Chapter 16

Stochastic Volatility

We have spent a good deal of time looking at vanilla and path-dependent options on QuantStart so far. We have created separate classes for random number generation and sampling from a standard normal distribution. We're now going to build on this by generating correlated time series paths.

Correlated asset paths crop up in many areas of quantitative finance and options pricing. In particular, the Heston Stochastic Volatility Model requires two correlated GBM asset paths as a basis for modelling volatility.

16.1 Motivation

Let's motivate the generation of correlated asset paths via the Heston Model. The original Black-Scholes model assumes that volatility, σ, is constant over the lifetime of the option, leading to the stochastic differential equation (SDE) for GBM:

$$dS_t = \mu S_t dt + \sigma S_t dW_t \tag{16.1}$$

The basic Heston model now replaces the constant σ coefficient with the square root of the instantaneous variance, $\sqrt{\nu_t}$. Thus the full model is given by:

$$dS_t = \mu S_t dt + \sqrt{\nu_t} S_t dW_t^S \tag{16.2}$$

$$d\nu_t = \kappa(\theta - \nu_t)dt + \xi\sqrt{\nu_t} dW_t^\nu \tag{16.3}$$

Where dW_t^S and dW_t^ν are Brownian motions with correlation ρ. Hence we have two correlated stochastic processes. In order to price path-dependent options in the Heston framework by Monte Carlo, it is necessary to generate these two asset paths.

16.2 Process for Correlated Path Generation

Rather than considering the case of two correlated assets, we will look at N seperate assets and then reduce the general procedure to $N = 2$ for the case of our Heston motivating example.

At each time step in the path generation we require N correlated random numbers, where ρ_{ij} denotes the correlation coefficient between the ith and jth asset, x_i will be the uncorrelated random number (which we will sample from the standard normal distribution), ϵ_i will be a correlated random number, used in the asset path generation and α_{ij} will be a matrix coefficient necessary for obtaining ϵ_i.

To calculate ϵ_i we use the following process *for each of the path generation time-steps*:

$$\epsilon_i = \sum_{k=1}^{i} \alpha_{ik} x_k, 1 \leq i \leq N \tag{16.4}$$

$$\sum_{k=1}^{i} \alpha_{ik}^2 = 1, 1 \leq i \leq N \tag{16.5}$$

$$\sum_{k=1}^{i} \alpha_{ik} \alpha_{jk} = \rho_{ij}, \forall j < i \tag{16.6}$$

Thankfully, for our Heston model, we have $N = 2$ and this reduces the above equation set to the far simpler relations:

$$\epsilon_1 = x_1 \tag{16.7}$$

$$\epsilon_2 = \rho x_1 + x_2 \sqrt{1 - \rho^2} \tag{16.8}$$

This motivates a potential C++ implementation. We already have the capability to generate paths of standard normal distributions. If we inherit a new class CorrelatedSND (SND for 'Standard Normal Distribution'), we can provide it with a correlation coefficient and an original time series of random standard normal variables draws to generate a new correlated asset path.

16.3 Cholesky Decomposition

It can be seen that the process used to generate N correlated ϵ_i values is in fact a matrix equation. It turns out that it is actually a Cholesky Decomposition, which we have discussed in the chapter on Numerical Linear Algebra. Thus a far more efficient implementation than I am constructing here would make use of an optimised matrix class and a pre-computed Cholesky decomposition matrix.

The reason for this link is that the correlation matrix, Σ, is symmetric positive definite. Thus it can be decomposed into $\Sigma = RR^*$, although R^*, the conjugate-transpose matrix simply reduces to the transpose in the case of real-valued entries, with R a lower-triangular matrix.

Hence it is possible to calculate the correlated random variable vector ϵ via:

$$\epsilon = \mathbf{Rx} \tag{16.9}$$

Where $\mathbf{x}$ is the vector of uncorrelated variables.

We will explore the Cholesky Decomposition as applied to multiple correlated asset paths later in this chapter.

16.4 C++ Implementation

As we pointed out above the procedure for obtaining the second path will involve calculating an uncorrelated set of standard normal draws, which are then recalculated via an inherited subclass to generate a new, correlated set of random variables. For this we will make use of statistics.h and statistics.cpp, which can be found in the chapter on Statistical Distributions.

Our next task is to write the header and source files for CorrelatedSND. The listing for correlated_snd.h follows:

```
#ifndef _CORRELATED_SND_H
#define _CORRELATED_SND_H

#include "statistics.h"

class CorrelatedSND : public StandardNormalDistribution {
 protected:
  double rho;
```

```
  const std::vector<double>* uncorr_draws;

  // Modify an uncorrelated set of distribution draws to be correlated
  virtual void correlation_calc(std::vector<double>& dist_draws);

 public:
  CorrelatedSND(const double _rho,
                const std::vector<double>* _uncorr_draws);
  virtual ~CorrelatedSND();

  // Obtain a sequence of correlated random draws from another set of SND
      draws
  virtual void random_draws(const std::vector<double>& uniform_draws,
                            std::vector<double>& dist_draws);
};

#endif
```

The class inherits from StandardNormalDistribution, provided in statistcs.h. We are adding two **protected** members, rho (the correlation coefficient) and uncorr_draws, a pointer to a const vector of doubles. We also create an additional virtual method, correlation_calc, that actually performs the correlation calculation. The only additional modification is to add the parameters, which will ultimately become stored as protected member data, to the constructor.

