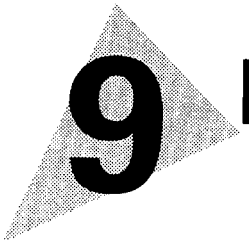


Mathematica and Diffusions

J. Michael Steele and Robert A. Stine

9.1 Introduction

A central aim of this chapter is to illustrate how symbolic computing can simplify or eliminate many of the tedious aspects of the stochastic calculus. The package `Diffusion.m` included with this book provides a suite of functions for manipulating diffusion models, and individuals with a basic knowledge of *Mathematica* should be able to use this package to expedite many of the routine calculations of stochastic calculus. After demonstrating the basic features of this package, we give an extensive example that applies the functions of the package to a problem of option-pricing.

This application offers a derivation of the well-known Black-Scholes formula for options pricing. The derivation exploits the idea of a self-financing portfolio (Duffie 1988). Our goal is show how *Mathematica* simplifies the *derivation* of the central expression in this problem. This task is to be distinguished from that engaged in Miller (1990) which shows a variety ways to describe and use the Black-Scholes formula.

In the next section we give a brief overview of the central idea of diffusions as required by options pricing theory, and we introduce the notation that is needed. This section is indeed very brief, and readers unfamiliar with diffusions should consider a text like that of Arnold (1974) or Duffy (1988) for the missing details and intuition. Section 9.3 introduces the new *Mathematica* functions that will be needed in Section 9.4 where we present a *Mathematica* derivation of the Black-Scholes formula. Section 9.5 demands more knowledge of *Mathematica* since it describes issues of implementation and includes further examples of the methods used in the extended example.

This package uses features found in Version 2 of *Mathematica*. Since some of these features are not available in earlier versions of the software, problems will occur if one attempts to use the package without having upgraded to at least Version 2.0.

9.2 Review of Diffusions and Itô's Formula

Diffusions form a class of stochastic processes that allow one to apply many of the modeling ideas of differential equations to the study of random phenomena. In simplest terms, a diffusion X_t is a Markov process in continuous time $t \geq 0$ that has continuous sample paths. A key feature of diffusions is that there exist two functions μ and σ which together with an initial value X_0 completely characterize each diffusion. Moreover, in many applications μ and σ have physical or financial interpretations.

For this article we will assume that the scalar-valued functions μ and σ are locally bounded, and let W_t denote the standard Weiner process. Under these conditions, a fundamental result of the Itô calculus is that there exists a well-defined process X_t having the suggestive representation

$$X_t = X_0 + \int_0^t \mu(X_s, s)ds + \int_0^t \sigma(X_s, s)dW_s. \tag{1}$$

The intuition behind this representation for X_t is that the function μ determines the instantaneous drift of X_t whereas σ controls its disperson or variability. If, as in later examples, X_t denotes the price of a stock at time t, then μ and σ may be interpreted as the rate of return and the instantaneous risk.

Since the process W_t is not of bounded variation, the second integral on the right-hand side of (1) cannot be interpreted naively. For example, suppose that we were to attempt to evaluate $\int_0^t W_s dW_s$ as the limit of an approximating sum over partitions of the interval $[0, t]$,

$$\sum_i W_{s_i}(W_{t_i} - W_{t_{i-1}}), 0 \leq t_{i-1} \leq s_i \leq t_i \leq t.$$

Since W_t is not of bounded variation, the value for the integral suggested by such approximation turns out to depend upon how we choose to locate s_i in the interval $[t_{i-1}, t_i]$. Throughout this chapter, we adopt the convention $s_i = t_{i-1}$. The resulting stochastic integral has a unique solution known as the Itô integral. A very important reason for choosing $s_i = t_{i-1}$ is that with this choice a stochastic integral with respect to W_t is always a martingale.

One of the most common and convenient ways to express a diffusion is to use a notation that is analogous to that of differential equations. Rather than use the stochastic integral to represent X_t as in (1), one often uses a shorthand notation:

$$dX_t = \mu(X_t, t)dt + \sigma(X_t, t)dW_t. \tag{2}$$

This compact description of the process X_t resembles a differential equation, but it must be interpreted appropriately as just shorthand for (1).

An important property of diffusions is their behavior under smooth transformations. Perhaps the central result in the theory of diffusions is that a smooth function of a diffusion produces another diffusion. Moreover, an explicit formula due to Itô identifies the new process by giving expressions for the drift and disperson functions. Suppose that the function f maps $(\Re, \Re^+) \rightarrow \Re$,

and consider applying this function to a diffusion, say $Y_t = f(X_t, t)$. Under modest conditions on f, the process Y_t is also a diffusion. One set of sufficient conditions is to require continuous first and second derivatives of f in its first argument and a single continuous derivative in the second. Denote these derivatives by $f_x = \partial f(x,t)/\partial x$, $f_{xx} = \partial^2 f(x,t)/\partial x^2$, and $f_t = \partial f(x,t)/\partial t$.

Under these restrictions on f, Itô's formula guarantees that Y_t is a diffusion and gives the functions that characterize this new diffusion. Specifically, the stochastic differential for $Y_t = f(X_t, t)$ is

$$dY_t = \{f_t(X_t,t) + \mu(X_t,t) f_x(X_t,t) + f_{xx}\sigma^2(X_t,t)/2\}dt + \{f_x(X_t,t)\sigma(X_t,t)\}dW_t. \tag{3}$$

This expression is not in the canonical form of (2) since the drift and dispersion functions of Y_t are functions of X_t rather than Y_t, but this problem is easily remedied. If the transformation f has an inverse g so that

$$y = f(x,t) \Leftrightarrow x = g(y,t),$$

then we can use this inverse to obtain the desired form. Substitution in (3) gives

$$dY_t = \{f_t(g(Y_t,t),t) + \mu(g(Y_t,t),t) f_x(g(Y_t,t),t) + f_{xx}\sigma^2(g(Y_t,t),t)/2\}dt + \{f_x(g(Y_t,t),t)\sigma(g(Y_t,t),t)\}dW_t$$

So indeed,

$$dY_t = \bar{\mu}(Y_t,t)dt + \bar{\sigma}(Y_t,t)dW_t \tag{4}$$

for the indicated values of $\bar{\mu}$ and $\bar{\sigma}$. The complexity of (4) suggests that we can avoid some tedium if we put *Mathematica* to work managing the transformation of the pair $\{\mu(x,t), \sigma(x,t)\}$ into $\{\bar{\mu}(x,t), \bar{\sigma}(x,t)\}$

