

Simulation & Stochastic Differential Equations

In recent years, the advent of new regulatory requirements, such as the Basel accord, and the dramatic growth of computing power have made Monte Carlo simulation an increasingly important and attractive technique with numerous applications in finance, especially in the area of risk management and derivative pricing.

More specifically, numerous financial clients have requested additional MATLAB tools to support the simulation of stochastic differential equations (SDEs). To address these requests, The MathWorks has been actively developing a host of new and enhanced tools to directly support Monte Carlo simulation and related techniques.

This presentation will preview new functionality specifically related to Monte Carlo simulation, such as:

- SDE simulation, stochastic interpolation, and Brownian bridges
- Variance reduction techniques, including antithetic and stratified sampling
- User-specified random number generators & interfaces
- Extreme value theory (EVT) and piecewise distributions
- Calibration and simulation of Gaussian & t copulas

Contents

- [Import the Supporting Historical Dataset](#)
- [Overview of Stochastic Differential Equations](#)
- [Incorporating Dynamic Behavior](#)
- [The Brownian Bridge & Stochastic Interpolation](#)
- [End-of-Period Processes: Black-Scholes Option Pricing](#)
- [User-Specified Random Number Generation: Stratified Sampling](#)
- [Pricing American Basket Options Using Copulas & Extreme Value Theory](#)

Import the Supporting Historical Dataset

To support the following presentation, first load a daily historical dataset of 3-month Euribor (converted to daily effective yield), the closing index levels of representative large-cap equity indices of Canada (TSX Composite), France (CAC 40), Germany (DAX), Japan (Nikkei 225), UK (FTSE 100), and US (S&P 500), and corresponding trading dates spanning the interval 07-Feb-2001 to 24-Apr-2006.

There are numerous sources from which the supporting data may be imported into MATLAB, including various data service providers by way of the Datafeed Toolbox, spreadsheets applications, and directly from native binary MATLAB files (i.e., MAT-files).

The following code segment uses the Database Toolbox to import the historical dataset. It first makes a database connection, then opens a cursor, and finally fetches the historical data formatted as a data structure.

```
setdbprefs('DataReturnFormat','structure')
connection = database('SDE_Data','','');
cursor     = exec(connection,'SELECT ALL Dates,Canada,France,Germany,Japan,UK,US,
Euribor3M FROM Simulation_Data ORDER BY Dates ASC');
```

```

cursor      = fetch(cursor);
close(cursor), close(connection)

```

```
SDE_Data = cursor.Data
```

```

SDE_Data =
    Dates: [1359x1 double]
    Canada: [1359x1 double]
    France: [1359x1 double]
    Germany: [1359x1 double]
    Japan: [1359x1 double]
    UK: [1359x1 double]
    US: [1359x1 double]
    Euribor3M: [1359x1 double]

```

The following plots illustrate the data just imported. Specifically, we plot the relative price movements of each index as well as the Euribor risk-free rate proxy. Notice that the initial level of each index has been normalized to unity to facilitate the comparison of relative performance over the historical record.

```

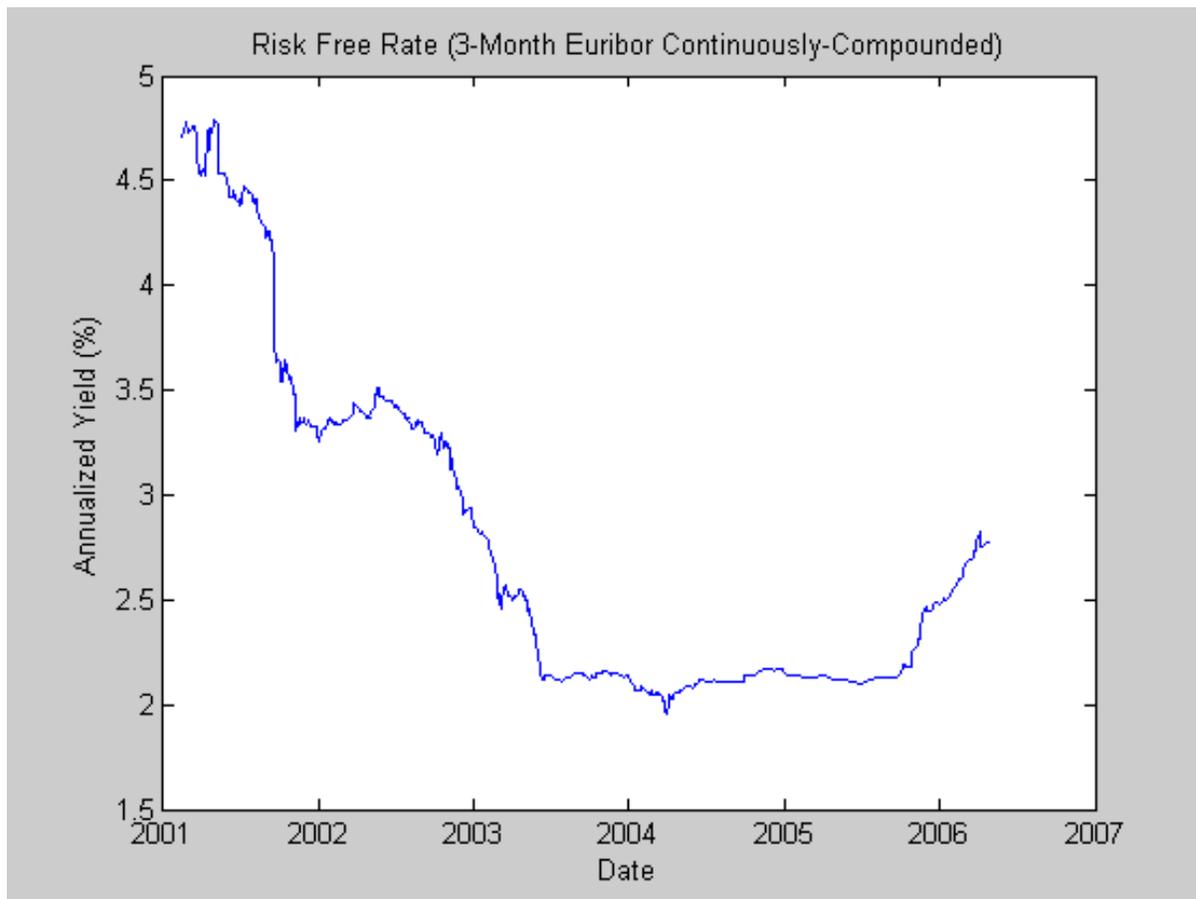
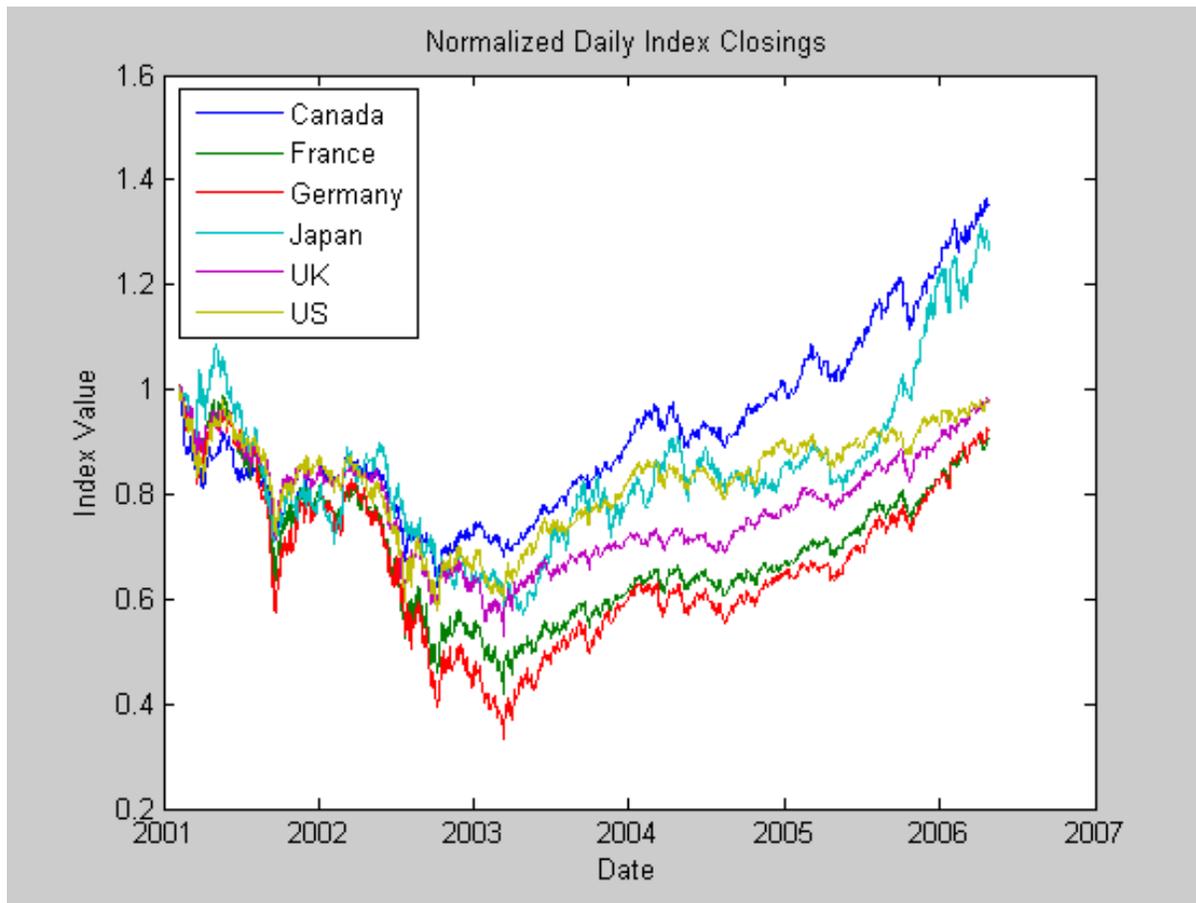
fields      = fieldnames(SDE_Data);
dates       = SDE_Data.Dates;
countries   = fields(2:end-1);
yields     = SDE_Data.Euribor3M;           % daily effective yields
yields     = 360 * log(1 + yields);        % continuous, annualized yields

for i = length(countries):-1:1
    prices(:,i) = SDE_Data.(countries{i}); % daily closing prices
end

figure, plot(dates, ret2price(price2ret(prices))), datetick('x')
xlabel('Date'), ylabel('Index Value'), title('Normalized Daily Index Closings')
legend(countries, 'Location', 'NorthWest')

figure, plot(dates, 100 * yields)
datetick('x'), xlabel('Date'), ylabel('Annualized Yield (%)')
title('Risk Free Rate (3-Month Euribor Continuously-Compounded)')

```



Overview of Stochastic Differential Equations

The following sections highlight important new features and functionality specifically designed to allow financial clients to simulate Stochastic Differential Equations (SDEs) in MATLAB.

Interestingly, most clients that participated in MATLAB advisory board/focus group discussions indicated that they were primarily interested in efficient simulation methods, and were largely content to calibrate their own models, citing the complexity associated with model calibration as *an art rather than a science*.

The architecture is completely new and makes extensive use of some of the latest MATLAB language features, including object oriented programming capabilities built upon the new MATLAB Common Object System (MCOS) infrastructure and data encapsulation using nested functions.

High Level Features

Although many of the following features are discussed below and demonstrated by way of a series of examples, it useful to highlight some of the more salient features here:

- Euler approximation default simulation method
- Approximate analytic solution for separable geometric Brownian motion and Hull-White/Vasicek models
- Specialized methods for efficient simulation of static, separable Geometric Brownian Motion and Brownian Motion multivariate models
- Vectorized methods for efficient simulation of static univariate models
- Stochastic interpolation & Brownian bridge simulation methods
- Full support for any combination of static and dynamic model parameters
- Full support for state and Brownian vectors of arbitrary dimensionality
- Optional user-specified random number generation & dependence/correlation structure
- Antithetic sampling
- End-of-period processing/state vector adjustments to perform virtually any type of path-dependent analysis
- Ability to sample an SDE at intermediate times without reporting those times, improving accuracy and reducing memory burden
- Ability to avoid storing state and noise time series to improve performance and memory efficiency

General Parametric Specification

The proposed SDE engine allows the simulation of generalized multivariate stochastic processes, and is designed to provide a flexible and powerful simulation architecture. In addition, the framework provides several utilities and model classes, offering users a variety of parametric specifications and interfaces.

The architecture is fully multi-dimensional in the both the state vector as well as the Brownian motion, and offers users the convenience of linear and mean-reverting drift rate specifications. Most parameters may be specified as traditional MATLAB arrays or as functions accessible by a common interface, thereby supporting rather general dynamic/non-linear relationships common in SDE simulations.

Specifically, the architecture allows users to simulate correlated paths of any number of state variables driven by a vector-valued Brownian motion of arbitrary dimensionality, thereby approximating the underlying multivariate continuous-time process by a vector-valued stochastic difference equation.

