A}_{ij} = \sum_{1 \le q < p \le 2^k} \Delta w_q^i \Delta w_p^j - \sum_{1 \le p < q \le 2^k} \Delta w_q^i \Delta w_p^j$$

2.3 Variable step size control and accuracy criteria

At first sight, the method we use resembles any standard step size control method for numerical integration of ODE's, with some simplifications, forced by the structure of the Brownian path. In particular, new step sizes are only derived from previous ones by halving or doubling.

The step size control routine takes one step of the required size, using a suitable discretisation scheme, and then checks whether the error made is acceptable, according to a chosen accuracy criterion. If the chosen condition is not satisfied then the step size is halved and the process repeated. Once the desired accuracy is reached, the next step size to be tried is set equal to twice the last one used,

as long as the structure of the Brownian tree allows for an increase in step size at that point.

The actual integration involved in "taking steps" can be performed using any discretisation method available, as long as convergence is guaranteed. (See Section 2.4 for details of the conditions required for convergence.) This lends flexibility to the method. It will be seen, however, that a certain class of discretisation methods, namely those usually known as asymptotically efficient, is particularly suitable.

The interesting question here is how to decide whether each step taken is accurate enough. We need both a reliable method for estimating the error introduced in a single step and a criterion for judging when the error made is acceptable.

2.3.1 Estimation of the Local Error

Let x_k be the true solution at time t_k and let the approximate solution be

$$\overline{x}_k = \phi(\overline{x}_{k-1}, \Delta w_k)$$

A method of local error estimation often used for ODE's is the following. Starting at time t_{k-1} one step of the required size h, and again at t_{k-1}, two steps of length $h/2$ are taken. Then the values obtained for $x(t_k)$ in the two cases, $\overline{x}_k$ and $\tilde{x}_k$, say, are compared, giving $|\overline{x}_k - \tilde{x}_k|$ as an estimate of the local error.

We have tried this method, but not found it very satisfactory. Although it does not yield a very bad error estimate, the total number of steps taken (including both those accepted and those rejected) is multiplied by at least one and a half.

A method of estimation of the one step error that seems more useful consists of expanding the error using a stochastic Taylor series (see Chapter 1) and then calculating the mean and variance of the resulting terms, conditional on our knowledge of the Brownian path. The necessary calculations can be carried out particularly easily using a computer algebra package (such as Reduce, Mathematica or Maple).

For any chosen integration method (such as Euler, Milshtein, Heun etc.) the procedure followed is roughly the same. First x_k is expanded as a series of integrals.

If necessary $\overline{x}_k$ is also expanded (this is the case for the explicit Heun method). Depending on whether the integration method being used converges to the solution of an Itô or a Stratonovich SDE, the expansion involves Itô or Stratonovich integrals. Then $\overline{x}_k$ is subtracted from x_k, to get the error terms. In general, the error terms include multiple stochastic integrals that cannot be calculated explicitly, so we do not immediately obtain an approximation to the error. However, we can calculate the mean and variance of the error conditional on the increments, Δw_k, of the Brownian path, (see Newton [44]). This gives us the best estimate of the error obtainable using the available information. Suitable estimates of the $\mathcal{P}_N-$conditional mean and variance are then obtained by retaining only the terms of lowest order.

2.3.2 Acceptance Criteria for the Local Error

Once the error introduced in one step has been estimated, the question remains as to what criterion to apply when accepting or rejecting the size of the error. Since our aim is to control the size of the final error, information is needed on the propagation of each local error along the trajectory. Another important consideration is how much each local error should be made to contribute to the final global error.

Error propagation

If we assume that step sizes are small enough that propagation of the errors is linear, we can obtain an SDE representing the propagation of an infinitesimal error, y, along a particular trajectory. Replacing x in (1.1) with $x + \epsilon y$ gives

$$dx + \epsilon dy = [g_0(x) + \epsilon g_0'(x)y]dt + \sum_{j=1}^{d}[g_j(x) + \epsilon g_j'(x)y]dw^j + o(\epsilon^2)(dw + dt)$$

and hence as ϵ tends to zero one obtains

$$dy = A(t)y\,dt + \sum_{j=1}^{d} B_j(t)y\,dw^j \tag{2.1}$$

where

$$A(t) \;=\; g_0'(x(t))$$

$$B_j(t) = g_j'(x(t))$$

We would like to know the size of the error that can be introduced at any given time $0 < t < T$, in order to obtain a contribution to the final error of at most a given amount. One way to get this information is to solve equation (2.1) backwards in time, using a fixed time step, starting from a chosen value for the final error. However, to solve (2.1), we need to know the solution to (1.1), so we start by calculating a rough solution to (1.1) using a sufficiently small fixed step size h, then use the values calculated for $\overline{x}_k$, $k = 1, \ldots, N$ while solving (2.1) backwards using the same step size. The final stage is then to solve the SDE (1.1) using variable time steps and using the information previously obtained to help choose tolerable errors at each step. The extra effort involved in the preliminary fixed time step solutions should be small compared with the computation time used subsequently to solve (1.1) accurately using variable time steps.

For an SDE in one dimension ($n = 1$), preliminary simultaneous forward solution of (1.1) and (2.1) over fixed time steps and rescaling of the solution to (2.1) would provide the necessary information on propagation of local errors. However, in more than one dimension ($n > 1$), this would involve inversion of matrices at each step, which would be computationally expensive and also problematic, since the determinant of the matrix could become very small.

Here we are supposing that the error is random and that the error at each step is independent of that at previous steps, something that can only be considered as an approximation to the truth when the step sizes and hence the errors are small enough.

Optimal contribution of local errors to the global error

At each time t_k an error $y(t_k)$ is introduced, which, after propagation along the trajectory, will add a contribution of

$$Y(t_k) = y(t_k)\omega(t_k)$$

to the final error at time T. Here $\omega(t_k)$ is the factor by which the error is multiplied between the time t_k and the final time, T. We need to choose the function $Y(t)$ so as to minimise the global error. We claim that the optimal choice, if one wishes to minimise the sum of the absolute errors for a given number of steps, is to let each local error contribute equally to the final error.

Sketch of proof:

We will presume that all steps are small enough that finite sums can be approximated accurately by integrals . Let $\rho(t)$ be the density of steps. Also suppose that the local error $y(t)$ can be expressed as

$$\psi(\rho) = c\rho^{-(\tau+1)}$$

We wish to minimise the total error

$$\int_0^T \psi(\rho)\omega(t)\rho(t)dt$$

subject to

$$\int_0^T \rho(t)dt = N$$

At the minimum we have

$$\int_0^T (\rho\psi(\rho))'h(t)\omega(t)dt = 0$$

giving

$$\int_0^T \psi(\rho)\omega(t)h(t)dt = 0$$

for all $h(t)$ with

$$\int_0^T h(t)dt = 0$$

Therefore

$$\psi(\rho)\omega(t) = constant$$

In the above we have chosen to minimise the sum of the absolute errors for a given number of steps. It is clear that the errors over individual steps may be of different signs and therefore cancelations may occur, so this is a kind of "worst case" solution. However, when estimating the local errors in Section 2.3.1, we

in fact estimated their $\mathcal{P}_N-$conditional means and variances. Variances, being positive, do all add up, so that the variance of the final error is indeed the sum of the contributions made by the variances of the errors over the different steps. What about the mean? Presumably we would like to minimise the mean of the global error as well as its variance, and, again, the one-step means can be of different signs. The way round this is to use a discretisation scheme for taking steps that reduces the $\mathcal{P}_N-$conditional mean of the error to a higher order than the standard deviation, thereby making the mean error negligible beside the standard deviation. The schemes known as asymptotically efficient schemes have this property. (See Chapter 1 and below for definition and references.)

2.3.3 Outline of Method

At this point a method for step size control has emerged. A maximum value C must be chosen for the contribution of each local error to the $\mathcal{P}_N-$conditional variance of the global error. C is a vector of dimension n, so this choice involves weighting the errors allowed for the different components of x. Initially we calculate a rough solution to the SDE (1.1) using a small enough fixed step size. Using this solution, we solve the linear SDE (2.1) backwards in time from the chosen final value of $y(T) = C$. (In the first two examples that follow, the SDE is simple enough that these initial stages can be replaced by analytical calculations, but this will not usually be the case.) This yields a set of permissible values $y(t_k, C)$ for the $\mathcal{P}_N-$conditional variance of errors introduced at points along the trajectory. In the final stage we proceed to solution of the original SDE using variable step size. To take steps we use any discretisation scheme that converges, that leads to local errors of the optimal order and using which the order of the square of the $\mathcal{P}_N-$conditional mean of the local error is higher than the order of the $\mathcal{P}_N-$conditional variance. At each step the conditional variance of the local error is estimated and compared with the value required. The step size is halved and the step repeated until the local error is small enough.

2.4 Convergence of variable step size algorithms and choice of discretisation schemes

In the preceding sections we have outlined an algorithm for numerical solution of SDE's using variable time steps. Naturally the question arises as to whether this algorithm converges to the true solution of each SDE. Various authors have already shown convergence of variable time step methods ([3], [45], [15], [26]), but in all cases the discretisation times have been stopping times. In this chapter the discretisation points are clearly not stopping times, since, on the one hand, the choice of acceptable errors is determined by prior investigation of an approximation of the whole trajectory and, on the other hand, it is only once a step has been taken and the error estimated that the decision is made to continue or to retreat and take a smaller time step. This leads us to look for general criteria that guarantee convergence of algorithms obtained by discretisation, to the solution of an SDE, whether the time steps are chosen deterministically or randomly, and, if randomly, whether the discretisation times are stopping times or not.

In this section the representation of Brownian paths described in Section 2.2 is crucial: the algorithms that we consider all represent a Brownian path as an infinite binary tree, so, however the time steps are chosen, the step size is equal to 2^{-n}, for some $n \in I\!N$.

We will consider first the particular case of SDE's driven by a one-dimensional Brownian path, or satisfying the commutativity condition, and then go on to consider the general case of SDE's driven by a multi-dimensional Brownian path, where the commutativity condition is not satisfied.

2.4.1 SDE's driven by a one-dimensional Brownian path or satisfying the commutativity condition

From [35] it follows that in the case of an SDE driven by a one-dimensional Brownian path, or an SDE satisfying the commutativity condition (1.19), in which

case the solution does not depend on Lévy area integrals, we have the following theorem:

Theorem 2.4.1 *Let the SDE (1.2) have $d = 1$ or satisfy the commutativity condition (1.19), then any variable step size algorithm that breaks $[0, T]$ into disjoint intervals each of length 2^{-k} for some $k \in \mathbb{N}$ and on each interval $[t_j, t_{j+1}]$, with $t_{j+1} = t_j + h_{j+1}$, takes as approximate solution, $\overline{x}(t)$, to (1.2) the solution for unit time of the ODE*

$$\begin{cases} \dfrac{du}{dt} = f_0 h_{j+1} + \displaystyle\sum_{i=1}^{d} f_i(u)\Delta w_{j+1}^i \\ u(0) = \overline{x}(t_j) \end{cases}, \tag{2.2}$$

will converge to the solution of (1.2), as long as

$$\max_j h_j \to 0$$

However, if we want an approximation that not only guarantees convergence of the variable step scheme, but also reduces the conditional mean of the local error sufficiently, then it would be sufficient to solve on each interval the ODE

$$\begin{cases} \dfrac{du}{dt} &= f_0(u)h_{j+1} + \sum_{i=1}^{d} f_i(u)\Delta w_{j+1}^1 - \dfrac{h_{j+1}^2}{12}\displaystyle\sum_{i=1}^{d} f^{(i0i)}(u) \\ u(0) &= \overline{x}_j \end{cases} \tag{2.3}$$

where $f^{(i0i)} = [f_i, [f_0, f_i]]$ shown in [5] to be a first-order asymptotically efficient approximation in the case of fixed step-sizes. Here we are using the notation $[u, v]$, where u and v are vector fields in $\mathbb{R}^n$, to denote the vector field

$$[u, v] = \sum_{i=1}^{n} u^i \frac{\partial v}{\partial x^i} - \sum_{i=1}^{n} v^i \frac{\partial u}{\partial x^i}$$

It is not necessary (or possible) to solve the ODE (2.3) perfectly. It is sufficient that over each step of the solution to the SDE, the discretisation scheme chosen produce a local error of $O(h^{3/2})$. Such accuracy can be obtained by solving the ODE (2.3) using the fourth order Runge-Kutta method (see [5]). Equivalently, any of the first order asymptotically efficient schemes of Newton ([44], [46]) could be used to solve the SDE (1.1) or (1.2) (depending on the scheme chosen) directly on each time interval.

2.4.2 SDE's driven by a multi-dimensional Brownian path and not satisfying the commutativity condition

When SDE's do not satisfy the commutativity condition, it is not enough to evaluate the Brownian path only at the points in time where the approximate solution is calculated. To guarantee convergence when using random time steps, it is necessary to also generate increments of the Brownian path at intermediate points in time and to use the increments to approximate the Lévy area integrals, as defined in (1.17).

The following theorem is completely based on suggestions of T.J. Lyons.

Theorem 2.4.2 *Let $\tilde{A}_{ij}(k+1)$, be an approximation to the Lévy area $A_{ij}(t_k, t_{k+1})$, $(i,j = 1,\ldots,d)$, obtained by subdividing the time interval $[t_k, t_{k+1}]$, where $t_{k+1} - t_k = h_{k+1} = 2^{-j}$, into $2^{\phi(j)}$ equal subintervals and taking the area of the piecewise linear path obtained by generating the Brownian path at the ends of the subintervals. If $\phi(j) \geq C \log_2 j$, for $C > 1$, then any variable step size algorithm that breaks $[0,T]$ into disjoint intervals each of length 2^{-j} for some $j \in \mathbb{N}$ and on each interval $[t_k, t_{k+1}]$ takes as approximate solution, $\overline{x}_{k+1}$, to (1.2) at time t_{k+1} the solution for unit time of the ODE*

$$\begin{cases} \dfrac{du}{dt} = f_0(u)h_{k+1} + \displaystyle\sum_{i=1}^{d} f_i(u)\Delta w_{k+1}^i + \sum_{1\leq i<j\leq d} [f_i, f_j](u)\tilde{A}_{ij}(k+1) \\ u(0) = \overline{x}_k \end{cases} \tag{2.4}$$

will converge to the solution of (1.2) as long as

$$\max_k h_k \to 0$$

Proof. To simplify notation, we denote by $A(t_k, t_{k+1})$ and $\tilde{A}_{k+1}$ the d by d matrices with entries $A_{ij}(t_k, t_{k+1})$ and $\tilde{A}_{ij}(k+1)$ respectively. From [35] it is clear that as long as

$$|\tilde{A}_{k+1} - A(t_k, t_{k+1})| \leq o(h_{k+1}) \tag{2.5}$$

uniformly in the choice of partition of $[0,T]$, then the approximate solution given by solving (2.4) for unit time on each interval $[t_k, t_{k+1}]$ will converge to the solution

of (1.2) as

$$\max_k h_k \to 0$$

Therefore we have to show that the approximate Lévy area, $\tilde{A}_{k+1}$, obtained by splitting each $[t_k, t_{k+1}]$ into $2^{\phi(j)}$ equal pieces, will satisfy (2.5) when $\phi(j) = C \log_2 j$, with $C > 1$.

We can express the Lévy area as

$$A(t_k, t_{k+1}) = \sum_{i=1}^{\infty} \sum_{p=1}^{2^{i-1}} [\Delta w(k,j,i,p), \Delta w(k,j,i,p+1)] \tag{2.6}$$

where we define

$$\begin{aligned} \Delta w(k,j,i,p) &\doteq w(t_k + (2p-1)h_{ij}) - w(t_k + (2p-2)h_{ij}) \\ \Delta w(k,j,i,p+1) &\doteq w(t_k + 2ph_{ij}) - w(t_k + (2p-1)h_{ij}) \end{aligned}$$

with

$$h_{ij} = 2^{-(i+j)}$$

and the approximate area as

$$\tilde{A}_{k+1} = \sum_{i=1}^{\phi(j)} \sum_{p=1}^{2^{i-1}} [\Delta w(k,j,i,p), \Delta w(k,j,i,p+1)] \tag{2.7}$$

(See Figure 2–3.) Here we are using the convention

$$[u, v] = u \otimes v - v \otimes u$$

Therefore we wish to estimate the size of

$$2^j \sum_{i > \phi(j)} \sum_{p=1}^{2^{i-1}} [\Delta w(k,j,i,p), \Delta w(k,j,i,p+1)]$$

We will start by investigating the size of the ith term. It is clearly sufficient to estimate

$$I\!P\left(\left| \sum_{p=1}^{2^i} \xi_{pj} \eta_{pj} \right| > \lambda 2^{-j} \right)$$

where the ξ_{pj}, η_{pj} are all i.i.d. normal random variables with mean 0 and variance h_{ij}, or, by scaling,

$$I\!P\left(\frac{1}{n} \left| \sum_{p=1}^{n} \xi_p \eta_p \right| > \lambda \right)$$

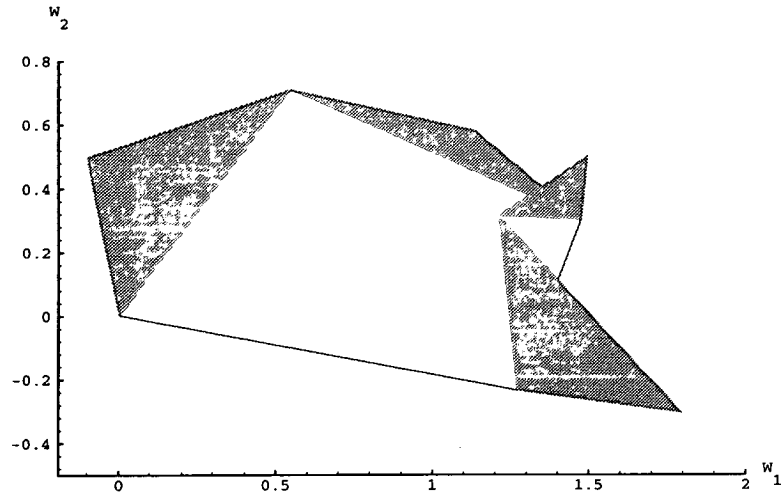


Figure 2–3: building up an approximate Lévy area

where the ξ_p, η_p are all $\mathcal{N}(0,1)$ (standard normal) random variables.

We have, for any $\alpha > 0$,

$$I\!E e^{\alpha(\xi\eta)} = \frac{1}{2\pi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{\alpha xy} e^{-\frac{x^2}{2}-\frac{y^2}{2}}\,dxdy = \frac{1}{\sqrt{1-\alpha^2}} \doteq \psi(\alpha)$$

and

$$I\!E e^{\alpha\sum_{p=1}^{n}(\xi_p\eta_p)} = I\!E\prod_{p=1}^{n} e^{\alpha\xi_p\eta_p} \leq \psi(\alpha)^n$$

Therefore, using large deviation theory,

$$\begin{aligned} I\!P\left(\frac{1}{n}\left|\sum_{p=1}^{n}\xi_p\eta_p\right| > \lambda\right) &= I\!P\left(e^{\alpha\sum_{p=1}^{n}\xi_p\eta_p} > e^{\alpha n\lambda}\right) \leq e^{-\alpha n\lambda} I\!E e^{\alpha\sum_{p=1}^{n}\xi_p\eta_p} \\ &\leq e^{-\alpha n\lambda}\psi(\alpha)^n = \left[e^{-\lambda\alpha}\psi(\alpha)\right]^n \end{aligned} \tag{2.8}$$

Since α is arbitrary in (2.8), we can take it so as to minimise the last expression. This is equivalent to minimising

$$-\log\sqrt{1-\alpha^2} - \alpha\lambda$$

So we must solve

$$\lambda = \frac{\alpha}{1-\alpha^2}$$

The positive root is

$$\alpha = -\frac{1}{2\lambda} + \sqrt{\frac{1}{4\lambda^2}+1}$$

which, using a Taylor series expansion, gives

$$I\!P\left(\frac{1}{n}\left|\sum_{p=1}^{n}\xi_p\eta_p\right| > \lambda\right) \leq \left(e^{-\frac{\lambda^2}{2}+O(\lambda^4)}\right)^n \tag{2.9}$$

We now need a suitable choice of $\lambda(i)$, to ensure that , with our hypothesis on $\phi(j)$, (2.5) will hold whenever

$$\left|\sum_{p=1}^{2^{i-1}}[\Delta w(k,j,i,p), \Delta w(k,j,i,p+1)]\right| \leq \lambda(i)$$

Let $\lambda(i) = 2^{-j}\frac{1}{i^m}$ with $m > 1$. Then

$$\begin{aligned} 2^j|\tilde{A}_{k+1} - A(t_k,t_{k+1})| &= 2^j\left|\sum_{i>\phi(j)}\sum_{p=1}^{2^{i-1}}[\Delta w(k,j,i,p), \Delta w(k,j,i,p+1)]\right| \\ &\leq 2^j\sum_{i>\phi(j)}\left|\sum_{p=1}^{2^{i-1}}[\Delta w(k,j,i,p), \Delta w(k,j,i,p+1)]\right| \\ &\leq \sum_{i>\phi(j)}\frac{1}{i^m} \sim \frac{1}{\phi(j)^{m-1}} = \frac{1}{(C\log_2 j)^{m-1}} \end{aligned}$$

which tends to 0 as $j \to \infty$.

Finally, we can use a Borel-Cantelli lemma, to show that, for a given Brownian path, (2.5) holds on every subinterval of $[0,T]$ with only finitely many exceptions. For each choice of j, there are 2^jT possible subintervals of length 2^{-j}. So, (remembering that the ξ_p and η_p are standard normal,) we need to show that

$$\sum_{j=1}^{\infty}2^jT\sum_{i>\phi(j)}I\!P\left(\frac{1}{2^i}\left|\sum_{p=1}^{2^i}\xi_p\eta_p\right| > \frac{1}{i^m}\right) < \infty \tag{2.10}$$

with $m > 1$. We can rewrite the left hand side of (2.10) as

$$T\sum_{i=1}^{\infty}\left(\sum_{\phi(j)<i}2^j\right)I\!P\left(\frac{1}{2^i}\left|\sum_{p=1}^{2^i}\xi_p\eta_p\right| > \frac{1}{i^m}\right)$$

With $\phi(j) = C\log_2 j$, for each i, there exists ϵ such that

$$\sum_{\phi(j)<i}2^j \leq 2^{(2-\epsilon)^i}$$

By (2.9),

$$I\!P\left(\frac{1}{2^i}\left|\sum_{p=1}^{2^i}\xi_p\eta_p\right| > \frac{1}{i^m}\right) \leq \left[\exp\left(-\frac{1}{2i^{2m}} + O(\frac{1}{i^{4m}})\right)\right]^{2^i}$$

Combining the last two expressions gives (2.10) for all m. □

As mentioned in Section 2.3.2, when using variable time steps it makes life easier if we choose a discretisation scheme to approximate the solution with the property that the mean error over each step, conditional on the available information about the Brownian path, is negligible beside the standard deviation. Then it is only the conditional variance that need be used as an accuracy criterion when choosing the length of the step. As seen above, for an SDE driven by a one-dimensional Brownian path, or satisfying the commutativity condition (1.19), there is a class of discretisation scheme available (those schemes known as first-order asymptotically efficient) that have the required property and also assure convergence by satisfying the hypotheses of Theorem 2.4.1. When it comes to a general SDE, (for which no $\mathcal{P}_N-$conditional approximation can assure a better order of accuracy than $O(\sqrt{h})$), we need to derive a numerical scheme that not only uses approximate Lévy areas, in order to fulfill the requirements of Theorem 2.4.2, but also produces local errors with small enough conditional means. We show here that both the true solution to the ODE (2.4), and good enough approximations to that solution, do give rise to local errors with reduced conditional means.

Since we have chosen to approximate Lévy areas over the time-steps of the numerical solution, by generating increments of the Brownian path over sub-intervals, the amount of information available at each time step, to be taken into account when examining conditional expectations of the local error, has increased. Whereas previously we have considered the filtration

$$\mathcal{P}_N = \sigma(w(t_k), k = 1, \ldots, N),$$

in this section we need to consider the augmented filtration

$$\tilde{\mathcal{P}}_N = \sigma(w(t_k), \{w(t_k + m\tilde{h}_k), m = 1, \ldots 2^{\phi(j)}\} m = 1, \ldots, N;),$$

where $h_k = 2^{-j}$, $\phi(j)$ is as defined in Theorem 2.4.2 and

$$\tilde{h}_k = \frac{h_k}{2^{\phi(k)}}.$$

If we expand the solution to the ODE (2.4) over a small time interval $[t_k, t_{k+1}]$, we obtain

$$\begin{aligned} x^i_{k+1} = \; & x^i_k + f^i_0(x_k)h_{k+1} + \sum_p f^i_p(x_k)\Delta w^p_{k+1} \\ & + \frac{1}{2}\sum_{j=1}^{n}\sum_{p=1}^{d} \frac{\partial f^i_p}{\partial x^j} f^j_p(x_k)\left((\Delta w^p_{k+1})^2 - h_{k+1}\right) \\ & + \sum_{j=1}^{n}\sum_{0<p<q\le d} \frac{1}{2}\left(\frac{\partial f^i_q}{\partial x^j} f^j_p + \frac{\partial f^i_p}{\partial x^j} f^j_q\right)(x_k) B_{pq}(k, k+1) \\ & + \sum_{j=1}^{n}\sum_{0<p<q\le d} \frac{1}{2}\left(\frac{\partial f^i_q}{\partial x^j} f^j_p - \frac{\partial f^i_p}{\partial x^j} f^j_q\right)(x_k) \tilde{A}_{pq}(k, k+1) + R \end{aligned}$$

for $i = 1, \ldots, n$, where the remainder, R, contains terms of order $O(h^{3/2})$ and higher. The expansion of the true solution of the SDE is identical as far as the remainder terms, except for containing the true Lévy areas, $A_{pq}(k, k+1)$, rather than the approximations, $\tilde{A}_{pq}(k, k+1)$.

The local error is therefore

$$\sum_{j=1}^{n}\sum_{0<p<q\le d} \frac{1}{2}\left(\frac{\partial f^i_q}{\partial x^j} f^j_p - \frac{\partial f^i_p}{\partial x^j} f^j_q\right)(x_k)(A_{pq}(k, k+1) - \tilde{A}_{pq}(k, k+1)) + R \quad (2.11)$$

Since the $\tilde{\mathcal{P}}_N$−conditional mean of the Lévy area $A(k, k+1)$ is the approximate area $\tilde{A}(k, k+1)$, the $\tilde{\mathcal{P}}_N$−conditional mean of the local error contains only terms of order $O(h^{3/2})$ and higher. The square of the conditional mean is therefore of order $O(h^3)$.

Now for the $\tilde{\mathcal{P}}_N$−conditional variance. Writing A_{pq} for $A_{pq}(k, k+1))$, we have

$$E\left[\left(A_{pq} - \tilde{A}_{pq}\right)^2 \middle| \tilde{\mathcal{P}}_N\right] = E\left[\left(\sum_{i=0}^{2^{\phi(k)}} A_{pq}(i\tilde{h}, (i+1)\tilde{h})\right)^2 \middle| \tilde{\mathcal{P}}_N\right]$$

Since

$$E\left[A_{pq}(i\tilde{h}, (i+1)\tilde{h}) A_{pq}(j\tilde{h}, (j+1)\tilde{h}) \middle| \tilde{\mathcal{P}}_N\right] = 0$$

for all $i \neq j$, we have

$$\begin{aligned} & E\left[\left(A_{pq} - \tilde{A}_{pq}\right)^2 \middle| \tilde{\mathcal{P}}_N\right] = \\ & \sum_{i=0}^{2^{\phi(k)}} E\left[A_{pq}(i\tilde{h}, (i+1)\tilde{h})^2 \middle| \Delta w(i\tilde{h}, (i+1)\tilde{h})\right] \end{aligned} \quad (2.12)$$

Using

$$E(A_{pq}^2(t,t+h)|\Delta w(t,t+h)) = \frac{1}{12}h(h + (\Delta w^p(t,t+h))^2 + (\Delta w^q(t,t+h))^2),$$

(2.12) becomes

$$E\left[\left(A_{pq} - \tilde{A}_{pq}\right)^2 \middle| \tilde{\mathcal{P}}_N\right] = \frac{\tilde{h}}{12}\left[h + \sum_{i=0}^{2^{\phi(k)}} \left(\Delta w^p(i\tilde{h},(i+1)\tilde{h})^2 + \Delta w^p(i\tilde{h},(i+1)\tilde{h})^2\right)\right],$$

which, when substituted into (2.11), gives the conditional variance of the local error as

$$\begin{aligned} &\sum_{j=1}^{n} \sum_{0<p<q\leq d} \frac{\tilde{h}}{12}\left(\frac{\partial g_q^i}{\partial x^j} g_p^j - \frac{\partial g_p^i}{\partial x^j} g_q^j\right)^2 (x_k) \\ &\left[h + \sum_{i=0}^{2^{\phi(k)}} \left(\Delta w^p(i\tilde{h},(i+1)\tilde{h})^2 + \Delta w^p(i\tilde{h},(i+1)\tilde{h})^2\right)\right] + R. \end{aligned} \tag{2.13}$$

2.5 Illustrative numerical applications

2.5.1 First example

For the first example we have chosen the one-dimensional equation

$$dx = \beta x \circ dw \tag{2.14}$$

because the solution can be explicitly calculated as

$$x(t) = x(0)e^{\beta w(t)}$$

When β is sufficiently large the solution becomes hard to approximate by pathwise integration.

Since the drift term in (2.14) is zero and the Brownian path one-dimensional, the solution to this equation can be approximated as accurately as wished using $\mathcal{P}_N$−measurable approximations. For such an equation the local error is always equal to its $\mathcal{P}_N$ conditional mean and standard deviation. The asymptotically efficient approximation would be the discretisation scheme given by the infinite series

$$\overline{x}_{k+1} = \overline{x}_k \sum_{i=0}^{\infty} \frac{\beta^i}{i!}(\Delta w_{k+1})^i$$

which is hardly a practical proposition. We will therefore choose a simple first-order scheme for the purpose of comparing fixed and variable time steps.

We will first estimate the one step error and then examine how errors propagate, hence deducing an accuracy criterion to be applied at each step.