Itô's formula (3) can be made more revealing if one also introduces the notion of the infinitesimal generator of the diffusion X_t. The infinitesimal generator of a diffusion measures the instantaneous rate of change in the expected value of a transformation of a diffusion as a function of its starting value. Under some natural conditions on the function f, we can define the infinitesimal generator as

$$A_X f(x) = lim_{t \to 0} \frac{E_x f(X_t,t) - f(x,0)}{t},$$

where E_x denotes the expectation conditional on the starting value $X_0 = x$. This limit turns out to be closely related to the drift expression in Itô's formula, and one can show under mild conditions that A_X is the differential operator that has the explicit form

$$A_X = \frac{\partial}{\partial t} + \mu(x,t)\frac{\partial}{\partial x} + \frac{1}{2}\sigma(x,t)\frac{\partial^2}{\partial x^2}. \tag{5}$$

By using the infinitesimal generator associated with the diffusion X_t, we obtain a more compact form of Itô's formula,

$$dY_t = df(X_t,t) = A_X f dt + f_x(X_t,t)\sigma(X_t,t)dW_t.$$

9.3 Basic Mathematica Operations

9.3.1 Introduction to the Package

This section introduces basic functions that permit us to build and manipulate diffusions. In order to follow the operations described here, one must first import the package that defines the needed functions. The package itself is located in the file named "`Diffusion.m`" which should be placed in a directory which is easily, if not automatically, searched. The following command imports the package.

```
In[1]:= << Diffusion`
```

The names of most of the functions in this package begin "Diffusion", though some *also* have shorter, more convenient names. This convention allows for an easy listing of the available functions via the built-in **`Names`** function.

```
In[2]:= Names["Diffusion*"]
Out[2]:= {DiffusionA, DiffusionChangeSymbol, DiffusionDispersion,
          DiffusionDrift, DiffusionExpand, DiffusionExpandRules,
          DiffusionFeynmanKac, DiffusionInitialValue,
          DiffusionIto, DiffusionLabel, DiffusionMake,
          DiffusionPrint, DiffusionSimulate, DiffusionSymbol,
          DiffusionWeinerProcess, DiffusionWeinerProcessMake,
          DiffusionWeinerProcessSymbol}
```

A brief synopsis of each function is available via the standard *Mathematica* request for help; one precedes the name of the function of interest with a question mark. For example, the following command reveals the syntax and a brief description of the function which prints diffusions.

```
In[3]:= ?DiffusionPrint
DiffusionPrint[d] prints a formatted version of the
   diffusion object d using subscripts and symbolic names.
```

The next section describes how to build the diffusion objects that this function requires as arguments.

9.3.2 Building a Diffusion

The first order of business in building a diffusion is to specify a Weiner process. In this and many other examples, we begin by calling the **`Clear`** function. This function removes any value that might already be bound to the named symbol, as might occur when experimenting with *Mathematica* outside of the range of commands shown here. Thus, we first clear the name `W` which we intend to use to represent a Wiener process.

```
In[1]:= Clear[W]
```

The next command associates the name W with a Weiner process. Until we say otherwise, the name W will denote a Weiner process to the system.

```
In[2]:= WeinerProcessMake[W]
```

We can now use this Weiner process to define more complex diffusions. To build each diffusion, our convention requires a symbolic identifier (or name) for the diffusion and for the underlying Weiner process, two expressions (the drift μ and the dispersion σ), and an initial value. As an example, we first consider the lognormal diffusion. This process is used extensively in financial modeling and it will play a central role in Section 9.4 when we derive the Black-Scholes formula. In the classical notation of the stochastic calculus, a lognormal diffusion can be specified by

$$dX_t = \alpha X_t dt + \beta X_t dW_t, X_0 = v,$$

for scalars α and $\beta > 0$. To represent this diffusion in *Mathematica*, use the command **DiffusionMake**. Again, we clear the name X before we make the new process.

```
In[3]:= Clear[X]
In[4]:= DiffusionMake[X, W, alpha X, beta X, v]
```

To test the success of this command, we recall that typing the name of any object in *Mathematica* reveals the value of that object. In the case of a diffusion, the value is a list that holds the definition of the process.

```
In[5]:= X
Out[5]:= diffusion[X, W, alpha X, beta X, v]
```

While revealing to the *Mathematica* programmer, this form is probably not very appealing to one more familiar with the standard mathematical notation. The function **DiffusionPrint** ameliorates this problem by producing a more familiar rendering of a diffusion, complete with subscripting. Here are examples for the two diffusions created so far. The printed form for a Weiner process is pretty simple.

```
In[6]:= DiffusionPrint[W]
W  = Weiner Process with scale 1
 t
```

The format in general for printing a diffusion matches the notation of equation (2).

```
In[7]:= DiffusionPrint[X]
dX  = alpha X  dt + beta X  dW    ; X  = v
  t          t            t   t      0
```

9.3.3 Simulating Diffusions

In order to experiment with a diffusion model, it is often useful to simulate realizations of the process. The function **DiffusionSimulate** generates sim-

ulated realizations by building a discrete-time approximation to the continuous-time process. The method implemented in this software is a little naive—it resembles Euler's method for approximating solutions of a differential equation—but it functions quite usefully in most instances. To illustrate, we begin by asking *Mathematica* to reveal its summary of **DiffusionSimulate**.