Consider the following general stochastic differential equation,

$$dX_t = F(t, X_t)dt + G(t, X_t)dW_t$$

where X is an $NVARS \times 1$ state vector of process variables (e.g., short rates, equity prices) we wish to simulate, dW is an $NBROWNS \times 1$ Brownian motion vector (also referred to as a Wiener process), F is an $NVARS \times 1$ vector-valued drift rate function, and G is an $NVARS \times NBROWNS$ matrix-valued diffusion rate function.

Notice that the drift and diffusion rates, F and G , respectively, are rather general functions of a real-valued scalar sample time t and state vector $X(t)$. Also, notice that static (non-time-varying) coefficients are simply a special case of the more general dynamic (time-varying) situation, in the same manner as a function may be a trivial constant, e.g., $f(t,X) = 4$.

The above SDE is quite general, and so it is often useful to implement derived classes which impose additional structure on the drift and diffusion rate functions.

Consider the following linear drift rate specification,

$$F(t, X_t) = A(t, X_t) + B(t, X_t)X_t$$

where A is an $NVARS \times 1$ vector-valued function and B is an $NVARS \times NVARS$ matrix-valued function.

As an alternative, consider a drift rate specification expressed in mean-reverting form,

$$F(t, X_t) = S(t, X_t)[L(t, X_t) - X_t]$$

S is an $NVARS \times NVARS$ matrix-valued function of mean reversion speeds (i.e., rates of mean reversion), and L is an $NVARS \times 1$ vector-valued function of mean reversion levels (i.e., long run average level).

Similarly, consider the following diffusion rate specification,

$$G(t, X_t) = D(t, X_t^{\alpha(t, X_t)})V(t, X_t)$$

where D is an $NVARS \times NVARS$ diagonal matrix-valued function in which each diagonal element is the corresponding element of the state vector raised to the corresponding element of an exponent α , which is also an $NVARS \times 1$ vector-valued function. V is an $NVARS \times NBROWNS$ matrix-valued function of instantaneous volatility rates in which each row corresponds to a particular state variable and each column to a particular Brownian source of uncertainty, and associates the exposure of state variables with sources of risk.

The parametric specifications for the drift and diffusion rate functions are designed to impose a sense of structure, more intuitively associating parametric restrictions with familiar models derived from the general SDE class.

Common Interface and Behavior

As already mentioned, much of the design is driven by a desired look and feel. In fact, most models and utilities outlined below are MATLAB objects, and share common behavior obtained by intentionally blurring the lines between methods, user-defined functions, object properties, and fields of traditional data structures.

Although the following examples elaborate in more detail, it is important to highlight the fact that

dynamic (i.e., time-varying) behavior is associated with function evaluation. Moreover, this dynamic behavior is accessible by passing time and state (t, X) to a common, published interface, and is pervasive throughout the SDE class system. Although this (t, X) interface appears limited, this seemingly simple function evaluation approach may be used to model or construct powerful analytics, especially when used with concert with nested functions.

Throughout many of the examples that follow, most model members may be evaluated, or invoked, as would any MATLAB function. Thus, although it may be helpful to examine and access object members in a manner similar to that of data structures, it is often more convenient and useful to think of them as functions that perform an action.

Incorporating Dynamic Behavior

As previously mentioned, object members are designed to be evaluated as if they were MATLAB functions accessible by a common interface. This accessibility, in turn, gives users the impression of dynamic behavior regardless of whether or not the underlying parameters are truly time-varying. Since members are accessible by a common interface, seemingly simple, linear constructs may in fact represent complex, non-linear designs.

Moreover, when members are entered as functions object constructors may only verify that they return arrays of correct size by evaluating them at the initial time and state, but otherwise have no knowledge of any particular functional form. This is both flexible and potentially dangerous.

Most users will create objects by entering traditional MATLAB arrays that explicitly represent constant (non-dynamic) parameters, or will need to convert arrays that represent historical time series.

Since time series arrays represent dynamic behavior, and dynamic behavior must be captured by functions accessible by the (t, X) interface, users will need utilities to convert traditional time series arrays into callable functions of time and state. To address this, a special conversion function called **ts2func** (i.e., time series to function) is available.

To illustrate dynamic behavior, the following example works with the daily historical dataset of 3-month Euribor rates and closing levels of the French CAC 40. Assume we wish to simulate risk-neutral sample paths of the CAC 40 index using a geometric Brownian motion (GBM) model,

$$dX_t = r_t X_t dt + \sigma X_t dW_t$$

in which

$$r_t$$

represents evolution of the risk-free rate of return.

Furthermore, assume we wish to annualize the relevant information derived from the daily data, and that each calendar year is composed of 250 trading days.

```
clc
dt      = 1 / 250;           % time increment = 1 day = 1/250 years
returns = price2ret(SDE_Data.France); % daily log returns of CAC 40
sigma   = std(returns) * sqrt(250); % annualized volatility
```

```
yields = SDE_Data.Euribor3M;
yields = 360 * log(1 + yields);           % continuously-compounded, annual yield
```

Now suppose we wish to compare the resulting sample paths obtained from two risk-neutral historical simulation approaches in which the daily Euribor yields serve as a proxy for the risk-free rate of return.

The first approach specifies the risk-neutral return as the sample average of Euribor yields, and therefore assumes a constant (non-dynamic) risk-free return.

```
nPeriods = length(yields);               % # of simulated observations

randn('state', 25)
obj = gbm(mean(yields), diag(sigma), 'StartState', 100)
[X1,T] = obj.simulate(nPeriods, 'DeltaTime', dt);
```

```
obj =
Class GBM: Generalized Geometric Brownian Motion
-----
Dimensions: State = 1, Brownian = 1
-----
StartTime: 0
StartState: 100
Correlation: 1
Drift: drift rate function F(t,X(t))
Diffusion: diffusion rate function G(t,X(t))
Simulation: simulation method/function simByEuler
Return: 0.0278117
Sigma: 0.231875
```

In contrast, the second approach specifies the risk-neutral return as the historical time series of Euribor yields, and therefore assumes a dynamic, yet deterministic, rate of return (i.e., this example does *not* illustrate stochastic interest rates). To illustrate this dynamic effect, use the **ts2func** utility,

```
r = ts2func(yields, 'Times', (0:nPeriods - 1)'), clc
```

```
r =
@ts2func/vector2Function
```

Notice that the **ts2func** utility packages a specified time series array inside a callable function of time and state, and also synchronizes it with an optional time vector. For instance,

```
r(0,100)
```

```
ans =
0.0470
```

evaluates the function at $(t = 0, X = 100)$ and returns the first observed Euribor yield. However, notice that the resulting function may also be evaluated at any intermediate time t and state X ,

```
r(7.5, 200)
```

```
ans =  
    0.0472
```

Furthermore, notice that the following command produces exactly the same result when called with time alone,

```
r(7.5)
```

```
ans =  
    0.0472
```

The equivalence of these last two commands highlights some important features of the architecture.

When members are specified as functions, they must evaluate properly when passed a scalar, real-valued sample time (t) followed by a state vector (X), and generate an array of appropriate dimensions (which in the first case is a scalar constant, and in the second is a scalar, piecewise constant function of time alone).

Notice that there is absolutely no obligation to use either time (t) or state (X). In fact, the caller has no knowledge of any implementation details. In this respect the architecture is only publishing an interface: it specifies what the inputs and outputs must be, but otherwise offers users complete flexibility regarding implementation.

In the current example, the function does indeed evaluate properly when passed time followed by state, thereby satisfying the minimal requirements. The fact that it also evaluates correctly when passed only time simply indicates that the function does not require the state vector X . The important point to make is that it works when passed (t, X) !

Furthermore, notice that the utility **ts2func** performs a zero-order-hold (ZOH) piecewise constant interpolation, and that the notion of piecewise constant parameters is pervasive throughout the SDE architecture.

To complete the comparison, perform the second simulation using the same initial random number state,

```
randn('state', 25), clc  
obj = gbm(r, diag(sigma), 'StartState', 100)  
X2 = obj.simulate(nPeriods, 'DeltaTime', dt);
```

```
obj =  
Class GBM: Generalized Geometric Brownian Motion
```

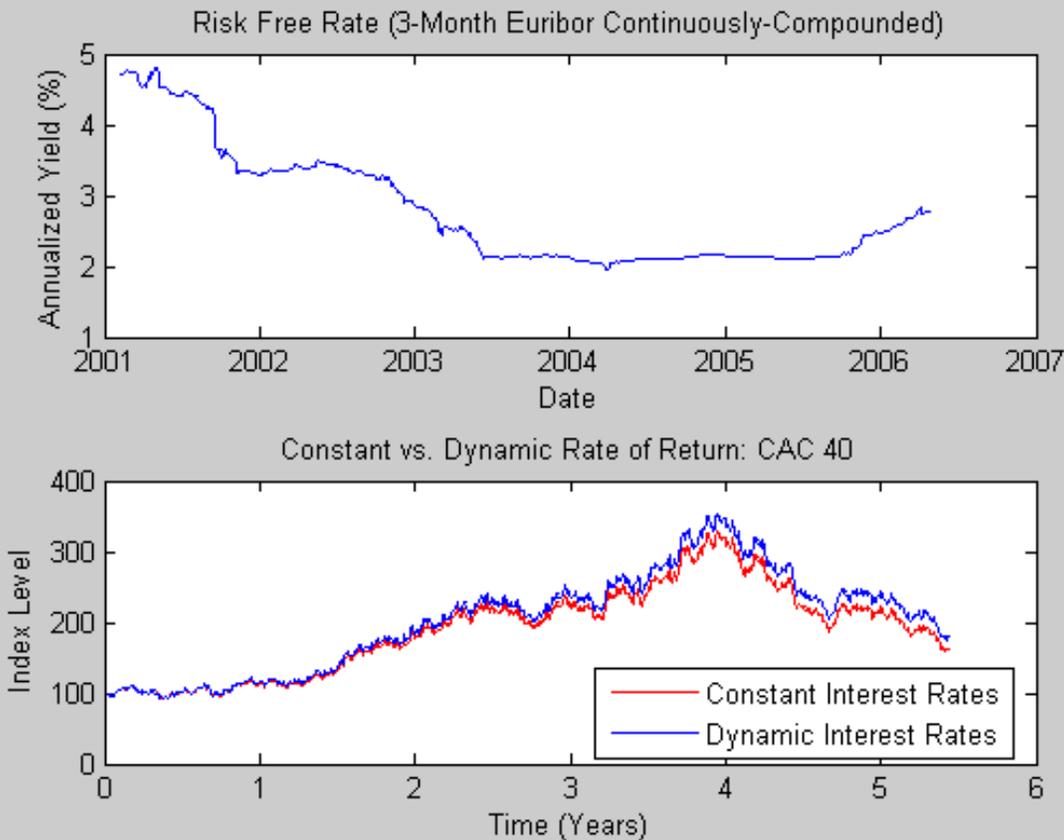
```
-----
Dimensions: State = 1, Brownian = 1
-----
```

```
StartTime: 0
StartState: 100
Correlation: 1
  Drift: drift rate function F(t,X(t))
  Diffusion: diffusion rate function G(t,X(t))
Simulation: simulation method/function simByEuler
  Return: function ts2func/vector2Function
  Sigma: 0.231875
```

Finally, plot the series of risk-free reference rates and compare the two simulation trials,

```
figure, subplot(2,1,1)
plot(SDE_Data.Dates, 100 * yields)
datetick('x'), xlabel('Date'), ylabel('Annualized Yield (%)')
title('Risk Free Rate (3-Month Euribor Continuously-Compounded)')

subplot(2,1,2)
plot(T, X1, 'red', T, X2, 'blue')
xlabel('Time (Years)'), ylabel('Index Level')
title('Constant vs. Dynamic Rate of Return: CAC 40')
legend({'Constant Interest Rates' 'Dynamic Interest Rates'}, 'Location', 'Best')
```



Notice that the paths are quite close, but not identical. The blue line in the bottom plot uses all the historical Euribor data, and illustrates a single trial of an historical simulation.

The Brownian Bridge & Stochastic Interpolation

Many applications involve post-processing of simulated paths, and may require knowledge of the state vector at intermediate sample times initially unavailable. One way to approximate these intermediate states is to perform some sort of deterministic interpolation. However, deterministic interpolation techniques fail to capture the correct probability distribution at these intermediate times.

A better technique that captures the correct joint distribution performs a Brownian, or stochastic, interpolation by sampling from a conditional Gaussian distribution. This sampling technique is sometimes referred to as a *Brownian bridge*.

For reference, notice that all simulation methods require users to specify a time grid by specifying the number of periods, optionally augmented with a scalar or vector of strictly positive time increments, and a number of intermediate time steps. The combination of these, together with an initial sample time associated with the object, uniquely determines the sequence of times at which the state vector is sampled. Thus, simulation methods traverse the time grid from beginning to end (i.e., from left to right).

In contrast, interpolation methods allow the time grid to be traversed in any order, allowing both forward and backward movements in time. To do so, they allow users to specify a vector of interpolation times, the elements of which are not required to be unique.

Since samples may be generated in any order, a traditional Monte Carlo simulation is just a special case of interpolation: an open-ended interpolation that involves sampling from a Gaussian distribution conditioned on the previous state alone.