We will use the (explicit) Heun method (1.20):

$$\overline{x}_{k+1} = \overline{x}_k + 1/2(f_0(\overline{x}_k) + f_0(\hat{x}))h + 1/2(f_1(\overline{x}_k) + f_1(\hat{x}))\Delta w_{k+1}$$

where

$$\hat{x} = \overline{x}_k + f_0(\overline{x}_k)h + f_1(\overline{x}_k)\Delta w_{k+1}$$

Expanding $f_1(\hat{x})$ as a series around $\overline{x}_k$ and setting $f_0(x) = 0$, we can write (1.20) as

$$\begin{aligned}\overline{x}_{k+1} = \overline{x}_k + f_1(\overline{x}_k)\Delta w_{k+1} + \frac{1}{2}f_1(\overline{x}_k)f_1'(\overline{x}_k)(\Delta w_{k+1})^2 \\ +\frac{1}{4}(f_1(\overline{x}_k))^2 f_1''(\overline{x}_k)(\Delta w_{k+1})^3 + O((\Delta w_{k+1})^4)\end{aligned}$$

The true value of x_{k+1} given $x_k = \overline{x}_k$ is

$$\begin{aligned}x_{k+1} = \overline{x}_k + f_1(\overline{x}_k)\Delta w_{k+1} + \frac{1}{2}f_1(\overline{x}_k)f_1'(\overline{x}_k)(\Delta w_{k+1})^2 \\ +\frac{1}{6}\left[(f_1(\overline{x}_k))^2 f_1''(\overline{x}_k) + (f_1(\overline{x}_k)f_1'(\overline{x}_k))^2\right](\Delta w_{k+1})^3 + O((\Delta w_{k+1})^4)\end{aligned}$$

Then our error estimate is

$$\frac{1}{12}(2f_1 f_1'^2 - f_1^2 f_1'')(\overline{x}_k)(\Delta w_{k+1})^3 + O((\Delta w_{k+1})^4)$$

With $f_1(x) = \beta x$ the error estimate is

$$y(t_k) = \left|\frac{1}{6}\beta^3 \overline{x}_k (\Delta w_{k+1})^3\right|$$

For this particular equation, the error as given by (2.1) is solution of the same SDE again:

$$dy = \beta y \circ dw$$

Let $y(t)$ be the error in x at an intermediate time t, $0 < t < T$. Then that error will lead to a final error $y(T)$ given by

$$y(T) = y(t)e^{\beta(w(T)-w(t))} = \frac{y(t)x(T)}{x(t)}$$

If we wish the local error at each step to contribute equally to the final global error, we should require

$$\left|\frac{y(t)x(T)}{x(t)}\right| < \epsilon$$

hence

$$|y(t)| < \epsilon \left|\frac{x(t)}{x(T)}\right|$$

The accuracy criterion is now

$$\frac{1}{6}|\beta^3 \overline{x}_k (\Delta w_{k+1})^3| < \frac{\epsilon}{|x(T)|}\,|\overline{x}_k|$$

or

$$\left|(\Delta w_{k+1})^3\right| < C$$

We give below in table form the results obtained for $\beta = 5$ and $x(0) = 1$ along one particular Brownian path, using first fixed step sizes of various lengths (Table 2–1) and then variable step sizes with different accuracy constraints (Tables 2–2–2–4). In all cases the discretisation method used is the Heun scheme.

In order to compare the relative performance of the fixed and variable step size methods, we report the number of steps taken in each case. For the variable step size method two numbers are given: the number of steps performed successfully (ie. that meet the chosen accuracy criterion) and the total number of steps tried, including those that were too large for the required accuracy and so had to be repeated using a smaller step size. The first number gives an indication of any possible reduction in the number of steps required to reach a final point with a given accuracy, once all steps do not need to be of equal length. The second number indicates how much work the variable step size routine performs (ie the number of function evaluations etc.) and therefore gives an indication of the computing time involved as compared with the time used by the fixed step size routine.

The accuracy criteria used with variable step sizes to produce the results in Tables 2–2, 2–3 and 2–4 below are, in order,

$$|\overline{x}_k - \tilde{x}_k| < C \tag{2.15}$$

$$|\overline{x}_k - \tilde{x}_k| < C|\overline{x}_{k-1}| \tag{2.16}$$

$$|\Delta w^3| < C \tag{2.17}$$

Table 2–1: Fixed step size

Step size	$x(1)$	No. steps	Error (%)
2^{-7}	34379.89	128	80
2^{-8}	34682.04	256	81
2^{-9}	31709.06	512	66
2^{-10}	22217.40	1024	16
2^{-11}	20340.67	2048	6
2^{-12}	19807.45	4096	3.5
2^{-13}	19394.61	8192	1.4

Table 2–2: Variable step size using $|\overline{x}_k - \tilde{x}_k| < C$

C	$x(1)$	No. steps taken	No. steps tried	Error (%)
10.0	31269.71	370	1680	63
1.0	27048.22	1505	6834	41
0.1	21959.87	6072	27303	15

Table 2–3: Variable Step Size using $|\overline{x}_k - \tilde{x}_k| < C|\overline{x}_{k-1}|$

C	$x(1)$	No. steps taken	No. steps tried	Error (%)
0.1	31226.03	38	168	63
0.01	31347.92	202	900	64
0.001	20598.17	809	3615	7.6
0.0007	19444.64	1078	4812	1.6

For the path used $w(1) = 1.971848$, hence the true solution is $x(1) = 19134.34$.

The final table for this example, Table 2–5, reports average figures over fifty different Brownian paths. The numbers of steps are the numbers needed to obtain at least 2% accuracy in $x(1)$. The third row of the table compares the number of steps taken using variable step sizes with the number using a fixed step size. The

Table 2–4: Variable step size using $|\Delta w^3| < C$

C	$x(1)$	No. steps taken	No. steps tried	Error (%)
0.1	6614.52	28	31	65
0.01	25456.50	56	69	33
0.001	20159.48	206	262	5.4
0.0001	19518.13	958	1216	2.0
0.00009	19353.70	1026	1302	1.1

Table 2–5: Comparison of Methods

	Fixed	VS (2.15)	VS (2.16)	VS (2.17)
Steps taken:	16998	18024	6970	3101
Steps tried:	16998	35864	15733	3956
Steps taken/fixed:	1.00	2.11	0.93	0.23
Taken/tried:	1.00	0.50	0.44	0.78

last row shows the proportion of steps accepted as meeting the relevant accuracy criterion to the total number of steps tried. The third accuracy criterion, which is the one described in detail above, is clearly the most efficient. The total number of steps taken and the proportion of steps tried but rejected are both lower for this method than for any other method tried.

2.5.2 Second Example

The second example is similar to the first, but with $n = 2$ (and $d = 1$ again). We take the system of Stratonovich equations

$$\begin{cases} du &= \beta^2 v \circ dw \\ dv &= u \circ dw \end{cases} \tag{2.18}$$

with initial conditions

$$\begin{cases} u(0) &= 0 \\ v(0) &= 2 \end{cases}$$

(We have taken $x = (x^1, x^2) = (u, v)$, to simplify notation.) The solution can be calculated analytically as

$$\begin{cases} u(t) &= -\beta e^{-\beta w(t)} + \beta e^{\beta w(t)} \\ v(t) &= e^{-\beta w(t)} + e^{\beta w(t)} \end{cases}$$

As in the first example, the solution can be approximated as accurately as desired using a $\mathcal{P}_N$ approximation, so asymptotic efficiency is not really relevant.

Let y and z be the errors made in u and v respectively. Then, replacing u and v in (2.18) with $u + \epsilon y$ and $v + \epsilon z$ and taking the limit as ϵ tends to zero, we see that, as in the previous example, y and z satisfy the original system :

$$\begin{cases} dy &= \beta^2 z \circ dw \\ dz &= y \circ dw \end{cases}$$

Hence

$$\begin{cases} y(t) &= -\beta A e^{-\beta w(t)} + \beta B e^{\beta w(t)} \\ z(t) &= A e^{-\beta w(t)} + B e^{\beta w(t)} \end{cases}$$

If we are interested in the contribution at time T of errors $y(t)$ and $z(t)$ introduced at time t, then we must take

$$\begin{aligned} A &= \frac{\beta z(t) - y(t)}{2\beta} e^{\beta w(t)} \\ B &= \frac{\beta z(t) + y(t)}{2\beta} e^{-\beta w(t)} \end{aligned}$$

giving

$$\begin{cases} y(T) &= \frac{y(t)}{2} E_1 + \frac{z(t)}{2} \beta E_2 \\ z(T) &= \frac{z(t)}{2} E_1 + \frac{y(t)}{2\beta} E_2 \end{cases}$$

where

$$\begin{aligned} E_1 &= e^{-\beta(w(T)-w(t))} + e^{\beta(w(T)-w(t))} \\ E_2 &= e^{\beta(w(T)-w(t))} - e^{-\beta(w(T)-w(t))} \end{aligned}$$

It can be shown straightforwardly that

$$\begin{aligned} E_1 &= \frac{\beta^2 v(T)v(t) - u(T)u(t)}{2\beta^2} \\ E_2 &= \frac{u(T)v(t) - v(T)u(t)}{2\beta} \end{aligned}$$

Therefore errors introduced at time t lead to errors at time T given by

$$\begin{cases} y(T) &= \frac{1}{4\beta^2}[y(t)(\beta^2 v(T)v(t) - u(T)u(t)) + \beta^2 z(t)(u(T)v(t) - v(T)u(t))] \\ z(T) &= \frac{1}{4\beta^2}[y(t)(u(T)v(t) - v(T)u(t)) + z(t)(\beta^2 v(T)v(t) - u(T)u(t))] \end{cases} \tag{2.19}$$

For the Heun method our error estimate for x^i, $i = 1, 2$ is

$$\frac{1}{12}\left(2\frac{\partial f_1^i}{\partial x_j}\frac{\partial f_1^j}{\partial x_k}f_1^k - \frac{\partial f_1^i}{\partial x^j \partial x^k}f_1^j f_1^k\right)(x^i(t))(\Delta w_{k+1})^3 + O((\Delta w_{k+1})^4)$$

giving

$$y(t) = \left|\frac{1}{6}\beta^4 v(t)(\Delta w_{k+1})^3\right|$$
$$z(t) = \left|\frac{1}{6}\beta^2 u(t)(\Delta w_{k+1})^3\right|$$

Since u is really only an intermediary variable and it is v that interests us, we only demand accuracy in v. Requiring as before that the local error at each step contribute equally to the final error, we obtain an accuracy criterion for each step of

$$\frac{1}{24}\left|u(T)(\Delta w_{k+1})^3\left[\beta^2 v(t)^2 - u(t)^2\right]\right| < C$$

or

$$\left|(\Delta w_{k+1})^3\left[\beta^2 v(t)^2 - u(t)^2\right]\right| < C \tag{2.20}$$

We present below in table form the comparative results of solving the system (2.18) using fixed and variable time steps. As for the first example, the discretisation method used is that of Heun. We have taken $\beta = 4$. Tables 2–6 and 2–7 give detailed results using one particular Brownian path, for which $w(1) = 0.080107$, giving $x^2(1) = 2.10356$. The last table, Table 2–8 again compares the efficiency of fixed and variable step sizes over fifty Brownian paths.

Motivated by the results obtained in example 1, only one accuracy criterion has been used to control step size, the criterion (2.20) derived above.

Table 2–6: Fixed Step Size

Step size	$x^2(1)$	No. steps	Error (%)
2^{-7}	4.34977	128	107
2^{-8}	2.94662	256	40
2^{-9}	2.53534	512	21
2^{-10}	2.28233	1024	8.5
2^{-11}	2.19512	2048	4.4
2^{-12}	2.14895	4096	2.2
2^{-13}	2.12498	8192	1.0

Table 2–7: Variable Step Size

c	$x^1(1)$	No. steps taken	No. steps tried	Error (%)
0.1	2.45227	192	246	17
0.01	2.16454	684	867	2.9
0.002	2.12298	1975	2529	0.9

Table 2–8: Comparison of fixed and variable steps

	Fixed step size	Variable step size (2.20)
Steps taken:	5530	738
Steps tried:	5530	927
Prop of fixed:	1.00	0.17
Taken/tried:	1.00	0.80

2.5.3 Third Example

For our final example we have chosen an SDE that is of one dimension in x but two dimensions in w ($n = 1$, $d = 2$),

$$d\theta = cos(\theta)dw^1 + sin(\theta)dw^2 \tag{2.21}$$

This is the 'gradient flow' on the circle mentioned by Rogers and Williams [53]. It is easily verified that the Itô and Stratonovich forms of this equation are identical and that the equation does not satisfy the commutative property.

We will use the method of solution suggested in Section 2.4.2, that involves approximating the Lévy areas by generating increments of the Brownian motion over finer time steps than the steps of the numerical solution, and that reduces the order of the mean one step error.

For this method, with $d = 2$, the variance of the local error generated in one step conditional on (all) the increments of the Brownian path is

$$\begin{aligned} &E[(x_{k+1} - \overline{x}_{k+1})^2 | \Delta w_{k+1}] = \\ &\frac{1}{12}(f_1'(x_k)f_2(x_k) - f_2'(x_k)f_1(x_k))^2 \tilde{h} \left[h + \sum_{i=1}^{K}((\Delta w_i^1)^2 + (\Delta w_i^2)^2) \right] \end{aligned} \tag{2.22}$$

where K is the number of equal subintervals that the time step has been divided into for the purpose of approximating the area integrals and Δw_i denotes the increment of the Brownian path over the ith subinterval. The mean of the local error is $O(h^{3/2})$.

In the case of the SDE (2.21) the local variance is

$$v_k = \frac{1}{12}\tilde{h}(h + \sum_{i=1}^{K}(\Delta w_i^1)^2 + \sum_{i=1}^{k}(\Delta w_i^2)^2) \tag{2.23}$$

From (2.1), the propagation of the local error is given by

$$dy = y(cos(\theta)dw^2 - sin(\theta)dw^1) \tag{2.24}$$

This equation cannot be solved analytically, so one way to proceed is to solve it numerically simultaneously to equation (2.21), giving the system

$$\begin{cases} d\theta &= cos(\theta)dw^1 + sin(\theta)dw^2 \\ dy &= y(cos(\theta)dw^2 - sin(\theta)dw^1) \end{cases} \tag{2.25}$$

with $y(0)$ chosen arbitrarily. If an error $e(t)$ is introduced at time t, then this will lead to a contribution to the final error at time T of

$$e(T) = \frac{y(T)}{y(t)} e(t)$$

and the variance, $v(t)$, of the local error will give a contribution to the variance of the final error of

$$v(T) = \left(\frac{y(T)}{y(t)}\right)^2 v(t)$$

which in this example becomes, from (2.23),

$$v(T) = \frac{1}{12}\left(\frac{y(T)}{y(t)}\right)^2 \tilde{h}(h + \sum_{i=1}^{K}(\Delta w_i^1)^2 + \sum_{i=1}^{K}(\Delta w_i^2)^2)$$

Hence we have an accuracy criterion of

$$\frac{1}{12}\left(\frac{y(T)}{y(t)}\right)^2 \tilde{h}(h + \sum_{i=1}^{K}(\Delta w_i^1)^2 + \sum_{i=1}^{K}(\Delta w_i^2)^2) < C$$

or

$$y(t)^{-2}\tilde{h}(h + \sum_{i=1}^{K}(\Delta w_i^1)^2 + \sum_{i=1}^{K}(\Delta w_i^2)^2) < C \tag{2.26}$$

There is a problem here: the accuracy criterion (2.26) includes Δw_i, $i = 1, \ldots, k$, but an efficient routine to generate and access the Brownian path will only return the increment of the Brownian path and the approximate Lévy area over each time step, not the increments over smaller steps used to approximate the integrals. The best way round this, seems to be to approximate the sums of $(\Delta w_i^j)^2$ by h.

Let

$$S_j = \sum_{i=1}^{K}(\Delta w_i^j)^2$$

The mean error, conditional on Δw_{k+1}, made by replacing S_j with h is

$$\mu = E[S_j - h|\Delta w_{k+1}] = \frac{1}{K}((\Delta w_{k+1})^2 - h)$$

and the conditional variance of the error can be shown to be

$$E[(S_j - h - \mu)^2|\Delta w_{k+1}] = \frac{4h^2}{K} - \frac{h^2}{K^2}$$

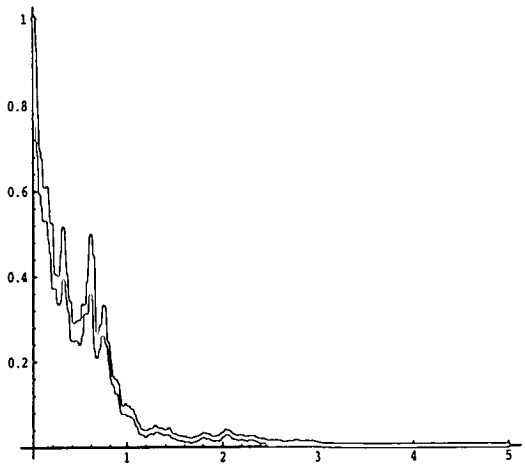


Figure 2–4: error propagation

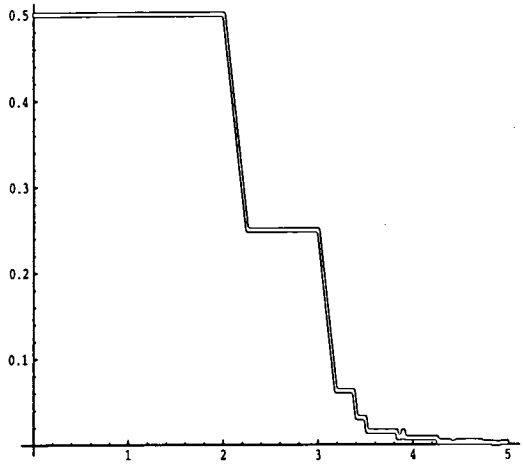


Figure 2–5: step size against time

Therefore, if we wish (2.26) to be satisfied with a certain probability, when approximating S_j, $j = 1, 2$, by h, we need to replace C in (2.26) with $C - \alpha h/\sqrt{K}$, giving the final criterion

$$y(t)^{-2}\tilde{h}h + \alpha\frac{h}{\sqrt{K}} < C \tag{2.27}$$

The choice of $\alpha \in I\!R$ depends on the probability with which we want (2.26) to hold.

As can be seen in Figure 2–4, the path followed by local errors suggests exponential decay. Using criterion (2.27) to control step size therefore results in initially large steps decreasing more or less exponentially in size with time. The time steps obtained for one particular Brownian path are shown in Figure 2–5.

2.6 Implementation

As mentioned above, we have chosen to generate approximations to Brownian paths using a tree-like structure and to store all parts of the tree generated in a data file. The file consists of a set of records, stored in the order in which they are generated. The first record in the file is unique and contains basic information about the whole file, such as the total number of records, the dimension of the Δw

vectors, the number M of vectors in each record, etc. Each of the other records consists of two blocks of data of fixed length and structure. The first block is a set of "labels", that define clearly the position in the tree of the set of Δw vectors contained in the record and that also specify where certain other nodes of the tree can be found, such as the parent node, the right and left children and the previous and next nodes on the same level of the tree. The labels are particularly important when it comes to the branches of the Brownian tree that are added as needed during the solution of the SDE, as these do not constitute full levels and therefore must be properly linked to the rest of the tree. There are several possible sets of labels that could be used, from a minimal set to much more extensive ones. Here a balance needs to be reached between the amount of data stored and the time needed for accessing data within the file. Although it might seem sensible not to store more in the file than necessary, much computing time could be lost searching through the file for the required record, when a label in the last record used could have given the location.

The second block of data contained in each record is a set of M Δw vectors constituting a node in the Brownian tree. The parameter M is chosen suitably large, allowing enough of the Brownian path to be kept in working memory at any one time to minimise the number of file accesses required, but not so large that paging is forced when using virtual memory.

At the beginning of the numerical solution, an area of working memory is initialised. A particular number of records is read from the file and stored in memory. Throughout the numerical integration the number of records kept in memory will remain unchanged. (This parameter must again be chosen, in conjunction with M, in such a way as to minimise access time.) The first record taken from the file is the one containing the Δw vector needed initially by the integration process. The other records are all parents, grandparents etc. of the first one. In this way if, a bit later in the solution, the step size should increase, the necessary nodes will already be present in memory.

After initialisation, each time that a path element is required, the first place to search for it is in memory. If the algorithm is tuned properly, then a large

proportion of the time the search will end there. When the required node is not in memory, then either the node is contained in the file, in which case the relevant record is read in and replaces another record already in memory, or else the node has not yet been generated and existing records are used to generate new nodes. In some instances, generation of one node demands generation of nodes on one or more higher levels. All newly generated nodes are placed in records along with the correct labels and appended to the file containing the Brownian tree.At the same time, in order to preserve the tree structure, other already existing records need to be updated. For example, parent records of newly generated records must be corrected to include pointers to their children.

The rules we have chosen state that the nodes kept in memory must at any time all be from different levels of the tree and be direct descendants of the one on the highest level. Therefore, when any one record is replaced, other records may also need to be replaced.

At the end of each step in the numerical integration, a decision is made as to whether in the next time interval the step size should be increased, decreased or remain unchanged. When accuracy tests suggest that the step size could be increased, it is not always possible to do so immediately. This is a question of data structure. An increase in step size can only be allowed when the current time is the initial point of a subinterval of length the larger (double) step size. For example, if an initial step of length 0.125 is taken at time $t = 0$, then the second step, taken at time $t = 0.125$ cannot be of length 0.25, since the Δw elements corresponding to steps of that length are only generated over the intervals $[0, 0.25]$,$[0.25, 0.5]$ .... The correct procedure at this point is to take another step of the previous size and to then increase the step size, if the situation still warrants it. On the other hand, step size can be decreased at any point in time.

The computer program written in C, corresponding to the method described here, is given in Appendix A.1.

Chapter 3

Random Generation of Stochastic Area Integrals

3.1 Definition of the problem

We have seen, in Chapter 1, that one way to obtain higher order approximations to strong solutions of the stochastic differential equation (1.1) or (1.2) is to simulate not only increments

$$\Delta w_{k+1}^i = w^i(t_{k+1}) - w^i(t_k) \tag{3.1}$$

along the Brownian paths, but also stochastic integrals involving the Brownian motion. To obtain accuracy of order $O(h)$ it suffices to generate and include in the numerical scheme the area integrals, as defined in (1.17),

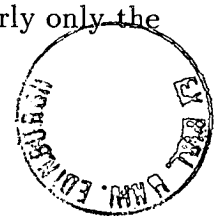
$$A_{i,j}(k,k+1) = \int_{t_k}^{t_{k+1}} \int_{t_k}^{s} dw^i(r)dw^j(s) - \int_{t_k}^{t_{k+1}} \int_{t_k}^{s} dw^j(r)dw^i(s), \quad i = 1,\ldots,n; j > i$$

where, $t_{k+1} - t_k = h$, $k = 0,\ldots,N-1$,

This chapter describes a method of random generation of the integrals

$$A_{1,2}(k,k+1) = \int_{t_k}^{t_{k+1}} \int_{t_k}^{s} dw^1(r)dw^2(s) - \int_{t_k}^{t_{k+1}} \int_{t_k}^{s} dw^2(r)dw^1(s)$$

Use of these integrals allows first-order approximation of strong solutions to any SDE based on a two-dimensional Brownian path. This, although clearly only the



beginning of the story, is a definite improvement. The work in this chapter has been published in [18].

Since the increments of the Brownian path, Δw_{k+1}^1 and Δw_{k+1}^2 are clearly not independent of the area integrals $A_{1,2}(k, k+1)$, the increments and the areas have to be generated jointly. At first sight it seems that we are faced with the generation of a random vector in $\mathbb{R}^3$, but in fact the problem can be reduced to the generation of a random vector in $\mathbb{R}^2$ and an independent uniformly distributed random variable, as shown by the following theorem.

Theorem 3.1.1 *(Lévy) Let* $r^2(t) = \Delta w^1(0,t)^2 + \Delta w^2(0,t)^2$. *The joint density function of* $r = r(1)$ *and* $a = A_{1,2}(0,1)$ *is*

$$f(r,a) = \frac{r}{\pi}\int_0^\infty \frac{x}{\sinh(x)} \exp\left(\frac{-r^2 x}{2\tanh(x)}\right)\cos(ax)dx \tag{3.2}$$

Proof. We will follow the proof given by Lévy in [31], where he obtains the density of the area integral. The essential ingredient is that $w^1(t)$ and $w^2(t)$ can be given by almost surely convergent Fourier series, as proved by Wiener in [57]. We can write

$$w^1(t) = \frac{t\bar{\xi}}{\sqrt{2\pi}} + \sum_{n=1}^{\infty}\frac{1}{n\sqrt{\pi}}\left[\xi_n(\cos nt - 1) + \bar{\xi}_n \sin nt\right] \tag{3.3}$$

$$w^2(t) = \frac{t\bar{\eta}}{\sqrt{2\pi}} + \sum_{n=1}^{\infty}\frac{1}{n\sqrt{\pi}}\left[\eta_n(\cos nt - 1) + \bar{\eta}_n \sin nt\right] \tag{3.4}$$

where $\bar{\xi}, \xi_1, \xi_2, \ldots, \bar{\xi}_1, \bar{\xi}_2, \ldots, \bar{\eta}, \eta_1, \eta_2, \ldots, \bar{\eta}_1, \bar{\eta}_2, \ldots$ are all independent standard normal random variables. For convenience of calculation, we will derive the joint characteristic function of $w^1(2\pi), w^2(2\pi), A_{1,2}(0,2\pi)$. Let $\bar{a} = A_{1,2}(0,2\pi)$. By straightforward calculation we get

$$w^1(2\pi) = \sqrt{2\pi}\bar{\xi} \tag{3.5}$$

$$w^2(2\pi) = \sqrt{2\pi}\bar{\eta} \tag{3.6}$$

$$\bar{a} = 2\sum_{n=1}^{\infty}\frac{1}{n}\left[\xi_n(\bar{\eta}_n - \bar{\eta}\sqrt{2}) - \eta_n(\bar{\xi}_n - \bar{\xi}\sqrt{2})\right] \tag{3.7}$$

Set

$$(r(t))^2 = (w^1(t))^2 + (w^2(t))^2$$

We will first consider the conditional characteristic function of $\bar{a}$, given $\bar{\xi}\sqrt{2} = \alpha$, $\bar{\eta}\sqrt{2} = \beta$, that is given $r^2(2\pi) = (\alpha^2 + \beta^2)\pi$. When ξ is standard normal

$$I\!E(e^{iu\xi}) = e^{-u^2/2}$$

so, if η is also standard normal, the characteristic function of $\xi(\eta - \beta)$ is

$$I\!E\left(e^{i\xi(\eta-\beta)z}\right) = I\!E\left(e^{-(\eta-\beta)^2z^2/2}\right) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} e^{\left[(y-\beta)^2z^2+y^2\right]/2}\,dy = \frac{1}{\sqrt{1+z^2}}e^{-\beta^2z^2/(2(1+z^2))}$$

and therefore the conditional characteristic function of $\xi_n(\bar{\eta}_n - \beta) - \eta_n(\bar{\xi}_n - \alpha)$ is

$$\frac{1}{1+z^2}e^{-\rho^2z^2/(1+z^2)}$$

where $\alpha^2 + \beta^2 = 2\rho^2$. The conditional characteristic function of $\bar{a}$ is then

$$\phi(z|\rho^2) = \left\{\prod_{n=1}^{\infty}\frac{n^2}{n^2+4z^2}\right\}\exp\left[-\rho^2\sum_{n=1}^{\infty}4z^2/(n^2+4z^2)\right]$$

which can be written

$$\phi(z|\rho^2) = \frac{2\pi z}{\sinh(2\pi z)}\exp\left[\rho^2(1 - 2\pi z\coth(\pi z))/2\right]$$

The density of $\bar{a}$ conditional on ρ^2 is

$$f(\bar{a}|\rho^2) = \frac{1}{2\pi}\int_{-\infty}^{\infty} e^{-iz\bar{a}}\phi(z|\rho^2)\,dz$$

Since the density is real, we have

$$f(\bar{a}|\rho^2) = 2\int_0^{\infty}\cos(z\bar{a})\frac{z}{\sinh(2\pi z)}\exp\left[\rho^2(1 - 2\pi z\coth(2\pi z))/2\right]\,dz$$

Since ρ^2 is the sum of the squares of 2 standard normal random variables, its density function is $\frac{1}{2}\exp(-\rho^2/2)$. The (unconditional) joint density of $\bar{a}$ and ρ^2 is therefore

$$f(\bar{a},\rho^2) = \int_0^{\infty}\cos(z\bar{a})\frac{z}{\sinh(2\pi z)}\exp\left[-\rho^2\pi z\coth(2\pi z)\right]\,dz$$

Replacing $2\pi z$ with x, the joint density of ρ and $\bar{a}$ becomes

$$f(\bar{a},\rho) = \frac{\rho}{2\pi^2}\int_0^{\infty}\cos(\frac{x\bar{a}}{2\pi})\frac{x}{\sinh(x)}\exp\left[-\rho^2x\coth(x)/2\right]\,dx$$

The random variables ρ and $r(1)$ have the same distribution, so ρ can be replaced with $r(1)$ in the above expression, but $\bar{a} = A_{1,2}(0, 2\pi)$ is distributed like $2\pi a$, so it remains to replace $\bar{a}$ with $2\pi a$ to obtain the required joint density of a and r. □

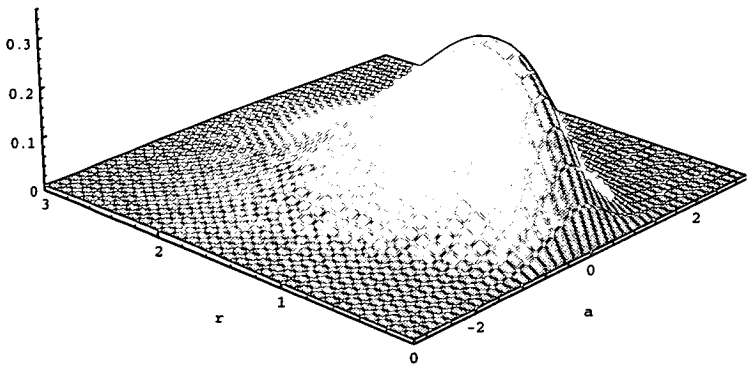


Figure 3–1: The graph of $f(r, a)$

The problem has become that of generating the vector (r, a) with the distribution given in (3.2). It is then trivial to obtain correctly distributed $\Delta w^1(0,1)$ and $\Delta w^2(0,1)$ by generating θ uniformly distributed on $[0, 2\pi]$ and taking

$$\Delta w^1(0,1) = r\cos\theta$$
$$\Delta w^2(0,1) = r\sin\theta$$

See Figure 3–1 for the graph of $f(r, a)$.