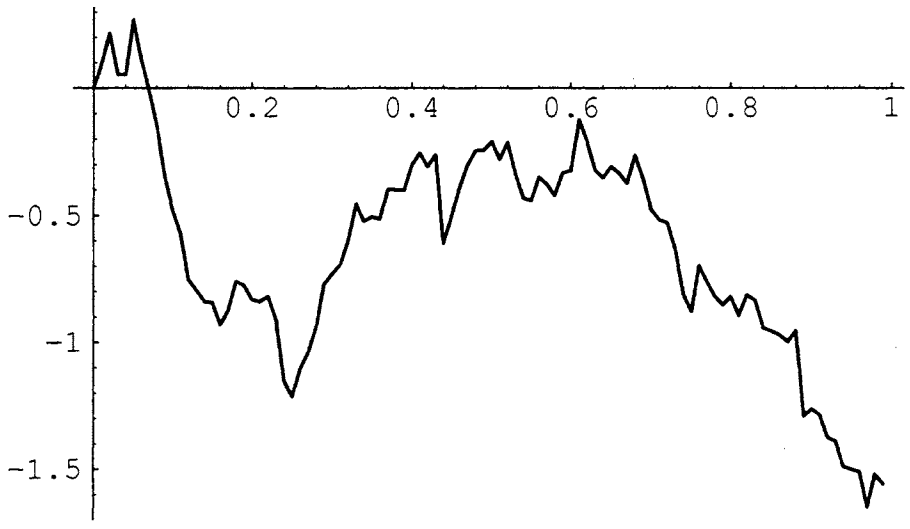
```
In[1]:= ?DiffusionSimulate
DiffusionSimulate[d_diffusion, n, dt] simulates a
   diffusion.  It returns a list of n pairs
   {,,,{i*dt,x[i*dt]},,,} of the  diffusion d. Each realized
   series is an independent realization.  Responds with an
   error if the diffusion contains symbolic paramters.
```

As a simple example of the use of this function, we will simulate 100 values of a Weiner process at times $\{0, 0.01, \ldots, 0.99\}$. Assuming that we are simulating a diffusion labelled X_t, the simulated realization begins with the pair $\{0, X_0\}$ and adds $n-1$ pairs of the form $\{i\,dt, X_{idt}\}, i = 1, 2, \ldots, n-1$, separated in time by the chosen step size dt which is the last argument to the function. In this example, the semicolon at the end of the command suppresses the printing of the complete simulated realization. The function **Short** reveals the start and end of the list, and the bracketed number indicates that 97 of the items in the list are hidden.

```
In[2]:= simW = DiffusionSimulate[W,100,0.01];

In[3]:= Short[simW]

Out[3]:= {{0, 0}, {0.01, 0.0984664}, <<97>>, {0.99, -1.55817}}
```

The list structure of the simulated diffusion makes it quite easy to have *Mathematica* plot this artificial realization versus time.

```
In[4]:= ListPlot[simW, PlotJoined -> True]
```




```
Out[4]:= -Graphics-
```

Simulations of other diffusions can be obtained just as easily. For example, suppose that we wished to plot the simulation of the diffusion

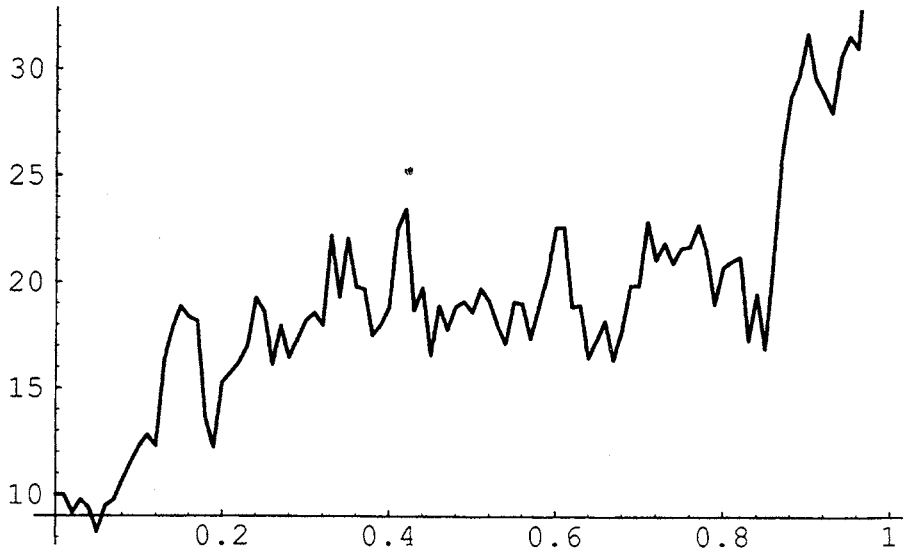
$$dY_t = Y_t dt + Y_t dW_t, Y_0 = 10.$$

First we build the diffusion using **DiffusionMake**.

```
In[5]:= DiffusionMake[Y,W, Y,Y,10]
```

Next we generate the partial realization using **DiffusionSimulate**, and again we note that **Short** reveals just the extremes of the list. Since the time spacing is not specified in the call to **DiffusionSimulate**, this program sets the spacing to a default value of 0.01.

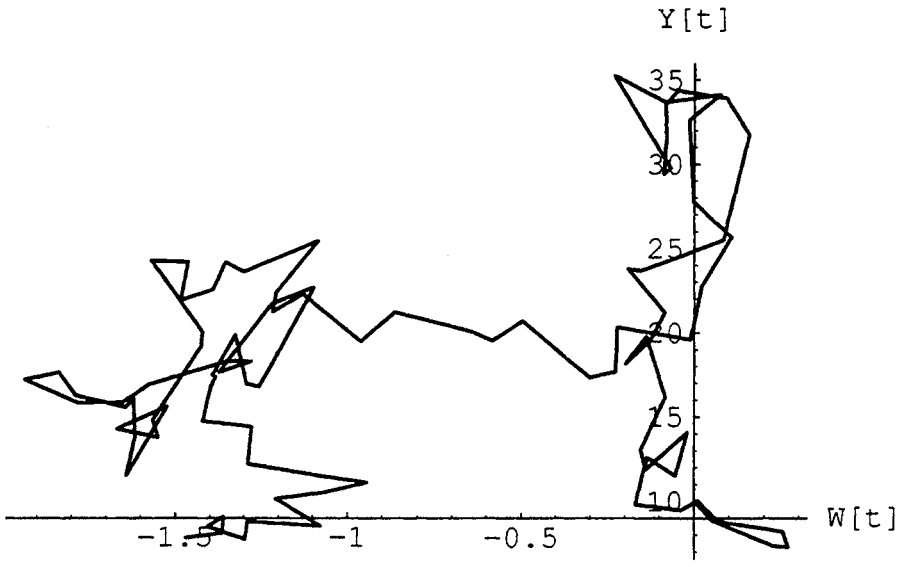
```
In[6]:= simY = DiffusionSimulate[Y,100];
        Short[simY]

Out[6]:= {{0, 10.}, {0.01, 9.99454}, <<97>>, {0.99, 37.4779}}

In[7]:= ListPlot[simY,
            PlotJoined -> True]
```




```
Out[7]:= -Graphics-
```