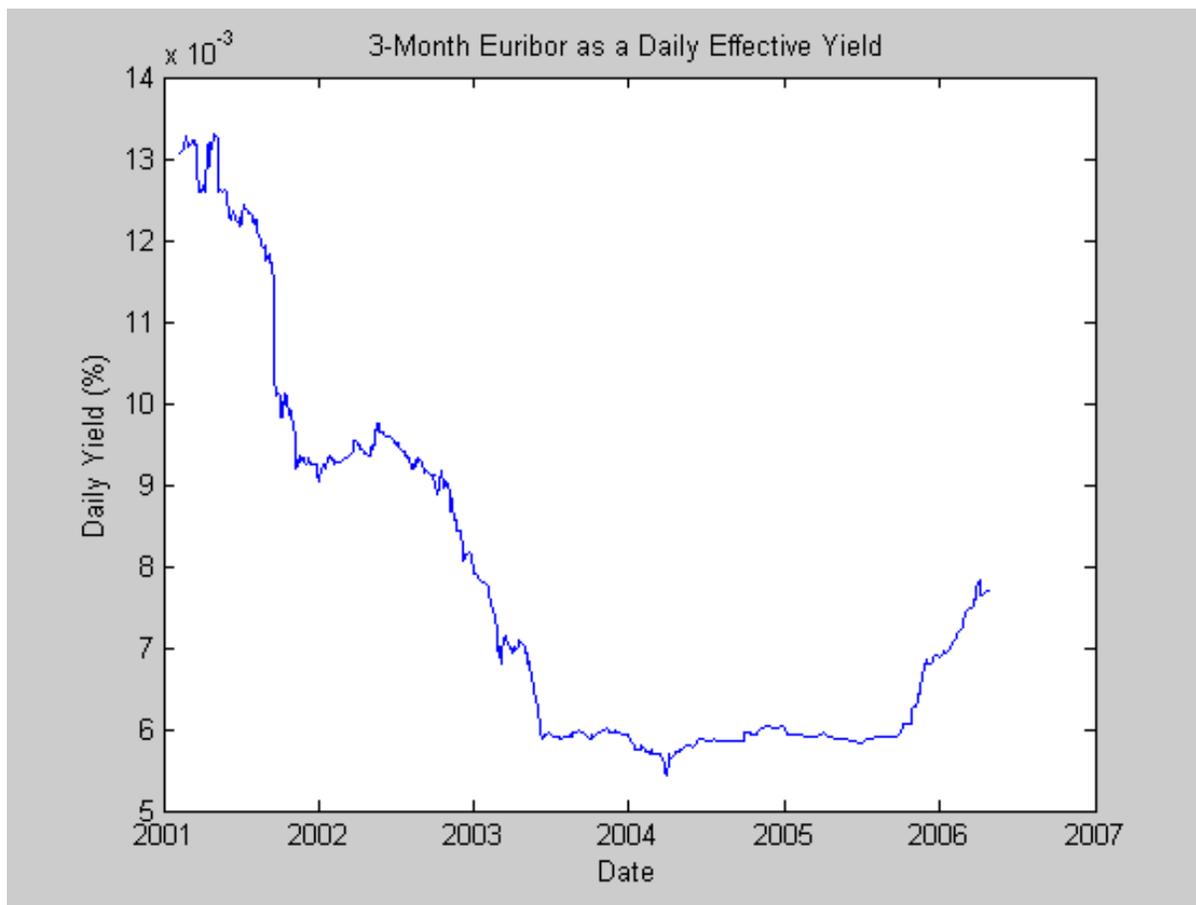
Note that many references define *The Brownian Bridge* as a conditional simulation combined with a scheme for traversing the time grid, effectively merging 2 distinct algorithms. In contrast, the interpolation method offered here provides additional flexibility by intentionally separating the algorithms.

One popular algorithm for moving about a time grid involves an initial Monte Carlo simulation to sample the state at the terminal time, then successively sampling intermediate states by stochastic interpolation. This algorithm allows the first few samples to determine the overall behavior of the paths, while later samples progressively refine the structure. Such algorithms are often cited as *variance reduction techniques*.

This algorithm is particularly simple when the number of interpolation times is a power of 2. In this case, each interpolation falls midway between 2 known states, refining the interpolation in a manner similar to bi-section. The following example highlights the flexibility of refined interpolation by implementing this popular power-of-two algorithm.

To illustrate this approach, we again work with the daily series of 3-month Euribor rates.

```
clf, plot(SDE_Data.Dates, 100 * SDE_Data.Euribor3M)
datetick('x'), xlabel('Date'), ylabel('Daily Yield (%)')
title('3-Month Euribor as a Daily Effective Yield')
```



Now consider fitting a simple univariate Vasicek model to the daily equivalent yields of the 3-month Euribor data,

$$dX_t = S(t, X_t)[L(t, X_t) - X_t]dt + V(t, X_t)dW_t$$

in which

$$S(t, X_t) = S, L(t, X_t) = L, V(t, X_t) = \sigma$$

are scalar constants. Given initial conditions, the distribution of the short rate at some time T in the future is Gaussian with mean

$$E(X_T) = X_0 e^{-ST} + L(1 - e^{-ST})$$

and variance

$$\text{Var}(X_T) = \sigma^2(1 - e^{-2ST})/2S$$

To calibrate this simple short rate model, re-write the model in more familiar regression format,

$$y_t = \alpha + \beta x_t + \epsilon_t$$

in which

$$y_t = dX_t, \alpha = SLdt, \beta = -Sdt,$$

and perform an ordinary linear regression in which the model volatility is proportional to the standard error of the residuals,

$$\sigma = \sqrt{\text{Var}(\epsilon_t)/dt}$$

```
yields      = SDE_Data.Euribor3M;
regressors  = [ones(length(yields) - 1, 1)  yields(1:end-1)];

[coefficients, intervals, residuals] = regress(diff(yields), regressors);

dt      = 1;           % time increment = 1 day
speed   = -coefficients(2) / dt;
level   = -coefficients(1) / coefficients(2);
sigma   = std(residuals) / sqrt(dt); clc
```

Now create a HWV object with an initial StartState set to the most recently observed short rate,

```
obj = hwv(speed, level, sigma, 'StartState', yields(end))
```

```
obj =
Class HWV: Hull-White/Vasicek
-----
Dimensions: State = 1, Brownian = 1
-----
StartTime: 0
StartState: 7.70408e-005
Correlation: 1
    Drift: drift rate function F(t,X(t))
    Diffusion: diffusion rate function G(t,X(t))
Simulation: simulation method/function simByEuler
    Sigma: 4.77637e-007
    Level: 6.00424e-005
    Speed: 0.00228854
```

Assume, for example, we wish to simulate the fitted model over 64 trading days (a power of 2), but using a refined Brownian bridge with the power-of-two algorithm instead of the usual beginning-to-end Monte Carlo simulation approach.

Furthermore, assume that the initial time and state coincide with those of the last available observation of the historical data, and the terminal state is the expected value of the Vasicek model 64 days into the future. In this case, we may assess the behavior of various paths that all share the same initial and terminal states, perhaps to support pricing path-dependent interest rate options over a 3-month interval.

The following code segment creates a vector of interpolation times to traverse the time grid by moving both forward and backward in time. Specifically, the first interpolation time is set to the most recent short rate observation time, the second interpolation time is set to the terminal time, then subsequent interpolation times successively sample intermediate states.

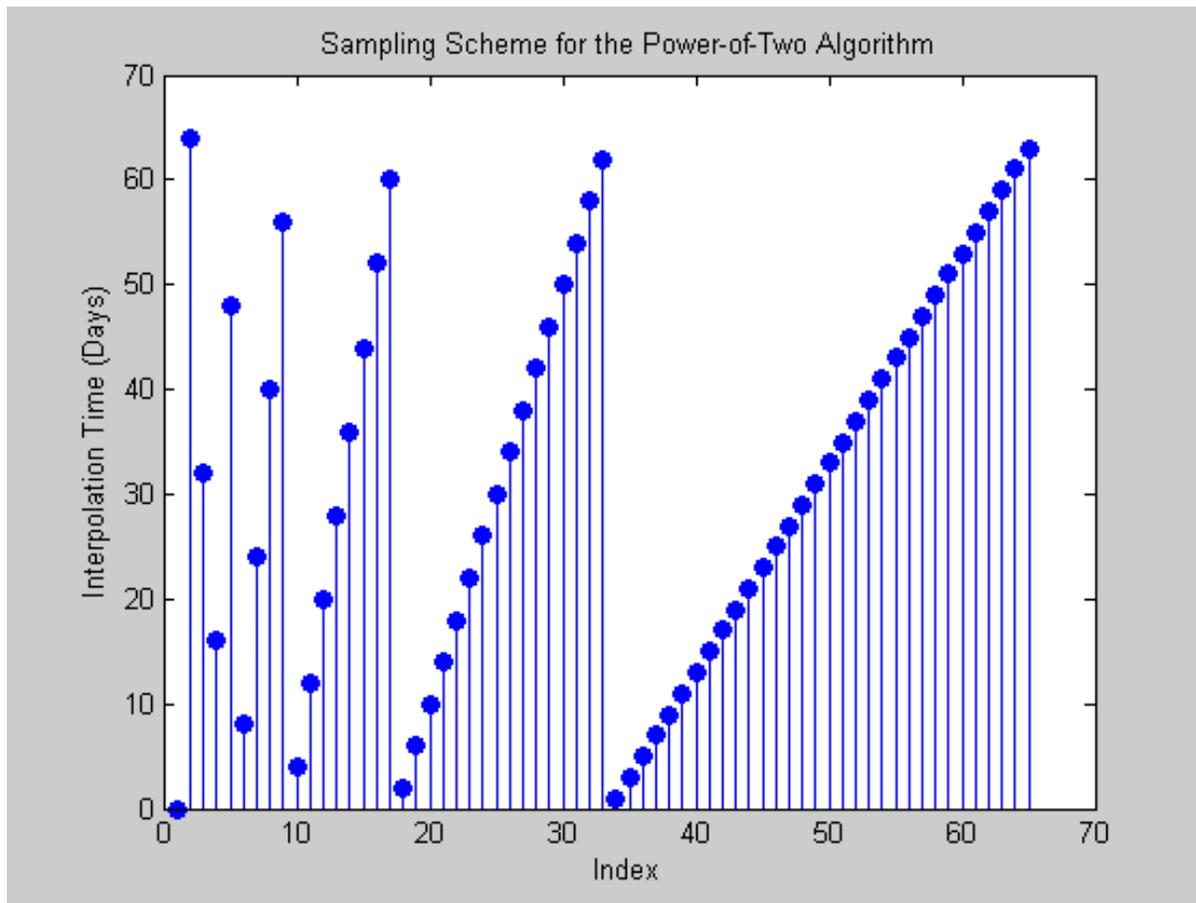
```
T          = 64;                % Terminal time in 64 days (periods)
times     = (1:T)';           % Sample times re-arranged by the bridge
t         = NaN(length(times) + 1, 1); % Pre-allocate the interpolation times
t(1)      = obj.StartTime;    % 1st time = last observation time
t(2)      = T;                % 2nd time = terminal time

delta     = T;
jMax      = 1;
iCount    = 3;

for k = 1:log2(T)
    i = delta / 2;
    for j = 1:jMax
        t(iCount) = times(i);
        i         = i + delta;
        iCount    = iCount + 1;
    end
    jMax = 2 * jMax;
    delta = delta / 2;
end
```

It is instructive to examine the sequence of interpolation times generated by this popular algorithm,

```
stem(1:length(t), t, 'filled')
xlabel('Index'), ylabel('Interpolation Time (Days)')
title('Sampling Scheme for the Power-of-Two Algorithm')
```



Notice that the first few samples are widely-separated in time and determine the course structure of the paths, while later samples are closely-spaced and progressively refine the detailed structure.