Note that we can generate (r, a) over unit time steps in the first instance. Eventual scaling of a by h and r by $\sqrt{h}$ suffices to produce a sequence of vectors $(\Delta w^1(t, t+h), \Delta w^2(t, t+h), A_{1,2}(t, t+h))$ for any required time step h.

Another obvious simplification consists of generating only the half of the distribution that corresponds to $a > 0$ and then giving a a random sign.

3.2 Outline of the Method

The integral in (3.2) can only be calculated numerically, so there is no 'quick and easy' method of generation available. The method we have chosen is based on Marsaglia's 'rectangle-wedge-tail' method, generalised to higher dimensions (see Marsaglia [38] and [37], or Knuth [29] for an outline of the method).

In one dimension Marsaglia's method involves dividing an area in $\mathbb{R}^2$ into equal rectangles and setting up tables with an entry for each rectangle. With

modern computers, the amount of memory used for storage of the tables is not large. However, once we are in three dimensions, and start dividing a region in $\mathbb{R}^3$ into equal pieces, the number of table entries needed becomes prohibitive. We have therefore been forced to a slightly more sophisticated analysis of the method to reduce storage requirements while retaining benefits of speed. The final implementation enables one to generate the vector (a, b, c) in about 4.6 times the time it takes to generate a vector of 3 independent numbers from a normal distribution.

Following Marsaglia's rectangle-wedge-tail method, the aim is to express the required density function $f(r, a)$ as a combination of three other densities,

$$f(r,a) = p_1 f_1(r,a) + p_2 f_2(r,a) + p_3 f_3(r,a) \tag{3.8}$$

where p_1, p_2, p_3 are probabilities that sum to 1, p_1 is as close to 1 as possible and the time needed for generating numbers from the distribution corresponding to $f_1(r, a)$ is very small. The distributions corresponding to $f_2(r, a)$ and $f_3(r, a)$ may be hard to generate, but they will be used sufficiently seldom that the average running time for the whole routine will remain acceptably low.

The volume

$$V = \{(r,a,z) | z < f(r,a), 0 \leq r \leq r_M, 0 \leq a \leq a_M\} \tag{3.9}$$

where r_M and a_M are chosen suitably large, is packed with as many parallelopipeds, $B_i = \{(r,a,z) | r_i^1 < r < r_i^2, a_i^1 < a < a_i^2, z_i^1 < z < z_i^2\}, i = 1 \ldots N$,as feasible. These will be referred to as 'boxes'. (The boxes correspond to the rectangles in Marsaglia's method for generating a single random variable.) The volume under $f_1(r, a)$ is defined to be the total volume occupied by all the boxes. The 'easy' density $f_1(r, a)$ is therefore a sum of uniform densities:

$$f_1(r,a) = \sum_i^N q_i \phi_i(r,a)$$

where ϕ_i is the density of points in B_i, q_i is the probability of a point (r, a) lying in box B_i (so q_i is twice the volume of B_i,) and N is the number of boxes used.

The 'wedges' are then the pieces left above the surface $f_1(r, a)$ and below $f(r, a)$. The 'tail' is the set of points $T = \{(r,a,z) | z < f(r,a)$ and $(r > r_M$ or $a > a_M)\}$.

The densities $f_2(r,a)$ and $f_3(r,a)$ are the densities of points in the wedges and in the tail respectively.

3.3 The Boxes

Determining the density function $f_1(r,a)$ involves first packing the volume below the surface $f(r,a)$ with boxes and then grouping the boxes in order to reduce both memory requirements and execution time of the code developed.

3.3.1 The packing problem

The problem is to pack the volume V, defined in (3.9), with boxes in such a way as to maximise the total volume occupied and satisfy the various accuracy and programming constraints imposed at later stages of the exercise.

The first decision taken is to set $r_M = a_M = 4$ in (3.9). This gives a volume for V of 0.49866, containing 99.732% of the distribution to be generated.

The next decision is to divide the volume under $f(r,a)$ into boxes in such a way that the dimensions of each box (length, width and height) are all integer multiples of a chosen $l = 2^{-n}$. In this way the volume of each box and hence the probability of a point being in it, can be represented on the computer as a binary number using $3n$ bits, introducing no rounding errors at this stage.

A computer program was written to generate the required boxes. The volume V was subdivided using an increasingly fine mesh. As a first step the r,a and z axes were divided into intervals $\Delta r_1 = 2^{-n_r}$, $\Delta a_1 = 2^{-n_a}$ and $\Delta z_1 = 2^{-n_z}$. Each box defined by this mesh was tested for complete inclusion in V. The boxes found to be in V were labeled as being part of the chosen partition. In the next step the intervals were halved, giving $\Delta r_2 = 2^{-n_r-1}$ etc. and the testing repeated for all boxes not already accepted. This procedure was repeated, halving the intervals at each step, until such time as a large enough fraction of the volume had been used up or the number of boxes defined was as large as thought practical. The

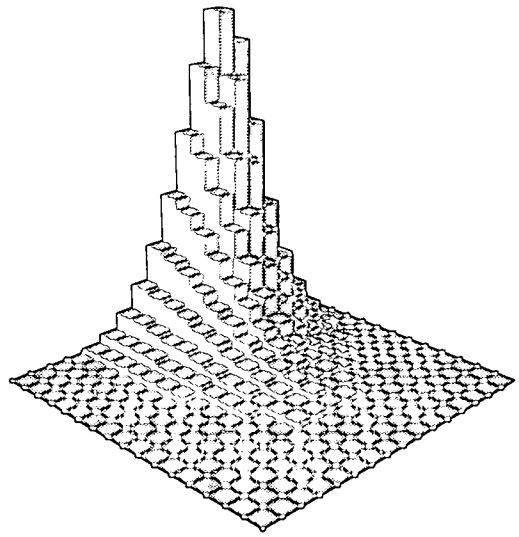


Figure 3–2: The packing at step 3

values of n_r, n_a, n_z, the volume occupied by boxes and the total number of boxes can be thought of as the parameters of the program. Another possible parameter, the origin of the mesh, was fixed at $(0,0,0)$ for the sake of simplicity. After experimentation, the parameter values considered acceptable were $n_r = n_a = 0$, $n_z = 6$, with a total of 13574 boxes making up 91.19% of the volume to be generated. This corresponds to $p_1 = 0.9119$ in (3.8). The numbers of boxes for each mesh size are as follows:

Step:	1	2	3	4	5
$\Delta r = \Delta a$:	1	1/2	1/4	1/8	1/16
Δz :	1/64	1/128	1/256	1/512	1/1024
Number of boxes:	3	86	519	2431	10535

Since the dimensions used in the final mesh were $\Delta r_5 = \Delta a_5 = 2^{-4}$, $\Delta z_5 = 2^{-9}$, 17 bits are sufficient to identify each box. The 13574 boxes of various sizes can be broken up into 119519 boxes of the smallest size.

Figure 3–2 shows all the boxes that have been allocated by the end of the third step, broken into boxes of size Δr_3 by Δa_3 by Δz_3.

Note that rather than using a decreasing sequence of meshes, we could simply use one mesh, that with the smallest mesh size. We would obtain the same density function $f_1(r,a)$. Using the parameter values specified above, $f_1(r,a)$ would be expressed as the sum of the densities over the 119519 boxes of smallest size. The

reason for using the multi-step approach is that it facilitates the next part of the exercise, which involves grouping the boxes together to form larger ones.

During this first part of the exercise, the first questions arose on the generation of multivariate random deviates using Marsaglia's method. If we fix the percentage of volume that we wish to fill with boxes, how is the number of boxes required related to the joint distribution function? Presumably the number of boxes increases with the surface area?

3.3.2 The entropy problem

The boxes packed under the surface $f(r, a)$ need to be grouped and ordered in an efficient fashion.

Whereas in a two-dimensional problem it is these days straightforward to subdivide the required area into identical small rectangles and keep a separate record in memory for each rectangle, in a three-dimensional setting the number of identical boxes is so large that the memory requirements would be prohibitive. In our example 119519 records would be required. We therefore decided to group boxes together in such a way that each subset of boxes itself formed a box.

When grouping boxes an effort was made not only to reduce the number of boxes but also to reduce the entropy of the partition. If it were possible to use only boxes of equal volume, then a random number consisting of the right number of bits (17 in our application) would uniquely identify the correct box to use. However, once the boxes vary in volume, a set of tests is required to determine which box each point falls in and therefore which $\phi_i(r, a)$ density function to use. The tests form a binary decision tree, which can be constructed in an optimal (time minimising) way using Huffman's method (see [22] for the original paper or Knuth [28] for a description). This method guarantees an average number of tests performed no greater than one plus the entropy of the partition. The entropy was therefore taken into account when choosing a partition. The entropy can be written as

$$-\sum_{i}^{N} q_i log_2(q_i)$$

where q_i, the probability of a point being in box B_i, is the volume of the box divided by the total volume occupied by boxes.

One possible way of grouping the boxes would be in columns, ie

$$B_{i,j} = \{(r,a,z)|i\Delta r \leq r < (i+1)\Delta r, j\Delta a \leq a < (j+1)\Delta a, 0 \leq z < h_{i,j}\},$$
$$i = 0 \ldots r_M/\Delta r, j = 0 \ldots a_M/\Delta a$$

where $h_{i,j}$ is the total height of boxes piled up on the square

$$\{(r,a)|i\Delta r \leq r < (i+1)\Delta r, j\Delta a \leq a < (j+1)\Delta a\}$$

With $\Delta r = \Delta a = 1/16$ and $r_M = a_M = 4$ there would be 4096 such columns, but if only columns with non-zero height are considered the number goes down to 2894. The entropy for this partition is however quite high, namely 10.13 for the chosen packing.

Another method of grouping the boxes, the one that was eventually chosen, is the following. In the first instance generate boxes of several sizes, using the method outlined in Section 3.3.1 above. This is equivalent to replacing groups of small boxes with larger ones. Then in the second step group together any boxes of the same size that are piled up one on top of another. This creates pieces of columns, that are in general shorter than the columns described above, but many of which have larger cross sections. In this way we reduced the number of boxes from the 13574 in Section 3.3.1 to 2975, giving a partition with entropy 7.14 .

Once the partition has been chosen, it remains to build the decision tree needed for choosing a box with the correct probability. A computer program was written to input the partition and output the decision tree using Huffman's method. Further software details can be found in the next section.

There is no reason to believe that, given the packing, the chosen partition has minimum entropy. Designing an algorithm to generate a partition with minimum or nearly minimum entropy demands a substantial amount of further work. We plan to attempt this some time in the future, and therefore to improve on the performance of the present software and perhaps extend it to higher dimensions.

We conjecture that the problem of devising a partition with minimum entropy in three or more dimensions may be NP-complete. Other questions arise for general multivariate distributions, such as asking what function of the distribution the minimal entropy is.

3.3.3 Programming details

The part of the routine that generates points in the boxes, that is points with the density $f_1(r, a)$, is only a few lines of executable code that relies on the data stored in one large binary tree. The tree, generated once and for all using Huffman's method, is stored in an array of records. The length of the array, equal to the number of nodes in the tree, is $2N - 1$, where N is the number of boxes (2975). The extra $N - 1$ nodes correspond to the tests needed to determine which box to use.

Each record holds the following data: a probability, the record numbers of the left and right children, the r and a coordinates of a corner of the box and the width of the box. (Note that in our application the length of each box is equal to the width. If this were not the case, the box length would also need to be given.) The records corresponding to boxes are at the ends of branches. They hold the value -1 for the record numbers of their non-existent children. The other records, those corresponding to decisions, hold zeroes in place of box coordinates and widths. This uses some extraneous memory, but produces relatively simple and fast code. On a computer with little memory it would be possible to use two separate arrays for the decisions and the boxes. We would like to investigate the trade-off between time and memory involved.

The 'probability' held at each node of the tree is cumulative and is expressed as an integer. It gives the numerator of a probability expressed as a fraction with denominator 2^n, where $n = 17$ is the number of random bits being used to determine the position.

The first step in the algorithm is to generate the n random bits, giving an integer p. If p is less than $p_B = 119519$, where $p_B/2^n$ is the total probability of

being in a box, then the point to be generated is in a box, otherwise it is in a wedge or in the tail. If the point is in a box, then the correct box must be chosen.

The search for the correct box starts at the root of the tree. If p is less than the probability stored in the left son, then the left branch is taken, otherwise the right branch is taken. This step is repeated until arrival at a terminal node.

The point (r, a) is then generated uniformly within the box, using the information on the position and dimensions of the box held in the node.

3.4 The Wedges

The region in the volume V, defined in (3.9) above, and outside the union of all the boxes, can be thought of as a set of disjoint pieces $W = \{w^i, i = 1 \ldots n_w\}$ that, in parallel with the usage in two dimensions and for lack of a better word, we will call wedges. The density function of the set W, $f_2(r, a)$, can therefore also be expressed in terms of n_w separate functions, one for each wedge. The wedges make up 8.55% of the whole distribution.

The packing of V with cubes, as described in Section 3.3.1, divides the square $\{(r, a, z)|z = 0, 0 \leq r \leq r_M, 0 \leq a \leq a_M\}$ into a number of identical rectangles, of sides Δr and Δa. With the values chosen of $r_M = a_M = 4$ and $\Delta r = \Delta a = 1/16$, we have $64^2 = 4096$ base squares, upon each of which there sits a column of boxes of total height $h_{i,j}$, $i, j = 1 \ldots 64$. The piece we are calling wedge is the set of points remaining above any one column and below the surface $f(r, a)$. There are therefore $n_w = 4096$ wedges.

We chose to generate the wedges using a simple rejection method. Each wedge can be enclosed in a box

$$C_{i,j} = \{(r, a, z)|(i-1)\Delta r \leq r \leq i\Delta r, (j-1)\Delta a \leq a \leq j\Delta a, h_{i,j} \leq z \leq \tilde{f}_{i,j}\}$$

where

$$\tilde{f}_{i,j} = \max\{f(r, a), (i-1)\Delta r \leq r \leq i\Delta r, (j-1)\Delta a \leq a \leq j\Delta a\}$$

Then, having first chosen a wedge with probability proportional to its volume, it is just a question of generating points uniformly distributed in the box containing the wedge and testing whether the point is under the surface $f(r,a)$. In case of failure, another point is generated in the box, and this is repeated until the test succeeds.

On average less than half the points generated in the boxes need to be rejected. However, if each test for acceptance of a point demanded evaluation by numerical integration of the function $f(r,a)$, this would be extremely costly in execution time. We have therefore chosen easy to calculate upper and lower approximations, $f^l_{i,j}(r,a)$ and $f^u_{i,j}(r,a)$, for $f(r,a)$ on each base square $r_i \le r \le r_{i+1}, a_j \le a \le a_{j+1}$ with $r_i = i\Delta r, a_j = j\Delta a$, $i,j = 1,\ldots,64$. All points (r,a) below $f^l(r,a)$ can be accepted without having to evaluate $f(r,a)$ and similarly all points above $f^u(r,a)$ can be immediately rejected. This is the squeeze method, so named by Marsaglia in [36].

The approximations to $f(r,a)$ are obtained by simple interpolation. Let $f^a_{i,j}(r,a)$ be defined by

$$\begin{aligned} f^a_{i,j}(r,a) = \; & (1-t)(1-u)f(r_i,a_j) + t(1-u)f(r_{i+1},a_j) \\ & + tuf(r_{i+1},a_{j+1}) + (1-t)uf(r_i,a_{j+1}) \end{aligned}$$

where

$$\begin{aligned} t &= (r-r_i)/(r_{i+1}-r_i) \\ u &= (a-a_j)/(a_{j+1}-a_j) \end{aligned}$$

Then we set

$$\begin{aligned} f^l_{i,j}(r,a) &= f^a_{i,j}(r,a) + \epsilon^1_{i,j} \\ f^u_{i,j}(r,a) &= f^a_{i,j}(r,a) + \epsilon^2_{i,j} \end{aligned}$$

with

$$\begin{aligned} \epsilon^1_{i,j} &= \min\{f(r,a) - f^a_{i,j}(r,a) | r_i \le r \le r_{i+1}, a_j \le a \le a_{j+1}\} \\ \epsilon^2_{i,j} &= \max\{f(r,a) - f^a_{i,j}(r,a) | r_i \le r \le r_{i+1}, a_j \le a \le a_{j+1}\} \end{aligned} \tag{3.10}$$

The data needed for generation of the wedges is stored in two arrays. The first array has a tree structure very similar to the data structure used for the generation of the boxes in Section 3.3.3 above. The tree is used when choosing which wedge should contain the point to be generated. The tree was again prepared once and for all using Huffman's method for reducing the average number of tests performed when finding the correct wedge.

The second array contains the information necessary for the generation of the various uniform deviates. Each entry is a pair of numbers representing the base height $h_{i,j}$ and the upper bound $\tilde{f}_{i,j}$ for a wedge. The width, length and r and a coordinates of each box do not need to be stored, since the width, Δr, and length, Δa, are constant over all the boxes. The use of this second array avoids some of the unnecessary use of memory mentioned in Section 3.3.3.

Two data sets are also needed for calculating the approximations $f^l_{i,j}$ and $f^u_{i,j}$. The first set consists of the function values $f(r_i, a_j)$ at the grid points, used in the interpolation. The other set contains the pairs of constants $(\epsilon^1_{i,j}, \epsilon^2_{i,j})$, as in (3.10), needed to ensure that the two approximations stay below and above $f(r, a)$.

3.5 The Tail

The tail is the last, smallest, but most difficult part of the distribution. It consists of the set of points under the surface $f(r, a)$ that have $r > r_M$ or $a > a_M$. With $r_M = a_M = 4$, the tail only accounts for 0.27% of the distribution.

Yet again, the tail has been divided up into a number of pieces. In each piece, points are generated using the rejection method. This time, rather than generating points from a uniform density, various non-linear density functions have been chosen, as being reasonably good approximations to $f(r, a)$ in the regions considered. In the table below details are given of the regions into which the tail has been divided and the density functions used. The rate reported in the last column indicates the average number of points generated in order to find one point

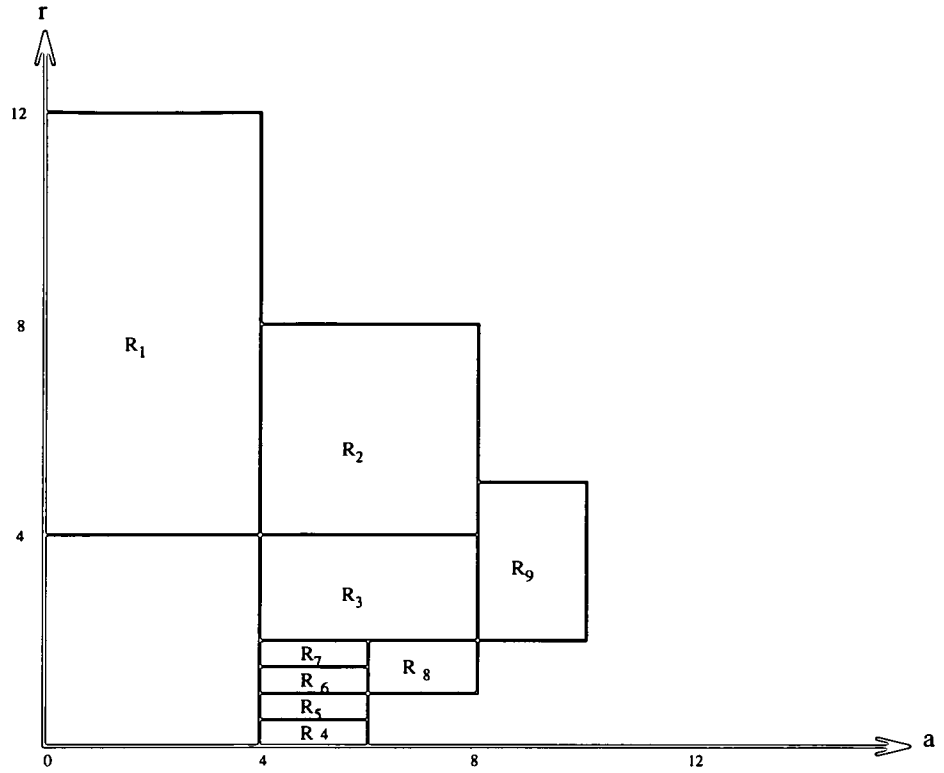


Figure 3–3: The regions in the tail

in the relevant region. The points in the last region, which has probability less than one in a million, are not generated with the right distribution.

Region	r_1	r_2	a_1	a_2	Probability	Comparison function	Rate
R_1	4	12	0	4	$2.98 \cdot 10^{-4}$	$\exp(-r^2/2)$	2.3
R_2	4	8	4	8	$3.65 \cdot 10^{-5}$	$3/2\exp(-r^2/2 - 3a^2/2r^2)$	2.0
R_3	2	4	4	8	$1.80 \cdot 10^{-3}$	$\exp(-\frac{\pi}{2}a)$	2.6
R_4	0	0.5	4	6	$1.61 \cdot 10^{-6}$	$15r\exp(-\pi a)$	2.6
R_5	0.5	1	4	6	$1.96 \cdot 10^{-5}$	$15r\exp(-2.8a)$	2.8
R_6	1	1.5	4	6	$1.20 \cdot 10^{-4}$	$25r\exp(-2.6a)$	3.0
R_7	1.5	2	4	6	$3.86 \cdot 10^{-4}$	$25r\exp(-2.4a)$	3.2
R_8	1	2	6	8	$6.57 \cdot 10^{-6}$	$40r\exp(-2.4a)$	4.2
R_9	2	5	8	10	$4.15 \cdot 10^{-6}$	$0.7\exp(-\frac{\pi}{2}a)$	2.4
	Remainder				$3.81 \cdot 10^{-7}$		

The various regions in the tail are depicted in Figure 3–3.

3.6 Performance

The results of time tests are laid out in the table below. The time taken to generate $(\Delta w^1, \Delta w^2, A_{1,2})$ triples is compared with the time taken to generate triples of uniform deviates and to generate triples of normally distributed random numbers using an application of Marsaglia's method. The same pseudo-random uniform number generator was used throughout. All the code was written in C and run on a Sun IPC Sparc workstation.

Number of triples	Time taken		
	Uniform deviates	Normal deviates	(w^1, w^2, a)
10000	<1s	<1s	3s
100000	2s	6s	28s
1000000	15s	1min1s	4min42s

The generation of area integrals therefore takes between 4 and 5 times as long as the generation of increments along the Brownian path. If one were to obtain a strong solution of order $O(h)$ by subdividing the time steps and generating increments of the Brownian path over steps of length $h' < h$, the condition on the size of h' would be $h' \leq h^2$. For $h < 1/4$ we get $h > 4h'$. So for any reasonably small step size, h, it is much quicker to obtain a solution of order $O(h)$ by generation of the $A_{1,2}(k, k+1)$ integrals than by generating Δw^1_{k+1} and Δw^2_{k+1} over smaller steps.

3.7 An example of application

As an illustration, we have used area integrals generated by our method in the numerical solution of the bilinear Itô SDE

$$dx = Ax\ dw^1 + Bx\ dw^2 \tag{3.11}$$

where A and B are constant 2 by 2 real matrices. The matrices A and B can be reduced to one of several canonical forms, in which case they depend on 4 real parameters. For example, we can take

$$A = \begin{bmatrix} a & 0 \\ 0 & -a \end{bmatrix} \qquad B = \begin{bmatrix} b & c \\ c & d \end{bmatrix}$$

This may be considered as a fundamental example, since locally any SDE can be thought of as linear.

The ease of numerical solution of equation (3.11) depends on the stability of the system and hence on the parameters a, b, c, d. When the system is stable it is possible to obtain a good approximate solution with a discretisation scheme that does not involve the area integrals using quite large time steps. However when the system is unstable it can become impossible to obtain a good solution without including the area integrals in the discretisation scheme, unless extremely small time steps are used (sometimes too small to be feasible on a computer).

We show what happens in a 'difficult' case. We have taken $a = c = 2$, $b = d = 1$ and compared the results using three different discretisation schemes. The approximate solutions given by these schemes all converge to the solution of the SDE (1.1) taken in the Itô sense, (and can only be used for a Stratonovich SDE if it is first converted into an Itô SDE). The first scheme, scheme A, is the Euler-Maruyama scheme (1.13). The second scheme, scheme B, is :

$$\tilde{x}^i_{k+1} = \tilde{x}^i_k + \left[g^i_0(\tilde{x}_k) - \frac{1}{2}\sum_{r=1}^{n}\sum_{j=1}^{d} \frac{\partial g^i_j}{\partial x^r}(\tilde{x}_k) g^r_j(\tilde{x}_k) \right] h$$
$$+ \sum_{j=1}^{d} g^i_j(\tilde{x}_k)\Delta w^j_{k+1} + \frac{1}{2}\sum_{r=1}^{n}\sum_{j,p=1}^{d} \frac{\partial g^p_i}{\partial x^r}(\tilde{x}_k) g^r_j(\tilde{x}_k)\Delta w^j_{k+1}\Delta w^p_{k+1}$$

and scheme C, the Milshtein scheme (see [26]), is the same as the previous scheme with added terms involving the area integrals and is given by:

$$\tilde{x}^i_{k+1} = \tilde{x}^i_k + \left[g^i_0(\tilde{x}_k) - \frac{1}{2}\sum_{r=1}^{n}\sum_{j=1}^{d} \frac{\partial g^i_j}{\partial x_r}(\tilde{x}_k) g^r_j(\tilde{x}_k) \right] h$$
$$+ \sum_{j=1}^{d} g_{ij}(\tilde{x}_k)\Delta w^j_{k+1} + \frac{1}{2}\sum_{r=1}^{n}\sum_{j=1}^{d} \frac{\partial g^i_j}{\partial x_r}(\tilde{x}_k) g^r_j(\tilde{x}_k) \left(\Delta w^j_{k+1}\right)^2$$
$$+ \frac{1}{2}\sum_{r=1}^{n}\sum_{j<p} \frac{\partial g^i_p}{\partial x_r}(\tilde{x}_k) g^r_j(\tilde{x}_k) \left[\Delta w^j_{k+1}\Delta w^p_{k+1} - A_{j,p}(k,k+1)\right]$$

$$+\frac{1}{2}\sum_{r=1}^{n}\sum_{j>p}\frac{\partial g_p^i}{\partial x_r}(\tilde{x}_k)g_j^r(\tilde{x}_k)\left[\Delta w_{k+1}^j\Delta w_{k+1}^p - A_{p,j}(k,k+1)\right]$$

The Euler-Maruyama scheme converges to the true solution with rate $O(\sqrt{h})$ in the general multi-dimensional set up. Scheme B converges with the same rate, but is asymptotically efficient (see [44]), ie the leading coefficient of the variance of the error in each step is minimal.

In the table below we present the approximate value obtained for x_1 at time $t = 5$ for one particular simulation of the Brownian path, using each of the three discretisation schemes and using a succession of step sizes ranging from $h = 2^{-4}$ to $h = 2^{-16}$.

	$\tilde{x}_1(5)$		
$-\log_2(h)$	Scheme A	Scheme B	Scheme C
4	2.447207	-0.017475	1.251238
5	-0.111236	0.030939	3.066002
6	0.008536	0.074882	4.243542
7	0.642300	1.281455	4.759604
8	1.096747	1.903669	3.751482
9	4.383487	3.538671	3.430095
10	5.292467	3.883448	3.430359
11	4.034335	3.030876	3.483880
12	3.704273	2.955552	3.460112
13	2.879655	2.705867	3.428692
14	3.036604	3.062775	3.431992
15	3.330327	3.167863	3.425553
16	3.415980	3.377057	3.424928

When the time step is greater than about 2^{-8}, the results obtained without area integrals, that is using scheme A or B, are completely wrong after about $t = 3$. This is illustrated in the form of a graph in Figure 3–4, depicting the

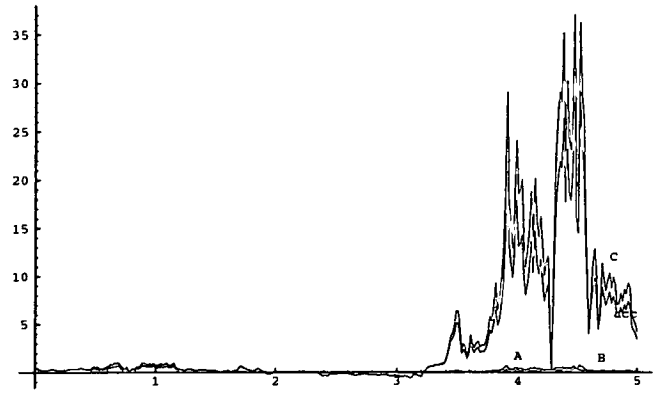


Figure 3–4: Approximations with and without area integrals

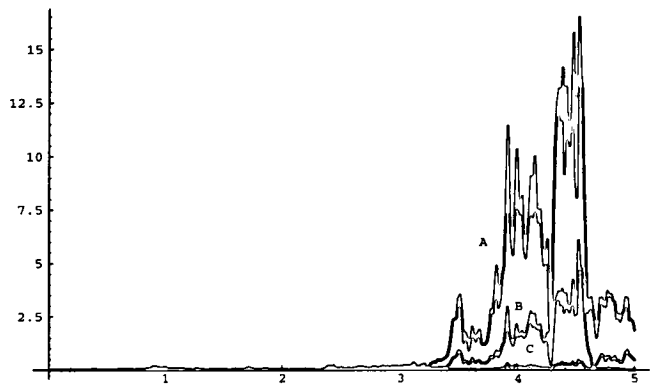


Figure 3–5: Differences between approximations and an accurate solution

three approximate trajectories obtained for x_1 using a time step $h = 2^{-6}$ and an accurate solution, obtained by using scheme C and $h = 2^{-16}$.

It is only once the time step is small enough that it makes sense to measure the difference between the various approximations. In Figure 3–5 we show the differences between the approximate solutions for x_1 obtained using each of the three discretisation schemes with a time step of $h = 2^{-10}$ and the same accurate solution used previously.

In order to obtain the same accuracy using scheme A or B as using scheme C, it is necessary to square the time step and therefore the number of time steps. If using scheme A, the Euler-Maruyama scheme, one wishes to obtain the same accuracy as with scheme C, using area integrals, with the step size $h = 2^{-8}$, then one must take $h = 2^{-16}$, hence multiplying the number of steps by 256. The computing time is

not increased by a factor of 256, since more function evaluations are performed per time step using scheme C than using scheme A. Timed simulations using equation (3.11) give a factor of 160. On a Sun Sparc station it took roughly 3 minutes to perform 1000 simulations with scheme C and $h = 2^{-8}$ and 8 hours for the same number of simulations with scheme A and $h = 2^{-16}$.