The function **RandomSeed** makes it possible to generate correlated realizations. By default, each realization of a diffusion is independent of other realizations. This structure may not be desired for many problems. As an example, consider the relationship of the Weiner process realization and the diffusion for the process $dY_t = Y_t dt + Y_t dW_t$ considered above. Independent realizations fail to capture the relationship between the series and suggest little relationship between the two. After all, these simulated realizations are independent. Here is a plot of Y_t on W_t with the plots joined in time sequence order.

```
In[8]:= ListPlot[Table[{simW[[i,2]], simY[[i,2]]},{i,1,100}],
                 AxesLabel -> {"W[t]", "Y[t]"},
                 PlotJoined -> True]
```



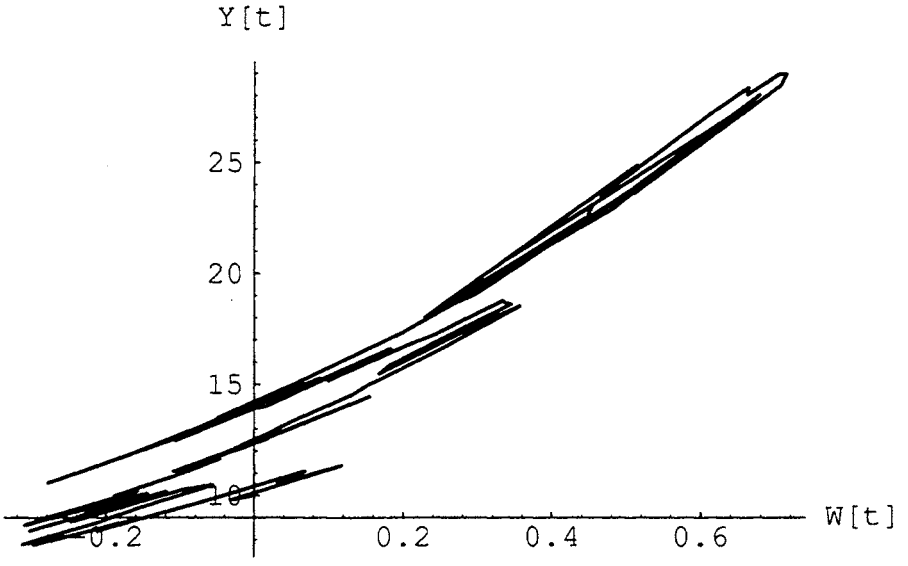
Out[8]:= -Graphics-

By explicitly setting the random seed used by *Mathematica,* we force the realization of Y_t to be based on the same random values used to simulate the Weiner process W_t. This connection gives the realizations we would have expected.

```
In[9]:= SeedRandom[732712];
        simW = DiffusionSimulate[W,100];
        SeedRandom[732712];
        simY = DiffusionSimulate[Y,100];
```

The plot of Y_t on W_t now exhibits the strong relationship between the two simulated realizations that is missing without matching the simulation seeds.

```
In[10]:= ListPlot[Table[{simW[[i,2]], simY[[i,2]]},{i,1,100}],
                  AxesLabel->{"W[t]", "Y[t]"},
                  PlotJoined -> True]
```



Out[10]:= -Graphics-

9.4 Deriving the Black-Scholes Formula

To illustrate the use of these tools in a problem of considerable historical interest, we will use our tools to derive the Black-Scholes option-pricing formula. Our approach parallels the development in Duffie (1988) and uses the notion of a self-financing trading strategy. Following the tradition in elementary option pricing theory, we will ignore the effects of transaction costs and dividends. Readers who are curious for further details and intuition should consider Duffie (1988) for a more refined discussion than space permits here.

The model begins with a simple market that consists of two investments, a stock and a bond. The bond is taken to be a risk free asset with rate of return ρ, so in our differential notation the bond price B_t at time t obeys $dB_t = \rho B_t dt$. The stock is assumed to have rate of return μ as well as some risk, so we can model the stock price S_t as a diffusion for which $dS_t = \mu S_t dt + \sigma S_t dW_t$ with the scalar $\sigma > 0$. We can set up these processes and clear the needed symbols as follows:

```
In[1]:= Clear[W, S,S0, B,B0, V, mu,sigma,rho, g,P]

        WeinerProcessMake[W]
        DiffusionMake[S, W, mu S, sigma S, S0]
        DiffusionMake[B, W, rho B, 0, B0]
```

Before continuing, we should perhaps view our processes in more conventional form by getting the printed version of each of the three diffusions.

```
In[2]:= DiffusionPrint[W]
        DiffusionPrint[S]
        DiffusionPrint[B]

W  = Weiner Process with scale 1
 t
dS  = mu S  dt + sigma S  dW   ; S  = S0
  t       t              t   t      0
dB  = rho B  dt  ; B  = B0
  t        t        0
```

With these processes as building blocks, we can define the equations that permit us to give the explicit value of the European option. Suppose that there exists a function $V(s,t)$ such that $V(S_t,t)$ is the value at time $t, 0 \le t \le T$, of an option to purchase the stock modeled by S_t at a terminal time T at the strike price P. Our goal is to find an expression for this function in terms of μ, σ, P, T, and t. Since the value of the option is a function of the diffusion S_t, Itô's formula gives an expression for $V(S_t,t)$. Even though we do not explicitly know $V(S_t,t)$, we can use the **Ito** function to identify this new diffusion symbolically in terms of derivatives of V. In this case, we want to avoid putting the diffusion into the canonical form (4) since doing so would conceal how $V(S_t,t)$ depends upon S_t. The use of the **ItoInvert** option avoids the inversion to the canonical diffusion form, thereby retaining the

stock symbol S in the expressions for the drift and dispersion of the new diffusion.

```
In[3]:= Ito[V[S,t], ItoInvert->False];

                (0,1)                    (1,0)
dV[S, t] = (V      [S, t] + mu S V      [S, t] +

        2  2  (2,0)
   sigma  S  V      [S, t]                    (1,0)
   ------------------------) dt + (sigma S V      [S, t]) dW
             2                                              t
```

Now equate the value of the option to that of a self-financing trading portfolio. That is, assume that the value $V(S_t, t)$ of this stock option can be reproduced by a portfolio consisting of a_t shares of stock and b_t shares of bond, $V(S_t, t) = a_t S_t + b_t B_t$. Here, of course, we are assuming that a_t and b_t are both stochastic processes. An argument for the existence of such a portfolio in this problem appears in Duffy (1988). The matching of the value of the option to that of a portfolio gives a second expression for V in addition to that from Itô's formula. If we equate these two expressions for $V(S_t, t)$, we can solve for a_t and b_t. The manipulations that support the elimination of a_t and b_t are very easy in *Mathematica* since the diffusion package defines some simple algebraic operations for diffusions.