Now that the sequence of interpolation times has been generated, initialize a course time series grid to initiate the interpolation. The sampling process begins at the last observed time and state taken from the historical short rate series, and ends 64 days into the future at the expected value of the Vasicek model derived from the calibrated parameters.

```
average = obj.StartState * exp(-speed * T) + level * (1 - exp(-speed * T));
X       = [obj.StartState ; average];
```

Now generate 5 sample paths setting the Refine input flag to TRUE, indicating that each new interpolated state is inserted into the time series grid as it becomes available. Notice that interpolation is performed on a trial-by-trial basis, and since the input time series X has 5 trials (each page of the 3-D time series represents an independent trial), the interpolated output series Y has 5 pages too.

```
nTrials = 5;
randn('state', 0)

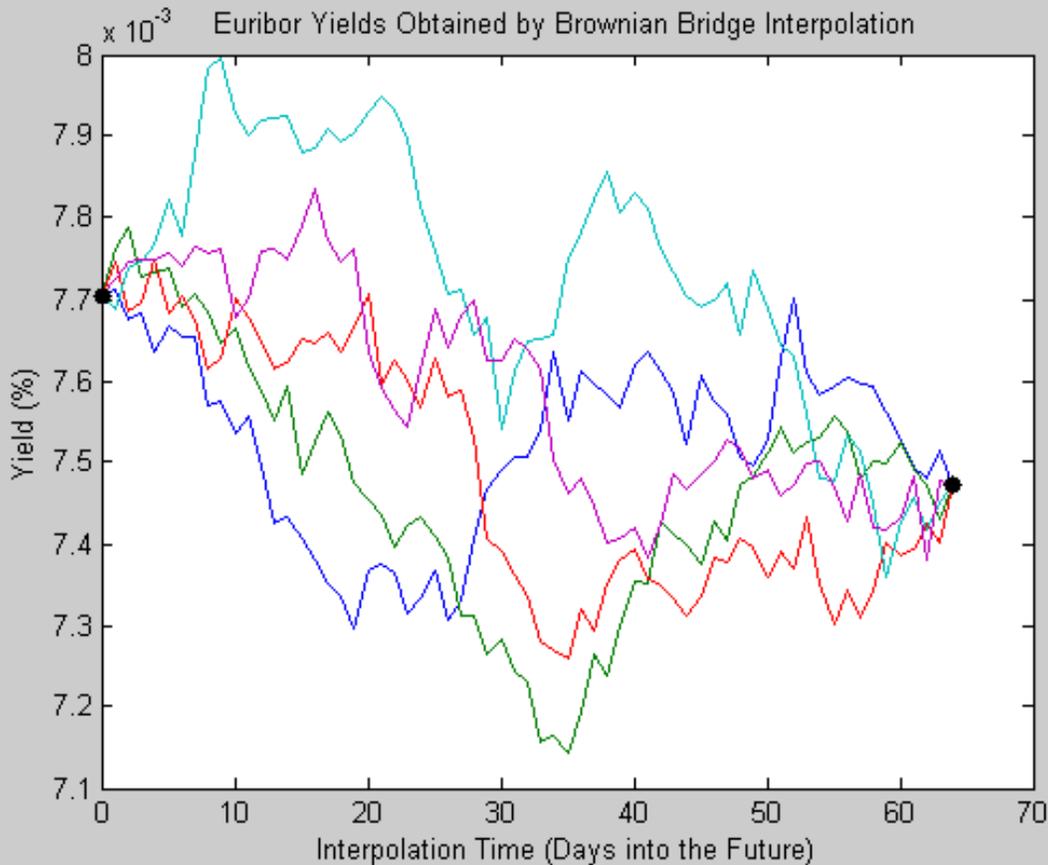
Y = obj.interpolate(t, X(:, :, ones(nTrials, 1)), 'Times', [obj.StartTime T], ...
                  'Refine', true);
```

Finally, plot the resulting sample paths. Since the interpolation times are not monotonically increasing, sort the times and re-order the corresponding short rates.

```

[t,i] = sort(t);
Y      = squeeze(Y);
Y      = Y(i,:);
plot(t, 100 * Y), hold('on')
plot(t([1 end]), 100 * Y([1 end],1), '. black', 'MarkerSize', 20)
xlabel('Interpolation Time (Days into the Future)')
ylabel('Yield (%)'), hold('off')
title ('Euribor Yields Obtained by Brownian Bridge Interpolation')

```



The short rates shown above represent alternative sample paths that share the same initial and terminal values, and illustrate a special, albeit simplistic, case of a broader sampling technique known as stratified sampling. For a more sophisticated example of stratified sampling, see *User-Specified Random Number Generation: Stratified Sampling*.

Although this simple example simulated a univariate Vasicek interest rate model, it is applicable to problems of any dimensionality.

End-of-Period Processes: Black-Scholes Option Pricing

All simulation and interpolation methods allow users to specify a sequence of functions, or background processes, to evaluate at the end of every sample time. These functions are specified as callable functions of time and state, and must return an updated state vector X ,

$$X_t = f(t, X_t)$$

If more than one processing function is specified, they must be entered as a cell array of functions, and are invoked in the order found in the cell array.

Notice that processing functions are not obligated to use time (t) or state (X), nor are they required to update or change the input state vector in any way. In fact, simulation/interpolation methods have no knowledge of any implementation details, and in this respect only adhere to a published interface.

At first glance, the presence of such processing functions may seem strange, yet they offer users a powerful modeling tool with applications limited only by the imagination. Such functions allow users to, for example, specify boundary conditions, accumulate statistics, plot graphs, and price path-dependent options.

Moreover, processing functions also allows users to avoid simulation outputs altogether.

As an example, consider pricing a European stock option by Monte Carlo simulation within a Black-Scholes-Merton framework. Assume the following characteristics:

- stock is currently trading at 100
- stock pays no dividends
- stock volatility is 50% per annum
- option strike price is 95
- option expires in 3 months
- risk-free rate is constant at 10% per annum

To solve this problem, model the evolution of the underlying stock by a univariate geometric Brownian motion (GBM) model with constant parameters,

$$dX_t = 0.1X_t dt + 0.5X_t dW_t$$

Furthermore, assume we are interested in simulating the stock price on a daily basis, and that each calendar month is composed of 21 trading days.

```

strike    = 95;           % exercise price
rate      = 0.1;         % annualized risk-free rate
sigma     = 0.5;         % annualized volatility
dt        = 1 / 252;     % time increment = 1 day = 1/252 years
nPeriods  = 63;          % # of simulation periods in 3 months = 63 trading days
T         = nPeriods * dt; % time to expiration = 3 months = 0.25 years

obj = gbm(rate, sigma, 'StartState', 100);

```

The goal is to simulate independent paths of daily stock prices, and calculate the price of European options as the risk-neutral sample average of the discounted terminal option payoff at expiration 63 days from now. This example calculates option prices by two approaches.

The first approach performs a traditional Monte Carlo simulation, explicitly requesting the simulated stock paths as an output. These output paths are then post-processed to price the options.

The second approach specifies an end-of-period processing function, accessible by time and state, which records the terminal stock price of each sample path. This processing function is implemented as a nested function with access to shared information.

Before simulation, first invoke the example file to access the end-of-period processing function,

```
clc
nTrials = 2000;           % # of independent trials (i.e., paths)
f       = blackScholesExample(nPeriods, nTrials)

f =
    BlackScholes: @blackScholesExample/saveTerminalStockPrice
    CallPrice: @blackScholesExample/getCallPrice
    PutPrice: @blackScholesExample/getPutPrice
```

Now simulate independent trials (sample paths). Notice that the simulated stock price paths are requested as an output and that an end-of-period processing function is specified,

```
randn('state', 0)

X = obj.simBySolution(nPeriods, 'DeltaTime', dt, 'nTrials', nTrials, ...
    'Processes', f.BlackScholes); clc
```

Now calculate the option prices directly from the simulated stock price paths. Since these are European options, the following code segment simply ignores all intermediate stock prices,

```
call = mean(exp(-rate * T) * max(squeeze(X(end, :, :)) - strike, 0))
put  = mean(exp(-rate * T) * max(strike - squeeze(X(end, :, :)), 0))
```

```
call =
    14.2566
put  =
    6.0388
```

Now price the options indirectly by invoking the nested functions found in the structure of function handles,

```
f.CallPrice(strike, rate)
f.PutPrice (strike, rate)
```

```
ans =
    14.2566
ans =
    6.0388
```

For reference, the theoretical call and put prices computed from the Black-Scholes option formulas are 13.6953 and 6.3497, respectively.

Although the same option prices are obtained, the latter approach works directly with the terminal stock prices of each sample path, and is therefore much more memory efficient. In fact, in this example there is no compelling reason to request an output at all.

Although this is a rather simplistic example, it illustrates several important features.

First, notice that the Black-Scholes example file consists of several utility functions nested within an outer, or primary, function. The outer function is invoked with problem-specific parameters needed to initialize and pre-allocate information shared by the nested functions and retained from one call to the next.

In other words, we have effectively created an object that encapsulates information and allows users to access and manipulate this information by invoking nested functions. In fact, the same effect could have been implemented by formally designing a class and creating an object with various methods, but a simple structure with nested functions offers a convenient alternative.

In this respect, the outer function does not calculate anything per se, but rather posts shared information. The output is not option prices, but rather a data structure whose fields encapsulate the details of the nested functions. The user is then able to access, update, and manipulate this shared information by invoking nested functions, at least one of which must be accessible by the (t, X) interface.

User-Specified Random Number Generation: Stratified Sampling

All simulation methods allow users to directly specify the random noise process used to generate the Brownian motion vector (Wiener process) which, in turn, drives the Monte Carlo simulation. Noise processes must also be specified as callable functions of time and state,

$$z_t = Z(t, X_t)$$

Similar to antithetic sampling, *stratified sampling* is also a variance reduction technique, yet operates by constraining a proportion of sample paths to specific subsets, or *strata*, of the sample space rather than inducing negative dependence between paired samples.

Specifically, this example specifies a noise function to stratify the terminal value of a univariate equity price series. Starting from known initial conditions, the function first stratifies the terminal value of a standard Brownian motion, then samples the process from beginning to end by drawing conditional Gaussian samples via a Brownian bridge.

The stratification assumes that each path is associated with a single stratified terminal value such that the number of paths is equal to the number strata, a technique referred as *proportional sampling*. This example is similar to that found in *The Brownian Bridge & Stochastic Interpolation*, yet more sophisticated.

Since stratified sampling requires knowledge of the future, it also requires more sophisticated time synchronization. Specifically, the example function requires knowledge of the entire sequence of sample times.

The function implements proportional sampling by partitioning the unit interval into bins of equal

probability by first drawing a random number uniformly distributed in each bin. These stratified uniform draws are then transformed by the inverse cumulative distribution function of a standard $N(0,1)$ Gaussian distribution. Finally, the resulting stratified Gaussian draws are scaled by the square root of the terminal time to stratify the terminal value of the Brownian motion.

Notice that the noise function above does **not** return the actual Brownian paths, but rather the Gaussian draws $Z(t,X)$ that drive it. Since the simulation method will scale $Z(t,X)$ by the square root of the current time increment dt , the function divides by the square root of dt .

To introduce the sampling technique, suppose we wish to stratify the terminal value of a univariate, zero-drift, unit-variance-rate Brownian motion (BM) model,

$$dX_t = dW_t$$

Furthermore, assume we are interested in simulating 10 paths of the process on a daily basis over a 3 month period, and that each calendar month and year is composed of 21 and 252 trading days, respectively,

```
clc, randn('state', 10), rand('twister', 0)

dt      = 1 / 252;           % time increment = 1 day = 1/252 years
nPeriods = 63;              % # of simulation periods in 3 months = 63 trading days
T       = nPeriods * dt;    % time to expiration = 3 months = 0.25 years
nPaths  = 10;               % # of simulated paths

obj      = bm(0, 1, 'StartState', 0);
sampleTimes = cumsum([obj.StartTime ; dt(ones(nPeriods,1))]);
z       = stratifiedExample(nPaths, sampleTimes)
```

```
z =
    @stratifiedExample/stratifiedSampling
```

Now simulate the standard Brownian paths by explicitly passing the stratified sampling function to the simulation method, and re-order the output sample paths for convenience,

```
X = obj.simulate(nPeriods, 'DeltaTime', dt, 'nTrials', nPaths, 'Z', z);
X = squeeze(X); % Re-order the 3-D output to a 2-D equivalent array.
```

To verify the stratification, re-create the uniform draws with proportional sampling, transform them to obtain the terminal values of standard Brownian motion, and plot the terminal values and output paths on the same figure,