This shows that if all one wants is to perform one single simulation, then using a simple scheme and small time steps is not going to take a lot of computing time, but if one wishes to do many simulations, varying, say, the starting point or the parameter values, then the time factor will become very important. (A million different simulations would take 50 hours or 8000 hours, to obtain the same accuracy with and without area integrals.) The time factor would clearly be increased further if scheme C were replaced by a Runge-Kutta type discretisation scheme, still using area integrals but not involving evaluation of derivatives at each time step.

In the future we foresee that the use of area integrals when simulating strong solutions to SDE's will become as automatic as the use of random numbers from a normal distribution is today. After all, once a good routine has been developed and implemented in numerical libraries, the ordinary user will only need to call this routine from each program and will not need to be concerned with the details of how the routine works.

3.8 The way forward

This is of course just the tip of the iceberg. On the one hand, to obtain a strong numerical solution of order $O(h)$ to a stochastic differential equation dependent on a Brownian path of dimension $n > 2$, not just one set of integrals may be needed, but several, and the sets are all correlated. In the most general case, we would need to generate the $n(n+1)/2$ correlated random variables

$$\Delta w^i(t, t+h),\ A_{i,j}(t, t+h),\ i = 1, \ldots, n;\ j < i$$

It is theoretically possible to reduce the problem to the generation of n correlated random variables and one random element of $\mathbb{O}_n$, which latter can be generated in n steps. Let A be the matrix $A_{ij} = \int_{t_0}^{t_1} \int_{t_0}^{s} dw^i(r)dw^j(s) - \int_{t_0}^{t_1} \int_{t_0}^{s} dw^j(r)dw^i(s)$. An orthogonal matrix $S \in \mathbb{O}_n$ can be chosen so that, $S^t w = \bar{w}$ and $S^t A S = \bar{A}$ with

$$\bar{w} = \begin{pmatrix} \bar{w}^1 \\ 0 \\ \bar{w}^2 \\ 0 \\ . \\ . \\ \bar{w}^m \\ 0 \end{pmatrix} \qquad \bar{A} = \begin{pmatrix} 0 & \lambda_1 & 0 & 0 & \dots & 0 & 0 \\ -\lambda_1 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & \lambda_2 & \dots & 0 & 0 \\ 0 & 0 & -\lambda_2 & 0 & \dots & 0 & 0 \\ . & & & & & & \\ . & & & & & & \\ 0 & 0 & 0 & 0 & \dots & 0 & \lambda_m \\ 0 & 0 & 0 & 0 & \dots & -\lambda_m & 0 \end{pmatrix}$$

if $n = 2m$, or

$$\bar{w} = \begin{pmatrix} \bar{w}^1 \\ 0 \\ \bar{w}^2 \\ 0 \\ . \\ . \\ \bar{w}^m \\ 0 \\ \bar{w}^{m+1} \end{pmatrix} \qquad \bar{A} = \begin{pmatrix} 0 & \lambda_1 & 0 & 0 & \dots & 0 & 0 \\ -\lambda_1 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & \lambda_2 & \dots & 0 & 0 \\ 0 & 0 & -\lambda_2 & 0 & \dots & 0 & 0 \\ . & & & & & & \\ . & & & & & & \\ 0 & 0 & 0 & 0 & \dots & 0 & \lambda_m \\ 0 & 0 & 0 & 0 & \dots & -\lambda_m & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 \end{pmatrix}$$

if $n = 2m + 1$. Therefore, if we can obtain the joint distribution function of

$$\bar{v} = \{\bar{w}^1, \dots, \bar{w}^m, \lambda_1, \dots, \lambda_m\}$$

from the joint distribution of

$$v = \{w^1, \dots, w^n, A_{12}, A_{13}, \dots, A_{n-1,n}\}$$

we can then generate the vector v and a random matrix $S \in \mathbb{O}_n$ and simply recover w and A by setting $w = S\bar{w}$ and $A = S\bar{A}S^t$.

We would also like to derive and implement the bridge that would allow us to generate sets of points $(\Delta w^1(t_k, t_{k+1}), \Delta w^2(t_k, t_{k+1}), A_{1,2}(t_k, t_{k+1}))$ over steps of length $h/2$ given the corresponding set for step size h. All this will demand a lot more work and imagination.

On the other hand, we would like to think that the research necessary to get us this far has uncovered various questions and suggested various techniques that are relevant to the random generation of deviates from multivariate distributions in general. (See Devroye [12] for a summary of work on multivariate distributions). By demonstrating that Marsaglia's method for speedy random number generation can be applied to multivariate distributions, we have raised general questions (see the end of Sections 3.3.1 and 3.3.2) on the time and memory constraints involved.

The C routine we have written to generate area integrals can be found in Appendix A.2.

Chapter 4

The Algebra of Iterated Stochastic Integrals

4.1 Motivation

In this chapter we will look at the algebra of Ito or Stratonovich iterated stochastic integrals of the type

$$I_J(t) = I_{j_1 j_2 \dots j_n} = \int_{\Sigma_n(0,t)} dw^{j_1}(t_1) dw^{j_2}(t_2) \dots dw^{j_n}(t_n)$$

or

$$S_J(t) = S_{j_1 j_2 \dots j_n} = \int_{\Sigma_n(0,t)} \circ dw^{j_1}(t_1) \circ dw^{j_2}(t_2) \dots \circ dw^{j_n}(t_n)$$

introduced in Chapter 1. We will refer to the set of all integrals of type (1.3) as $I = I_t = \{I_J(t)\}$ and to the set of all integrals of type (1.4) as $S = S_t = \{S_J(t)\}$. Since integration will mainly be over the interval $[0, t]$, we will write I_J and S_J for $I_J(t)$ and $S_J(t)$.

This study is motivated by the desire for a better understanding of the structure of solutions to stochastic differential equations needed for analysis and improvement of numerical schemes for the approximation of solutions to SDE's. Consider again the SDE (1.2)

$$\begin{cases} dx_t &= f_0(x,t)dt + \sum_{i=1}^{d} f_i(x,t) \circ dw_t^i \\ x(0) &= x_0 \end{cases}$$

written in the Stratonovich sense, or the equivalent SDE, (1.1), written in the Itô sense

$$\begin{cases} dx_t &= g_0(x,t)dt + \sum_{i=1}^{d} g_i(x,t)\ dw_t^i \\ x(0) &= x_0 \end{cases}$$

As already stated in Chapter 1, in (1.7), (1.8) and (1.23), the solution to (1.2)can be approximated locally as a Stratonovich or Ito stochastic Taylor series,

$$x(t) = x_0 + \sum_{m=1}^{p} \sum_{||J||=m} S_J(t)\ f_J(x_0) + t^{(p+1)/2} R_{p+1}(t)$$

or

$$x(t) = x_0 + \sum_{m=1}^{p} \sum_{||J||=m} I_J(t)\ g_J(x_0) + t^{(p+1)/2} R_{p+1}(t)$$

or else as the exponential Lie series,

$$x(t) = \exp(\zeta^p(t))(x_0) + t^{\frac{p}{2}} \overline{R}_p(t)$$

where

$$\zeta^p(t) = \sum_{m=1}^{p-1} \sum_{||J||=m} c^J(t)\ X^J,$$

(See Section 1.4 for the relevant notation.) Both ways of expressing the solution involve the stochastic integrals in S or I defined above.

We felt a need for further information about these iterated integrals for two distinct reasons. Firstly it is known (see [55] and [26]) that it is not necessary to generate all iterated integrals: some of the required integrals can be expressed as polynomials in the other integrals. We wished to examine the available bases (ie subsets of the set of iterated integrals that can be used to generate the whole set) and hoped to find a 'natural' basis. In [55] Sussmann obtains a basis by using a Philip Hall basis. Here we describe a basis using a particular Hall basis, the Lyndon basis, that turns out to be more suited to our purposes.

Secondly we wished to calculate the mean and variance of the error produced when omitting the remainder term from (1.7), (1.8) or (1.23), such truncation being a natural way of generating discrete numerical schemes. Hence we wanted explicit formulae for the variances and covariances of the iterated stochastic integrals.

The variance and covariance of iterated Ito integrals was first given by Platen and Wagner in [48]. A simplified proof of the variance formula was given by Meyer in [42]. An explicit formula for the covariance of Stratonovich integrals would also be useful.

The work in this chapter has been published in [17]. This work has led to computer algebra routines for use with Mathematica, some of which are given in Appendix A.3.

4.2 Algebraic structure

4.2.1 Concatenation

Let

$$A = \{a_0, a_1, a_2, \ldots, a_d\}$$

be a set of $d+1$ distinct elements known as letters, endowed with the ordering $a_0 < a_1 < \ldots < a_d$, and let $A^\star$ be the free monoid generated by A, that is the set of all words formed with the letters in A. There is a one-to-one mapping of both sets of iterated integrals to $A^\star$, given by

$$\phi : I \to A^\star, \quad \phi(I_J) = \phi(I_{j_1 j_2 \ldots j_n}) = a_{j_1} a_{j_2} \ldots a_{j_n}$$

and

$$\tilde{\phi} : S \to A^\star, \quad \tilde{\phi}(S_J) = \tilde{\phi}(S_{j_1 j_2 \ldots j_n}) = a_{j_1} a_{j_2} \ldots a_{j_n}$$

As is usual, we denote the concatenation of two words $u, v \in A^\star$ as

$$w = uv$$

omitting any operation sign. Let the operation of concatenation be defined on I and S as the composition of operators: For $J = j_1 j_2 \ldots j_m$ and $K = k_1 k_2 \ldots k_n$, we write

$$I_J(t) \cdot I_K(t) = \int_0^t \int_0^{t_n} \ldots \int_0^{t_2} I_J(t_1) dw_{k_1}(t_1) dw_{k_2}(t_2) \ldots dw_{k_n}(t_n)$$
$$= \int_0^t \int_0^{t_{m+n}} \ldots \int_0^{t_2} dw_{j_1}(t_1) \ldots dw_{j_m}(t_m) dw_{k_1}(t_{m+1}) \ldots dw_{k_n}(t_{m+n}) = I_L(t)$$

where $L = j_1 \dots j_m k_1 \dots k_n$ and similarly

$$S_J \circ S_K = S_L$$

The functions ϕ and $\tilde{\phi}$ preserve the operations of concatenation :

$$\phi(I_J \cdot I_K) = \phi(I_J)\phi(I_K) = a_{j_1} \dots a_{j_m} a_{k_1} \dots a_{k_n}$$

and similarly for $\tilde{\phi}$. Concatenation is clearly associative.

Let $Z\langle I\rangle$ and $Z\langle S\rangle$ be the sets of finite linear combinations of integrals with integer coefficients. The operation of concatenation extends in a natural way to $Z\langle I\rangle$ and $Z\langle S\rangle$, hence:

Proposition 4.2.1 *$(Z\langle I\rangle, \cdot)$ and $(Z\langle S\rangle, \circ)$ are algebras over Z which are isomorphic to the free algebra $Z\langle A\rangle$ via ϕ and $\tilde{\phi}$.*

4.2.2 Shuffle products

On $Z\langle A\rangle$ we can provide another operation, the well-known shuffle product (see [34],[41],[33] and others). This can be defined by

(i) $1 \star 1 = 1$

(ii) $1 \star a = a \star 1 = a$

(iii) $ua \star vb = (u \star vb)a + (ua \star v)b$

$\forall a, b \in A, \forall u, v \in A^\star$

where 1 represents the 'empty' word formed of no letters.

Inspired by the work on iterated path integrals of Chen [6], Ree [51] and, more recently, Fliess [16], we have the following proposition:

Proposition 4.2.2 *The product of two Stratonovich integrals is a shuffle product.*

Proof. The proof simply uses integration by parts. (The Stratonovich calculus follows the same rules as ordinary calculus.) We will use the symbol $\times$ for multiplication temporarily during the proof and then replace it with the shuffle

product symbol $\star$ once we have proved we can do so. Consider any two multi-indices, $J = j_1 \dots j_{m-1}$ and $K = k_1 \dots k_{n-1}$ of length $m-1$ and $n-1$ respectively $(m, n > 1)$. We can obtain two new multi-indices $J' = j_1 \dots j_{m-1} j_m$ and $K' = k_1 \dots k_{n-1} k_n$, by adding a new index to each of J and K. Then we have

$$\int_0^t \left[S_J(s) \int_0^s S_K(r) \circ dw_{k_n}(r) \right] \circ dw_{j_m}(s)$$
$$= \int_0^t S_J(s) \circ dw_{j_m}(s) \int_0^t S_K(s) \circ dw_{k_n}(s) - \int_0^t \left[S_K(s) \int_0^s S_J(r) \circ dw_{j_m}(r) \right] \circ dw_{k_n}(s)$$

Hence

$$S_{J'} \times S_{K'} = (S_J \times S_{K'}) \circ S_{j_m} + (S_{J'} \times S_K) \circ S_{k_n}$$

□

Here is an example of using the shuffle product to find the pointwise product of two iterated Stratonovich integrals:

$$S_{12} \star S_{34} = S_{1234} + S_{1324} + S_{1342} + S_{3124} + S_{3142} + S_{3412}$$

The consequence of Proposition 4.2.2 is that, as will be seen in Section 4.3, various nice results on the free shuffle algebra can be applied directly to the algebra of Stratonovich integrals with the shuffle product, $(Z\langle S \rangle, \star)$.

In the product of two Ito integrals extra terms appear:

Proposition 4.2.3 *The product of two Ito integrals in $I(t)$ is a modified shuffle product, defined recursively by*

(i) $1 * 1 = 1$

(ii) $1 * I_i = I_i * 1 = I_i$

(iii) $(u \cdot I_i) * (v \cdot I_j) = (u * (v \cdot I_j)) \cdot I_i + ((u \cdot I_i) * v) \cdot I_j + \overline{\delta}_{ij}(u * v) \cdot I_0$

$\forall u, v \in I$ *and* $\forall i, j = 0, \dots, d,$

where

$$\overline{\delta}_{ij} = \begin{cases} 1; & i = j \neq 0 \\ 0; & \textit{otherwise} \end{cases}$$

Proof. The proof is again based on integration by parts, with the difference that this time we are doing Ito calculus and Ito's formula must be used. Again let

$J = j_1 \dots j_{m-1}$, $K = k_1 \dots k_{n-1}$, $J' = j_1 \dots j_{m-1} j_m$ and $K' = k_1 \dots k_{n-1} k_n$. We have

$$\begin{aligned}
&\int_0^t \left[I_J(s) \int_0^s I_K(r) dw_{k_n}(r) \right] dw_{j_m}(s) \\
&= \int_0^t I_J(s) dw_{j_m}(s) \int_0^t I_K(s) dw_{k_n}(s) - \int_0^t \left[I_K(s) \int_0^s I_J(r) dw_{j_m}(r) \right] dw_{k_n}(s) \\
&- \int_0^t I_J(s) I_K(s) < dw_{j_m}(s), dw_{K_n}(s) >
\end{aligned}$$

so

$$I_{J'} \times I_{K'} = (I_J \times I_{K'}) \cdot I_{j_m} + (I_{J'} \times I_K) \cdot I_{k_n} + \overline{\delta}_{j_m k_n} (I_J \times I_K) \cdot I_0$$

□

As an example:

$$I_{12} * I_{13} = 2I_{1123} + 2I_{1132} + I_{1213} + I_{1312} + I_{023} + I_{032}$$

Note that the last two results generalize the results (5.2.3 and 5.2.10) of Kloeden and Platen in [26].

Using the Wick product on I rather than the usual product (see eg. [20], [32]), Proposition 4.2.3 becomes:

Proposition 4.2.4 *The Wick product of two Ito integrals is a shuffle product, defined recursively by*

(i) $1 \diamondsuit 1 = 1$

(ii) $1 \diamondsuit I_i = I_i \diamondsuit 1 = I_i$

(iii) $(u \cdot I_i) \diamondsuit (v \cdot I_j) = (u \diamondsuit (v \cdot I_j)) \cdot I_i + ((u \cdot I_i) \diamondsuit v) \cdot I_j$
$\forall u, v \in I$ *and* $\forall i, j = 0, \dots, d$.

We will define a *shuffle* to be a permutation of two tuples or multi-indices that results from shuffling, ie that preserves the order of the elements of each of the tuples. If we have $|J| = p$ and $|K| = q$, then there are

$$\frac{(p+q)!}{p!q!}$$

ways of shuffling J with K, although not all the resulting shuffles will be distinct.

4.3 Bases

4.3.1 A basis for Stratonovich integrals

We give here a transcendence basis for the shuffle algebra $Q\langle S, \star\rangle$ obtained directly from a transcendence basis of $Q\langle A, \star\rangle$ via the isomorphism g defined in Section 4.2.1. (This means that we will show that each iterated integral in S can be expressed as a polynomial with rational coefficients in the basis elements. The basis will be a subset of S.)

Let the set $A^\star$ be totally ordered using the usual lexicographical ordering induced by the ordering on A. If the word $w \in A^\star$ can be written as $w = uv$ where both u and v are non-empty words, then v is a *proper right factor* of w. A word that is strictly less than any of its proper right factors is called a *Lyndon word.* Let L be the set of all Lyndon words in $A^\star$. We will need the following theorem ([33], Theorem 5.1.5):

Theorem 4.3.1 *(Lyndon) Any word $w \in A^\star$ can be uniquely written as the concatenation of a non-increasing sequence of Lyndon words,*

$$w = l_1 l_2 \dots l_n \quad (l_1, \dots, l_n \in L;\ l_1 \geq \dots \geq l_n),$$

the Lyndon factorisation *of w.*

By a result of Radford [50] the set of Lyndon words, L, forms a transcendence basis of the shuffle algebra $Q\langle A, \star\rangle$. We give here the theorem as stated and proved by Melançon and Reutenauer in [41], since their proof has the added benefit of providing an iterative method for rewriting words as sums of shuffle products of Lyndon words.

Theorem 4.3.2 *(Radford, Melançon and Reutenauer) $\forall w \in A^\star$,*

$$w = l_1^{i_1} l_2^{i_2} \dots l_k^{i_k} \quad (l_1 > \dots > l_k; i_1, \dots, i_k \geq 1),$$

w can be written as

$$w = \frac{1}{i_1! \dots i_k!}(l_1)^{i_1} \star \dots \star (l_k)^{i_k} - \sum_{u<w} \alpha_u u$$

where $(l)^i$ is the shuffle product of i copies of l and $\alpha_u \in Z$. Hence L forms a transcendence basis of the shuffle algebra $Q\langle A, \star \rangle$.

Using Propositions 4.2.1 and 4.2.2, we have immediately:

Corollary 4.3.3 *The set of Stratonovich integrals defined by*

$$L_S = \{S_J | S_J \in S, \tilde{\phi}(S_J) \in L\}$$

is a transcendence basis for $Q\langle S, \star \rangle$.

As an illustration, let the dimension of the Brownian path $d = 2$. The elements S_J of L_S for which $|J| \leq 3$ are:

$$\{S_0, S_1, S_2, S_{01}, S_{02}, S_{12}, S_{001}, S_{002}, S_{011}, S_{012}, S_{021}, S_{022}, S_{112}, S_{122}\}$$

For $d = 1$ the basis elements with $|J| \leq 5$ are

$$\{S_0, S_1, S_{01}, S_{001}, S_{011}, S_{0001}, S_{0011}, S_{0111}, S_{00001}, S_{00011}, S_{00101}, S_{00111}, S_{01011}, S_{01111}\}$$

The rewriting rule given in Theorem 4.3.2 is of immediate practical use. Each u can be rewritten in the same way as the initial word w and, since $u < w$ the procedure will terminate when all the words on the right-hand side are Lyndon words.

For example, for the integral S_{4321} one obtains

$$S_{4321} = S_1 S_2 S_3 S_4 - S_3 S_4 S_{12} - S_1 S_4 S_{23} - S_1 S_2 S_{34} + S_{12} S_{34} + S_4 S_{123} + S_1 S_{234} - S_{1234}$$

A theorem of Witt tells us how many basis elements there are of each length. (See [58] for the original paper or [52] for a recent proof.) Let $r = d + 1$. Then the number of basis elements of length n is

$$M_r(n) = \frac{1}{n} \sum_{k|n} \mu(k) r^{n/k}$$

where $\mu(k)$ is the Möbius function on Z^+ defined by $\mu(1) = 1$ and for $k = p_1^{m_1} p_2^{m_2} \dots p_s^{m_s}$, where the p_i are distinct primes, $\mu(k) = 0$ if any $m_i > 1$ and $\mu(k) = (-1)^s$ otherwise.

4.3.2 A basis for Ito integrals

It is clear that Lyndon words also provide the transcendence basis

$$L_I = \{I_J | I_J \in I, \phi(I_J) \in L\}$$

for the algebra of Ito integrals with the Wick product $Z\langle I, \diamond\rangle$. Moreover, more usefully, although ordinary multiplication of Ito integrals is not a shuffle product, we can show that the set L_I is also a transcendence basis for $Q\langle I, *\rangle$.

We will need the fact that each Stratonovich integral in S can be expressed as a $Q-$linear combination of integrals in I. The conversion formula, given in [2], can be written recursively as:

$$\begin{array}{lll} (i) & S_i = I_i & \forall i = 0, \ldots, d \\ (ii) & S_J \circ S_i \circ S_j = S_J \circ S_i \cdot I_j + \frac{1}{2}\overline{\delta}_{ij} S_J \cdot I_0 & \forall i, j = 0, \ldots, d;\ \forall J \end{array} \tag{4.1}$$

Proposition 4.3.4 *The set L_I is a transcendence basis for the algebra $Q\langle I, *\rangle$.*

Proof. The conversion formula (4.1) provides a natural isomorphism between $Q\langle I, *\rangle$ and $Q\langle S, \star\rangle$. $\tilde{L}_I$, the image of the set L_S under this isomorphism must be a transcendence basis for $Q\langle I, *\rangle$. This basis is not a practical one, but we will show that each element of $\tilde{L}_I$ can be expressed as a polynomial in the elements of L_I. The proof is by induction on the length of the elements of L_S. (The integral S_J is said to have length n if $|J| = n$.) The images of the Stratonovich integrals in L_S of length 1 are simply the Ito integrals of length one, each of which is in L_I. Suppose that the images of the integrals in L_S of length n and less can all be expressed in terms of elements of L_I of length n or less. This implies that each element of I of length at most n can be expressed using elements of L_I of length at most n. Consider the image in $\tilde{L}_I$ of an element of L_S of length $n+1$. Since each such polynomial is the sum of an element of L_I of length $n+1$ and terms involving Ito integrals of shorter length, the proof is complete. □

In a recent paper, Melançon [40] has shown that the Lyndon basis used in Sections 4.3.1 and 4.3.2 is a special case of a Hall basis. Therefore there is a family of available bases, of which the Lyndon basis seems to be the most practical.

Reutenauer's recent book, [52], is a comprehensive guide to the algebra used in this chapter and gives many further references.

4.3.3 An example of application

Consider the SDE

$$\begin{cases} dx &= \frac{1}{2}x(w_1 \circ dw_2 - w_2 \circ dw_1) \\ x(0) &= 1 \end{cases}$$

The solution of this SDE is $x(t) = \exp(A(t))$ where $A(t)$ is the Lévy area, $A(t) = 1/2(S_{12} - S_{21})$.

Expanding the solution as a series using (1.8) gives

$$x(t+h) = \frac{1}{8}\,x(t)\,(8 + 4\,S_{12} - 4\,S_{21} + 4\,S_2\,w_1(t) + 4\,S_{122}\,w_1(t) - 4\,S_{221}\,w_1(t)$$

$$+2\,S_{22}\,w_1(t)^2 + S_{222}\,w_1(t)^3 - 4\,S_1\,w_2(t) - 4\,S_{112}\,w_2(t) + 4\,S_{211}\,w_2(t)$$

$$-2\,S_{12}\,w_1(t)\,w_2(t) - 2\,S_{21}\,w_1(t)\,w_2(t) - S_{122}\,w_1(t)^2\,w_2(t) - S_{212}\,w_1(t)^2\,w_2(t)$$

$$-S_{221}\,w_1(t)^2\,w_2(t) + 2\,S_{11}\,w_2(t)^2 + S_{112}\,w_1(t)\,w_2(t)^2 + S_{121}\,w_1(t)\,w_2(t)^2$$

$$+S_{211}\,w_1(t)\,w_2(t)^2 - S_{111}\,w_2(t)^3) + R_4(t+h)$$

but, when this expression is rewritten in terms of the Lyndon basis we obtain

$$x(t+h) = \frac{1}{48}\,x(t)\,(48 - 24S_1S_2 + 48S_{12} + 24S_2w_1(t) - 12S_1S_2^2w_1(t)$$

$$+24S_2S_{12}w_1(t) + 6S_2^2w_1(t)^2 + S_2^3w_1(t)^3 - 24S_1w_2(t) + 12S_1^2S_2w_2(t)$$

$$-24S_1S_{12}w_2(t) - 12S_1S_2w_1(t)w_2(t) - 3S_1S_2^2w_1(t)^2w_2(t)$$

$$+6S_1^2w_2(t)^2 + 3S_1^2S_2w_1(t)w_2(t)^2 - S_1^3w_2(t)^3) + R_4(t+h)$$

involving no integrals other than S_1, S_2, and S_{12}, as expected.

4.4 Moments

The propositions in Section 4.2.2 show precisely how the product of any two iterated integrals can be expressed as a sum of iterated integrals. Since it is straightforward to calculate the mean of integrals in I and S, the variance and covariance of such integrals can then be calculated exactly. However the number of terms in the product grows very quickly with the length of the integral and it is computationally quicker to have explicit formulae for the variances and covariances. Platen and Wagner prove a formula for the covariance of iterated Ito integrals in [48] (Proposition 2). (See also [26] Lemma 5.7.2.) Using shuffle products it is possible to give an alternative proof of this result with a combinatorial flavour. (This proof is a bit similar to Meyer's proof for the variance in [42].)

Theorem 4.4.1 *The covariance of two Ito integrals $I_J(t)$ and $I_K(t)$ is given by*

$$E\left(I_J(t) \times I_K(t)\right) = \begin{cases} 0; & \overline{J} \neq \overline{K} \\ \dfrac{t^{l(J,K)}}{l(J,K)!} \displaystyle\prod_{i=1}^{|J|+1} \dfrac{(p_i+q_i)!}{p_i!q_i!}; & \text{otherwise} \end{cases}$$

where, for any two multi-indices J and K, $l(J,K)$ is the total number of zeroes in J and K plus half the total number of non-zero indices, and $\overline{J}$ is the multi-index obtained by leaving out the zeroes in J. For J and K as given, p_i and q_i are defined to be the number of zeroes in J and K respectively, positioned between indices $(i-1)$ and (i) of $\overline{J}$, respectively $\overline{K}$.

Proof. The only integrals in I with non-zero mean are integrals $I_J(t)$ with $J = (0,0,\ldots,0)$, $|J| = n$, the mean being $t^n/n!$. To calculate the covariance of two elements of I it is therefore sufficient to calculate the terms in the product $I_J * I_K$ that have multi-indices consisting entirely of zeroes. The formula given in Proposition 4.2.3 (iii) shows that the multi-indices of terms in the product either contain a shuffle of the indices in J and K or else contain the same indices with some pairs of non-zero indices replaced by zeroes. The replacement by zeroes only occurs when a non-zero index in J is through shuffling brought next to an identical

non-zero index in K. Therefore the only way to obtain all-zero multi-indices is to bring every non-zero index in J next to an identical non-zero in K and vice-versa. Since shuffling preserves the order within J and K, this can only be done if J and K contain the same non-zero indices in the same order. Hence if $\overline{J} \neq \overline{K}$, $E(I_J \times I_K) = 0$.

It is now clear that the only all-zero multi-index occurring in the product of I_J and I_K will have length $l(J,K)$ and therefore mean $\frac{t^{l(J,K)}}{l(J,K)!}$. It remains to calculate the coefficient of the term containing the all-zero index , ie to count the number of ways in which this multi-index can be produced. The need to pair all indices in $\overline{J}$ with the indices in $\overline{K}$ restricts the number of shuffles. The only freedom is in shuffling the zero indices, and even here the possibilities are restricted in that any zeroes between index $(i-1)$ and index (i) of $\overline{J}$ can only be shuffled with the zeroes between the same indices of $\overline{K}$. The lengths of these two blocks of zeroes being p_i and q_i, the number of such shuffles is $\frac{(p_i+q_i)!}{p_i!q_i!}$ and we are done. □

Example: $I_J = I_{101}$ and $I_K = I_{1001}$

$$I_J * I_K = 3I_{00000} + 6I_{000011} + 3I_{000101} + I_{001001} + 2I_{100001} + I_{100100} + 3I_{101000}$$
$$+6I_{110000} + 2I_{1001011} + 2I_{1001101} + 6I_{1010011} + 3I_{1010101} + 2I_{1011001} + 12I_{1100011}$$
$$+6I_{1100101} + 2I_{1101001}$$

$$E(I_J * I_K) = 3E(I_{00000}) = \frac{t^5}{40}$$

Here $p_2 = 1$, $q_2 = 2$ and all other $p_i, q_i = 0$.

Using similar arguments, it should be possible to obtain a formula for the covariance of two Stratonovich integrals S_J and S_K, but we have not yet succeeded in calculating it. Note that as well as calculating the covariance of S_J and S_K by using the shuffle product, it is also possible to convert each of the integrals into a linear combination of Ito integrals using (4.1) and then apply the above theorem.

Thanks are due to Devlin ([11]) for pointing out the relevant work on deterministic path integrals.

Vlad Bally has proved independently in unpublished work that the product of Stratonovich iterated integrals is a shuffle product.

Chapter 5

Partitions with Minimum Entropy of Regions in $\mathbb{R}^2$

5.1 Motivation

In this chapter, we present a polynomial time algorithm for choosing, among a set of rectangular or triangular partitions of the interior of a polygon in $\mathbb{R}^2$, the partition with the minimum entropy. We were led to look for such an algorithm by our work, presented in Chapter 3, with the generation of random variates using patchwork methods. Since Marsaglia et al. ([38], [37]) first invented the technique in 1964, it has been known that the fastest method for generating random deviates from a normal distribution, and from most other continuous univariate probability distributions, is the 'rectangle-wedge-tail' method, more recently known as the patchwork method. This method, described in Chapter 3, involves partitioning an area in $\mathbb{R}^2$ into rectangles, or other simple pieces such that generating points uniformly distributed in each piece is straightforward. Our use of the patchwork method for generating a bivariate density has shown us that entropy is a major consideration when choosing how to partition an area in $\mathbb{R}^d$.