By default, diffusions retain their symbolic form in algebraic expressions. For example, if we enter a sum of two diffusions, the sum is retained and no attempt is made to combine the drift of one with that of the other.

```
In[4]:= at S + bt B

Out[4]= bt B + at S
```

This behavior is consistent with the way *Mathematica* handles many other symbolic expressions, such as the way a product is not expanded unless the user makes an explicit request:

```
In[5]:= (a + b) (c + d)

Out[5]= (a + b) (c + d)

In[6]:= Expand[%]

Out[6]= a c + b c + a d + b d
```

The analogous behaviour is needed in the algebra of diffusions. In order to equate the two expressions for $V(S_t, t)$, we require the drift and dispersion of a diffusion that is the sum of two diffusions. The function **DiffusionExpand** combines several diffusions, though the resulting "diffusion" is not of the canonical form. Since we are only interested in the drift and dispersion, the absence of the canonical form is not a problem.

```
In[7]:= diff = DiffusionExpand[V - (at S + bt B)]
```

```
Out[7]:=                                                          (0,1)
         diffusion[$3, W, -(bt rho B) - at mu S + (V)      [S, t]

                                                    2  2     (2,0)
                              (1,0)           sigma  S   (V)      [S, t]
           + mu S (V)      [S, t] + ------------------------------------,
                                                    2

                                                 (1,0)
           -(at sigma S) + sigma S (V)      [S, t],

           -(bt B0) - at S0 + V[S0, t]]
```

In order for the difference $V(S_t, t) - (a_t S_t + b_t B_t)$ to be zero, both the drift and dispersion of this new diffusion must be identically zero. This gives two equations in two unknowns, and thus the number of stock and bond shares in the matching portfolio, a_t and b_t. The built-in function **Solve** gives a set of rules that define the solution of the system of two equations.

```
In[8]:= roots = Solve[{DiffusionDrift[diff]==0,
              DiffusionDispersion[diff]==0}, {at,bt}]
Out[8]:=                (1,0)
         {{at -> (V)      [S, t], bt ->

                        (1,0)
                  mu S (V)      [S, t]
                -(-------------------) +
                        rho B

                       (0,1)                          (1,0)
                (2 (V)      [S, t] + 2 mu S (V)      [S, t] +

                          2  2     (2,0)
                     sigma  S   (V)      [S, t]) / (2 rho B)}}
```

For convenience, we next extract the values of a_t and b_t implied by these rules and assign them appropriately. The additional simplification eases later manipulations and seems unavoidable in some implementations of *Mathematica*. The function **First** in the next two expressions extracts the solution from the single-element list in which it is embedded.

```
In[9]:= at = Simplify[ First[at /. roots] ]
Out[9]=     (1,0)
          (V)      [S, t]

In[10]:= bt = Simplify[ First[bt /. roots] ]
Out[10]=       (0,1)                2  2     (2,0)
          2 (V)      [S, t] + sigma  S   (V)      [S, t]
         ------------------------------------------------
                          2 rho B
```

In more conventional notation,

$$a_t = V_x(S_t, t), b_t = \frac{V_t(S_t, t) + \frac{1}{2}\sigma^2 S_t^2 V_{xx}(S_t, t)}{2\rho B_t},$$

where V_x denotes the partial derivative of $V(x,u)$ with respect to its first argument, and V_{xx} denotes the second partial derivative in the first argument. Similarly, V_t is the first partial in the second argument.

To find a partial differential equation for the value of the option, we substitute these expressions for a_t and b_t back into the relation $V(S_t,t) = a_t S_t + b_t B_t$. The use of the function **Expand** assures that the function **Coefficient** extracts the proper term.

```
In[11]:= pde = Expand[ at S + bt B - V[S,t] ]
Out[11]=                    (0,1)
                          (V)      [S, t]          (1,0)
          -V[S, t] + ------------------------ + S (V)      [S, t] +
                              rho

                2  2     (2,0)
               sigma  S  (V)      [S, t]
             ------------------------------
                        2 rho
```

To set things up for the next step, normalize this PDE so that the coefficient of V_t is 1.

```
In[12]:= pde = Expand[ pde / Coefficient[pde, D[V[S,t],t]] ]
Out[12]=                                (0,1)                    (1,0)
          -(rho V[S, t]) + (V)      [S, t] + rho S (V)      [S, t]

                2  2     (2,0)
               sigma  S  (V)      [S, t]
          + ------------------------------
                          2
```

These equations and the boundary conditions discussed shortly are sufficient to determine V, thus solving the option pricing problem. The problem now faced is the purely mathematical one of solving our PDE. A variety of tools exist for solving PDE's that arise in the application of diffusions, and one of the most powerful is based upon the Feymann-Kac theorem. For our purposes, the Feynman-Kac theorem expresses the solution of a certain second-order PDE as an expectation with respect to a related diffusion. The first question one has to resolve before applying this method is whether the PDE of interest is of the appropriate type. The second issue is to make an explicit correspondence between one's PDE and the form of the Feynman-Kac result. The function **FeynmanKac** performs both of these tasks. As a side-effect, it also prints out the terms used in its matching using the notation of Duffie (1988).