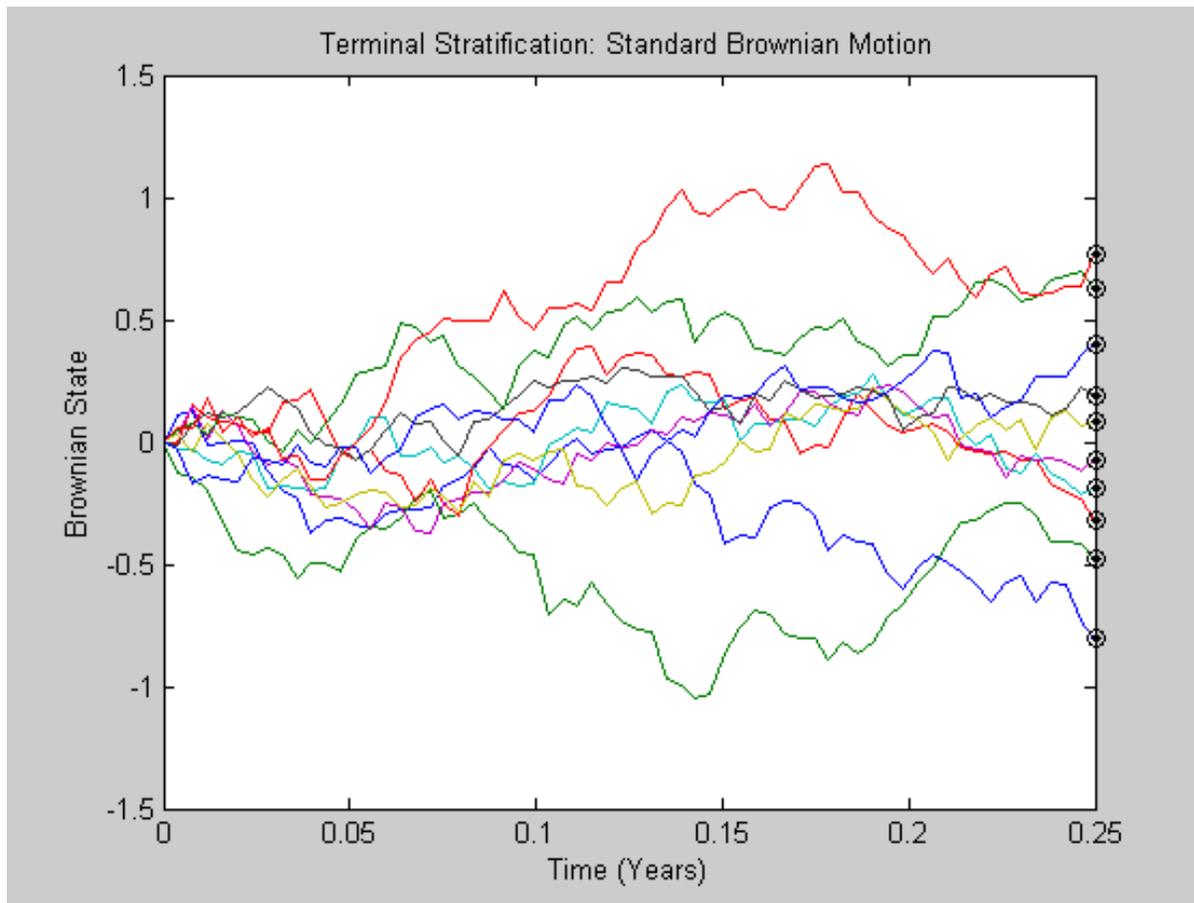
```
rand('twister', 0) % Use the same initial state.
U = ((1:nPaths)' - 1 + rand(nPaths,1))/nPaths; % Stratified uniforms in each bin.
WT = norminv(U) * sqrt(T); % Stratified Brownian motion.

plot(sampleTimes, X), hold('on')
xlabel('Time (Years)'), ylabel('Brownian State')
```

```

title('Terminal Stratification: Standard Brownian Motion')
plot(T, WT, '. black', T, WT, 'o black')
hold('off')

```



Notice that the last value of each sample path (i.e., the last row of the output array X) exactly coincides with the corresponding element of the stratified terminal value of the Brownian motion. This occurs because the simulated model and the noise generation function both represent the same standard Brownian motion.

However, the same stratified sampling function may also be used to stratify the terminal price of constant-parameter geometric Brownian motion models. In fact, the stratified sampling function may be used to stratify the terminal value of any constant-parameter model driven by Brownian motion, provided the model's terminal value is a monotonic transformation of the terminal value of the Brownian motion.

To illustrate this, assume we wish to simulate risk-neutral sample paths of the FTSE 100 index using a geometric Brownian motion (GBM) model with constant parameters,

$$dX_t = rX_t dt + \sigma X_t dW_t$$

in which Euribor yields represent the risk-free rate of return. Furthermore, assume we wish to annualize the relevant information derived from the daily data, and that each calendar year is composed of 252 trading days.

```

returns = price2ret(SDE_Data.UK);           % daily log returns of FTSE 100

```

```

sigma = std(returns) * sqrt(252); % annualized volatility
rate = SDE_Data.Euribor3M;
rate = mean(360 * log(1 + rate)); % continuously-compounded, annual yield

```

Now create the GBM model assuming the FTSE 100 starts at 100, determine the sample times, and simulate the price paths,

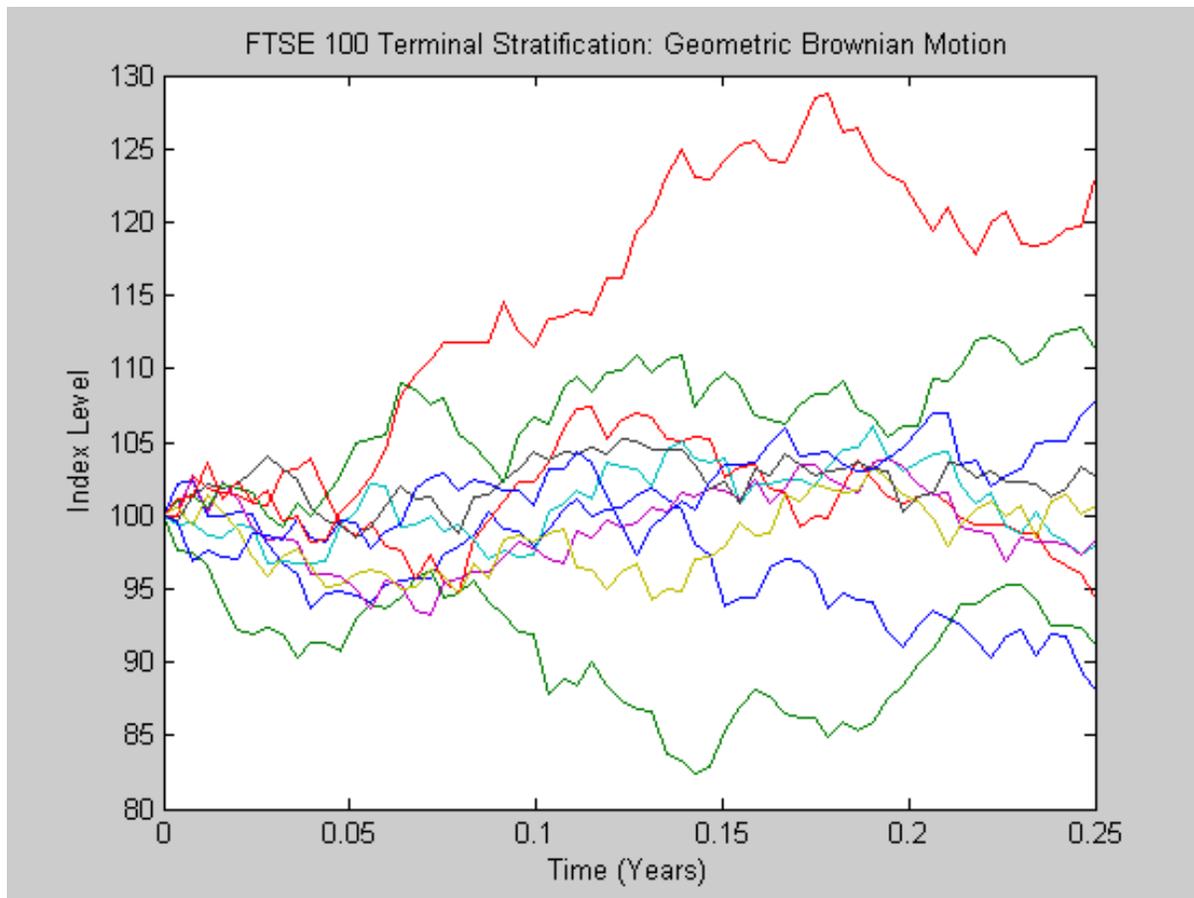
```

obj = gbm(rate, sigma, 'StartState', 100);
nSteps = 1;
sampleTimes = cumsum([obj.StartTime ; dt(ones(nPeriods * nSteps,1))/nSteps]);
z = stratifiedExample(nPaths, sampleTimes);

randn('state', 10), rand('twister', 0)
[Y, Times] = obj.simBySolution(nPeriods, 'nTrials', nPaths, ...
                             'DeltaTime', dt, 'nSteps', nSteps, 'Z', z);
Y = squeeze(Y); % Re-order the 3-D output to a 2-D equivalent array.

figure, plot(Times, Y)
xlabel('Time (Years)'), ylabel('Index Level')
title('FTSE 100 Terminal Stratification: Geometric Brownian Motion')

```



Although the terminal value of the Brownian motion shown in the first graph is normally-distributed, and the terminal price in the second graph is lognormally-distributed, notice the similarity between the corresponding paths of each graph.

As in the antithetic sampling example, this example could avoid outputs entirely provided the user specifies an end-of-period processing function to query the terminal state, thereby dramatically reducing memory usage.

Pricing American Basket Options Using Copulas & Extreme Value Theory

This is an advanced example, motivated by a series of related client requests, designed to illustrate some of the more salient features of the SDE architecture, including

- Customized random number generation functions, including a comparison of Brownian motion and Brownian copulas
- Customized end-of-period processing functions to form the price of an equity portfolio (basket) and retain that price series for subsequent option valuation
- Piecewise probability distributions and extreme value theory
- Early exercise of American options on baskets of equity indices, based on the method of Longstaff & Schwartz.

In recent months several corporate clients, facing increased exposure to the volatility of international equity markets, have inquired about the ability to model and price an American put option to protect the value their global portfolio.

Furthermore, several clients have also recently requested additional information related to modeling the fat tails of individual asset returns using Extreme Value Theory (EVT), and aggregating the joint distribution of these disparate asset returns with a Brownian copula. Ideally, they would also like to examine the impact of the resulting joint distribution on the value of the put as compared to a traditional multi-variate Brownian motion (i.e., a correlated Gaussian vector scaled by the square root of time).

To address these related requests, this example compares alternative implementations of a separable multivariate geometric Brownian motion (GBM) process often referred to as a *multi-dimensional market model*, and will do so by simulating risk-neutral sample paths of an equity index portfolio driven by correlated random draws derived from a Gaussian distribution, a Gaussian copula, and a t copula.

The risk-neutral market model we want to simulate is

$$dX_t = rX_t dt + \sigma X_t dW_t$$

in which the risk-free rate is assumed constant over the life of the option. Notice that any dividend yields are ignored to simplify the model and data collection.

Since this is a separable multi-variate model, the risk-free return, r , is a diagonal matrix in which the same riskless return is applied to all indices.

In contrast, the specification of the exposure matrix, σ , depends upon whether the driving source of uncertainty is modeled directly as a Brownian motion (i.e., correlated Gaussian random numbers implicitly mapped to Gaussian margins) or derived from a Brownian copula (i.e., correlated Gaussian or t random numbers explicitly mapped to semi-parametric margins).

Overview of the Modeling Framework

The ultimate objective of this analysis is a comparison of basket option prices derived from various noise processes. The first noise process is a traditional Brownian motion model in which the price (i.e., level) of

each individual asset (i.e., equity index) is driven by correlated Gaussian random draws.

As alternatives, the Brownian motion benchmark is compared to noise processes driven by Gaussian and Student's t copulas, referred to collectively as a *Brownian copula*. These copula draws produce dependent random variables, which are then transformed to individual variables (i.e., the margins) by a semi-parametric probability distribution with generalized Pareto tails tailored to the historical record of each index.

A copula is a multi-variate cumulative distribution function (CDF) with uniformly-distributed margins. Although the theoretical foundations were established decades ago, copulas have experienced a tremendous surge in popularity over the last few years, primarily as a technique for modeling non-Gaussian portfolio risks.

Although numerous families exist, all copulas represent a statistical device for modeling the dependence structure between 2 or more random variables. In addition, important statistics, such as *rank correlation* and *tail dependence*, are properties of a given copula and are unchanged by monotonic transforms of its margins.

Since the CDF and inverse CDF (i.e., quantile function) of univariate distributions are both monotonic transforms, a copula provides a very convenient technique to simulate dependent random variables with dissimilar and arbitrarily-distributed margins. Moreover, since a copula defines a given dependence structure irrespective of its margins, calibration is typically much easier than estimating the joint distribution function.

Once sample paths have been simulated, the options are priced by the least squares regression method of Longstaff & Schwartz (see *Valuing American Options by Simulation: A Simple Least-Squares Approach*, The Review of Financial Studies, Spring 2001). This approach uses least squares to estimate the expected payoff of an option if not immediately exercised, doing so by regressing the discounted option cash flows received in the future on the current price of the underlier associated with all in-the-money sample paths. The continuation function is estimated by a third order polynomial in which all cash flows and prices involved in the regression are normalized by the option strike price improve numerical stability.

To illustrate the modeling framework, we first characterize the distribution of each margin. Although the distribution of each index return series may be characterized parametrically, it is often useful to fit a semi-parametric approach using a piecewise distribution with generalize Pareto tails, which makes use of Extreme Value Theory to better characterize the behavior in each tail.

Extreme Value Theory & Piecewise Probability Distributions

In recent years, numerous financial clients have requested additional functionality related to the use of *Extreme Value Theory*, a statistical tool for modeling the fat-tailed behavior of financial data such as asset returns and insurance losses.

In response, 2 univariate probability distributions are now supported by the Statistics Toolbox:

- Generalized Extreme Value (GEV) distribution, which lends itself to a modeling technique known as the *block maxima or minima* method. In this approach, an historical dataset is divided into a set of sub-intervals, or blocks, and the largest or smallest observation in each block is recorded and fitted to a GEV distribution.
- Generalized Pareto (GP) distribution, which lends itself to a modeling technique known as the *distribution of exceedances* or *peaks over threshold* method. In this approach, an historical dataset is sorted, and the amount by which those observations which exceed a specified threshold is fitted

to a GP distribution.

Since the latter approach is more popular in risk management applications, the following analysis highlights the Pareto distribution.