In Chapter 3 we described a method of generation of Levy area integrals jointly with the increments along a two-dimensional Brownian path, based on an extension of Marsaglia's 'rectangle-wedge-tail' method to a two-dimensional distribution.

The joint density function in question was so complicated, indeed it was in the form of an improper integral that could only be calculated numerically, that no other method was feasible. Previously it has been considered that, since patchwork methods demand a lot of programming time and result in lengthy code, they are only suitable for use when speed is very important (see [12]). Our experience shows that there are densities for which patchwork methods seem to be the only option. Generalisation of the patchwork method to higher dimensions brought to light several questions that had not arisen in the case of a one-dimensional distribution and that deserve further exploration. One of these questions was how to choose a partition with low, if not minimum, entropy.

The part of Marsaglia's method with which we are concerned here, is the 'rectangle' part. Marsaglia et al. divide an area $A \subset \mathbb{R}^2$ under the graph of $f(x)$ into equal rectangles, each with area 2^{-n} for some $n \in \mathbb{N}$, and set up tables with an entry for each rectangle. A uniform random number consisting of n bits is then sufficient to choose a table entry with the correct probability. For a one-dimensional distribution the amount of memory used for storage of the tables is acceptable, but for a two-dimensional distribution the number of table entries needed becomes prohibitive. In Chapter 3, we decided to regroup many of the boxes into larger ones. This, while reducing memory requirements, increases the computing time, because once the boxes vary in area, a set of tests is required to determine which box each point falls in. The tests form a binary decision tree, which can be constructed in an optimal (time minimising) way using Huffman's method (see [22] for the original paper or Knuth [28] for a description). Huffman's method guarantees an average number of tests performed no greater than one plus the entropy of the partition. The entropy must therefore be taken into account when choosing a partition. The entropy can be written as

$$-\sum_{i}^{N} r_i \log_2(r_i)$$

where r_i, the probability of a point in a rectangle being in rectangle R_i, is the area of R_i divided by the total area occupied by rectangles.

In three or more dimensions the question of finding the partition into boxes of a given region that minimises the entropy seems to be a very hard one, so we

have chosen to start by considering what happens in two dimensions. Despite the fact explained above, that it is not necessary when generating a random deviate from a one dimensional distribution, to take rectangles of varying sizes and, therefore, not necessary to worry about the entropy of the partition, our hope is that a better understanding of the entropy of partitions in $\mathbb{R}^2$ will lead to a better understanding of the situation in higher dimensions, where entropy really becomes a consideration.

We will consider not only rectangular partitions, but also partitions into triangles. Triangles yield a much closer fit to a smooth curve than rectangles, so not nearly so many of them are required to fill the area under it. In [12], Devroye mentions that triangles can be used for patchwork methods and describes how to generate points uniformly distributed in a general triangle by linear transformation of points generated uniformly in the basic triangle $((0,0),(1,0),(0,1))$. It should be possible, when desired, to generate the whole of some distributions using triangles, to within the accuracy of a computer, so removing the need for wedges and tails, thereby both simplifying the programming and shortening the computation time. Optimal rectangular partitions remain of particular interest when considering the generation of discrete random variates.

The problem we have set ourselves is a complicated optimisation problem. Given a density function $f : \mathbb{R} \to \mathbb{R}$ we wish to maximise the area under the graph of $y = f(x)$ covered by rectangles or triangles and at the same time both minimise the number of rectangles or triangles used and minimise the entropy of the partition. The first aim clearly conflicts with the other two: the larger the area covered, the more rectangles or triangles will be needed, and the higher the entropy will become. Since we can see no way to obtain a fully optimal solution, we have chosen to separate the problem into stages and optimise one variable at each stage.

In the first instance we fix an area that we wish to cover with rectangles or triangles. If we wish to follow tradition and use other generation methods, such as rejection, for areas not contained within the rectangles, then covering say 90% or 99% of the area under $f(x)$ is suitable. If, however, we aim to cover the whole area

to within the accuracy of a particular computer, then the target area would be something like 99.9999%. Having chosen a target area, we want to fill in a region containing that area using as few rectangles or triangles as possible. In the second stage, we take as given the polygonal region formed by the union of the rectangles or triangles, and re-partition the region, keeping the total number of pieces fixed, in such a way as to minimise the entropy of the partition. There are, of course, other ways in which such a partial optimisation could be done, but we have been unable to come up with anything more practical.

The remainder of the chapter is structured as follows: in Section 5.2 we consider rectangular partitions of areas under a density curve. Initially we only consider monotone increasing or decreasing densities. After introducing the necessary notation in Section 5.2.1, we detail the two stages of optimisation: finding the partition with minimal entropy in Section 5.2.2 and minimisation of the number of rectangles in Section 5.2.3. The main theorem, presenting a polynomial time algorithm for finding the partition with minimum entropy, is in Section 5.2.2. In Section 5.2.4, we generalise to unimodal densities. In Section 5.3, we show how results obtained for rectangular partitions can be extended to triangular partitions. We have to consider convex and concave polygonal regions separately. Throughout, we illustrate results with applications to well-known distributions, such as the normal distribution.

5.2 Rectangular partitions

Let $y = f(x)$ be a continuous monotone decreasing or increasing density function defined on an interval $[a, b]$. (The interval could be $[a, \infty)$ for a decreasing function or $(\infty, b]$ in the case of an increasing function.) We wish to pack the region $\{(x, y)|y < f(x), a \leq x \leq b\}$ with rectangles in a way that minimises the number of rectangles, maximises the total area covered with rectangles and minimises the entropy of the partition. These aims are conflicting. For example, it is clearly impossible to cover more than a certain area with any fixed number of rectangles.

We will therefore split the task into two parts. First we will fix a total area that we wish the rectangles to cover and try to find a way of covering that area using the smallest number of rectangles. Secondly, we will take as given a region made up of the union of a number of rectangles and try to find the rectangular partition of that region with the lowest entropy.

If, instead of a continuous density function, we are considering the generation of a random variate from a discrete distribution, then clearly only the second stage of the method is needed.

5.2.1 Notation

Definition 5.2.1 *A* staircase *is the interior of a closed piecewise linear curve, obtained by joining linearly a set of points*

$$\{(x_0,y_0),(x_0,y_1),(x_1,y_1),(x_1,y_2),\ldots,(x_{i-1},y_i),(x_i,y_i),(x_i,y_{i+1}),$$
$$\ldots,(x_n,y_n),(x_n,y_0),(x_0,y_0)\}$$

where $x_{i+1} > x_i$, $i = 0,\ldots,n-1$, $y_{i+1} < y_i$, $i = 1,\ldots,n-1$ *and* $y_0 < y_n$ *(a descending staircase) or else* $x_{i+1} < x_i$, $i = 0,\ldots,n-1$, $y_{i+1} < y_i$, $i = 1,\ldots,n-1$ *and* $y_0 < y_n$ *(an ascending staircase).* n *is the number of steps in the staircase. The point* (x_0,y_0) *is the* root *of the staircase.*

Any staircase S with n steps can be identified uniquely by giving the set of $n+1$ points $\{P_i = (x_i,y_i), i = 0,\ldots,n\}$. For simplicity, we will often stretch notation and write $S = \{P_i = (x_i,y_i), i = 0,\ldots,n\}$. Figure 5–1 shows a descending staircase.

Definition 5.2.2 *A* rectangular partition, $\mathcal{P}$, *of a staircase,* S, *is a set of disjoint rectangles with sides parallel to the* x *and* y *axes, the union of which is* S.

A rectangle $R = \{(x,y)|x_1 < x < x_2, y_1 < y < y_2\}$ can be uniquely identified by giving the two points $P_1 = (x_1,y_1)$ and $P_2 = (x_2,y_2)$, so we will write $R = (P_1,P_2)$.

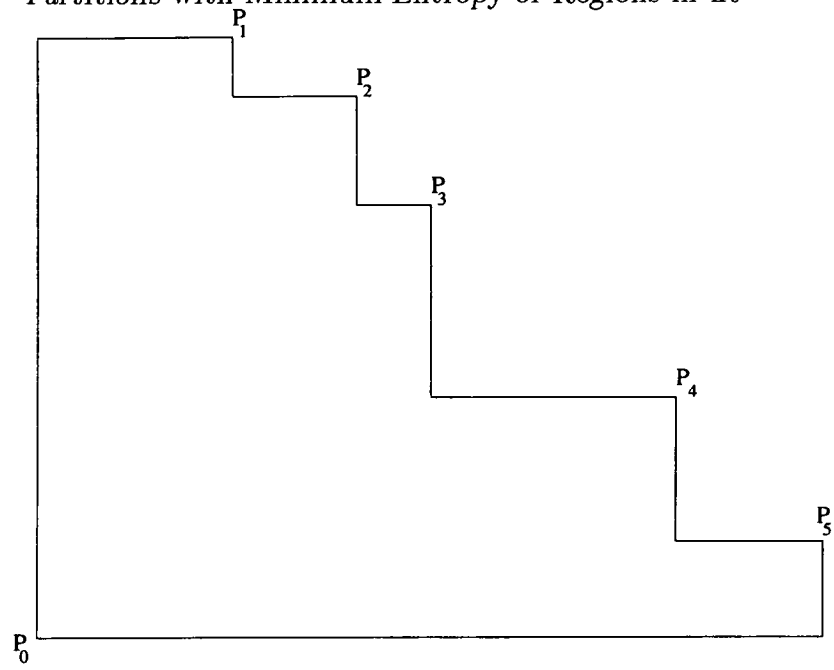


Figure 5–1: A staircase

Since we will mainly be considering rectangular partitions of a staircase S with n steps that contain precisely n rectangles, we will often write 'rectangular partition of S' for 'rectangular partition of S containing n rectangles'. A rectangular partition can be identified by listing the rectangles it contains:

$$\mathcal{P} = \{R_1, R_2, \ldots, R_n\}$$

Definition 5.2.3 *Given a staircase* $S = (P_0, P_1, \ldots, P_n)$, *a* basic rectangle *of* S *is a rectangle* $R_i = (P_0, P_i)$ *for any* $1 \leq i \leq n$.

Definition 5.2.4 *A sub-staircase of a staircase* $S = (P_0, P_1, \ldots, P_n)$, *is a staircase* $S' \subset S$ *where* $S' = (Q_0, Q_1, \ldots, Q_m)$ *and* $Q_i \in \{P_1, P_2, \ldots, P_n\}$, $1 \leq i \leq m$.

5.2.2 Optimal partition

We are now interested in choosing, from all rectangular partitions of a staircase where the number of rectangles in the partition is equal to the number of steps (see Figure 5–2), the partition with minimum entropy.

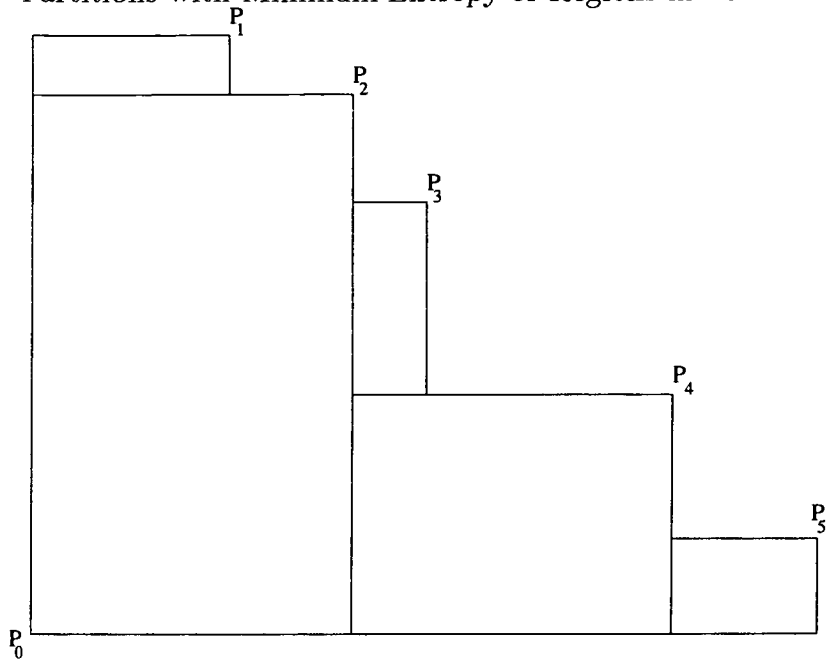


Figure 5–2: A rectangular partition of a staircase

Proposition 5.2.5 *Given a staircase S with n steps, the number of different partitions into exactly n rectangles of S is the Catalan number*

$$a_n = \frac{1}{n+1}\begin{pmatrix} 2n \\ n \end{pmatrix} \tag{5.1}$$

Proof. Each partition into n rectangles necessarily includes precisely one basic rectangle. This observation divides the partitions of the whole region into n disjoint sets, according to the choice of the basic rectangle (P_0, P_i). (This argument will also be essential further on.)

Consider all possible partitions that include the basic rectangle (P_0, P_i), for a fixed $0 < i \leq n$. Removal of the rectangle (P_0, P_i) leaves two sub-regions, one of which may be empty, to be divided into $i-1$ and $n-i$ rectangles respectively. The number of different partitions of each of these two regions is a_{i-1} and a_{n-i}. Therefore the number of partitions that include the rectangle (P_0, P_i) is $a_{i-1}a_{n-i}$ and hence the total number of possible partitions is

$$a_n = \sum_{i=0}^{n-1} a_i a_{n-i-1}$$

where we let $a_0 = 1$. From this expression, it is clear that a_n is the well-known Catalan number. (See eg. [10] for a proof that a_n satisfies (5.1)).

□

Note that we have

$$a_{n+1} = \frac{2(2n+1)}{n+2} a_n$$

and therefore

$$\lim_{n\to\infty} \frac{a_{n+1}}{a_n} = 4$$

Proposition 5.2.5 tells us that finding the partition with minimum entropy of a staircase by simply going through all possible partitions one at a time and calculating the entropy of each, takes time exponential in the number of steps. When programmed, this method only yielded an answer within a reasonable length of time for very small numbers of steps (less than 10, say). However it was pointed out to me by Mark Jerrum, that it is easy to turn this algorithm into a polynomial time algorithm, simply by storing the entropy of all sub-partitions as they are calculated. To prove this, we will need some simple results.

Let $E(\mathcal{P})$ denote the entropy of a rectangular partition $\mathcal{P} = \{R_1 \dots R_n\}$. So

$$E(\mathcal{P}) = -\sum_{i}^{N} r_i \log_2(r_i)$$

where $r_i = A_i/A$, A_i is the area of rectangle R_i, $i = 1, \dots, n$ and A is the total area

$$A = \sum_{i=1}^{n} A_i$$

Proposition 5.2.6 *Using the above notation, for any rectangular partition $\mathcal{P} = \{R_1 \dots R_n\}$ of a staircase S define*

$$\tilde{E}(\mathcal{P}) = -\sum_{i=1}^{n} A_i \log A_i$$

Then, the rectangular partition of S with minimum entropy is also the partition $\mathcal{P}$ for which $\tilde{E}(\mathcal{P})$ is minimum.

Proof. The entropy can be expressed as

$$E(\mathcal{P}) = \frac{1}{\log 2}\left(\frac{1}{A}\tilde{E}(\mathcal{P}) + \log A\right)$$

therefore, since A is constant, $E(\mathcal{P})$ and $\tilde{E}(\mathcal{P})$ have the same minimum. □

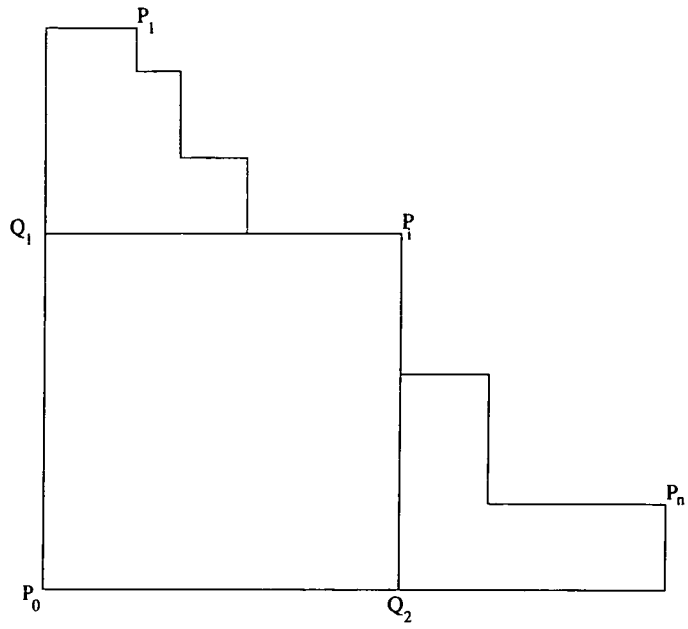


Figure 5–3: Removing a rectangle to leave two smaller staircases

Proposition 5.2.7 *Given a staircase $S = (P_0, P_1, \ldots, P_n)$ the rectangular partition of S into n rectangles with the minimum entropy can be found by, for each $i = 1, \ldots, n$, removing the basic rectangle $R_i = (P_0, P_i)$ and finding the partition, $\mathcal{P}_i$, with the lowest entropy of the remaining region $S - R_i$. Then the partition with minimum entropy is $\mathcal{P} = \{R_j\} \cup \mathcal{P}_j$ where $\mathcal{P}_j$ is such that $E(\mathcal{P}) = \min_i E(\{R_i\} \cup \mathcal{P}_i)$.*

Proof. As shown in Proposition 5.2.5, the set of all rectangular partitions of S into n rectangles can be divided into n disjoint subsets, according to which basic rectangle they contain. Therefore the partition with minimum entropy can be found by choosing the optimal partition from each of the n subsets and then taking the optimal partition from among those n. It remains to note that

$$\tilde{E}(\{R_1, \ldots, R_i, \ldots, R_n\}) = \tilde{E}(\{R_i\}) + \tilde{E}(\{R_1, \ldots, R_{i-1}, R_{i+1}, \ldots, R_n\})$$

so, applying Proposition 5.2.6, the optimal partition in each subset can be found by discarding the rectangle R_i and calculating the entropy of each possible partition of the remaining region. □

Theorem 5.2.8 *Let S be a staircase with n steps. There is an algorithm for finding the partition into n rectangles of S that has minimum entropy, that takes time polynomial in n.*

Proof. We will give details of the algorithm and show that the time is indeed polynomial in n. The algorithm is recursive.

Let $S = (P_0, P_1, \ldots, P_n)$ and consider what happens when the basic rectangle $R_i = (P_0, P_i)$, $0 < i \leq n$ is removed from S. If $1 < i < n$, we are left with two disjoint sub-staircases, $S_1 = (Q_1, P_1, P_2, \ldots, P_{i-1})$ and $S_2 = (Q_2, P_{i+1}, P_{i+2}, \ldots, P_n)$, where $Q_1 = (x_0, y_i)$, $Q_2 = (x_i, y_0)$. (This is illustrated in Figure 5–3.) If $i = 1$, then S_1 is the empty set, and if $i = n$, then S_2 is the empty set. So removing a basic rectangle from a staircase always leaves one or two sub-staircases.

Proposition 5.2.7 can now be applied recursively, first to the original staircase S, then to each of the sub-staircases created by removing a basic rectangle, etc. It is clear that, on the one hand, if we know the entropy of every sub-staircase contained in S we have enough information to find the partition of S with minimal entropy, and on the other hand, some sub-staircases of S are going to turn up more than once during the recursion. We claim that if we calculate and store the optimal partition and its entropy for each sub-staircase the first time that we need it, then simply retrieve the stored information on each subsequent occasion that we need it, the recursive algorithm will take time polynomial in n. To show this we simply need to count the number of sub-staircases that arise.

As seen above, it is only a certain type of sub-staircase that can be obtained by removing a basic rectangle from a staircase. Apart from the root, all the points defining the sub-staircase are a consecutive subset of the set of points defining the staircase taken as an ordered set. The root is uniquely determined by the other points defining the sub-staircase. Therefore, the number of sub-staircases with m steps of the staircase S, is the number of subsets of m consecutive points of the ordered set $\{P_1, P_2, \ldots, P_n\}$, which is clearly $n - m + 1$. So in total we need to find the optimal partition of

$$\sum_{m=2}^{n} (n - m + 1) = \frac{n(n-1)}{2}$$

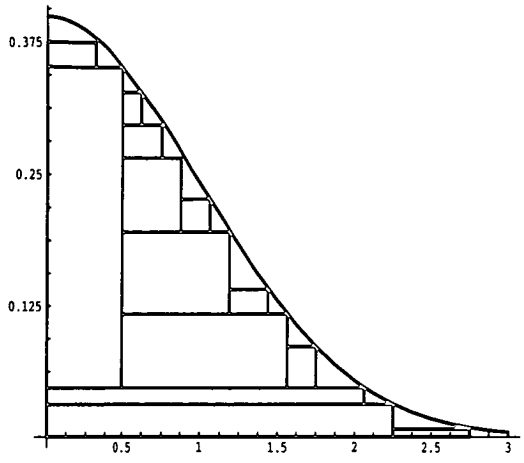


Figure 5–4: a random partition

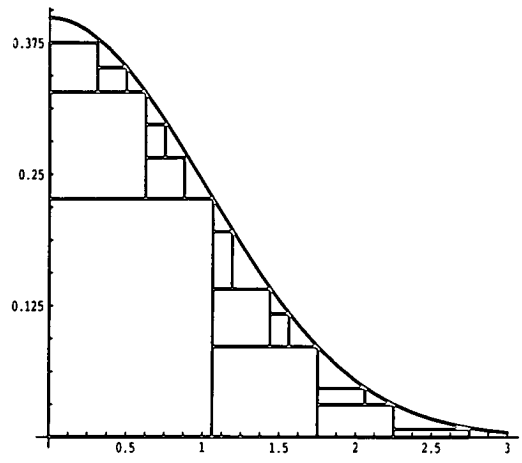


Figure 5–5: the optimal partition

staircases. The memory storage required by the algorithm is therefore quadratic in the size of the staircase. Given a sub-staircase with m steps, choosing the partition with minimum entropy amounts to comparing the previously calculated optimal entropies of each of the m sub-sets obtained by removing a basic rectangle. The amount of time involved for that one sub-staircase is therefore linear in m. The total amount of time needed to find the optimal partition of the staircase S is then proportional to

$$\sum_{m=2}^{n} m(n-m+1) = \frac{1}{6}n(n-1)(n+4)$$

So the computation time is cubic in n. □

In Figures 5–4 and 5–5, we show two different partitions of the staircase illustrated in Figure 5–6. In the first picture, the partition has been chosen randomly, (by choosing a basic rectangle randomly and then repeating recursively on the two remaining sub-staircases). The entropy of the partition is 2.83. The partition in the second picture is the optimal partition, calculated using the algorithm of Theorem 5.2.8 and has an entropy of 2.32. The entropy goes up to 3.27, for the partition obtained by dividing the staircase into vertical strips (columns).

In Table 5–1, we give the computation time used to find the partition with minimum entropy of staircases of various sizes using an implementation of the algorithm given in Theorem 5.2.8. We programmed the algorithm in C and ran

Table 5–1: Computation time for optimal partition

No. steps	100	200	300	400	500	600	700	800	900	1000
Time (s)	5	34	116	275	569	919	1464	2201	3120	4237

it on a Sun Sparc workstation. A copy of the main part of the program can be found in Appendix A.4.

5.2.3 Generating the Staircase

Having shown how to partition a staircase in such a way as to minimise the entropy of the partition, in this section we suggest a way in which a suitable staircase could be placed under a density function. Let $f(x) : \mathbb{R} \to \mathbb{R}$ be a monotone increasing or decreasing density function on $[0, \infty]$. The area under f is $\int_0^\infty f(x)dx = 1$. We are interested in finding a staircase S that contains a chosen large percentage q of the area under f, where the number of steps, n, is as small as possible, and such that when the staircase is divided into n rectangles, all the rectangles have areas that are integer multiples of a unit area equal to 2^{-m} for some $m \in \mathbb{N}$. (The constraint on the areas allows us to work with bits and therefore with integers on a computer, so avoiding loss of accuracy.) The numerical examples at the end of this section will show that, as long as the staircase contains an area at least q and very close to q, the choice of the initial staircase does not change the minimum entropy significantly. We suggest the following method.

First choose large values $m_1, m_2 \in \mathbb{N}$, and cover the quadrant $\{(x, y)|x > 0, y > 0\}$ with a rectangular grid with mesh size $\Delta x = 2^{-m_1}$,$\Delta y = 2^{-m_2}$. For simplicity we will assume that the origin is a grid point. Shade in all the rectangles that lie completely under f. The boundary of the shaded region is a staircase S_0 lying under f. At this stage the area contained by S_0 should be larger than the desired value q and S_0 should have a number of steps, N, several times larger than the number of steps envisaged for the final staircase S. (If this is not so, then choose larger values for m_1 and m_2 and repeat.) Let S_0 have root P_0 and steps with

corners at P_i, $i = 1, \ldots, N$, so $S_0 = (P_0, P_1, P_2, \ldots P_N)$ We will now define an iterative procedure which removes one step at each stage until at stage K the resulting staircase $S = S_K$ has area at least q and the staircase S_{K+1} does not have this property. For staircase $S_j = (P_0, Q_1, Q_2, \ldots, Q_{N-j})$, obtained at stage j, define $\tilde{S}_j^i = (P_0, Q_1, Q_2, \ldots, Q_{i-1}, Q_{i+1}, \ldots, Q_{N-j})$, the staircase obtained by removing step i from S_j. At step $j+1$, from all possible $\tilde{S}_j^i$, $i = 1, \ldots, N-j$, choose the one that includes the greatest area and call this one S_{j+1}. Repeat this process, removing at each stage the step that reduces the area by the smallest amount, until at stage K it is no longer possible to remove a step and still retain an area at least as big as q. The total number of comparisons needed up to stage K is

$$\sum_{j=1}^{K} N - j + 1$$

so, in the worst case, when $K = N$, the computation time is quadratic in the initial number of points, N. The relevant areas could be stored at the outset, simply removing one and up-dating two others at each stage.

This algorithm is illustrated in Figure 5–6. We have used the positive half of the normal density and a mesh size of 2^{-4} by 2^{-7}. The aim was to produce a staircase with few steps containing at least 90% of the area under the curve. The outer staircase is S_0 and the inner one the staircase S, obtained from S_0 by removing steps. There are 13 steps in S.

Tables 5–2 and 5–3 show how, for a fixed target area, the entropy of the optimal partition varies with the number of equal pieces into which the relevant interval on the $x-$axis is initially divided. The columns N_1, Area 1, and N_2, Area 2, show the numbers of rectangles and the total area covered when the interval is first divided into equal sub-intervals and then after as many points have been removed as possible without going below the target area. The notation $E_{optimal}$ denotes the entropy of the optimal partition of the resulting staircase and E_{column} denotes the entropy of the partition of the same staircase into columns, for the sake of comparison.

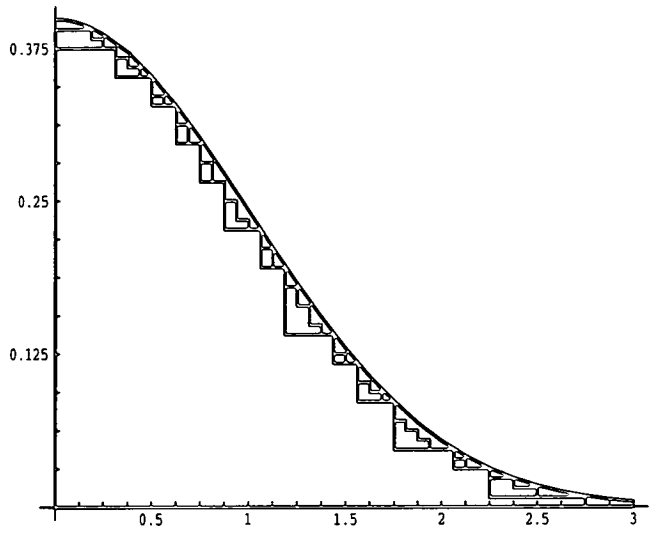


Figure 5–6: Fitting a staircase under a curve

Table 5–2: Standard normal distribution

Interval	Target	N_1	Area 1	N_2	Area 2	$E_{optimal}$	E_{column}
$[0,5]$	90%	50	.460000	15	.450000	2.386157	3.501112
		100	.479062	13	.452813	2.350116	3.299084
		200	.489687	12	.451541	2.302467	3.189303
		500	.495650	11	.450138	2.263739	3.052353
		1000	.497762	12	.453328	2.308525	3.165404
		2000	.498838	12	.452621	2.300580	3.238747
$[0,5]$	95%	100	.479062	32	.475313	2.702003	4.494936
		200	.489687	27	.475703	2.652975	4.239397
		500	.495650	24	.475113	2.621831	4.158726
		1000	.497762	25	.475825	2.635692	4.226906
$[0,5]$	99%	500	.495650	175	.495012	3.008597	6.919783
		1000	.497762	136	.495019	2.997530	6.577358
		2000	.498838	128	.495036	2.995460	6.444160
		5000	.499525	125	.495009	2.992606	6.417679

Table 5–3: Cauchy distribution

Interval	Target	N_1	Area 1	N_2	Area 2	$E_{optimal}$	E_{column}
$[0, 10]$	90%	500	.452850	64	.450100	3.542023	5.775558
		1000	.460338	50	.450150	3.511929	5.407273
		2000	.464422	47	.450084	3.493660	5.300923
$[0, 20]$	95%	3000	.475144	295	.475006	3.873419	7.946205
		5000	.478354	145	.475036	3.856674	6.548353
		10000	.481294	118	.475044	3.843877	6.548353

5.2.4 Rectangular partitions of a general region in $\mathbb{R}^2$

We have so far only considered monotone increasing or decreasing density functions. Naturally we would wish to extend our results to a general density on $\mathbb{R}$. Let $f(x) : \mathbb{R} \to \mathbb{R}$ be a unimodal density function, increasing on $(-\infty, a]$ and decreasing on $[a, \infty)$.

If f is symmetric, we would naturally choose to generate points $x \geq a$, using the right half of the distribution only, and then with probability $1/2$ take $a - x$ instead of x. This reduces the problem to partitioning a descending staircase.

However, if f is not symmetric, we need a way to generalise the previous section to a staircase that is first ascending and then descending.