The use of this function also requires identifying boundary conditions. Let $g(x)$ denote the payout function for the option. For example, the payout function for a European option with exercise price P is the piecewise linear function $g(x) = (x-P)^+$ where $(x-P)^+ = x - P$ if $x > P$ and is zero otherwise. The payout function determines the needed boundary condition, $V(x,T) = g(x)$. Here we apply the **FeynmanKac** function, using the symbol g to denote

the payout function. In the following output, the symbol Ave stands for the expected value operator since the symbol E denotes the base of the natural log e in *Mathematica*.

```
In[13]:= soln = FeynmanKac[pde,g]

f = V;  rho = rho;  u = 0
dX  = rho X  dt + sigma X  dW   ; X  = x
  t         t              t   t      0

Out[13]=      g[X[-t + T]]
          Ave[--------------]
                rho (-t + T)
               E
```

The "solution" given by the Feynman-Kac theorem is rather abstract and the task of rendering it concrete is not always easy. The result of this function indicates that the solution of our PDE is the expected discounted payout

$$e^{-\rho(T-t)}Eg(X_{T-t}),$$

where X_t is the diffusion that satisfies $dX_t = \rho X_t dt + \sigma X_t dW_t$ with initial value $X_0 = x$ and Weiner process W_t. The process X_t is just the familiar lognormal diffusion, as confirmed by use of Itô's formula.

We will next confirm that the exponential of the normal diffusion $dY_t = a\,dt + b\,dW_t$ is indeed a lognormal diffusion, and we use *Mathematica* to make the required identification of coefficients in both drift and dispersion.

```
In[14]:= Clear[Y];
         DiffusionMake[Y,W, a,b,0];
         Ito[Y][Exp[Y]]

                    2   Y
  Y     (2 a + b ) E              Y
dE  = (-------------) dt + (b E ) dW
              2                    t

Inversion rule... {Y -> Log[Z]}

Out[14]=                              2
                                (2 a + b ) Z
          diffusion[Z, W, -------------, b Z, 1]
                                    2
```

Clearly, b corresponds to σ. Equating the drift coefficients $(2a + b^2/2) = \rho$ shows that $a = \rho - (s^2/2)$.

```
In[15]:= a = Simplify[a/.First[ Solve[(2 a+sigma^2)/2 == rho,
         a]]]

Out[15]=              2
                  sigma
          rho - -------
                    2
```

This calculation implies that the diffusion X_t in the solution from the Feynman-Kac theorem is the exponential of a normal diffusion with constant drift $\rho -$

$(\sigma^2/2)$ and dispersion σ. Recalling that $X_0 = x$, at any time $t \geq 0$ X_t satisfies

$$X_t = xe^{(\rho - \frac{\sigma^2}{2})t + \sigma W_t}.$$

Hence for any t, X_t has the same distribution as $xe^{(\rho-(\sigma^2/2))t+\sigma\sqrt{T}Z}$ where Z is a standard normal random variable with mean zero and variance one. Finally we see that the value of the option at time $t = 0$ is

$$V(x,0) = e^{-\rho T} E g(X_T) = \frac{e^{-\rho T}}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(xe^{(\rho - \frac{\sigma^2}{2})T + \sqrt{T}\sigma z}) e^{-z^2/2} dz.$$

For the European option with exercise price P, we have $g(x) = (x - P)^+$ and this integral becomes

$$\frac{e^{-\rho T}}{\sqrt{2\pi}} \int_{z_0}^{\infty} (xe^{(\rho - \frac{\sigma^2}{2})T + \sqrt{T}\sigma z} - P) e^{-z^2/2} dz,$$

where z_0 solves

$$xe^{(\rho - \frac{\sigma^2}{2})T + \sqrt{T}\sigma z_0} = P.$$

Calculation of this integral is somewhat tedious, but its evaluation makes for a nice illustration of the integral solving capabilities of *Mathematica*. To make the coding a little more modular, we first define a function $h(z) = xe^{(\rho-(\sigma^2/2))T+\sqrt{T}\sigma z} - P$ and use it to locate the lower bound z_0.

```
In[16]:= h[z_] := x Exp[(rho-sigma^2/2.)T+Sqrt[T] sigma z] - P
         rule = Solve[ h[z]==0, z];
         z0 = Simplify[ First[z /. rule] ]
```

$$\text{Out[16]:} \quad \frac{-(\text{rho}\ T) + 0.5\ \text{sigma}^2\ T + \text{Log}[-]}{\text{sigma Sqrt[T]}}$$

To simplify the input further, we define the Gaussian kernel gauss(z)=$e^{-z^2/2}/\sqrt{2\pi}$.

```
In[17]:= gauss[z_] := E^(-(z^2)/2)/Sqrt[2 Pi]
```

Given these definitions of the functions **h** and **gauss**, it is quite simple to describe the needed integral. We call the integration function with specific limits a and b rather than symbolic infinite limits. The integration routines seem to behave more robustly in this application with these limits rather than infinite limits.

```
In[18]:= Clear[a,b]
         intab = Integrate[E^(-rho T) h[z] gauss[z], {z,a,b}]
```

$$\text{Out[18]:} \quad \frac{P\ \text{Erf}\left[\frac{a}{\text{Sqrt[2]}}\right]}{2\ E^{\text{rho}\ T}} - \frac{P\ \text{Erf}\left[\frac{b}{\text{Sqrt[2]}}\right]}{2\ E^{\text{rho}\ T}} -$$

```
Out[18](cont.)
                         2                      2
   -(rho T) + (sigma  T)/2 + (rho - 0.5 sigma ) T
(E                                                    x

     a - sigma Sqrt[T]
 Erf[-----------------]) / 2 +
        Sqrt[2]

                         2                      2
   -(rho T) + (sigma  T)/2 + (rho - 0.5 sigma ) T
(E                                                    x

     b - sigma Sqrt[T]
 Erf[-----------------]) / 2
        Sqrt[2]
```

Now we can apply two substitution rules that specify the range of integration, $a = z_0$ and $b = \infty$, and we have the desired integral.

```
In[19]:= int = intab /. {a->z0, b->Infinity};
         Simplify[int]
Out[19]:=
                                           2           P
                     -(rho T) - 0.5 sigma  T + Log[-]
                                                       x
                 x Erf[----------------------------------]
   -P       x                Sqrt[2] sigma Sqrt[T]
 ------ + - - ----------------------------------------------
  rho T    2                      2
 2 E

                           2           P
          -(rho T) + 0.5 sigma  T + Log[-]
                                        x
     P Erf[------------------------------]
                Sqrt[2] sigma Sqrt[T]
  + ----------------------------------------
                     rho T
                  2 E
```