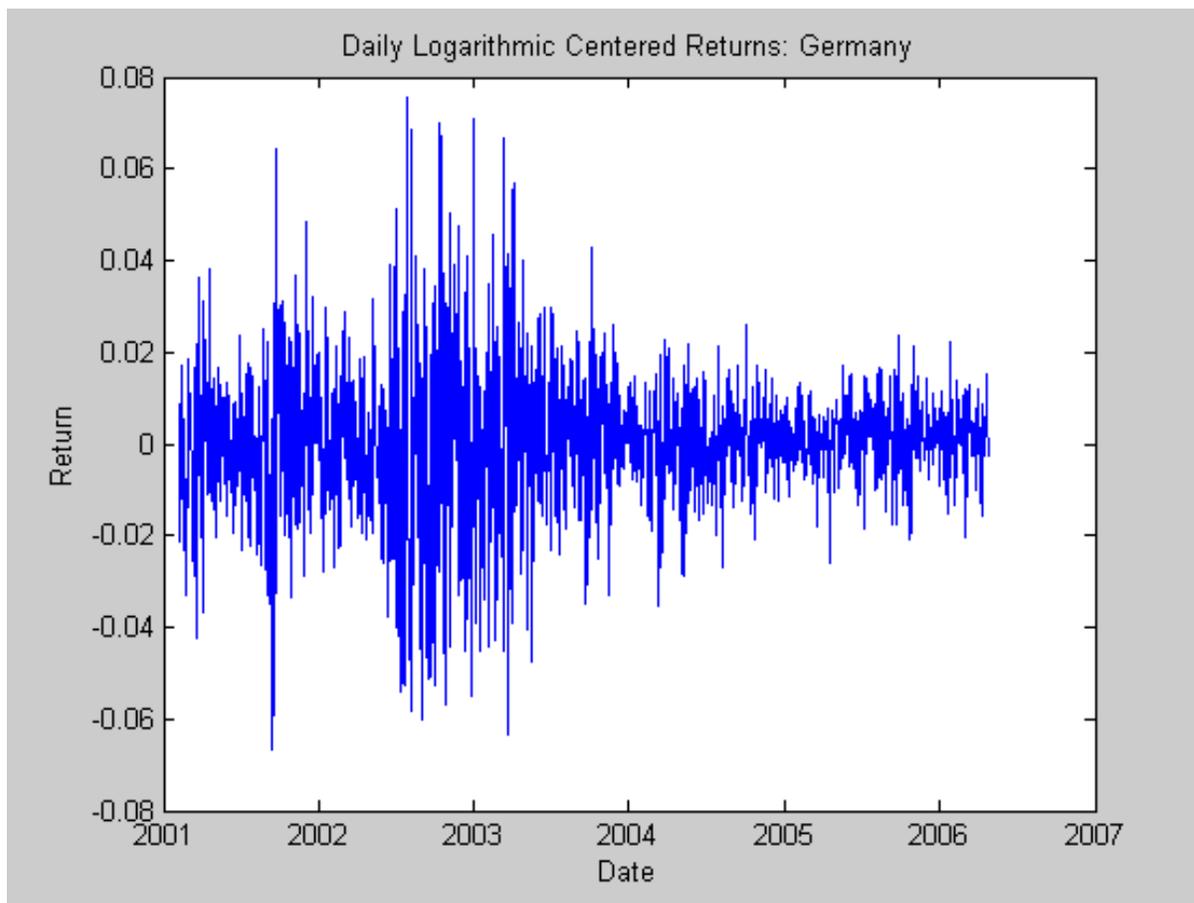
In particular, suppose we wish to provide a complete statistical description of the probability distribution of daily asset returns of any one of the equity indices. Furthermore, assume that this description is provided by a piecewise semi-parametric distribution in which the asymptotic behavior in each tail is characterized by a generalized Pareto distribution.

Ultimately, the copula will be used to generate random numbers to drive the simulation. The CDF and inverse CDF transforms will capture the volatility of simulated returns as part of the diffusion term of the SDE, but the mean return of each index is governed by the riskless rate and incorporated in the drift term of the SDE. For this reason, the following code segment centers the returns (i.e., extracts the mean) of each index.

Since the following analysis applies extreme value theory to characterize the distribution of each individual equity index return series, it is helpful to examine the details for a particular country. The code segment below can be changed to examine the details for any country.

```
returns = price2ret(prices); % Logarithmic returns.
returns = bsxfun(@minus, returns, mean(returns)); % Center the returns.
nIndices = size(returns,2); % # of indices.
index = strmatch('Germany', countries); if isempty(index), index = 1; end

figure, plot(dates(2:end), returns(:,index)), datetick('x')
xlabel('Date'), ylabel('Return')
title(['Daily Logarithmic Centered Returns: ' countries{index}])
```



Given the centered returns shown above, estimate the empirical, or non-parametric, CDF of each index with a Gaussian kernel. This smooths the CDF estimates, eliminating the staircase pattern of unsmoothed sample CDFs. Although non-parametric kernel CDF estimates are well suited for the interior of the distribution where most of the data is found, they tend to perform poorly when applied to the upper and lower tails. To better estimate the tails of the distribution, apply EVT to those returns that fall in each tail.

Specifically, find upper and lower thresholds such that 10% of the returns is reserved for each tail. Then fit the amount by which those extreme returns in each tail fall beyond the associated threshold to a parametric GP distribution by maximum likelihood.

The following code segment creates objects of type **paretotails**, one such object for each index return series. These Pareto tail objects encapsulate the estimates of the parametric GP lower tail, the non-parametric kernel-smoothed interior, and the parametric GP upper tail to construct a composite semi-parametric CDF for each index.

The resulting piecewise distribution object allows interpolation within the interior of the CDF and extrapolation (function evaluation) in each tail. Extrapolation is very desirable, allowing estimation of quantiles outside the historical record, and is invaluable for risk management applications.

Moreover, Pareto tail objects also provide methods to evaluate the CDF and inverse CDF (quantile function), and to query the cumulative probabilities and quantiles of the boundaries between each segment of the piecewise distribution.

Also, notice that collections of Pareto tail objects are stored in cell arrays, high-level MATLAB data containers designed to store disparate data types.

```

tailFraction = 0.1;           % decimal fraction allocated to each tail
OBJ          = cell(nIndices,1); % cell array of Pareto tail objects

for i = 1:nIndices
    OBJ{i} = paretotails(returns(:,i), tailFraction, 1 - tailFraction, 'kernel');
end

```

Having estimated the three distinct regions of the piecewise distribution, graphically concatenate and display the result. Again, note that the lower and upper tail regions, displayed in red and blue, respectively, are suitable for extrapolation, while the kernel-smoothed interior, in black, is suitable for interpolation.

The code below calls the CDF and inverse CDF methods of the Pareto tails object of interest with data other than that upon which the fit is based. Specifically, the referenced methods have access to the fitted state, and are now invoked to select and analyze specific regions of the probability curve, acting as a powerful data filtering mechanism.

For reference, the plot also includes a zero-mean Gaussian CDF of the same standard deviation. In some sense, the variation in options prices may be thought of as the extent to which the distribution of each asset differs from this normal curve.

```

figure, hold('on'), grid('on')

minProbability = OBJ{index}.cdf((min(returns(:,index))));
maxProbability = OBJ{index}.cdf((max(returns(:,index))));

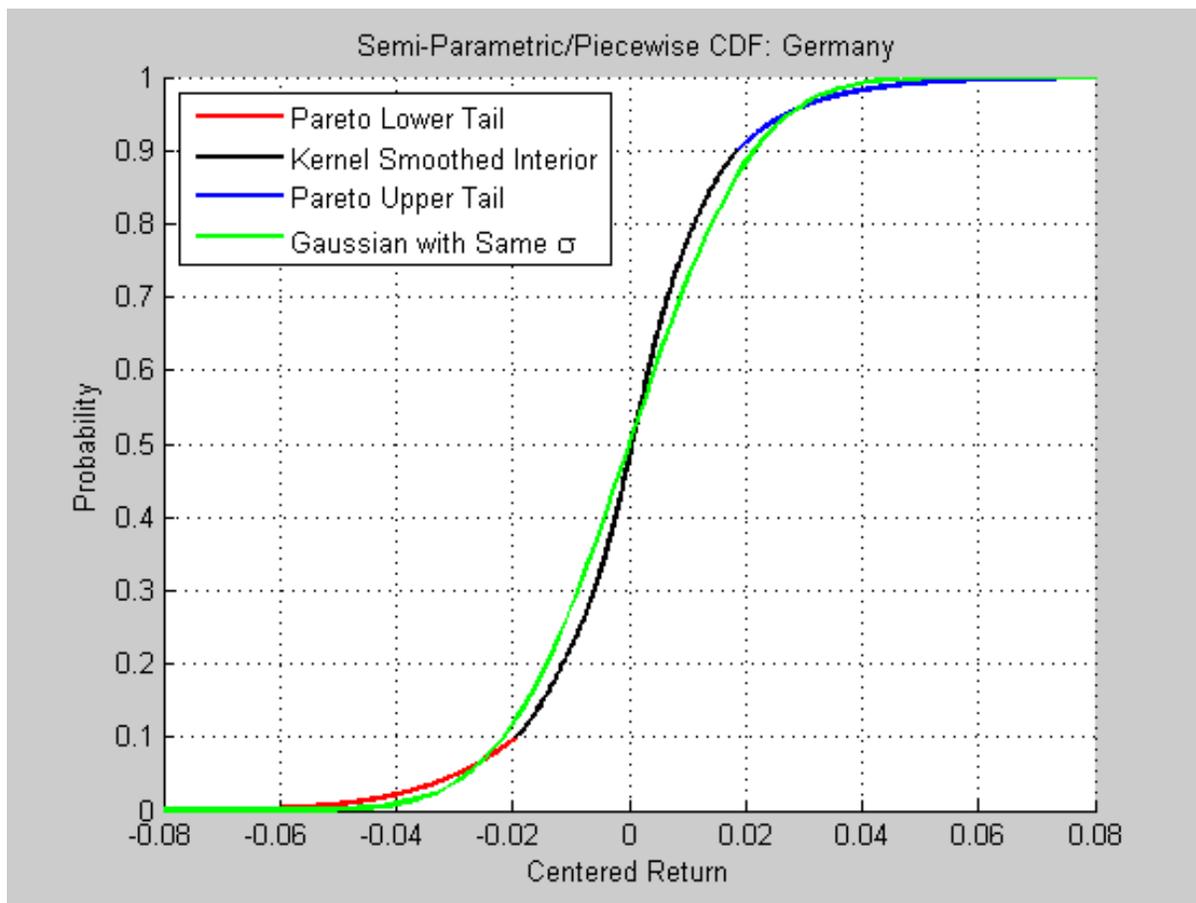
pLowerTail = linspace(minProbability, tailFraction, 200); % lower tail
pUpperTail = linspace(1 - tailFraction, maxProbability, 200); % upper tail
pInterior = linspace(tailFraction, 1 - tailFraction, 200); % interior

plot(OBJ{index}.icdf(pLowerTail), pLowerTail, 'red', 'LineWidth', 2)
plot(OBJ{index}.icdf(pInterior), pInterior, 'black', 'LineWidth', 2)
plot(OBJ{index}.icdf(pUpperTail), pUpperTail, 'blue', 'LineWidth', 2)

limits = axis; x = linspace(limits(1), limits(2));
plot(x, normcdf(x, 0, std(returns(:,index))), 'green', 'LineWidth', 2)

xlabel('Centered Return'), ylabel('Probability')
title(['Semi-Parametric/Piecewise CDF: ' countries{index}])
legend({'Pareto Lower Tail' 'Kernel Smoothed Interior' ...
       'Pareto Upper Tail' 'Gaussian with Same \sigma'}, 'Location', 'NorthWest')

```



Copula Calibration

The Statistics Toolbox includes functionality specifically related to the calibration and simulation of Gaussian and t copulas.

Given the daily index returns from above, now estimate the parameters of the Gaussian and t copulas using the Statistics Toolbox function `copulafit`. Since a t copula becomes a Gaussian copula as the scalar degrees of freedom parameter (DoF) becomes infinitely large, the two copulas are really of the same family, and therefore share a linear correlation matrix as a fundamental parameter.

Although calibration of the linear correlation matrix of a Gaussian copula is straightforward, the calibration of a t copula is more difficult. For this reason, the Statistics Toolbox offers 2 techniques to calibrate a t copula.

The first technique performs maximum likelihood estimation (MLE) in a two-step process. The inner step maximizes the log-likelihood with respect to the linear correlation matrix, given a fixed value for the degrees of freedom. That conditional maximization is placed within a 1-D maximization with respect to the degrees of freedom, thus maximizing the log-likelihood over all parameters. The function being maximized in this outer step is known as the profile log-likelihood for the degrees of freedom.

The second technique is derived by differentiating the log-likelihood function with respect to the linear correlation matrix, assuming the degrees of freedom is a fixed constant. The resulting expression is a non-linear equation that can be solved iteratively for the correlation matrix. This technique approximates the profile log-likelihood for the degrees of freedom parameter for large sample sizes. This technique is usually significantly faster than the true maximum likelihood technique outlined above, however, it should be used with caution because the estimates and confidence limits may not be accurate for small or

moderate sample sizes.