Definition 5.2.9 *An* f-staircase *is the interior of a closed piecewise linear curve, obtained by joining linearly a set of points*

$$\begin{aligned}
&\{(x_0, y_0), (x_0, f(x_0)), (x_1, f(x_0)), (x_1, f(x_1)), \ldots, \\
&(x_j, f(x_j)), (x_{j+1}, f(x_j)), (x_{j+1}, f(x_{j+1})), \ldots, \\
&(x_p, f(x_p)), (x_{p+1}, f(x_p)), (x_{p+1}, f(x_{p+2})), \ldots, \\
&(x_k, f(x_k)), (x_k, f(x_{k+1})), (x_{k+1}, f(x_{k+1})), \ldots, \\
&(x_{n-1}, f(x_n)), (x_n, f(x_n)), (x_n, y_0), (x_0, y_0)\}
\end{aligned}$$

where $x_{i+1} > x_i$, $(i = 0, \ldots, n-1)$, $f(x_i) \geq y_0$, $(i = 0, \ldots, n)$, $x_p < a < x_{p+1}$ *and* $f(x_p) = f(x_{p+1})$.

Naturally, if f is defined only on a subset of $\mathbb{R}$, then the x_i $(i = 0, \ldots, n)$, in the above definition, must be taken in that subset. For simplicity, we will assume that for $i \neq j$ we only have $f(x_i) = f(x_j)$ when $i = p$ and $j = p+1$ (or $j = p$, $i = p+1$), i.e. no two steps are of the same height. This time there is no root, but there is a line $(x_0, y_0), (x_n, y_0)$ which can be called the *base* of the staircase.

Theorem 5.2.8 is dependent on the concept of similarity that leads to recursion: each time a basic rectangle is removed from a staircase, we are left with one or two pieces that are again staircases and give identical, but smaller, optimisation problems to the original one. If the staircase is to be allowed to ascend and then descend, we need a satisfactory new definition of a basic rectangle, ie. a piece that serves to classify the partition and when removed leaves one or more general staircases with fewer steps. It can be shown that the following definition meets these criteria:

Definition 5.2.10 *Given an f-staircase $S = (P_0, \ldots, P_n)$, a* basic rectangle *of S is a rectangle $R_i = ((\tilde{x}, y_0), (x_i, f(\tilde{x}))$ for one $0 < i \leq n$, where*

$$\tilde{x} = \begin{cases} x_0; & f(x_0) < f(x_n) \\ x_n; & f(x_n) < f(x_0) \end{cases}$$

If $i > 1$ and $i < n$, then removal of the basic rectangle R_i leaves a single region that is not an f-staircase. However, if removal of R_i is accompanied by splitting the remaining region vertically along the line $x = x_i$, then the region is subdivided into two staircases, as required. This splitting is validated by the fact that any rectangular partition of the remaining region not including the line segment $[(x_i, y_0), (x_i, f(x_i))]$, would necessarily contain more than n rectangles. Furthermore, any partition of a staircase contains one and only one basic rectangle.

It is now clear that the number of rectangular partitions of an f-staircase with n steps, as defined in Definition 5.2.9, is again given by the formula in Proposition 5.2.5 and that a polynomial-time algorithm for finding the partition with minimum entropy exists and is essentially the same as that given in Theorem 5.2.8. The main difference is that, using the new definition of basic rectangle, the number of subsets of the region for which the optimal partition needs to be calculated and stored, in

order to find the optimal partition of the whole region, is no longer quadratic in the number of steps, n, but cubic. Therefore the algorithm becomes quartic in n.

If the density f is not unimodal, the best option seems to be to split the density into unimodal pieces and partition each piece separately. Perhaps this is worth further study.

5.3 Triangular partitions

Rather than partitioning an area in $\mathbb{R}^2$ into rectangles, it could be preferable to partition it into triangles. The great advantage is that the number of triangles needed to cover a given percentage of the area under a curve will generally be of the order the square root of the number of rectangles. The drawbacks are that the areas will no longer be multiples of a given area and that the operations involved are more complex.

5.3.1 Triangular partitions of a convex region

Let $f(x) : \mathbb{R} \to \mathbb{R}$ be positive and concave on an interval $[a, b]$. Since the area under f is finite, it does not really matter whether the area is 1 or not. Note that for this section $f(x)$ need not be monotone. Consider the area enclosed between $f(x)$ and the chord AB, where $A = (a, f(a))$, $B = (b, f(b))$. We aim to divide this area into triangles in an efficient way. (The remaining area under $f(x)$ and above $[a, b]$ is a trapezium and can be cut into 2 triangles or a rectangle and a right-angled triangle.)

The placing of points on the curve can be carried out in a similar fashion to the division into rectangles of the region under a decreasing function described in Section 5.2.3. Furthermore, a similar result to Theorem 5.2.8 can be given concerning the partition with the minimum entropy.

Whereas for rectangular partitions we only considered rectangles with sides parallel to the coordinate axes, we will not insist that triangles have any side par-

allel to an axis. However, we will put a constraint on the positioning of triangles: each triangle in the partition must have all three vertices on the curve $f(x)$. As before, we will only look at partitions where the number of pieces is minimal. In other words, there will be the same number of triangles as intermediate points between A and B. This leads to the following definitions:

Definition 5.3.1 *A* convex f-polygon *is the interior of the closed piecewise linear curve, obtained by joining linearly a set of n points*

$$\{(x_1, f(x_1)), (x_2, f(x_2)), \ldots, (x_i, f(x_i)), (x_{i+1}, f(x_{i+1})), \ldots, (x_n, f(x_n))\}$$

where $a \leq x_1$, $x_n \leq b$, the function f is concave on the interval $[a, b]$ and $x_{i+1} > x_i$, $i = 1, \ldots, n$. The points $P_1 = (x_1, f(x_1))$ and $P_n = (x_n, f(x_n))$ are the extreme points *of the f-polygon.*

We will often write polygon for f-polygon.

Definition 5.3.2 *Given a convex f-polygon $S = (P_1, P_2, \ldots, P_n)$, a* basic triangle *of S is a triangle $T_i = (P_1, P_i, P_n)$ for any $1 < i < n$.*

The definition of *sub-polygon* can be deduced from that of sub-staircase.

In Figure 5–7 we give an example of such a triangularisation of an f-polygon.

All the propositions in Section 5.2 can be proved in this set up. For example:

Proposition 5.3.3 *(Catalan) Given a convex polygon S with $n + 2$ vertices the number of different partitions into exactly n triangles of S is*

$$a_n = \frac{1}{n+1} \begin{pmatrix} 2n \\ n \end{pmatrix}$$

Proof. The proof is identical to that of Proposition 5.2.5, with triangle substituted for rectangle throughout and replacing basic rectangle (P_0, P_i) with basic triangle (P_1, P_i, P_{n+2}). This is actually the original enumeration problem of Catalan. □

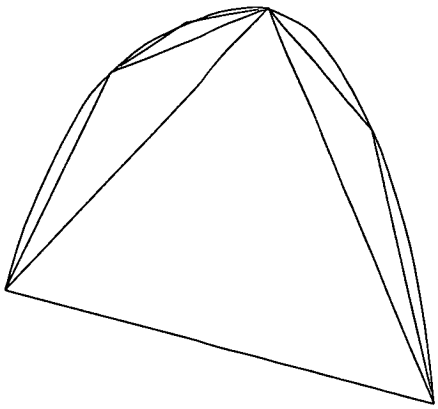


Figure 5–7: partition of a convex region into triangles

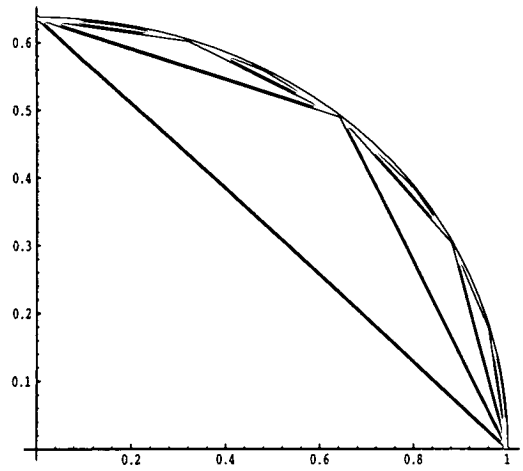


Figure 5–8: Optimal partition of a convex region into triangles

The theorem is then the following:

Theorem 5.3.4 *Let S be a convex f-polygon with $n+2$ vertices. There is a polynomial time algorithm for finding the partition into n triangles of S that has minimum entropy.*

Proof. The proof is completely parallel to that of Theorem 5.2.8. The algorithm consists in removing each basic triangle in turn, finding, by recursion, the optimal partition into triangles of each of the remaining sub-polygons and then choosing from the n resulting partitions that with the least entropy. □

In Figure 5–8, we show the optimal triangular partition of a convex f-polygon covering 99% of the area under the positive half of the curve $(2/\pi)\sqrt{1-x^2}$. There are only 6 triangles in the partition and the entropy is 1.29. The mean entropy over 100 random triangular partitions of the same area is 1.74.

When using triangles, the method used for placing the points on the curve, so determining the polygon to be partitioned, could be the method suggested above for placing a staircase under a curve: start with many points and then discard those points that reduce the area by the least, until a target area is covered. Experiments with particular densities show that, as long as the desired area is

Table 5–4: Circular distribution

Target	N_1	Area 1	N_2	Area 2	$E_{optimal}$
99%	12	.991018	11	.990238	1.303015
	15	.991570	9	.990141	1.293016
	50	.998942	8	.990473	1.291973
	100	.999626	9	.992651	1.341921
99.9%	55	.999083	38	.999000	1.485578
	100	.999626	25	.999014	1.483400
	200	.999868	23	.999016	1.482014
	500	.999967	23	.999073	1.483102

covered, the particular placing of the vertices of the polygon has little effect on the optimal entropy, but the number of triangles used can be decreased significantly. In Table 5–4, the density considered is $f(x) = \sqrt{4/\pi - x^2}$ on the the interval $[0, 2/\sqrt{\pi}]$, and the notation is that used in Table 5–2.

5.3.2 Triangular partitions of a concave region

Let $f(x) : \mathbb{R} \to \mathbb{R}$ be positive, convex and differentiable on an interval $[a, b]$, where a or b may be $\pm\infty$, and let the area $\int_a^b f(x)dx$ be finite. Consider the area bounded by $f(x)$, $x \in [a, b]$ and the tangents to $f(x)$ at $x = a$ and $x = b$. If $b = \infty$ or $f(a)$ is undefined, but $\lim_{x\to a} f(x)$ is defined, then one or both of the tangents will be asymptotes. Let the point Q_0 be the intersection of the two boundary tangents or asymptotes.

The triangles we will consider this time are triangles that have one side tangent to the curve $f(x)$. They are more complicated to construct, and involve knowledge of the derivative $f'(x)$.

Definition 5.3.5 *Given a set of n points*

$$\{(x_1, f(x_1)), (x_2, f(x_2)), \ldots, (x_i, f(x_i)), (x_{i+1}, f(x_{i+1})), \ldots, (x_n, f(x_n))\}$$

where $a \leq x_1$, $x_n \leq b$, the function f is convex on $[a, b]$ and $x_{i+1} > x_i$, $i = 1, \ldots, n$, define the line ℓ_i to be the tangent to $f(x)$ at $x = x_i$ and let $Q_i = \ell_i \cap \ell_{i+1}$, for

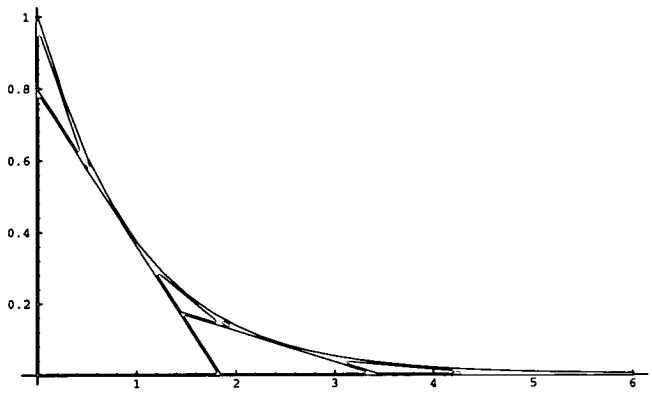


Figure 5–9: Optimal partition of a concave region into triangles

$i = 1, \ldots, n-1$ *and* $Q_0 = \ell_1 \cap \ell_n$. *Then the interior of the closed piecewise linear curve, obtained by joining linearly the set of n points* $\{Q_0, Q_1, Q_2, \ldots, Q_{n-1}\}$ *is a* concave f-polygon.

Definition 5.3.6 *Given a concave f-polygon* $S = (Q_0, Q_1, Q_2, \ldots, Q_m)$, *a* basic triangle *of S is a triangle* $T_i = (A_i, Q_0, B_i)$ *for any* $1 \leq i < m$, *where the points* A_i *and* B_i *are defined by* $A_i = Q_0Q_1 \cap Q_iQ_{i+1}$, $B_i = Q_iQ_{i+1} \cap Q_0Q_m$.

All the results proved for partition of the area under a decreasing curve into rectangles and for partition of the area under a concave curve into triangles can also be obtained in this third case.

As an illustration of the triangularisation of a concave region, we show the exponential density in Figure 5–9. The area covered by triangles is 99% of the total area under the curve. There are 11 triangles and for the partition shown, which is the optimal partition of the polygon, the entropy is 1.38.

Again, we can study the effect on the optimal entropy and on the number of triangles in the partition, of the choice of polygonal approximation. Using the exponential density, we have obtained the numbers given in Table 5–5, employing the same method and notation as in Section 5.3.1. This time the number of triangles has hardly been reduced by starting with a finer partition.

Table 5–5: Exponential distribution

Interval	Target	N_1	Area 1	N_2	Area 2	$E_{optimal}$
[0, 5]	99%	20	.994052	11	.990010	1.376467
		50	.996217	11	.990516	1.382802
		100	.996528	11	.990150	1.378803
		300	.996620	11	.990258	1.384259
[0, 8]	99.9%	75	.999009	37	.999041	1.467915
		100	.999565	36	.999009	1.467333
		250	.999789	36	.999013	1.468156

5.3.3 Triangular partitions of a general region

There does not seem to be any straightforward way of placing triangles under a curve $f(x)$ in an interval where f is sometimes concave and sometimes convex. The two methods for fitting triangles described in Sections 5.3.1 and 5.3.2 are so different that a hybrid of the two is hardly possible. The simplest idea is just to divide the whole interval on which $f(x)$ is defined into sub-intervals, on each of which f is only concave or only convex and to then place triangles separately under each piece of the curve. This amounts to insisting that all inflection points (points where $f''(x) = 0$) be used as points in the partition. Each convex region will be bounded by the curve $f(x)$ and a chord $[(x_1, f(x_1)), (x_2, f(x_2))]$, while each concave region will be bounded by the curve and two tangents, (one of which may be the tangent at infinity, in which case it is quite likely to be the $x-$axis). There may be some large pieces left over after removing these regions, which can easily be divided into a few rectangles or triangles.

The positive half of the normal density, for example, has a single inflection point at $x = 1$, so triangles should be fitted in two separate regions. The convex region is between the chord $[(0, f(0)), (1, f(1))]$ and the curve between $x = 0$ and $x = 1$. The concave region is between the curve, $f(x), x > 1$, the line $x = 1$ and the $x-$axis. Apart from these regions, there is a polygon that can be divided into a large rectangle and a right-angled triangle, which already contain between them more than 63% of the probability.

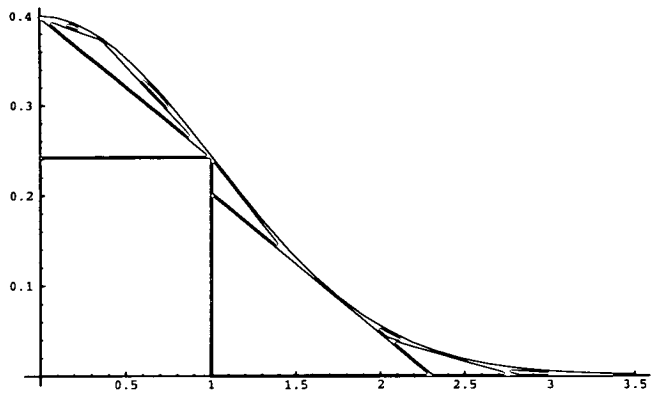


Figure 5–10: triangular partition of the normal distribution

In Figure 5–10 we show how the region under the normal density might be divided up into triangles. Since the density is symmetric about $x = 0$, it is of course sensible to generate points from the positive half and then assign a random sign to them. The partition illustrated covers 99.5% of the area under the curve and has an entropy of 1.97331. The convex part contains 4 triangles and the concave part 7, giving a total of 12 triangles and one rectangle. The partitions of both the convex and the concave parts have minimum entropy given the set of points on the curve to be used as vertices in the convex case and points of tangency in the concave case. This partition contains the same number of pieces as the rectangular partition in Figure 5–5, but has a lower entropy and covers much more of the distribution, showing clearly the advantage of using triangles.

All the well-known probability densities have a small number of inflection points, so breaking up the curve in this fashion is perfectly feasible. The problem comes when trying to generalise to higher dimensions: most pieces of surface are neither convex nor concave.

5.4 Conclusion

The work described above is a detailed description of improvements to the set up phase of the patchwork method for generating random deviates. Traditionally this phase is separate from the actual generation of random numbers and, since it needs to be performed only once for each distribution, it is worth taking time and effort to get it right.

In Chapter 6, we will present a method in which the set up phase is carried out in parallel to the random number generation. Rather than choosing a fixed area to fill with rectangles or triangles before starting, we will add in a new piece whenever it is needed, so that the number of pieces placed increases with the number of random deviates generated, thereby allowing for flexible accuracy.

Chapter 6

Generation of a General Univariate Probability Density

6.1 Introduction

Our aim in this chapter is to describe a method for generation of random deviates that is fast, as accurate as required and may be used for a wide class of continuous distributions, so could be called a black box method.

Our starting point is again the rectangle-wedge-tail method devised by Marsaglia et al. ([38], [37]), initially for generation of normal deviates, the fastest method available for generation of continuous random deviates. Speed is paid for by the use of lengthy look-up tables. On modern computers the memory requirements are no longer a problem, but the programming time and complexity is considerable. For each distribution, the programmer has to calculate in advance all the necessary constants.

We chose to design a similar method that, given any distribution, from as large a class as possible, could calculate its own constants as it went along, building up tables dynamically. In order, on the one hand, to simplify the structure of the program and, on the other hand, to handle as many different distributions as possible, we decided to throw away the wedges and the tail, keeping only triangles.

The reason for choosing triangles, not rectangles, was that, given that we wished to use the same method of generation everywhere, we had to cover not just 99% or 99.9% of the region under the density curve, but the whole region to within a certain accuracy, and were therefore forced to minimise the number of pieces required. The number of triangles needed to cover a given area under a smooth curve is of the order the square root of the number of rectangles required.

Accuracy was another essential consideration: our work with numerical solution of stochastic differential equations (SDE's) has convinced us of the need for a source of random deviates accurate to a desired number of bits, where the precise number of bits should be allowed to vary from application to application. When taking large time steps to approximate the solution of an SDE, it makes no sense to require extremely accurate random deviates, but as the time steps get smaller, in an attempt to increase the accuracy of the approximation, the random deviates used should get more accurate if the desired accuracy is to really be obtained.

The general idea is to place triangles under the density curve as needed, writing the necessary table entries for each triangle at the time it is placed. Most of the time points will be generated in already positioned triangles.

6.2 Placing the triangles

Let $f : \mathbb{R} \to \mathbb{R}$ be a continuous density function with a continuous derivative $f'(x)$ and second derivative $f''(x)$. Let the set of points $F = \{x | f''(x) = 0\}$ be finite and small. By 'small', we mean that we can handle 0, 1 or 2 inflection points easily, but that the more there are, the more complicated the program becomes. We can therefore split $\mathbb{R}$ into a few intervals, on each of which $f(x)$ is either concave or convex.

We wish to build an algorithm for returning random deviates with density $f(x)$, using as few inputs as possible. The necessary inputs are as follows: the interval $[a, b]$ on which the function $f(x)$ is defined, if this is not the whole of $\mathbb{R}$, the set F and either one or two black box functions (subroutines). The first function,

which must always be supplied, will, when given a value of x, return $f(x)$. The second black box function, only needed if $f(x)$ is convex on any interval, returns the derivative $f'(x)$. By black box function, we mean that the calling program will know only the name of the function, that the only input will be the point at which the function is to be evaluated, the only output the function value, and no knowledge of how the value is calculated is available to other program units. Note that the second derivative is not needed, only the information, given by F, about where it is zero.

If $f(x)$ is symmetric about $x = c$, then this information can also be supplied, and F need only contain the inflection points for $x > c$. In this case, the routine will generate, in the first instance, points $x > c$ and then return either x or $c - x$, with probability 1/2 each way.

It is not important if the function evaluations $f(x)$ and $f'(x)$ take a long time, since these functions will only be called when a new triangle is placed under $f(x)$, an event that does not happen very often.

The method used for placing triangles varies according to whether the triangle is in a convex or concave region. For a convex region, triangles are placed with all 3 vertices on the curve $f(x)$, whereas in a concave region triangles are placed with all 3 sides tangent to $f(x)$, as described in Chapter 5. No triangle must be allowed to be partly in a convex region and partly in a concave region. The easiest way of ensuring this is to use all the points of F as vertices and points of tangency of the first few triangles placed, as in Figure 5–10.

An immediate question is how we choose the points on the curve $f(x)$ that determine the placing of triangles. The method suggested in Chapter 5 cannot be used, because it involves placing all points once and for all, in order to cover a certain chosen percentage of the region under $f(x)$, whereas we now wish to place new points one at a time, leaving open how much of the region will eventually be covered. When only a few points are required from the distribution, only a few triangles will be needed, but the greater the number of points to be generated sequentially (as one sample, or as a set of independent samples all required together), the closer the region covered by triangles will get to being the whole

region. To choose a point on the curve, it naturally suffices to choose a point x from the interval $[a, b]$, since the second coordinate is then given by $f(x)$. The first points chosen can be the ends of the interval $[a, b]$, if a or b or both are finite (or $[c, b]$ in the symmetric case), and the points in F. Further points will then be chosen between these, in order, ie if in the first round points have been placed at $\{x_1, x_2, \ldots, x_n\}$, then in the second round they will be placed between x_1 and x_2, between x_2 and x_3 and so on. The term 'round' is used here to show that there is a hierarchy among the points placed on the curve and therefore among the triangles placed in the region under the curve. The first points are placed along the curve in order from left to right. Whenever the algorithm has to return to the left end of the interval, in order to place points between those previously placed, then a new round starts and a new set of triangles will be added.

What happens at the ends of the interval depends on whether $f(x)$ is defined on a finite or an infinite interval. If the interval is finite at any end, then that end of the interval will be used to place a point on $f(x)$ in the first round and subsequent points will be placed further in. This is the easy case. If, however, the interval is infinite in any direction, on the right say, which can only happen if $f(x)$ is convex at that end of the interval, it will never be possible to place a rightmost point. At each round there will be a temporary rightmost point, but in the next round another point must be placed further to the right of that one.

We are now faced with two important and difficult questions. Firstly, if points have already been placed at x_1 and x_2 and it is time to place a new point between them, where do we put it? Secondly, if $f(x)$ is defined on $[a, \infty)$ and is therefore convex on $[b, \infty)$ $(b >= a)$, and $x_1 > b$ is the rightmost point placed so far, where do we put the next point $x_2 > x_1$? We will refer to this second problem as the problem of placing a new tail triangle. We would wish to answer these questions in such a way that all the triangles placed in one round have areas that are very close to each other in size, and all the triangles added in the next round have much smaller areas. When placing a new tail triangle, we would like to cover about half of the remaining area under the tail.

We will first concentrate on the question of where to place new points on an

interval where $f(x)$ is concave. Ideally we would like to place a new point between two old ones in such a way that the two wedges left between the new triangle and the curve have equal areas. Even if we allow integration of $f(x)$, which does not seem a desirable option, it is still not easy to satisfy this demand. In practice, it is often satisfactory when placing a point on the curve $f(x)$, between points with abscissas x_1 and x_2, to simply take the point with abscissa $\bar{x} = (x_1 + x_2)/2$.

However this method is very unsatisfactory in the case of a purely concave curve on an interval $[a, b]$ with an infinite derivative at a or b. For an arc of a circle, it is easy to place a new point so that the areas of the wedges will be equal, by choosing equal angles from the centre of the circle, therefore subdividing the arc x_1x_2 into two arcs of equal length. For an ellipse, it suffices to subdivide the arc of the circle with same centre as the ellipse and with radius equal to the axis of the ellipse in the x direction, and then to project down from the circle on to the ellipse, parallel to the y-axis, as illustrated in Figure 6–1. We therefore propose this method for placing a point $\bar{x}$ between x_1 and x_2 in $[0, b]$ on a density curve $f(x)$ concave on $[0, b]$, when the value of the derivative is zero at 0 and infinity at b: project parallel to the y-axis the arc of $f(x)$ for $x \in [x_1, x_2]$ onto the circle of centre $(0,0)$ and radius b, divide the obtained arc of the circle into two equal pieces and project back down onto $f(x)$.

Using polar coordinates for the circle, we have $x_1 = b\cos(\theta_1)$, $x_2 = b\cos(\theta_2)$ and we wish to take $\bar{x} = b\cos((\theta_1 + \theta_2)/2)$. Simple development using trigonometric identities yields

$$\bar{x} = \sqrt{\frac{1}{2}\left[b^2 + x_1x_2 - \sqrt{(b^2 - x_1^2)(b^2 - x_2^2)}\,\right]}$$

so no angles need to be calculated explicitly. The class of densities for which this method is appropriate (with a translation of the origin where necessary), naturally includes those purely concave densities which are symmetric about some point $x = c$ and have infinite derivatives at the ends of the interval on which they are defined.

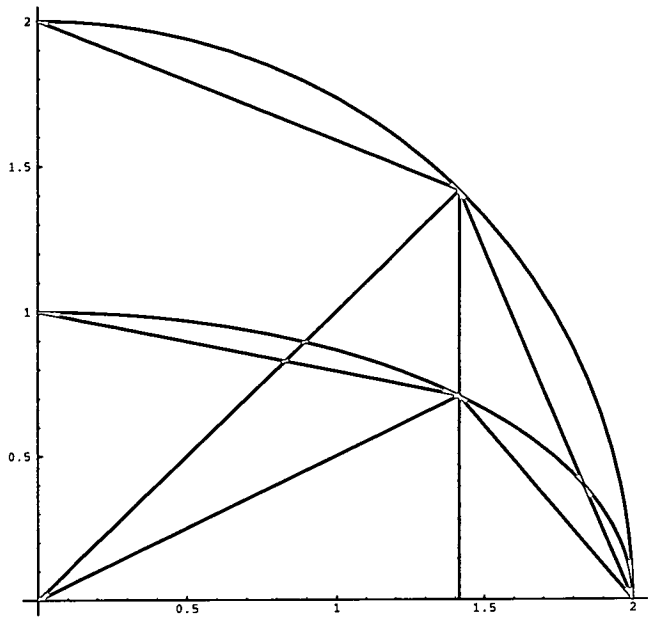


Figure 6–1:

6.3 Using uniform random numbers

So far, we have only discussed how to choose the vertices of new triangles to be placed under the curve $f(x)$. We have not said when new triangles should be added or how we choose the triangle in which to generate a point each time. Nor have we yet mentioned the data structures used to store information about the triangles as they are put in. We have implemented two different methods, one that relies on a generator of uniform random numbers from the interval $[0, 1]$ and another that relies on a supply of random bits. We will describe these two methods in detail, answering all the remaining questions for each method in turn. There are other possible ways of proceeding, including hybrids of the two methods we are going to describe. After reading the description of the two methods, the reader will be able to imagine how to make mixtures of the two.

The first method, that based on a uniform random number generator, is simple to program and to describe. It is more realistic, since everyone should be able to

Table 6–1: Inversion by sequential search

Generate a uniform $[0,1]$ random variate U.
Set $n \leftarrow 0$.
While $U > S_n$ do
    $n \leftarrow n+1$;
    if triangle T_n has not been placed then
      place a new triangle T_n;
      $S_n \leftarrow S_{n-1}+$ area of T_n
Generate the x-coordinate of a point uniformly distributed in T_n.

obtain a good 'cheap' source of uniform random numbers, whereas a good fast generator of random bits is harder to come by (if it exists at all). When using a high level programming language, such as C or Fortran, it is more natural to work with floating-point numbers than with bits. However, as will be seen below, the random bit model has various optimal features, making it a much more interesting model theoretically.

The method based on uniform random numbers is essentially the method for generation of a discrete random variate called 'inversion by sequential search' by Devroye [12]. The method is outlined in Table 6–1.

It is well-known that the expected number of iterations is bounded by the entropy of the discrete distribution, in this case by the entropy of the areas of the triangles. Theoretically the number of triangles is infinite, but the size of the triangles should decrease fast enough that the entropy stays finite and quite small.

The accuracy of this method depends on the accuracy of the uniform random number generator. However many decimal places or bits of accuracy are ensured by the random number generator, the rest of the calculations should be designed to give results to at least the same accuracy. Using any source of uniform deviates, there will always be a largest number $U_m < 1$ that can be generated. However many random numbers are generated with density $f(x)$, the number of triangles

used will never go beyond the number necessary to cover a region with total area U_m. Therefore, the only way to increase the overall accuracy, is to replace the uniform random number generator with a more accurate one, that will necessarily possess a larger maximum value U_m.

The storage requirements imposed by this method are an array to contain the x-coordinates of the vertices of the triangles (the y-coordinates are not necessary), an array, S, of partial sums of probabilities and a linked list of (x-coordinates of) points placed on the curve $f(x)$. So for each triangle inserted, there will be a total of 5 floating-point numbers and one pointer stored. All three lists of data need to be of maximum length equal to the maximum number of triangles that could be inserted using a particular random number generator. It is helpful to choose a programming language, such as C, that allows dynamic allocation of memory, since, whenever a new density f is used, it is impossible to know in advance how many triangles will be needed.

It is possible to write versions of this method that use separate data tables for convex and concave regions, and that first choose a convex or a concave interval with the right probability, before selecting a triangle within the chosen region. Such a version can be more efficient, but needs to be given the areas of each of the convex and concave regions as well as the basic inputs described at the beginning of Section 6.2. In any case, when placing a new triangle, the program must know whether the triangle is being placed under a convex or a concave part of the curve.

A detail that should be mentioned, is how points are generated uniformly in triangles. We have used the method of linear transformation mentioned by Devroye [12]. Points generated uniformly in the basic triangle $((0,0),(1,0),(0,1))$ are linearly transformed to place them in the triangle with vertices having x-coordinates x_1, x_2, x_3. We only need the x-coordinate of the random point. The algorithm is given in Table 6–2.