This expression involves the error function defined in *Mathematica* as $\mathrm{erf}(z) = \int_0^z e^{-x^2} dx$. We get a more familiar result by using the equivalence $\mathrm{erf}(z) = 2\Phi(\sqrt{2}z) - 1$, where $\Phi(z)$ denotes the cumulative standard normal distribution, $\Phi(z) = \int_{-\infty}^{z} (e^{(-x^2/2)}/\sqrt{2\pi}) dx$.

```
In[20]:= bs = Simplify[int /. Erf[x_] ->
              (2 NormalCDF[Sqrt[2]x] - 1)]
Out[20]:=
                                              2         P
                            -(rho T) - 0.5 s  T + Log[-]
    P                                                   x
-(------) + x - x NormalCDF[----------------------------] +
   rho T                            s Sqrt[T]
  E
```

```
Out[20](cont.)
                                                  2          P
                                    -(rho T) + 0.5 s  T + Log[-]
                                                             x
                  P NormalCDF[-------------------------------------]
                                          s Sqrt[T]
                 ------------------------------------------------------
                                        rho T
                                       E
```

We can express this result in yet more familiar form by using a rule that expresses the relationship $\Phi(x) = 1 - \Phi(-x)$.

```
In[21]:= Simplify[ bs/.NormalCDF[x_]->1-NormalCDF[-x]]
Out[21]=
                                                    2          P
                                      -(rho T) - 0.5 s  T + Log[-]
                                                               x
                  x NormalCDF[-(-------------------------------------)] -
                                            s Sqrt[T]

                                                      2          P
                                        -(rho T) + 0.5 s  T + Log[-]
                                                                 x
                    P NormalCDF[-(-------------------------------------)]
                                              s Sqrt[T]
                   ------------------------------------------------------
                                          rho T
                                         E
```

This is the Black-Scholes formula for pricing the European option (Duffie, 1988, p. 239). Miller (1990) discusses this function at length and shows various manipulations of it using *Mathematica*.

9.5 More Mathematica Details and Examples

The material of this section focusses on some issues that can be omitted at a first reading, but that might prove useful for the user who wishes to extend the methods or examples. The discussion begins with the underlying data structure used to represent a diffusion. Many of the ideas come from object-oriented programming. We next consider in some detail the use of our functions that produce the infinitesimal and Itô's formula. Further programming details appear as comments embedded in the code of the package itself.

The examples in this section use both of the *Mathematica* diffusion objects built in Section 9.3. In case these are not available, the following command rebuilds the Weiner process W_t and the associated lognormal diffusion X_t.

```
In[1]:= Clear[X,W];
        WeinerProcessMake[W];
        DiffusionMake[X,W,alpha X, beta X, v];
```

The data structure we use to represent a diffusion parallels the notation of Section 9.2. To represent an arbitrary diffusion, one must describe each of the

distinguishing components: drift, dispersion, Weiner process, and initial value In addition, we need to supply a symbol that names the diffusion itself. As a result, the list that we use to represent a diffusion has these five items: name of the diffusion, name of Weiner process, drift expression, dispersion expression, and initial value. This data structure is particularly simple for a Weiner process. The first two symbols which are the names of the diffusion and its underlying Weiner process match for a Weiner process. The drift and dispersion are the constants 0 and 1, respectively. The matching of the leading symbols identifies in the software the presence of a Weiner process.

Several accessor functions permit one to extract components of a diffusion without requiring detailed knowledge of its data structure. Although these functions simply index the list that holds the components of the diffusion, use of these accessors frees one from having to remember the arrangement of the list. Such abstraction offers the opportunity to change the data structure at a later point without having to rewrite code that uses accessor functions. Here are a few examples of the accessor functions.

```
In[2]:= DiffusionWeinerProcessSymbol[X]

Out[2]:= W
```

Notice that this function results in the symbol associated with the Weiner process rather than the Weiner process itself. A different accessor extracts the process.

```
In[3]:= DiffusionWeinerProcess[X]

Out[3]:= diffusion[W, W, 0, 1, 0]
```

The next three examples extract the remaining components of the diffusion.

```
In[4]:= DiffusionDrift[X]

Out[4]:= alpha X

In[5]:= DiffusionDispersion[X]

Out[5]:= beta X

In[6]:= DiffusionInitialValue[X]

Out[6]:= v
```

It is important to notice that the results of some of these functions include the symbol X which represents the diffusion. A sleight of hand is needed to obtain this behavior, and all is not quite as it appears. The built-in function **FullForm** reveals that the results of both **DiffusionDrift** and **DiffusionDispersion** contain so-called "held expressions" that one must use in order to keep the system from trying to evaluate the diffusion symbols that appear in the drift and dispersion expressions.

```
In[7]:= DiffusionDrift[X] // FullForm

Out[7]:= Times[alpha, HoldForm[X]]
```

Were it not for holding the evaluation of the symbol X in this expression, *Mathematica* would descend into an endless recursion, continually substituting the list representing the diffusion each time it encountered the symbol X in the list. Further discussion of this recursion appears in Steele and Stine (1991).

We have found that it is most easy to manipulate the drift and dispersion functions as expressions rather than *Mathematica* functions. Still, occasions arise when one wants a function. For example, one might want to differentiate or plot the drift. The accessors normally extract these components in a manner that preserves their symbolic content. That is, they return an expression which includes the symbol for the diffusion. When a function is desired, the accessors **DiffusionDrift** and **DiffusionDispersion** have an optional argument that forces the output to be returned as a function. The returned function has two arguments, the first for the diffusion and the second for time, as in $\mu(x,t)$ and $\sigma(x,t)$ respectively.

As example, consider generating a plot of the drift from the lognormal diffusion. First, extract the drift as a *Mathematica* function and give it a suggestive name.