The code segment below first transforms the daily centered returns to uniform variates by the piecewise, semi-parametric CDFs derived above, then fits the Gaussian and t copulas to the transformed data. When the uniform variates are transformed by the empirical CDF of each margin, the calibration method is often known as canonical maximum likelihood (CML).

```

U = zeros(size(returns));

for i = 1:nIndices
    U(:,i) = OBJ{i}.cdf(returns(:,i));    % transform each margin to uniform
end

options      = statset('Display', 'off', 'TolX', 1e-4);
[rhoT, DoF] = copulafit('t', U, 'Method', 'ApproximateML', 'Options', options);
rhoG        = copulafit('Gaussian', U); clc

```

If we examine the estimated correlation matrices, we see that they are quite similar but not identical.

```
corrcoef(returns)    % linear correlation matrix of daily returns
```

```

ans =
    1.0000    0.4813    0.5058    0.1854    0.4573    0.6526
    0.4813    1.0000    0.8485    0.2261    0.8575    0.5102
    0.5058    0.8485    1.0000    0.2001    0.7650    0.6136
    0.1854    0.2261    0.2001    1.0000    0.2295    0.1439
    0.4573    0.8575    0.7650    0.2295    1.0000    0.4617
    0.6526    0.5102    0.6136    0.1439    0.4617    1.0000

```

```
rhoG                % linear correlation matrix of the optimized Gaussian copula
```

```

rhoG =
    1.0000    0.4745    0.5018    0.1857    0.4721    0.6622
    0.4745    1.0000    0.8606    0.2393    0.8459    0.4912
    0.5018    0.8606    1.0000    0.2126    0.7608    0.5811
    0.1857    0.2393    0.2126    1.0000    0.2396    0.1494
    0.4721    0.8459    0.7608    0.2396    1.0000    0.4518
    0.6622    0.4912    0.5811    0.1494    0.4518    1.0000

```

```
rhoT                % linear correlation matrix of the optimized t copula
```

```

rhoT =
    1.0000    0.4671    0.4858    0.1907    0.4734    0.6521
    0.4671    1.0000    0.8871    0.2567    0.8500    0.5122
    0.4858    0.8871    1.0000    0.2326    0.7723    0.5877

```

0.1907	0.2567	0.2326	1.0000	0.2503	0.1539
0.4734	0.8500	0.7723	0.2503	1.0000	0.4769
0.6521	0.5122	0.5877	0.1539	0.4769	1.0000

Moreover, notice the relatively low degrees of freedom parameter obtained from the t copula calibration, indicating a significant departure from a Gaussian situation.

```
DoF          % scalar degrees of freedom parameter of the optimized t copula
```

```
DoF =
    4.8613
```

Copula Simulation

Now that the copula parameters have been estimated, simulate jointly-dependent equity index returns using the **copularnd** function of the Statistics Toolbox.

By extrapolating into the GP tails and interpolating into the smoothed interior, transform the uniform variates derived from **copularnd** to daily centered returns via the inverse CDF of each index. These simulated centered returns are consistent with those obtained from the historical dataset. The returns are assumed independent in time, but at any point in time possess the dependence and rank correlation induced by the given copula.

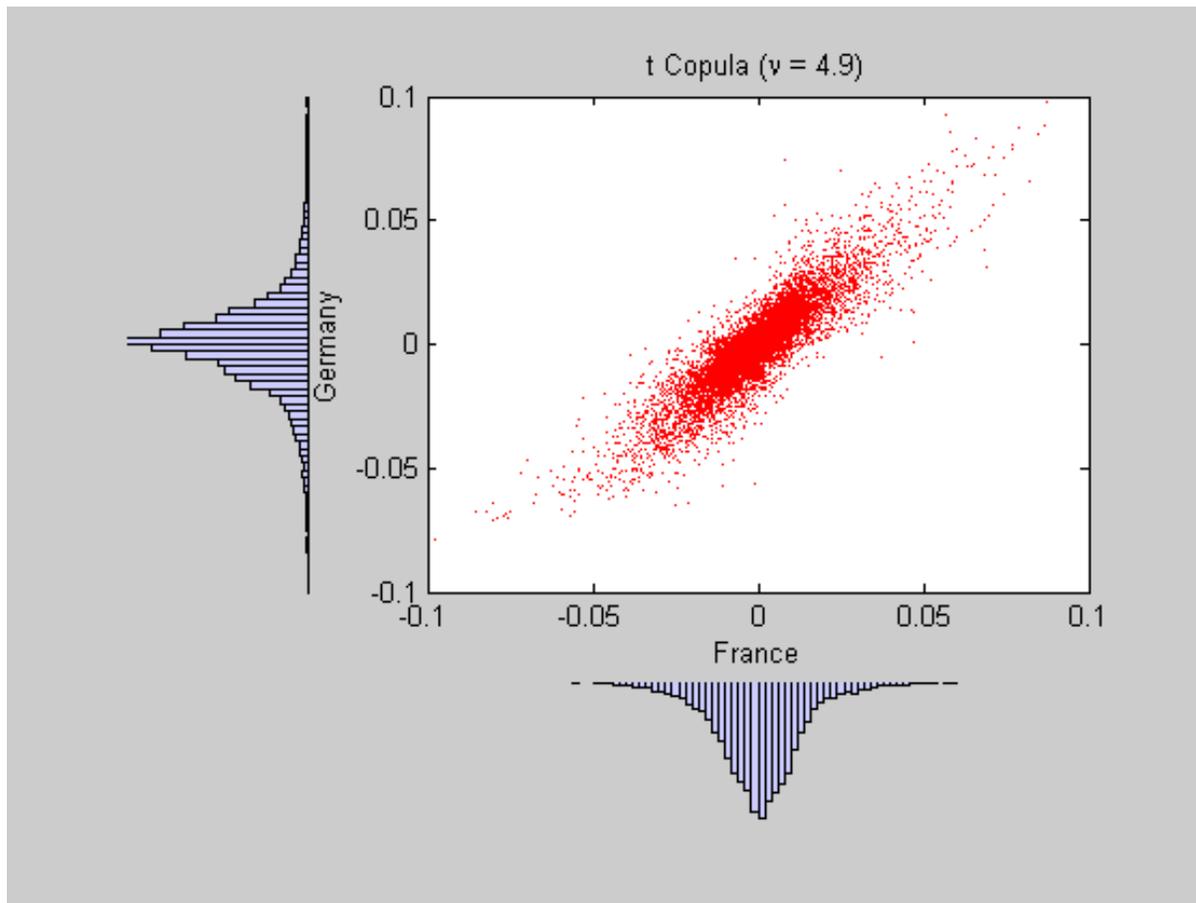
To illustrate the dependence structure, the following code segment simulates centered returns using the t copula, and plots a 2-D scatter plot with marginal histograms for the French CAC 40 and German DAX using the **scatterhist** function of the Statistics Toolbox (the French and German indices were chosen simply because they have the highest correlation of the available data).

```
nPoints = 10000;          % # of simulated observations
randn('state', 0), rand('twister', 0)

R = zeros(nPoints, nIndices); % pre-allocate simulated returns array
U = copularnd('t', rhoT, DoF, nPoints); % simulate U(0,1) from t copula

for j = 1:nIndices
    R(:,j) = OBJ{j}.icdf(U(:,j));
end

figure, h = scatterhist(R(:,2), R(:,3));
set(findobj(h(1), 'Type', 'line'), 'marker', '.', 'color', 'red', 'markerSize', 1)
y1 = get(h(1), 'ylim'); y3 = get(h(3), 'ylim');
set(h(1), 'xlim', [-.1 .1], 'ylim', [-.1 .1])
set(h(2), 'xlim', [-.1 .1])
set(h(3), 'ylim', [(y3(1) + (-0.1 - y1(1))) (y3(2) + (0.1 - y1(2)))]])
xlabel('France'), ylabel('Germany'), title(['t Copula (\nu = ' num2str(DoF,2) ')'])
```



Now simulate and plot centered returns using the Gaussian copula.

```

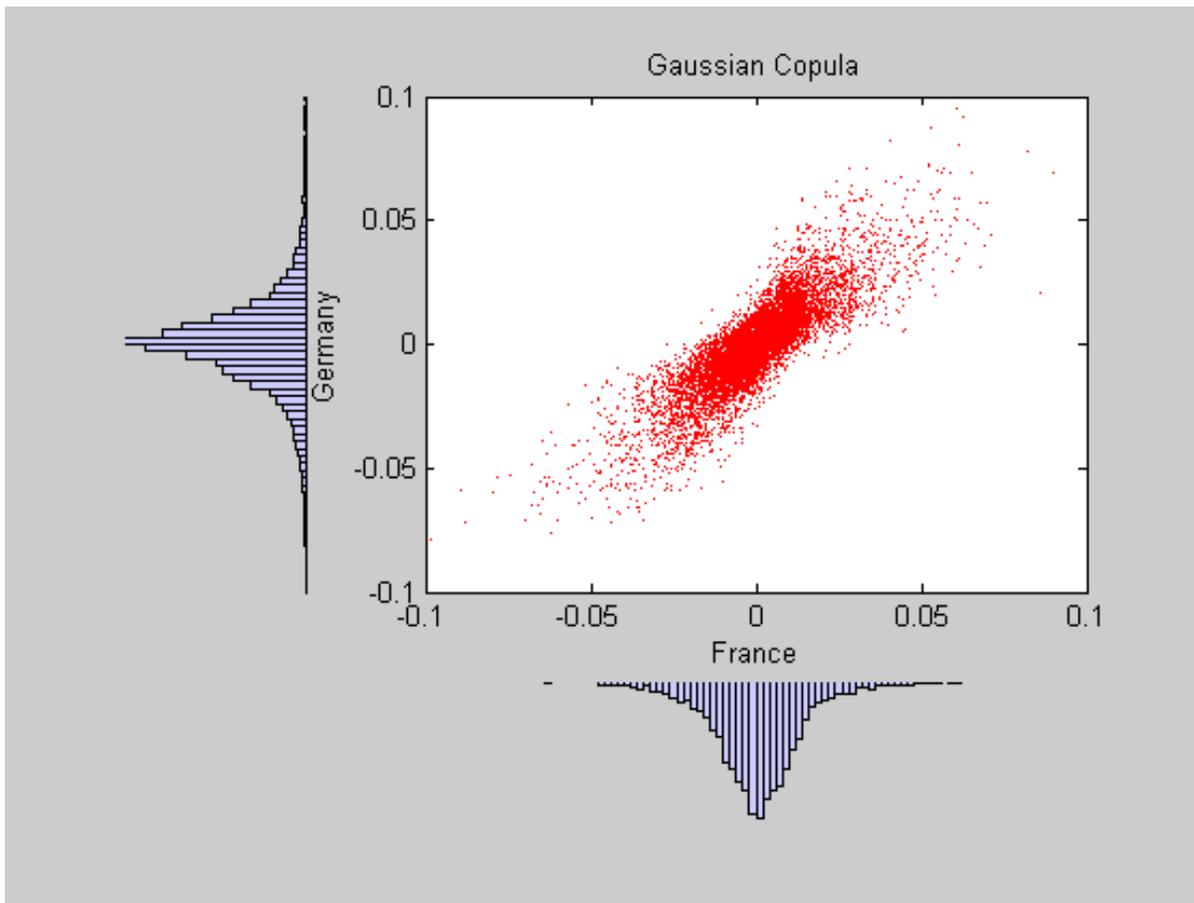
randn('state', 0), rand('twister', 0)

R = zeros(nPoints, nIndices);           % pre-allocate simulated returns array
U = copularnd('Gaussian', rhoG, nPoints); % simulate U(0,1) from Gaussian copula

for j = 1:nIndices
    R(:,j) = OBJ{j}.icdf(U(:,j));
end

figure, h = scatterhist(R(:,2), R(:,3));
set(findobj(h(1), 'Type', 'line'), 'marker', '.', 'color', 'red', 'markerSize', 1)
y1 = get(h(1), 'ylim'); y3 = get(h(3), 'ylim');
set(h(1), 'xlim', [-.1 .1], 'ylim', [-.1 .1])
set(h(2), 'xlim', [-.1 .1])
set(h(3), 'ylim', [(y3(1) + (-0.1 - y1(1))) (y3(2) + (0.1 - y1(2)))]))
xlabel('France'), ylabel('Germany'), title('Gaussian Copula')