Clearly generation of one random number from the distribution given by $f(x)$ using this method uses a total of 3 uniform random deviates: one to choose a triangle and two to generate a point within the triangle. It is possible to do as Marsaglia ([38], [37]) does for the fast generation of normal deviates, use some of

Table 6–2: Generation of a point in a triangle

Generate uniform $[0, 1]$ random variates U, V.
If $U > V$ then $U \leftrightarrow V$.
Return $Ux_1 + (V - U)x_2 + (1 - V)x_3$

the bits from a uniform random number for choosing the triangle and the remaining bits for placing a point in the triangle. However, whereas all the rectangles used by Marsaglia have areas that are integer multiples of a given area, so that there is a definite number of bits needed to choose a rectangle with perfect accuracy, the triangles we use do not have such a property. It only makes sense though to use as much accuracy when choosing a triangle as the accuracy with which the area has been calculated and stored.

6.4 Using random bits

The second method we have implemented uses random bits for choosing a triangle with the correct probability and is very much based on work of Knuth and Yao [27]. For this part, we have to presume the existence of a perfect random bit generator. Knuth and Yao prove the existence of and describe how to build optimal discrete distribution generating trees (DDG-trees).

Given a discrete density $P(X = i) = p_i$, $i = 1, \ldots, \infty$, a DDG-tree for that density is a binary tree that, traversed while reading a sequence of random bits, returns a value n with probability p_n. Starting at the root of the tree, the left or right branch is followed according to whether the first bit generated is 0 or 1. Either a terminal node is reached, or a node with 2 branches. In the latter case the process is repeated with subsequent bits until such time as a terminal node is reached. Each terminal node is labeled with a number n, indicating that the value n is to be returned. An optimal DDG-tree is one that, for a given distribution, minimises the expected number of bits needed to return each number. Following

Knuth and Yao, we shall use Borel's functions

$$\epsilon_k(x) = \lfloor 2^k x \rfloor \bmod 2$$

to specify the coefficient of 2^{-k} in the binary representation of x. Knuth and Yao show that an optimal DDG-tree is a DDG-tree with the property that level k of the tree contains one (and no more than one) terminal node with the label n if and only if $\epsilon_k(p_n) = 1$. We will call this property the 'DDG-tree optimality condition'.

The most useful part for our purposes is that Knuth and Yao also describe how, starting with an optimal DDG-tree for a discrete density p_i, $i = 1, \ldots, n$, it is possible to extend the tree to obtain an optimal DDG-tree for the density $p_1, \ldots, p_{n-1}, \bar{p}_n, \bar{p}_{n+1}$ with $\bar{p}_n + \bar{p}_{n+1} = p_n$. This allows us to build up dynamically, as the triangles are inserted, an optimal DDG tree for the density given by the triangles under the curve $f(x)$.

We start with a tree consisting only of the root node. At any stage, when we have a number n of triangles already positioned, we can write

$$\sum_{i=0}^{n} p_i = 1$$

where p_i, $i \geq 1$, is the area of triangle i and by p_0 we denote the total area under the curve f not yet covered by triangles. We use a DDG-tree where every node is labeled with an integer from the set $\{-1, 0, 1, 2, 3, \ldots\}$. Each node with a label $i > 0$ is a terminal node and points to triangle i. A node with the label -1 is an interior node (whose right and left children may or may not have been added to the tree yet) and a node labeled 0 is a terminal node, that may later become an interior node, pointing to the area under the curve outside the existing triangles.

If q is a node of the tree, let $L(q)$ denote the label stored at that node. The outline of the algorithm used to choose a triangle is given in Table 6–3. Two parts of the algorithm need to be described in detail: extension of the tree and refinement of the tree. These two tasks must be carried out in such a way as to ensure that the DDG-tree optimality condition always holds.

This is how we extend the tree: When an interior node without children is reached on level $k - 1$ of the tree, the children of that node must be added to level

Table 6–3: Choosing a triangle using a DDG-tree

Set decided $\leftarrow$ false, $q \leftarrow$ root;

Repeat

Generate a random bit, b.

If node q has no children then extend the tree at q;

If $b = 0$ then $q \leftarrow$ left child of q;

else $q \leftarrow$ right child of q;

While $L(q) = 0$ refine the tree;

If $L(q) > 0$ then

$n \leftarrow L(q)$;

decided $\leftarrow$ true

Until decided.

Generate the x-coordinate of a point uniformly distributed in triangle T_n.

k. If $\epsilon_k(p_i) = 1$ for some $0 \leq i \leq n$ and if there is as yet no terminal node, q, on level k, with $L(q) = i$, then we should create such nodes. We have to consider the p_i for inclusion in a clearly defined order, so we choose the order $1, 2, \ldots, n, 0$. An 'accounting' array, M, with one element for each level of the tree, keeps track of which probability we should next examine at level k. So we examine probability p_{M_k}, to see if $\epsilon_k(p_{M_k}) = 1$. If it does, we add a terminal node with label M_k. Then we examine the next probability, until either two terminal nodes have been added or else we have examined all probabilities, including p_0, and still have one or two children to add. Any remaining children are then interior nodes. Unlike the other probabilities, p_0 changes every time a new triangle is added. Therefore the accounting for p_0 is more complicated and we use a separate array, Z, again with an element for each level of the tree, to keep track of whether p_0 has been examined when adding nodes to each level. The outline of the extension algorithm is given in Table 6–4.

By refinement of the tree, we mean the addition of a new triangle and the updating operations involved. If a terminal node with label 0 is reached when

traversing the tree, a new triangle, with area p_{n+1}, must be inserted and p_0 must be decreased accordingly. Whereas extension is a local operation, carried out at one node of the tree only, refinement involves the whole tree generated so far. When the value of p_0 changes, all terminal nodes labeled 0 need to be updated. Each such node, at level k, may only retain the label 0 if the new value of p_0 contains the digit 1 in the kth binary place and if there is no probability p_i, $n+1 > i \geq M_k$, with $\epsilon_k(p_i) = 1$. If such a p_i exists, the label is changed to i, otherwise, if $\epsilon_k(p_0) = 0$, the label becomes -1 and the node therefore changes from a terminal node to an interior node. The refinement is done recursively, node by node. Starting with the root, each node is examined in turn. If the node is labeled zero, it is updated, if not, the right and left children are examined, if they exist. The basic routine to refine the tree when a new triangle is added is outlined in Table 6–5.

This routine calls the routine refine(q, k), which updates the sub-tree with root at node q on level k. The routine is given in Table 6–6.

An example of a computer program written using the above methods is given in Appendix A.5.

Table 6–4: Extension of the DDG-tree

Allocate storage for 2 new nodes, $q1$ and $q2$.

Set $q1_l, q1_r, q2_l, q2_r \leftarrow$ NULL;

$i \leftarrow M_k$;

While $i \leq n$ and $\epsilon_k(p_i) = 0$ do $i \leftarrow i + 1$;

If $i \leq n$

  then

    $L(q1) \leftarrow i$;

    $M_k \leftarrow i + 1$;

  else

    If $Z_k = 0$ and $\epsilon_k(p_0) = 1$

      then

        $L(q1) \leftarrow 0$;

        $Z_k \leftarrow 1$;

      else $L(q1) \leftarrow -1$

$i \leftarrow M_k$;

While $i \leq n$ and $\epsilon_k(p_i) = 0$ do $i \leftarrow i + 1$;

If $i \leq n$

  then

    $L(q2) \leftarrow i$;

    $M_k \leftarrow i + 1$;

  else

    If $Z_k = 0$ and $\epsilon_k(p_0) = 1$

      then

        $L(q2) \leftarrow 0$;

        $Z_k \leftarrow 1$;

      else $L(q2) \leftarrow -1$;

Left child of $q \leftarrow q1$

Right child of $q \leftarrow q2$

Table 6–5: Refinement of the DDG-tree

$n \leftarrow n + 1$;

place triangle T_n;

$p_0 \leftarrow p_0 -$ area of T_n;

refine($root, 0$);

Table 6–6: Routine refine(q,k)

If $L(q) = 0$

then

$Z_k \leftarrow 0$;

$i \leftarrow M_k$;

While $i \leq n$ and $\epsilon_k(p_i) = 0$ do $i \leftarrow i + 1$;

If $i \leq n$

then

$L(q) \leftarrow i$;

$M_k \leftarrow i + 1$;

else

If $\epsilon_k(p_0) = 1$

then $Z_k \leftarrow 1$;

else $L(q) \leftarrow -1$;

else

if q has a left child, q_l, then refine($q_l, k + 1$);

if q has a right child, q_r, then refine($q_r, k + 1$);

Bibliography

[1] R. Azencott. Formule de taylor stochastique et developpement asymptotique d'intégrales de feynmann. In *Séminaire de Probabilités XVI, Supplément: Géometrie differentielle stochastique*, pages 237–284. Springer-Verlag, 1980/81.

[2] G. Ben Arous. Flots et series de Taylor stochastiques. *Probab. Theory Related Fields*, 81:29–77, 1989.

[3] K. Bichteler. Stochastic integration and l^p theory of semimartingales. *Ann. of Prob.*, 9(1):49–89, 1981.

[4] F. Castell. Asymptotic expansion of stochastic flows. *Probab. Theory Related Fields*, 96(2):225–239, 1993.

[5] F. Castell and J. G. Gaines. The ordinary differential equation approach to asymptotically efficient schemes for solution of stochastic differential equations. To appear in Annales de l'I.H.P.

[6] K.-T. Chen. Integration of paths, geometric invariants and a generalized Baker-Hausdorff formula. *Ann. of Math.*, 65(1):163–178, 1957.

[7] J. M. C. Clark. An efficient approximation for a class of stochastic differential equations. In W. H. Fleming and L. G. Gorostiza, editors, *Advances in filtering and optimal stochastic control, Proceedings of IFIP-WG7/1 Working Conference, Cocoyoc, Mexico, 1982*, number 42 in Lecture Notes in Control and Information Sciences. Springer-Verlag, Berlin, 1982.

[8] J. M. C. Clark. The discretization of stochastic differential equations: A primer. In H. Neunzert, editor, *Road Vehicle Systems and Related Mathematics, Proc. Second Workshop Turino.* Series Mathematical Methods in Technology, 1987.

[9] J. M. C. Clark and R. J. Cameron. The maximum rate of convergence of discrete approximations for stochastic differential equations. In B. Grigelionis, editor, *Stochastic Differential Systems*, number 25 in Lecture Notes in Control and Information Sciences. Springer-Verlag, Berlin, 1980.

[10] L. Comtet. *Analyse Combinatoire, Tome Premier.* Number 4 in Le Mathématicien. Presses Universitaires de France, Paris, 1970.

[11] J. Devlin. Word problems related to periodic solutions of a non-autonomous system. *Math. Proc. Cambridge Philos. Soc.*, 108:127–151, 1990.

[12] Luc Devroye. *Non Uniform Random Variate Generation.* Springer-Verlag, 1986.

[13] C. Doléans-Dade. On the existence and unicity of solutions of stochastic integral equations. *Z. Wahrscheinlichkeitstheorie verw. Gebiete*, 36:93–101, 1976.

[14] O. Faure. Numerical pathwise approximation of stochastic differential equations. Technical report, CERMA-ENPC, 1990.

[15] O. Faure. *Simulation du mouvement brownien et des diffusions.* PhD thesis, Ecole Nationale des Ponts et Chaussées , CERMA-ENPC, 1991.

[16] M. Fliess. Fonctionnelles causales non-linéaires et indéterminées non-commutatives. *Bull. Soc. Math. France*, 109:3–40, 1981.

[17] J. G. Gaines. The algebra of iterated stochastic integrals. *Stochastics and Stochastic Reports*, 49:169–179, 1994.

[18] J. G. Gaines and T. J. Lyons. Random generation of stochastic area integrals. *SIAM J. of Applied Math.*, 54(4):1132–1146, 1994.

[19] T. C. Gard. *Introduction to Stochastic Differential Equations.* Monographs and Textbooks in Pure and Applied Mathematics. Marcel Dekker, Inc., New York and Basel, 1988.

[20] H. Gjessing, H. Holden, T. Lindstrøm, B. Øksendal, J. Ubøe, and T.-S. Zhang. The Wick product. In A. Melnikov, editor, *Frontiers in Pure and Applied Probability, Vol. I.* TVP Science Publishers, Moscow, 1993.

[21] Y. Z. Hu. Serie de Taylor stochastique et formule de Campbell-Haussdorff, d'apres Ben Arous. In J. Azema, P. A. Meyer, and M. Yor, editors, *Séminaire de Probabilités XXV*, number 1485 in Lecture Notes in Mathematics, pages 579–586. Springer-Verlag, 1991/92.

[22] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proc. IRE*, 40:1098–1101, 1952.

[23] K. Itô. Stochastic differential equations in a differentiable manifold. *Nagoya Math. Journ.*, 1:35–47, 1950.

[24] P. E. Kloeden and E. Platen. A survey of numerical methods for stochastic differential equations. *Journal of Stochastic Hydrology and Hydraulics*, 3:155–178, 1989.

[25] P. E. Kloeden and E. Platen. Stratonovich and Itô stochastic Taylor expansions. *Math. Nachr.*, 151:33–50, 1991.

[26] P. E. Kloeden and E. Platen. *Numerical Solution of Stochastic Differential Equations*, volume 23 of *Applications of Mathematics.* Springer-Verlag, 1992.

[27] D. E. Knuth and A. C. Yao. The complexity of nonuniform random number generation. In J. E. Traub, editor, *Algorithms and compexity: new directions and recent results*, pages 357–428, New York, N.Y., 1976. Academic Press.

[28] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA., 1973.

[29] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.

[30] P. Lévy. *Processus Stochastiques et Mouvement Brownien*. Monographies des Probabilités. Gauthier-Villars, Paris, 1948.

[31] Paul Lévy. Wiener's random function, and other Laplacian random functions. In J. Neyman, editor, *Second Berkeley Symposium on Mathematical Statistics and Probability*, 1951.

[32] T. Lindstrøm, B. Øksendal, and J. Ubøe. Stochastic differential equations involving positive noise. In M. T. Barlow and N. H. Bingham, editors, *Stochastic Analysis*, pages 261–303. Cambridge Univ. Press, 1991.

[33] M. Lothaire. *Combinatorics on Words*. Encyclopedia of Mathematics and its Applications. Addison-Wesley, Reading, Mass., 1983.

[34] R. C. Lyndon. On Burnside problem I. *Trans. Amer. Math. Soc.*, 77:202–215, 1954.

[35] T.J. Lyons. The interpretation and solution of ordinary differential equations driven by rough signals. *Maths. Research Letters I*, pages 451–464, 1994.

[36] G. Marsaglia. The squeeze method for generating gamma variates. *Comput. Math. Appl.*, 3:321–325, 1977.

[37] G. Marsaglia, K. Anantharayanan, and N. J. Paul. Improvements on fast methods for generating normal random variables. *Inform. Process. Lett.*, 5(2):27–30, 1976.

[38] G. Marsaglia, M. D. MacLaren, and T. A. Bray. A fast procedure for generating normal random variables. *Comm. ACM*, 7:4–10, 1964.

[39] Maruyama. Continuous Markov processes and stochastic equations. *Com Rend. Circ. Mat. Palermo*, 4:48–90, 1955.

[40] G. Melançon. Combinatorics of Hall trees and Hall words. *J. Combin. Theory Ser. A*, 59(2):285–308, 1992.

[41] G. Melançon and C. Reutenauer. Lyndon words, free algebras and shuffles. *Canad. J. Math.*, 41(4):577–591, 1989.

[42] P.-A. Meyer. Sur deux estimations d'intégrales multiples. In J. Azéma, P. A. Meyer, and M. Yor, editors, *Séminaire de Probabilités XXV*, pages 425–426. Springer-Verlag, 1991.

[43] G. N. Milshtein. Approximate integration of stochastic differential equations. *Th. Prob. Appl.*, 19, 1974.

[44] N. J. Newton. An asymptotically efficient difference formula for solving stochastic differential equations. *Stochastics*, 19:175–206, 1986.

[45] N. J. Newton. An efficient approximation for stochastic differential equations on the partition of symmetrical first passage times. *Stochastics and Stochastic Reports*, 29:227–258, 1989.

[46] N. J. Newton. Asymptotically efficient Runge-Kutta methods for a class of Itô and Stratanovich equations. *SIAM J. of Applied Mathematics*, 51(2):542–567, 1991.

[47] E. Pardoux and D. Talay. Discretization and simulation of stochastic differential equations. *Acta Appl. Math.*, 3:23–47, 1985.

[48] E. Platen and W. Wagner. On a taylor formula for a class of Itô processes. *Prob. and Math. Stats.*, 3:37–51, 1982.

[49] P. Protter. On the existence, uniqueness, convergence and explosions of solutions of systems of stochastic differential equations. *Ann. Prob.*, 5:243–261, 1977.

[50] D. E. Radford. A natural ring basis for the shuffle algebra and an application to group schemes. *J. Algebra*, 58:432–453, 1979.

[51] R. Ree. Lie elements and an algebra associated with shuffles. *Ann. of Math.*, 68:210–220, 1958.

[52] C. Reutenauer. *Free Lie Algebras.* London Mathematical Society Monographs, New Series 7. Oxford Science Publications, Oxford, 1993.

[53] L. C. G. Rogers and D. Williams. *Diffusions, Markov Processes and Martingales 2, Itô Calculus.* John Wiley and Sons, 1987.

[54] W. Rümelin. Numerical treatment of stochastic differential equations. *SIAM J. Numer. Anal.*, 19(3), 1982.

[55] H. J. Sussmann. Product expansions of exponential Lie series and the discretization of stochastic differential equations. In W. Fleming and P. L. Lions, editors, *Stochastic Differential Systems, Stochastic Control Theory and Applications*, volume 10 of *The IMA volumes in Mathematics and its Applications.* Springer-Verlag, 1988.

[56] D. Talay. Simulation and numerical analysis of stochastic differential systems: A review. Technical Report 1313, INRIA, 1990.

[57] N. Wiener. Un problème de probabilités dénombrables. *Bull. Soc. Math. France*, 52:569–578, 1924.

[58] E. Witt. Treue Darstellung Liescher Ringe. *Journal für die Reine und Angewandte Mathematik*, 177:152–160, 1937.

Appendix A

Computer Programs

All the code in this appendix is written in C, except for Section A.3.

A.1 Access and dynamic generation of a Brownian tree

```
#define K 5

typedef struct{ int recno,level,prerec,nexrec,parent;
                int lson,rson,spare[7];
                float t0,tp;} record;
typedef struct{ int np,nw,nlevels,nl1,seed,nrecs1,nrecs2;
                int spare[7];
                float sp[2];} header;

getdw(nw,dw,level,t,h,func,idum,cangoup)
int nw,level,idum,*cangoup;
float *dw,t,h,(*func)();
/* Routine to read and return elements from a Brownian
tree stored in a file, also to add elements to the tree
when necessary.
The external variables wfile and in, that may be used
by other program units, are pointers to the file
containing the Brownian tree and to the io-unit
connected to the file.
The routine keeps several (K) records in memory at all times,
so that file accesses are not performed during
most calls to the routine.
getdw presumes that it is being used to help solve an sde
forwards through time, ie it presumes that the next call
to getdw will be at the same period or the next one.
*/
```

```
{extern char *wfile;
 extern  FILE *in;
 static float ***w;
 static record r[K];
 static header head;
 static int l,np,recsize,init=0;
 static long size;
 float ***array3(),hh,tt,sd,**w2,**matrix();
 int i,j,k,tag;
 record r2;
 if(init==0||idum<0) /* Initialize by reading in K records, such that
                the last is on given level and the others are
                all parents. */
 {in=fopen(wfile,"r+");
  if(in==NULL) {printf(" Can't open file in getdw\n"); exit(1);}
  recsize=sizeof(record);
  fread((char *)&head,recsize,1,in);
  if(nw!=head.nw) {printf(" Noise vector wrong size in %s.\n",wfile);
                   printf(" nw=%d head.nw=%d\n",nw,head.nw);
                   exit(1);}
  np=head.np;
  if(head.nlevels<level)
  {printf(" Not enough levels in initial brownian path\n");
   exit(1);}
  if(init==0) w=array3(0,K-1,0,np-1,0,nw-1);
  size=recsize+np*nw*sizeof(float);
  fseek(in,size,0);
  readw(w[0],np,nw,r,in);
  while(r[0].level<level-K+1||t<r[0].t0) readw(w[0],np,nw,r,in);
  for(i=1;i<K;i++)
  {fseek(in,size*r[i-1].lson,0);
   readw(w[i],np,nw,r+i,in);
  }
  l=level;
  init=1;
 }
 while(level>l) /* Go down one or more levels */
 { for(i=0;i<K-1;i++)
   { r[i]=r[i+1];
     for(j=0;j<np;j++) for(k=0;k<nw;k++) w[i][j][k]=w[i+1][j][k];
   }
  if(r[K-1].lson>0)
  {fseek(in,size*r[K-1].lson,0);
   readw(w[K-1],np,nw,r+K-1,in);
   l++;
  }
  else /* Create new noise elements */
  {i=1<<l;
   hh=1.0/i;
   sd=sqrt(hh*0.5);
   w2=matrix(0,np-1,0,nw-1);
   brownchunk(w[K-2],np,nw,w[K-1],w2,sd,func);
   r[K-1].recno=head.nrecs1++;head.nrecs2++;
   l++; r[K-1].level=l;
   r[K-1].nexrec=r[K-1].recno+1;
   r[K-1].parent=r[K-2].recno;r[K-1].lson=r[K-1].rson=(-1);
```

```
   r[K-1].prerec=(-1);
   r[K-2].lson=r[K-1].recno;
   r[K-1].t0=r[K-2].t0;
   r[K-1].tp=r[K-1].t0+np*hh;
   fclose(in);
   in=fopen(wfile,"a+");
   i=savew(w[K-1],np,nw,r+K-1,in);
   fclose(in);
   in=fopen(wfile,"r+");
   r2.recno=head.nrecs1++;head.nrecs2++;r2.level=l;
   r2.prerec=r2.recno-1;
   r2.parent=r[K-1].parent;r2.lson=r2.rson=(-1);
   r2.nexrec=(-1);
   r[K-2].rson=r2.recno;
   r2.t0=r[K-1].tp;r2.tp=r2.t0+np*hh;
   fclose(in);
   in=fopen(wfile,"a+");
   i=savew(w2,np,nw,&r2,in);
   fclose(in);
   in=fopen(wfile,"r+");
   free_matrix(w2,0,np-1,0,nw-1);
   /* Rewrite header */
   fwrite((char *)&head,recsize,1,in);
   /* Rewrite record(K-2) to update lson,rson */
   fseek(in,r[K-2].recno*size,0);
   savew(w[K-2],np,nw,r+K-2,in);
  }
}

while(l-K+1>level) /* Go up one or more levels */
{ for(i=K-1;i>0;i--)
  { r[i]=r[i-1];
    for(j=0;j<np;j++) for(k=0;k<nw;k++) w[i][j][k]=w[i-1][j][k];
  }
  fseek(in,size*r[1].parent,0);
  readw(w[0],np,nw,r,in);
  l--;
}

if(t<r[0].t0) {printf(" Error: t<t0 in getw.\n"); exit(1);}
/* Get correct level */
  for(i=0;i<K;i++) if(r[i].level==level) {tag=i; break;}

  while(t>=r[tag].tp) /* Get correct time interval */
  { /* Move tag down to K-1 */
   r[K-1]=r[tag];
   for(j=0;j<np;j++) for(k=0;k<nw;k++) w[K-1][j][k]=w[tag][j][k];
   tag=K-1; l=level;
   if(r[tag].nexrec>0)
   {fseek(in,r[tag].nexrec*size,0);
    readw(w[tag],np,nw,r+tag,in);
   }
   else
   {r[0]=r[tag];
    initstack();
    while(r[0].nexrec<0)
```

```
     {push(r[0].recno);
      fseek(in,r[0].parent*size,0);
      readw(w[0],np,nw,r,in);
     }
     fseek(in,r[0].nexrec*size,0);
     readw(w[0],np,nw,r,in);
     while(r[0].level<level) /* Create more tree */
     {
      /* Rewrite parent */
      fseek(in,r[0].recno*size,0);
      r[0].lson=head.nrecs1;r[0].rson=head.nrecs1+1;
            fwrite((char *)r,recsize,1,in);
      hh=1.0/(1<<r[0].level);
      sd = sqrt(0.5*hh);
      brownchunk(w[0],np,nw,w[1],w[2],sd,func);
      r[1]=r[0];
      r[1].recno=head.nrecs1++;head.nrecs2++;
      r[1].parent=r[0].recno;
      r[1].prerec=pop(); printf(" popping %d\n",r[1].prerec);
      r[1].level++;r[1].nexrec=r[1].recno+1;
      r[1].lson=r[1].rson=(-1);
      r[1].tp=r[0].t0+np*hh;
      tellprerec(r[1].prerec,r[1].recno,recsize,size,in);
      fclose(in);
      in=fopen(wfile,"a+");
      savew(w[1],np,nw,r+1,in);
      fclose(in);
      in=fopen(wfile,"r+");
      r[2]=r[1];head.nrecs1++;head.nrecs2++;
      r[2].recno++;r[2].prerec=r[2].recno-1;
      r[2].nexrec=(-1);
      r[2].t0=r[2].tp;r[2].tp+=np*hh;
      fclose(in);
      in=fopen(wfile,"a+");
      savew(w[2],np,nw,r+2,in);
      fclose(in);
      in=fopen(wfile,"r+");
      for(k=0;k<np;k++) for(i=0;i<nw;i++)
       w[0][k][i]=w[1][k][i];
      r[0]=r[1];
      /* Rewrite header */
      fwrite((char *)&head,recsize,1,in);
     }
     r[K-1]=r[0];
     for(j=0;j<np;j++) for(k=0;k<nw;k++) w[K-1][j][k]=w[0][j][k];
     freestack();
    }
/*Fill in w and r */
     for(i=K-2;i>=0;i--)
     {fseek(in,r[i+1].parent*size,0);
      readw(w[i],np,nw,r+i,in);
     }
   }
/* Return correct dw */
 tt=r[tag].t0; i=0;
 while(tt<t) { i++; tt=r[tag].t0+i*h;}
```

```
 if(i>np-1) {printf(" Error in time.\n"); exit(1);}
 for(j=0;j<nw;j++) dw[j]=w[tag][i][j];
/* Check if possible to go up a level
   next time. It is only possible when the
   current time index = 3 mod 4. */
  *cangoup=((i%4==3&&level>5)?1:0);
}

tellprerec(prerec,nexrec,recsize,size,in)
int prerec,nexrec,recsize;
long size;
FILE *in;
/* Updates the record prerec so that the pointer
to the next record points to nexrec.
*/
{ record r;
  fseek(in,prerec*size,0);
  fread((char *)&r,recsize,1,in);
  r.nexrec=nexrec;
  fseek(in,(long)(-recsize),1);
  fwrite((char *)&r,recsize,1,in);
}

int *sp;

initstack()
/* Initialises a stack, pointed to by sp */
{extern int *sp;
 unsigned stacksize=20; /* Max. no of levels */
 sp=(int *)malloc(stacksize*sizeof(int));
 if(!sp) {printf(" allocation failure in initstack\n"); exit(1);}
}

freestack()
/* Releases the memory used by the stack
pointed to by sp */
{extern int *sp;
 free((char *)sp);
}

push(n) int n;
/* Adds the integer n to the stack */
{extern int *sp;
 *(sp++)=n;
}

int pop()
/* Gets an integer from the top of the stack */
{extern int *sp;
 return (*(--sp));
}
```