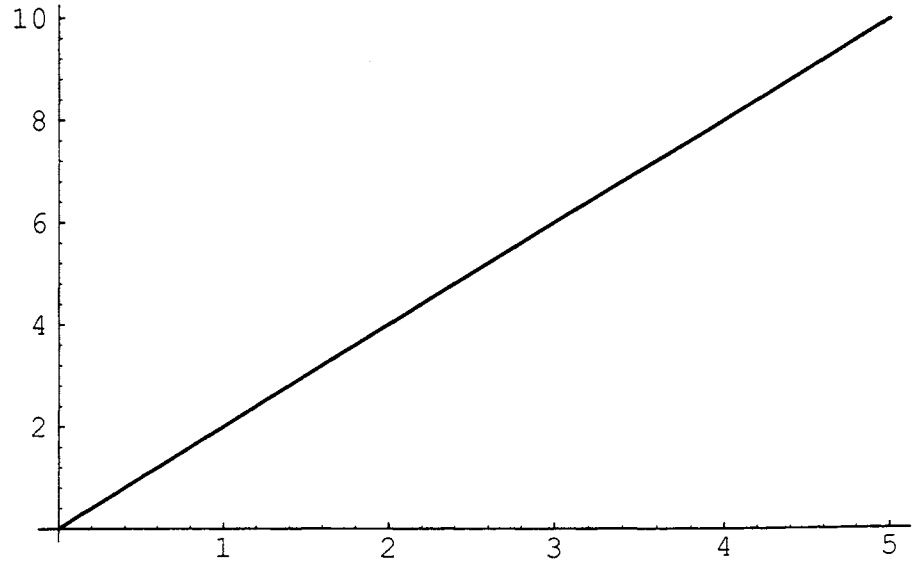
```
In[8]:= mu = DiffusionDrift[X,Function]

Out[8]:= Function[{x$, t$}, alpha x$]
```

We can treat this function just like any other, differentiating or plotting as we choose.

```
In[9]:= D[mu[x,t],x]

Out[9]:= alpha
```

Of course, if we expect to plot the drift function, we have to make sure that all of the symbolic terms have a value. Here we set $\alpha = 2$ so that $\mu(x,t) = 2x$.

```
In[10]:= Plot[ mu[x,t]/.alpha->2, {x,0,5} ]
```




```
Out[10]:= -Graphics-
```

In general, we would need to plot the drift $\mu(x,t)$ as a surface over the plane, bu in this and many other common circumstances this elaborate plot is not needed

As we noted in our review of Itô's formula, the infinitesimal generator o a diffusion process has an intimate relationship to the stochastic differentia representation of the diffusion. The argument given to the function which find the infinitesimal is an expression involving a diffusion. For example, the nex example determines the infinitesimal of the process defined by an arbitrary function g of W_t. We see that the infinitesimal is half of the second derivative of g.

```
In[11]:= A[g[W]]
Out[11]:= g''[W]
          ------
            2
```

For the lognormal diffusion X_t, the result is somewhat more complex, but the syntax of the commands is the same. We saw this expression in Section 9.4 in the option-pricing problem.

```
In[12]:= A[g[X]]
Out[12]=
                                    2  2
                                beta  X  g''[X]
          alpha X g'[X] + ------------------
                                    2
```

The function associated with Itô's formula provided in the accompanying package is more potent than the other functions of the package. The function **Ito** (or more elaborately, **DiffusionIto**) builds the new diffusion associated with the input expression which again is an expression which includes a diffusion. The program also assigns a default name to the new diffusion unless a new name is chosen.

As an example we show how to use Itô's formula to find the diffusion associated with half the square of a Weiner process, $Y_t = W_t^2/2$. The optional argument sets the name of the new diffusion to be the symbol Y which has been cleared of any prior value.

```
In[13]:= Clear[Y]
         Ito[1/2 W^2, ItoSymbol->Y]
   2
  W      1
d--  =  (-) dt + (W) dW
  2      2          t

Inversion rule... {W -> Sqrt[2] Sqrt[Y]}
Out[13]=                    1
          diffusion[Y, W, -, Sqrt[2] Sqrt[Y], 0]
                            2
```

Two pieces of intermediate output precede the final result in this example. The first portion of the output shows the diffusion associated with the transformation

in the form of (3) before conversion to canonical form. This portion of the output is occasionally useful in solving various stochastic integrals. In this example, the output expression $dW_t^2/2 = (1/2)dt + W_t dW_t$ suggests the solution to a stochastic integral,

$$\int_0^t W_s dW_s = W_t^2/2 - 1/2$$

Of course, one would have to know to look at Itô's formula applied to W_t^2 in order to find $\int_0^t W_s dW_s$ in this way. However, since $\int_0^t x dx = t^2/2$, this is not such a bad place to start looking for an answer.

The second piece of intermediate output in the example beginning "Inversion rule..." (above) gives the inverse transformation used to convert the result of Itô's formula (which is a function of W_t) into a function of Y_t. This is the inverse function g described in the introductory review.

Here is another example. This example shows that the exponential of a Weiner process is a lognormal diffusion. Notice as always that it is important to begin with an unbound symbol to use for the new diffusion.

```
In[14]:= Clear[Y]
         Ito[Exp[W], ItoSymbol->Y]

          W
  W      E            W
dE   = (---) dt + (E  ) dW
        2                 t

Inversion rule... {W -> Log[Y]}

Solve::ifun:
   Warning: Inverse functions are being used by Solve, so
     some solutions may not be found.

Out[14]=                    Y
          diffusion[Y, W, -, Y, 1]
                            2
```

The warning message in this example will often suggest a problem in the inversion process. In this example the inversion via logarithms works since $Y_t = e^{W_t}$ implies $W_t = \log_e Y_t$. Were the process W_t complex, for example, such inversion might not be appropriate. Since *Mathematica* does not assume W_t is real-valued, it displays a warning.

9.6 Summary and Concluding Remarks

This chapter began by reviewing some of the basic notions of the theory of diffusions such as the local drift μ, the local dispersion σ, and the specification of a diffusion process though the formalism of stochastic integraton. We then illustrated how the fundamental formula of Itô for functions of diffusions can lead to calculations that can be profitably performed by symbolic methods.

9.7 References

Arnold, L. (1974). *Stochastic Differential Equations: Theory and Applications*. Wiley, New York.

Duffie, D. (1988). *Security Markets*. Academic Press, New York.

Steele, J. M. and R. A. Stine (1991). "Applications of *Mathematica* to the stochastic calculus." In *American Statistical Association, Proceedings of the Statistical Computing Section.*, 11–19, Amercian Statistical Association, Washington, D.C.

Miller, R. (1990). "Computer-aided financial analysis: an implementation of the Black-Scholes model." *Mathematica Journal*, **1**, 75–79.