```



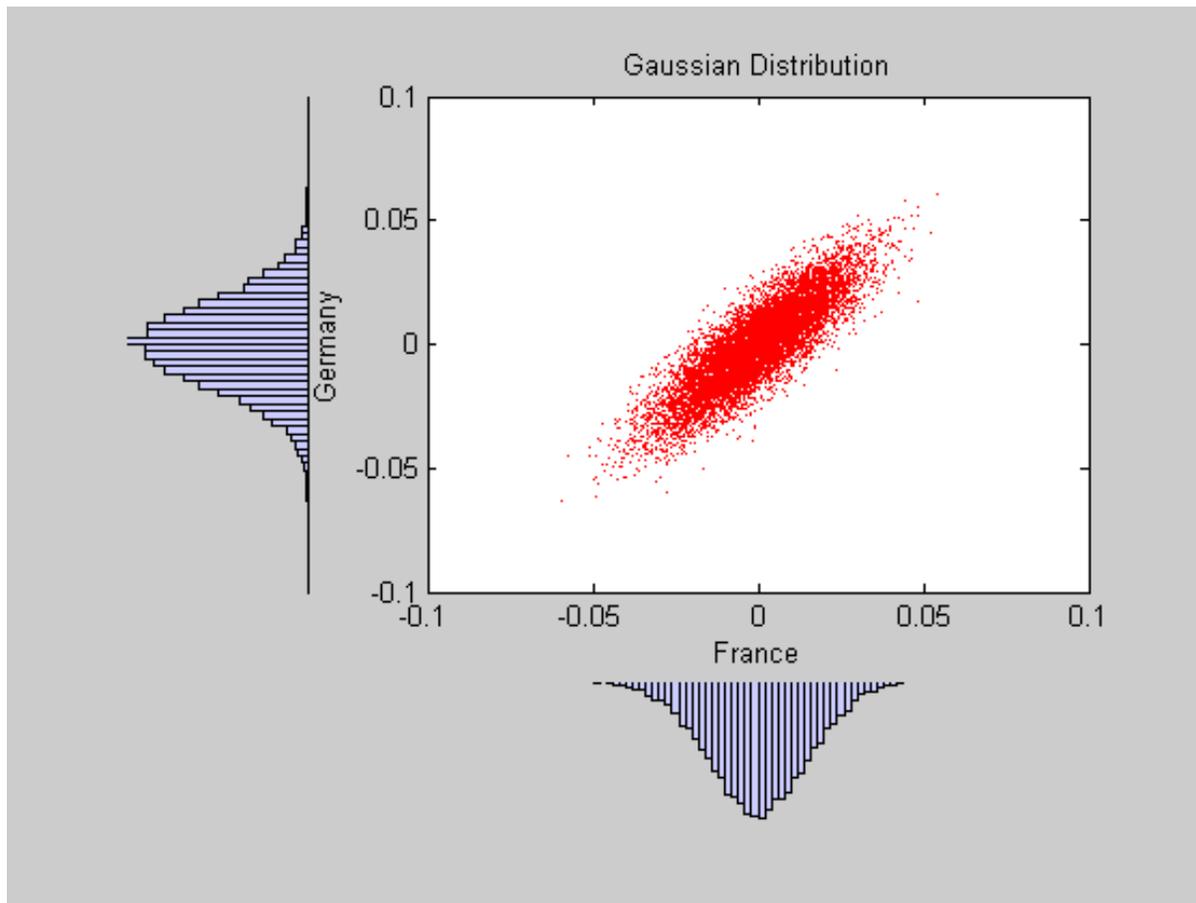
Now examine the 2 figures just created. In particular, notice the remarkable similarity between the miniature histograms on the corresponding axes of each figure. This similarity is no coincidence.

Since both copulas simulate uniform random variables, which are then transformed to daily centered returns by the inverse CDF of the piecewise distribution of each index, the simulated returns of any given index are identically distributed regardless of the copula. However, the scatter graph of each figure indicates the dependence structure associated with the given copula, and in contrast to the uni-variate margins shown in the histograms, the scatter graphs are quite distinct.

Once again, the copula defines a dependence structure irrespective of its margins, and therefore offers many appealing and convenient features not limited to calibration alone.

For reference, now simulate and plot centered returns using the Gaussian distribution which underlies the traditional Brownian motion model.

```
R = mvnrnd(zeros(1,nIndices), cov(returns), nPoints);
figure, h = scatterhist(R(:,2), R(:,3));
set(findobj(h(1), 'Type', 'line'), 'marker', '.', 'color', 'red', 'markerSize', 1)
y1 = get(h(1), 'ylim'); y3 = get(h(3), 'ylim');
set(h(1), 'xlim', [-.1 .1], 'ylim', [-.1 .1])
set(h(2), 'xlim', [-.1 .1])
set(h(3), 'ylim', [(y3(1) + (-0.1 - y1(1))) (y3(2) + (0.1 - y1(2))])
xlabel('France'), ylabel('Germany'), title('Gaussian Distribution')
```



American Option Pricing Using the Longstaff & Schwartz Approach

Now that the copulas have been calibrated, compare the price of an at-the-money American basket option derived from various approaches. To simplify the analysis, assume that all equity indices begin at 100, and that the basket portfolio holds a single unit, or share, of each index such that the value of the basket portfolio at any time is just the sum of the values of the individual indices.

In what follows, we simulate daily index levels and assume that the option may be exercised at the end of every day, which approximates the American option as a Bermudan option. We also assume the option expires in 3 months.

Furthermore, assume we wish to annualize the relevant information derived from the daily data illustrated above, and that each calendar year is composed of 252 trading days. Notice that the same results could also be obtained by working with unannualized (in this case, daily) centered returns and riskless rates, in which the time increment $dt = 1$.

Compute the basic data statistics needed as inputs to simulation methods.

```

clc
dt      = 1 / 252;           % time increment = 1 day = 1/252 years
yields  = SDE_Data.Euribor3M;
yields  = 360 * log(1 + yields); % continuously-compounded, annual yield
r       = mean(yields);     % historical 3M Euribor average
X       = repmat(100, nIndices, 1); % initial state vector
strike  = sum(X);           % initialize an at-the-money basket

```

```
nTrials = 200;      % # of independent trials (i.e., sample paths)
nPeriods = 63;     % # of simulation periods: 63/252 = 0.25 years = 3 months
```

Now create two separable (i.e., the riskless return and volatility exposure are both diagonal matrices) multi-dimensional market models.

While both are diagonal GBM models with identical risk-neutral returns, the first is driven by a correlated Brownian motion and explicitly specifies the sample linear correlation matrix of centered returns. This correlated Brownian motion process is then weighted by a diagonal matrix of annualized index volatilities or standard deviations.

As an alternative, the same model could be driven by an uncorrelated Brownian motion (i.e., *standard Brownian motion*) by specifying the correlation as an identity matrix or simply accepting the default. In this case, the exposure matrix `sigma` would be specified as the lower Cholesky factor of the index return covariance matrix. Since the copula-based approaches simulate dependent random numbers, the diagonal exposure form is chosen for consistency.

```
sigma      = std(returns) * sqrt(252);    % annualized volatility
correlation = corrcoef(returns);          % correlated Gaussian disturbances
GBM1       = gbm(diag(r(ones(1,nIndices))), diag(sigma), 'StartState', X, ...
                'Correlation', correlation)
```

```
GBM1 =
Class GBM: Generalized Geometric Brownian Motion
-----
Dimensions: State = 6, Brownian = 6
-----
StartTime: 0
StartState: 100 (6x1 double array)
Correlation: 6x6 double array
  Drift: drift rate function F(t,X(t))
  Diffusion: diffusion rate function G(t,X(t))
Simulation: simulation method/function simByEuler
  Return: 6x6 diagonal double array
  Sigma: 6x6 diagonal double array
```

Now create the second model driven by the Brownian copula with an identity `sigma` matrix. This may seem unusual, but this example highlights some of the flexibility of the architecture.

```
GBM2 = gbm(diag(r(ones(1,nIndices))), eye(nIndices), 'StartState', X)
```

```
GBM2 =
Class GBM: Generalized Geometric Brownian Motion
-----
Dimensions: State = 6, Brownian = 6
-----
StartTime: 0
StartState: 100 (6x1 double array)
```

```

Correlation: 6x6 diagonal double array
  Drift: drift rate function F(t,X(t))
  Diffusion: diffusion rate function G(t,X(t))
Simulation: simulation method/function simByEuler
  Return: 6x6 diagonal double array
  Sigma: 6x6 diagonal double array

```

When working with copulas, it is often convenient to allow the random number generator function $Z(t,X)$ to induce dependence (of which the traditional notion of linear correlation is a special case) with the copula, and to induce magnitude or scale of variation (similar to volatility or standard deviation) with the semi-parametric CDF and inverse CDF transforms.

This representation is conceptually similar to specifying `correlation` as a covariance matrix. Since the CDF and inverse CDF transforms of each index inherit the characteristics of historical returns, this also explains why the returns are now centered.

In what follows, statements such as

```
z = copulaExampleSub(returns * sqrt(252), nPeriods, 'Gaussian');
```

or

```
z = copulaExampleSub(returns * sqrt(252), nPeriods, 't');
```

fit the Gaussian and t copula dependence structures, respectively, and the semi-parametric margins to the centered returns scaled by the square root of the number of trading days per year (252). Notice that this scaling does **not** annualize the daily centered returns per se, but rather scales them such that the volatility remains consistent with the diagonal annualized exposure matrix `sigma` of the traditional Brownian motion model created above.

Now that the objects have been created, simulate independent trials of equity index prices over 3 calendar months using the default **simByEuler** method.

In this example, we specify an end-of-period processing function that accepts time followed by state (t,X) , and records the sample times and value of the portfolio as the single-unit weighted average of all indices. This function also shares this information with other functions designed to price American put and call options with a constant riskless rate using the least squares regression approach of Longstaff & Schwartz.

```
f = longstaffSchwartzExample(nPeriods, nTrials)
```

```

f =
  LongstaffSchwartz: @longstaffSchwartzExample/saveBasketPrices
  CallPrice: @longstaffSchwartzExample/getCallPrice
  PutPrice: @longstaffSchwartzExample/getPutPrice
  Prices: @longstaffSchwartzExample/getBasketPrices

```

Notice that no outputs are requested from the simulation methods; in fact the simulated prices of the indices underlying the basket option are unnecessary. Also, notice that call option prices are reported for convenience.

```

randn('state', 100), rand('twister', 100)

GBM1.simByEuler(nPeriods, 'nTrials' , nTrials, 'DeltaTime', dt, ...
               'Processes', f.LongstaffSchwartz);

BrownianMotionCallPrice = f.CallPrice(strike, r);
BrownianMotionPutPrice  = f.PutPrice (strike, r);

randn('state', 100), rand('twister', 100)

z = copulaExampleSub(returns * sqrt(252), nPeriods, 'Gaussian');
f = longstaffSchwartzExample(nPeriods, nTrials);

GBM2.simByEuler(nPeriods, 'nTrials' , nTrials, 'DeltaTime', dt, ...
               'Processes', f.LongstaffSchwartz, 'Z', z);

GaussianCopulaCallPrice = f.CallPrice(strike, r);
GaussianCopulaPutPrice  = f.PutPrice (strike, r);

```

Now repeat the copula simulation with the t copula dependence structure. Notice that we use the same model object for both copulas; only the random number generator and option pricing functions need to be re-initialized.

```

randn('state', 100), rand('twister', 100)

z = copulaExampleSub(returns * sqrt(252), nPeriods, 't');
f = longstaffSchwartzExample(nPeriods, nTrials);

GBM2.simByEuler(nPeriods, 'nTrials' , nTrials, 'DeltaTime', dt, ...
               'Processes', f.LongstaffSchwartz, 'Z', z);

tCopulaCallPrice = f.CallPrice(strike, r);
tCopulaPutPrice  = f.PutPrice (strike, r);

```

Finally, compare the American put and call option prices obtained from all models.

```

clc
disp(' ')
fprintf('          # of Monte Carlo Trials: %8d\n'      , nTrials)
fprintf('          # of Time Periods/Trial: %8d\n\n'   , nPeriods)
fprintf(' Brownian Motion American Call Basket Price: %8.4f\n' ,
BrownianMotionCallPrice)
fprintf(' Brownian Motion American Put  Basket Price: %8.4f\n\n',
BrownianMotionPutPrice)
fprintf(' Gaussian Copula American Call Basket Price: %8.4f\n' ,
GaussianCopulaCallPrice)
fprintf(' Gaussian Copula American Put  Basket Price: %8.4f\n\n',
GaussianCopulaPutPrice)
fprintf('          t Copula American Call Basket Price: %8.4f\n' , tCopulaCallPrice)

```

```
fprintf('          t Copula American Put Basket Price: %8.4f\n' , tCopulaPutPrice)
```

```

# of Monte Carlo Trials:      200
# of Time Periods/Trial:     63

Brownian Motion American Call Basket Price: 23.0427
Brownian Motion American Put Basket Price: 17.0683

Gaussian Copula American Call Basket Price: 22.2751
Gaussian Copula American Put Basket Price: 18.1843

t Copula American Call Basket Price: 24.0212
t Copula American Put Basket Price: 18.0952
```

Although instructive, the above analysis represents a relatively small-scale simulation. If the same analysis is repeated with 100,000 trials, the results are as follows:

```

# of Monte Carlo Trials:     100000
# of Time Periods/Trial:     63

Brownian Motion American Call Basket Price: 20.2214
Brownian Motion American Put Basket Price: 16.5355

Gaussian Copula American Call Basket Price: 20.6097
Gaussian Copula American Put Basket Price: 16.5539

t Copula American Call Basket Price: 21.1273
t Copula American Put Basket Price: 16.6873
```

Interestingly, the results agree quite closely. In particular, put option prices obtained from copulas exceed those of Brownian motion by less than 1%.

*Copyright 1999-2007 The MathWorks, Inc.
Published with MATLAB® 7.6*