A.2 Generation of Lévy areas

The tables of constants used by this routine are much too long to be included here, so we have given only the first two and the last elements of each array. The code is otherwise complete.

```
#define NCUBE 2975   /* Number of blocks easy part is broken into*/
#define DIMP 5949    /* Dimension of tree: 2*NCUBE-1 */
#define SCUBE 119519 /* Total number of blocks when all blocks are
                        broken down into smallest size blocks */
#define NW 4096      /* Number of wedges */
#define NWEDGE 4095  /* Number of wedges - 1 */
#define DIMW 8191    /* Dimension of wtree: 2*NWEDGE+1 */

#define NT 9        /* Number of parts of tail - 1 */
#define DIMT 19     /* Dimension of tailtree: 2*NT-1 */
/* Dimension of square matrix z (of function values) */
#define DIMZ 65

 /* Volume of easy part + volume of wedges */
#define PR1 0.99732406024821951
#define DL 0.0625
#define MAXRAN 4.294967296e9
#define H 0.298023225e-7
#define EPS 1.0e-4

typedef struct
 {unsigned long p; short int rson,lson; float r,a,h;} node;
static node tree[DIMP]=
{
  {119519,2,1,0.0,0.0,0.0},
  {69003,4,3,0.0,0.0,0.0},
  ...
  {107595,-1,-1,1.0,0.0,1.0}
};

typedef struct {double p; short int rson,lson;} wedge;
static wedge wtree[DIMW]=
{
  {9.973240602482193e-01,2,1},
  {9.611933921062400e-01,4,3},
  ...
  {9.361841639166600e-01,-1,-1}
};

typedef struct {double h,dh,base;} triple;
static triple W[NW]=
{
  {0.0,0.048896637226827,0.0},
  {0.0,0.048430181086172,0.0},
  ...
```

```
  {0.0,0.000068025297515,0.000053038700553}
};

typedef struct {double p; short int rson,lson,no;} tailstruct;
static tailstruct tailtree[DIMT]=
{
  {1.000000000000000e+00,1,16,0},
  {1.000000000000000e+00,12,2,0},
  ...
  {9.994249898245016e-01,-1,-1,1}
};

static double z[DIMZ][DIMZ]=
  {{0.0,0.0,0.0,0.0,0.0,
     ...
    0.000055417724037,0.000053038700553}
  };

typedef struct {double l,h;} pair;
static pair epsilon[NW]=
  {{0.0,0.000149092954277},
   {-0.0,0.000140455255677},
    ...
   {-0.000000282727450,0.0}
  };

double f(),f1(),f2(),f3(),f4(),f5(),f6(),f7(),f8(),f9(),f10();

gen_ra(r,a,func,eps)
float *r,*a,(*func)(),eps;
{float u,v,w;
 double xx;
 short int   lson,rson,i,sign;
 unsigned long bits,p;

  w=(*func)(1);
  bits=w*MAXRAN;
  sign=(bits & 0X80000000)>>31;
  p=(bits & 0X7FFFC000)>>14;

  if(p<=SCUBE)
  {lson=tree[0].lson;
   rson=tree[0].rson;
   while(lson>=0)
   {if(p<=tree[lson].p)
     {i=lson; lson=tree[i].lson; rson=tree[i].rson;}
    else
     {i=rson; lson=tree[i].lson; rson=tree[i].rson;}
   }
   u=(*func)(1);
   v=(*func)(1);
   *r=tree[i].r+tree[i].h*u;
   *a=tree[i].a+tree[i].h*v;
  }
  else
  {xx=((bits&0X7FFFFFC0)>>6)*H;
```

```
   if(xx<PR1)
   {i=choose_wedge2(xx);
    wedge_ra(i,func,r,a,eps);
   }
   else
    tail_ra(xx,func,r,a);
  }
  if(sign) *a=(-(*a));
}

choose_wedge2(x) double x;
{int i,rson,lson;
 lson=wtree[0].lson;
 rson=wtree[0].rson;
 while(lson>=0)
 {if(x<=wtree[lson].p)
   {i=lson; lson=wtree[i].lson; rson=wtree[i].rson;}
  else
   {i=rson; lson=wtree[i].lson; rson=wtree[i].rson;}
 }
 if(i<NWEDGE) {printf(" Mistake in choose_wedge2: i=%d\n",i);
               exit(1);
              }
/* in wtree the wedges are in the 2nd half
   and in reverse order */
 i=DIMW-1-i;
 return i;
}

wedge_ra(k,func,r,a,eps)
int k; float (*func)(),*r,*a,eps;
{ double z,zz,interpolate();
  int i,j,count=0,rejected;
  i=k/64;
  j=k-i*64;
  do
  {*r=DL*(i+(*func)(1));
   *a=DL*(j+(*func)(1));
    z=W[k].h+W[k].dh*(*func)(1);
   if(z<W[k].base) rejected=0;
   else
   {zz=interpolate(*r,*a,i,j);
    if(z<zz+epsilon[k].l-eps)   rejected=0;
    else
    {if(z>zz+epsilon[k].h+eps)  rejected=1;
     else
     {if(z<f(*r,*a)) rejected=0;
      else           rejected=1;
     }
    }
   }
  } while(rejected&&count<100);
  if(count>=100)
   printf("wedge %d, 100 numbers rejected: r=%f a=%f\n",k,*r,*a);
}
```

```
double interpolate(r,a,i,j)
 double r,a;
 int    i,j;
{double t,u,r1,r2,a1,a2;
 r1=i*DL;
 r2=(i+1)*DL;
 a1=j*DL;
 a2=(j+1)*DL;
 t=(r-r1)/(r2-r1);
 u=(a-a1)/(a2-a1);
 return
  (1-t)*(1-u)*z[i][j] + t*(1-u)*z[i+1][j] +
    t*u*z[i+1][j+1] + (1-t)*u*z[i][j+1] ;
}

tail_ra(x,func,r,a)
double x;
float (*func)(),*r,*a;
{double z;
 int i,rson,lson,tailno;
 lson=tailtree[0].lson;
 rson=tailtree[0].rson;
 while(lson>=0)
 {if(x<=tailtree[lson].p)
   {i=lson; lson=tailtree[i].lson; rson=tailtree[i].rson;}
  else
   {i=rson; lson=tailtree[i].lson; rson=tailtree[i].rson;}
 }
 if(i<NT) {printf(" Mistake in choosing tail: i=%d\n",i);
               exit(1);
              }
 i=tailtree[i].no;
 do
 {switch(i)
    { case 1: g1(r,a,&z,func); break;
      case 2: g2(r,a,&z,func); break;
      case 3: g3(r,a,&z,func); break;
      case 4: g4(r,a,&z,func); break;
      case 5: g5(r,a,&z,func); break;
      case 6: g6(r,a,&z,func); break;
      case 7: g7(r,a,&z,func); break;
      case 8: g8(r,a,&z,func); break;
      case 9: g9(r,a,&z,func); break;
      case 10: g10(r,a,&z,func); break;
    }
 } while(z>f(*r,*a));
}

double  pisq2=0.047746482927568600 73; /* h/2/Pi */
float   slownorm(),randexp();

double f(r,a)
float a,r;
{static double lim=31.0,h=0.3;
 double x,r2,aa,sum=0.0;
 int i=0;
```

```
 aa=(double)a;
 r2=(-0.5)*r*r;
 x=(-lim);
 while(x<lim)
 {x=i*h-lim;
  sum+= x/sinh(x)*exp(r2*x/tanh(x))*cos(aa*x) ;
  i++;
 }
 return pisq2*r*sum;
}

/* 4<r<12 0<a<4 */
g1(r,a,z,func) float (*func)(),*r,*a; double *z;
{*a=4.0*(*func)(1);
 do{
    *r=4.0+8.0*(*func)(1);
    *z=0.00034*(*func)(1);
   } while(*z>f1(*r,*a));
}

double f1(r,a) float a,r;
{return exp(-0.5*r*r);}

/* 4<r<8 4<a<8 */
g2(r,a,z,func) float (*func)(),*r,*a; double *z;
{
 do{
    *r=4.0+4.0*(*func)(1);
    *a=4.0+4.0*(*func)(1);
    *z=0.00012*(*func)(1);
   } while(*z>f2(*r,*a));
}

double f2(r,a) float a,r;
{static double c=1.5,lam=1.5;
 double rr;
 rr=r*r;
 return c*exp(-0.5*rr-lam*a*a/rr);
}

/* 2<r<4 4<a<8 */
g3(r,a,z,func) float (*func)(),*r,*a; double *z;
{static float lam=0.5*(M_PI);
 *r=2.0+2.0*(*func)(1);
 do{*a=4.0+randexp(lam,func);} while(*a>8.0);
 *z=f3(*r,*a)*(*func)(1);
}

double f3(r,a) float a,r;
{static float lam=0.5*(M_PI);
 return exp(-lam*a);
}

/* 0<r<0.5 4<a<6 */
g4(r,a,z,func) float (*func)(),*r,*a; double *z;
{static float lam=(M_PI);
```

```
 float x,y;
 x=0.5*(*func)(1);
 y=0.5*(*func)(1);
 *r=(x>y?x:y);
 do{*a=4.0+randexp(lam,func);} while(*a>6.0);
 *z=(*func)(1)*f4(*r,*a);
}

double f4(r,a) float a,r;
{static double lam=(M_PI),c=15.0;
 return c*r*exp(-lam*a);
}

/* 0.5<r<1 4<a<6 */
g5(r,a,z,func) float (*func)(),*r,*a; double *z;
{static float lam=2.8;
 float x,y;
 x=0.5+0.5*(*func)(1);
 y=0.5+0.5*(*func)(1);
 *r=(x>y?x:y);
 do{*a=4.0+randexp(lam,func);} while(*a>6.0);
 *z=(*func)(1)*f5(*r,*a);
}

double f5(r,a) float a,r;
{static double lam=2.8,c=15.0;
 return c*r*exp(-lam*a);
}

/* 1<r<1.5 4<a<6 */
g6(r,a,z,func) float (*func)(),*r,*a; double *z;
{static float lam=2.6;
 float x,y;
 x=1.0+0.5*(*func)(1);
 y=1.0+0.5*(*func)(1);
 *r=(x>y?x:y);
 do{*a=4.0+randexp(lam,func);} while(*a>6.0);
 *z=(*func)(1)*f6(*r,*a);
}

double f6(r,a) float a,r;
{static double lam=2.6,c=25.0;
 return c*r*exp(-lam*a);
}

/* 1.5<r<2 4<a<6 */
g7(r,a,z,func) float (*func)(),*r,*a; double *z;
{static float lam=2.4;
 float x,y;
 x=1.5+0.5*(*func)(1);
 y=1.5+0.5*(*func)(1);
 *r=(x>y?x:y);
 do{*a=4.0+randexp(lam,func);} while(*a>6.0);
 *z=(*func)(1)*f7(*r,*a);
}
```

```
double f7(r,a) float a,r;
{static double lam=2.4,c=25.0;
 return c*r*exp(-lam*a);
}

/* 1<r<2 6<a<8 */
g8(r,a,z,func) float (*func)(),*r,*a; double *z;
{static float lam=2.4;
 float x,y;
 x=1.0+(*func)(1);
 y=1.0+(*func)(1);
 *r=(x>y?x:y);
 do{*a=6.0+randexp(lam,func);} while(*a>8.0);
 *z=(*func)(1)*f8(*r,*a);
}

double f8(r,a) float a,r;
{static double lam=2.4,c=40.0;
 return c*r*exp(-lam*a);
}

/* 2<r<5 8<a<10 */
g9(r,a,z,func) float (*func)(),*r,*a; double *z;
{static float lam=0.5*(M_PI);
 *r=2.0+3.0*(*func)(1);
 do{*a=8.0+randexp(lam,func);} while(*a>10.0);
 *z=(*func)(1)*f9(*r,*a);
}

double f9(r,a) float a,r;
{static double lam=0.5*(M_PI),c=0.7;
 return c*exp(-lam*a);
}

/* remainder */
g10(r,a,z,func) float (*func)(),*r,*a; double *z;
{static float lam=(M_PI);
 do{*r=fabs(slownorm(0.0,10.0,func));
    *a=randexp(0.1,func);
   }while( (*a<6&&*r<8)       ||
           (*a<4&&*r<12)      ||
           (*r>1&&*a<8&&*r<8) ||
           (*r>2&&*r<5&&*a<10)   );
 *z=0.0;
}

double f10(r,a) float a,r;
{static double lam=(M_PI),c=1.0;
 return 0.0;
}
```

A.3 Mathematica: shuffle products and writing in terms of a Lyndon basis

The routines included here form part of a larger package, that performs algebraic manipulations of iterated Stratonovich and Ito integrals. This subset is however complete, in that it could be used without the rest of the package.

```
(* Calculates the shuffle product
   of Stratonovich multiple integrals
   and the modified shuffle product
   of Ito multiple integrals
 *)

Format[S[x_]]:=Subscripted[S[x]]
Format[ItoInt[x_]]:=Subscripted[ItoInt[x]]

Mult[x_+y_,z_]:=Mult[x,z]+Mult[y,z]
Mult[x_,y_+z_]:=Mult[x,y]+Mult[x,z]
Mult[n_Integer*x_,y_]:=Expand[n*Mult[x,y]]
Mult[x_,n_Integer*y_]:=Expand[n*Mult[x,y]]
Mult[n_Rational*x_,y_]:=Expand[n*Mult[x,y]]
Mult[x_,n_Rational*y_]:=Expand[n*Mult[x,y]]

pushinStrat[list_,x_]:=Module[
  {i,j,l,m,result={},y1,y2,y3,elt,head},
  l=Length[list];
  For[i=1,i<=l,i++,
   elt=Last[Part[list,i]];
   head=Drop[Part[list,i],-1];
   m=Length[elt]+1;
   For[j=1,j<=m,j++,
    y3=head;
    y1=Take[elt,j-1];
    y2=Take[elt,j-m];
    AppendTo[y1,x];
    AppendTo[y3,y1];
    AppendTo[y3,y2];
    AppendTo[result,y3]
   ]
  ];
  result
]

shuffleStrat[list1_,list2_]:=Module[
  {i,l,answer,ans={}},
  l=Length[list1];
  answer={{list2}};
  For[i=1,i<=l,i++,answer=pushinStrat[answer,Part[list1,i]]];
  l=Length[answer];
```

```
  For[i=1,i<=l,i++,AppendTo[ans,Flatten[Part[answer,i]]]];
  ans
]

Mult[S[l1_],S[l2_]]:=Module[
  {list,i,l,answer=0},
  list=shuffleStrat[l1,l2];
  l=Length[list];
  For[i=1,i<=l,i++,answer+=S[Part[list,i]]];
  answer
]

Mult[ItoInt[l1_],ItoInt[l2_]]:=Module[
  {list,i,l,answer=0},
  list=stretch[rshuffleIto[l1,l2]];
  l=Length[list];
  For[i=1,i<=l,i++,answer+=ItoInt[Part[list,i]]];
  answer
]

rshuffleIto[l1_,l2_]:=Module[
 {answer,q1,q2,a,b,i},
 If[l1=={},l2,
    If[l2=={},l1,
       q1=Drop[l1,1];
       q2=Drop[l2,1];
       a=First[l1];
       b=First[l2];
       answer={MyPrepend[rshuffleIto[q1,l2],a],
        MyPrepend[rshuffleIto[l1,q2],b]};
       If[a==b&&a!=0,
         AppendTo[answer,MyPrepend[rshuffleIto[q1,q2],0]]
        ];
       answer
      ]
   ]
]

MyPrepend[list_,x_]:=Module[
 {ans,l,i},
 If[list=={}||AtomQ[list[[1]]],
    ans=Prepend[list,x],
    l=Length[list];
    ans={};
    For[i=1,i<=l,i++,AppendTo[ans,MyPrepend[list[[i]],x]]]
   ];
 ans
]

stretch[list_]:=Module[
 {l,i,j,part,answer={},count=0},
 l=Length[list];
 For[i=1,i<=l,i++,
     part=list[[i]];
     If[AtomQ[part[[1]]],count++;AppendTo[answer,part],
        For[j=1,j<=Length[part],j++,
```

```
            AppendTo[answer,part[[j]]]
           ]
        ]
    ];
 If[count<1,stretch[answer],answer]
]

(* Routines for rewriting linear combinations of iterated
   integrals using an algebraic basis
 *)

Lyndon[list_]:=Module[
 {i,l,lyndon=True,word},
 word=StringJoin[Map[ToString,list,1]];
 l=StringLength[word];
 For[i=l-1,i>0,i--,
     If[OrderedQ[{StringDrop[word,i],word}]==True,lyndon=False]
    ];
 lyndon
]

LyndonSet[set_,n_]:=Block[
 {i,list,answer,word},
 answer={};
 list=pick[set,n];
 For[i=1,i<=Length[list],i++,
     word=Join[list[[i]]];
     If[Lyndon[word]==True,
        AppendTo[answer,word]
       ]
    ];
 answer
]

pick[set_,n_]:=Block[
 {i,j,k,l,answer,list},
 l=Length[set];
 answer=Table[{set[[i]]},{i,1,l}];
 list={};
 For[i=2,i<=n,i++,
     For[j=1,j<=Length[answer],j++,
         For[k=1,k<=l,k++,
             AppendTo[list,Append[answer[[j]],set[[k]]]]
            ]
        ];
     answer=list;
     list={}
    ];
 answer
]

LyndonBasis[set_,n_]:=Block[
 {i,answer},
 answer={};
 For[i=1,i<=n,i++,answer=Join[answer,LyndonSet[set,i]]];
 answer
```

```
]

LyndonFactor[list_]:=Module[
(* Breaks a word w into w=uv,
   where v is the longest right Lyndon factor of w.
   Returns {u,v}.
*)
 {word,srf},
 word=StringJoin[Map[ToString,list,1]];
 srf=First[Sort[Table[StringTake[word,-i],
           {i,1,StringLength[word]-1}]]];
 srf=Map[ToExpression,Characters[srf],1];
 {Drop[list,-Length[srf]],srf}
]

LyndonProduct[list_]:=Module[
 {word1,answer},
 word1=list;
 answer={list};
 While[Lyndon[word1]==False,
       answer=Join[LyndonFactor[word1],Drop[answer,1]];
       word1=First[answer]
      ];
 answer
]

Rewrite[S[J_]]:=Module[
 {list,i,n=1,c=1,p,pre,word,w,r},
 If[Lyndon[J]==True,Return[S[J]]];
 list=LyndonProduct[J];
 pre=list[[1]];
 p=S[pre];
 For[i=2,i<=Length[list],i++,
     w=list[[i]];
     If[w==pre,n++,c=c/n!;n=1;pre=w];
     p=Expand[Mult[S[w],p]]
    ];
 c=c/n!;
 p*=c;
 r=p-S[J];
 Expand[c*Product[S[list[[i]]],{i,1,Length[list]}]-r]
]

Rewrite[ItoInt[J_]]:=Module[
 {list,i,n=1,c=1,p,pre,word,w,r},
 If[Lyndon[J]==True,Return[ItoInt[J]]];
 list=LyndonProduct[J];
 pre=list[[1]];
 p=ItoInt[pre];
 For[i=2,i<=Length[list],i++,
     w=list[[i]];
     If[w==pre,n++,c=c/n!;n=1;pre=w];
     p=Expand[Mult[ItoInt[w],p]]
    ];
 c=c/n!;
 p*=c;
```

```
 r=p-ItoInt[J];
 Expand[c*Product[ItoInt[list[[i]]],{i,1,Length[list]}]-r]
]

rw1[x_]:=Rewrite[S[x]]
rw2[x_]:=Rewrite[ItoInt[x]]

WriteUsingStratBasis[x_]:=Expand[x//.S->rw1]
WriteUsingItoBasis[x_]:=Expand[x//.ItoInt->rw2]
```

A.4 Calculation of the partition with minimum entropy of a staircase

This is a recursive routine that needs to be called by another piece of code, that first sets up the necessary data structures. We have not included the calling program here. We have written a similar routine for triangular partitions.

```
float optimal_p(root,n1,l) point root; int n1,l;
{int n2,i,j,jmin,l1,l2;
 float a,e,min;
 point root1,root2;

 n2=n1+l-1;

 /* Look up answer in table E, if already calculated */
 if(E[l][n1].i>=0) return E[l][n1].e;
 else /* calculate minimum entropy */
 {
  if(l==1) /* only one rectangle : no minimisation necessary */
  {a=(S[n1].x-root.x)*(S[n1].y-root.y);
   e=-a*log(a);
   E[l][n1].i=n1; E[l][n1].e=e;
   return (e);
  }
  else /* at least 2 rectangles */
  {
   /* loop over number of steps */
   min=1000.0;
   for(i=0;i<l;i++)
   {j=n1+i;
    /* remove rectangle (S[j],root) leaving 1 or 2 sub-regions */
    /* area of rectangle removed */
    a=(S[j].x-root.x)*(S[j].y-root.y);
    e=-a*log(a);
    root1.x=root.x;  root2.y=root.y;
    root2.x=S[j].x; root1.y=S[j].y;
    l1=j-n1; l2=n2-j;
```

```
    if(l1>0) e+=optimal_p(root1,n1,l1);
    if(l2>0) e+=optimal_p(root2,j+1,l2);
    if(e<min) {min=e; jmin=j;}
   }
   /* store new info */
   E[l][n1].i=jmin; E[l][n1].e=min;
   return (min);
  }
 }
}
```

A.5 Generation of univariate random deviates from a general density

We give below a general routine for generating random deviates from any density, where the second derivative is zero at precisely one point, or from any symmetric density with two such points. It is clear how to modify the routine to handle fewer or more inflexion points. We also include the calling program, in order to show precisely what information about each density is needed and how this is passed to the subroutine which performs the generation.

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <malloc.h>

/* Version with optimal DDG-tree for choosing triangles. */

/* User supplied main program and subroutines defining density */

main(argc, argv)
  int       argc;
  char    **argv;
{int i,N,seed=-3;
 double x,x2,mean=0.0,var=0.0,m3=0.0,m4=0.0,xinfc,xinfn;
 float y,ran3();
 double trigen();
 double fn(),dfn(),fc(),dfc();
 FILE *out;

 if(argc>1) out=fopenw(argv[1]);

  getint(&N,"N");
  ran3(seed);
```

```
  /* inflection point for 0,1 cauchy distribution */
  xinfc=0.5773502691896259;
  /* inflection point for normal distribution */
  xinfn=1.0;

  for(i=1;i<=N;i++)
  {
   x=trigen(fc,dfc,xinfc,i);
   fprintf(out,"%17.15f\n",x);
  }

  trigen(fc,dfc,xinfc,-1);
}

#define C 0.3989422804014327        /* 1/sqrt(2*Pi) */

double fc(x) double x;
/* Cauchy density */
{ return ((M_1_PI)/(x*x+1.0)); }

double dfc(x) double x;
/* Derivative of Cauchy density */
{ double d;
  d=x*x+1.0;
  return ((M_1_PI)*(-2.0*x)/(d*d));
}

double fn(x) double x;
/* Normal density */
{ return C*exp(-0.5*x*x); }

double dfn(x) double x;
/* Derivative of normal density */
{ return -C*x*exp(-0.5*x*x); }

/* General purpose part */

#define FAC32 4.294967296e9        /* 2^32 */
#define MAXB 32                    /* Max. no. of bits */

struct pointnode{double x,y; int next;} ;
typedef struct pointnode pnode;
typedef struct {double x1,x2,x3;} triangle;

struct tnode{int n; struct tnode *left,*right;} ;
typedef struct tnode node;
int nodesize;
node *root;
int marker[MAXB],zmarker[MAXB];

double ran_tri(),tri_area(),g();
int    ran_bit(),p();
unsigned long *s;
pnode  *v;
pnode q1,q2;
```

```
triangle *t;
int    nmax,pnodesize,trisize,next;
double pn;

double trigen(f,df,xinf,init) double (*f)(),(*df)(),xinf; int init;
{
/* Routine to generate random deviates with density f */
/* df is the derivative of the density */
/* xinf is the (only) point where the second derivative is 0 */

 static int first=1;
 float     ran3();
 double    x,y,sum,ent;
 static double xc;
 int       n,i,lb,undecided,bit;
 node      *q;


  if(init<0 || first==1)
  {printf("Initialising trigen\n");
   if(nmax>0)
   {printf("Freeing allocated memory\n");
    freetree(root);
    free((char*)t);
    free((char *)s);
    free((char*)v);
    return (0.0);
   }

   /* initialisation of constants */
   first=0;
   pnodesize=sizeof(pnode);
   trisize=sizeof(triangle);
   nodesize=sizeof(node);
   for(i=1;i<MAXB;i++)
   {marker[i]=1; zmarker[i]=0;}
   /* base length of right angled triangle under concave part */
   xc=-(*f)(xinf)/(*df)(xinf);

   /* Initialisation of convex part */
   root=(node *)malloc(nodesize);
   root->n=-1;
   root->left=root->right=NULL;


   /* Initialisation of arrays */
   s=(unsigned long *)malloc((unsigned)4*sizeof(unsigned long));
   t=(triangle *)malloc(4*trisize);
   x=2.0*xinf*(*f)(xinf);
   pn=x;
   /* p of being in rectangle {{0,0}{xinf,f(xinf)}} */
   s[1]=FAC32*x;
   x=xinf*((*f)(0.0)-(*f)(xinf));
   pn+=x;
   s[2]=FAC32*x; /* p of being in triangle
                    {{0,f(xinf)},{xinf,f(xinf)},{0,f(0)}} */
```

```
   t[2].x1=0.0; t[2].x2=0.0; t[2].x3=xinf;
   x=xc*(*f)(xinf);
   pn+=x; printf("p1+p2+p3=%e\n",pn);
   s[3]=FAC32*x; /* p of being in triangle
                    {{xinf,0)},{xinf,f(xinf)},{xinf+xc,0}} */
   t[3].x1=xinf; t[3].x2=xinf; t[3].x3=xinf+xc;
   s[0]=FAC32*(1.0-pn);
   v=(pnode*)malloc(2*pnodesize);
   v[0].x=0.0; v[0].y=(*f)(0.0);
   v[0].next=1;
   v[1].x=xinf; v[1].y=(*f)(xinf);
   v[1].next=(-1);
   q1=v[0];
   nmax=3;
  }

  undecided=1;
  lb=0;
  q=root;
  while(undecided)
  {lb++;
   bit=ran_bit(1);
   if(q->left==NULL) extend_tree(q,lb);
   if(bit==0) q=q->left;
   else       q=q->right;
   while(q->n==0) refine_tree(f,df,xinf);
   if(q->n>0) {n=q->n; undecided=0;}
  }
 if(n==1)  x=xinf*ran3(1); /* random point in rectangle */
 else x=ran_tri(t[n].x1,t[n].x2,t[n].x3);
 if (ran_bit(1)==1) x=-x;
 return x;
}

static int jj=0;

int p(n,b) int n,b;
{ /* returns bit b of p(n) */
 if(n>nmax){printf("Error in p: n>nmax\n"); exit(1);}
 if(b>MAXB) {printf("b=%d>MAXB\n",b); exit(1);}
 return (s[n]>>(32-b))&1;
}

make_tri1(f) double (*f)();
/* Adds a triangle in convex region */
{double x,y,ta;
 q1=v[jj];
 q2=v[q1.next];
 x=0.5*(q1.x+q2.x);
 y=(*f)(x);
 ta=tri_area(q1.x,q1.y,x,y,q2.x,q2.y);
 nmax++;
 if((v=(pnode*)realloc(v,(unsigned)(nmax-1)*pnodesize))==NULL)
  {printf(" Out of memory 1.\n"); exit(1);}
 v[nmax-2].x=x; v[nmax-2].y=y; v[nmax-2].next=q1.next;
 v[jj].next=nmax-2;
```

```
 if((s=(unsigned long *)realloc(s,
     (unsigned)(nmax+1)*sizeof(unsigned long)))==NULL)
  {printf(" Out of memory 3.\n"); exit(1);}
 pn+=ta;
 s[0]=FAC32*(1.0-pn);
 s[nmax]=FAC32*ta;
 if((t=(triangle *)realloc(t,(unsigned)(nmax+1)*trisize))==NULL)
  {printf(" Out of memory 4.\n"); exit(1);}
  t[nmax].x1=q1.x;
  t[nmax].x2=x;
  t[nmax].x3=q2.x;
 jj=q1.next;
 }

make_tri2(f,df)  double (*f)(),(*df)();
/* Adds a triangle in concave region */
{extern pnode  *v;
 double x,y,xl,yl,xu,yu,x1,y1,x2,y2,x3,y3;
 double dfx,dfxl,dfxu,ta;
 q1=v[jj];
 xl=q1.x;  yl=(*f)(xl); dfxl=(*df)(xl);
 if(q1.next<0) /* will add tail triangle */
  x=xl+0.5*xl;
 else
 {q2=v[q1.next];
  xu=q2.x; yu=(*f)(xu); dfxu=(*df)(xu);
  x=0.5*(xl+xu);
 }
 y=(*f)(x);
 dfx=(*df)(x);
 x1=(dfxl*xl-dfx*x+y-yl)/(dfxl-dfx);
 y1=dfx*(x1-x)+y;
 if(q1.next>0)
 {x2=(dfxu*xu-dfx*x+y-yu)/(dfxu-dfx);
  y2=dfx*(x2-x)+y;
  x3=(dfxu*xu-dfxl*xl+yl-yu)/(dfxu-dfxl);
  y3=dfxl*(x3-xl)+yl;
 }
 else /* q1.next==(-1): new tail triangle */
 {x2=x-y/dfx;
  y2=0.0;
  x3=xl-yl/dfxl;
  y3=0.0;
 }
 ta=tri_area(x1,y1,x2,y2,x3,y3);
 nmax++;
 if((v=(pnode*)realloc(v,(unsigned)(nmax-1)*pnodesize))==NULL)
  {printf(" Out of memory 1.\n"); exit(1);}
 v[nmax-2].x=x; v[nmax-2].y=y; v[nmax-2].next=q1.next;
 v[jj].next=nmax-2;
 if((s=(unsigned long *)realloc(s,
     (unsigned)(nmax+1)*sizeof(unsigned long)))==NULL)
  {printf(" Out of memory 3.\n"); exit(1);}
 pn+=ta;
 s[0]=FAC32*(1.0-pn);
 s[nmax]=FAC32*ta;
```

```
 if((t=(triangle *)realloc(t,(unsigned)(nmax+1)*trisize))==NULL)
  {printf(" Out of memory 4.\n"); exit(1);}
  t[nmax].x1=x1;
  t[nmax].x2=x3;
  t[nmax].x3=x2;
 if(q1.next==(-1)) jj=0;
 else              jj=q1.next;
}

double  tri_area(x1,y1,x2,y2,x3,y3) double x1,x2,x3,y1,y2,y3;
{/* returns twice area of triangle : for even distributions */
 return fabs(x2*y1-x1*y2+x3*y2-x2*y3-x3*y1+x1*y3);
}

double ran_tri(x1,x2,x3) double x1,x2,x3;
{double u,v,temp;
 float ran3();
 u=ran3(1);
 v=ran3(1);
 if(u>v) {temp=v; v=u; u=temp;}
 return u*x1+(v-u)*x2+(1.0-v)*x3;
}

int ran_bit(i) int i;
{ static int count=7,init=0;
  static unsigned long v;
  float      ran3();
  int bit;

  if(count==7)
  {v=ran3(1)*FAC32;
   count=31;
  }
  bit=(v>>count)&1;
  count--;
  return bit;
}

double g(x) double x;
{return (-x*log(x));}

extend_tree(q,l) node *q; int l;
/* gives interior node q on level l two children */
{node *q1,*q2;
 int i;
 q->left=q1=(node *)malloc(nodesize);
 q->right=q2=(node *)malloc(nodesize);
 q1->left=q1->right=q2->left=q2->right=NULL;
 i=marker[l];
 while(i<=nmax && p(i,l)==0) i++;
 if(i<=nmax) {q1->n=i; marker[l]=i+1;}
 else
 {if(zmarker[l]==0 && p(0,l)==1)
   {q1->n=0; zmarker[l]=1;}
  else       q1->n=-1;
 }
```

```
 i=marker[l];
 while(i<=nmax && p(i,l)==0) i++;
 if(i<=nmax) {q2->n=i; marker[l]=i+1;}
 else
 {if(zmarker[l]==0 && p(0,l)==1)
   {q2->n=0; zmarker[l]=1;}
  else       q2->n=-1;
 }
}

refine_tree(f,df,xinf) double (*f)(),(*df)(),xinf;
{
 if(v[jj].x<xinf) make_tri1(f);
 else             make_tri2(f,df);
 refine(root,0);
}

refine(q,l) node *q; int l;
/* Updates all 0 nodes on level l and higher
   to take into account the inclusion of p(nmax).
*/
{
 int i;
 if(q->n==0) /* refine */
 {zmarker[l]=0;
  i=marker[l];
  while(i<=nmax && p(i,l)==0) i++;
  if(i<=nmax) {q->n=i; marker[l]=i+1;}
  else
  {if(p(0,l)==1) zmarker[l]=1;
   else q->n=-1; /* change to interior node */
  }
 }
 else
 {if(q->left!=NULL)  refine(q->left,l+1);
  if(q->right!=NULL) refine(q->right,l+1);
 }
}

freetree(p)
node *p;
{node *q;
 if(p!=NULL)
   {freetree(p->left);
    q=p->right;
    free((char *)p);
    freetree(q);
   }
}
```