Next up is the source file, correlated_snd.cpp:

```
#ifndef _CORRELATED_SND_CPP
#define _CORRELATED_SND_CPP

#include "correlated_snd.h"
#include <iostream>
#include <cmath>

CorrelatedSND::CorrelatedSND(const double _rho,
                             const std::vector<double>* _uncorr_draws)
  : rho(_rho), uncorr_draws(_uncorr_draws) {}

CorrelatedSND::~CorrelatedSND() {}
```

```
// This carries out the actual correlation modification. It is easy to see
    that if
// rho = 0.0, then dist_draws is unmodified, whereas if rho = 1.0, then
    dist_draws
// is simply set equal to uncorr_draws. Thus with 0 < rho < 1 we have a
// weighted average of each set.
void CorrelatedSND::correlation_calc(std::vector<double>& dist_draws) {
    for (int i=0; i<dist_draws.size(); i++) {
      dist_draws[i] = rho * (*uncorr_draws)[i] + dist_draws[i] * sqrt(1-rho
          *rho);
    }
}

void CorrelatedSND::random_draws(const std::vector<double>& uniform_draws,
                                 std::vector<double>>& dist_draws) {
  // The following functionality is lifted directly from
  // statistics.h, which is fully commented!
  if (uniform_draws.size() != dist_draws.size()) {
    std::cout << "Draws vectors are of unequal size in standard normal dist
        ."
      << std::endl;
    return;
  }

  if (uniform_draws.size() % 2 != 0) {
    std::cout << "Uniform draw vector size not an even number." << std::
        endl;
    return;
  }

  for (int i=0; i<uniform_draws.size() / 2; i++) {
    dist_draws[2*i] = sqrt(-2.0*log(uniform_draws[2*i])) *
      sin(2*M_PI*uniform_draws[2*i+1]);
    dist_draws[2*i+1] = sqrt(-2.0*log(uniform_draws[2*i])) *
      cos(2*M_PI*uniform_draws[2*i+1]);
  }
```

```
  // Modify the random draws via the correlation calculation
  correlation_calc(dist_draws);

  return;
}

#endif
```

The work is carried out in correlation_calc. It is easy to see that if $\rho = 0$, then dist_draws is unmodified, whereas if $\rho = 1$, then dist_draws is simply equated to uncorr_draws. Thus with $0 < \rho < 1$ we have a weighted average of each set of random draws. Note that I have reproduced the Box-Muller functionality here so that you don't have to look it up in statistics.cpp. In a production code this would be centralised elsewhere (such as with a random number generator class).

Now we can tie it all together. Here is the listing of main.cpp:

```
#include "statistics.h"
#include "correlated_snd.h"
#include <iostream>
#include <vector>

int main(int argc, char **argv) {

  // Number of values
  int vals = 30;

  /* UNCORRELATED SND */
  /* ================ */

  // Create the Standard Normal Distribution and random draw vectors
  StandardNormalDistribution snd;
  std::vector<double> snd_uniform_draws(vals, 0.0);
  std::vector<double> snd_normal_draws(vals, 0.0);

  // Simple random number generation method based on RAND
  // We could be more sophisticated an use a LCG or Mersenne Twister
  // but we're trying to demonstrate correlation, not efficient
```

```
  // random number generation!
  for (int i=0; i<snd_uniform_draws.size(); i++) {
    snd_uniform_draws[i] = rand() / static_cast<double>(RAND_MAX);
  }

  // Create standard normal random draws
  snd.random_draws(snd_uniform_draws, snd_normal_draws);

  /* CORRELATION SND */
  /* ================ */

  // Correlation coefficient
  double rho = 0.5;

  // Create the correlated standard normal distribution
  CorrelatedSND csnd(rho, &snd_normal_draws);
  std::vector<double> csnd_uniform_draws(vals, 0.0);
  std::vector<double> csnd_normal_draws(vals, 0.0);

  // Uniform generation for the correlated SND
  for (int i=0; i<csnd_uniform_draws.size(); i++) {
    csnd_uniform_draws[i] = rand() / static_cast<double>(RAND_MAX);
  }

  // Now create the -correlated- standard normal draw series
  csnd.random_draws(csnd_uniform_draws, csnd_normal_draws);

  // Output the values of the standard normal random draws
  for (int i=0; i<snd_normal_draws.size(); i++) {
    std::cout << snd_normal_draws[i] << ", " << csnd_normal_draws[i] << std
        ::endl;
  }

  return 0;
}
```

The above code is somewhat verbose, but that is simply a consequence of not encapsulating

the random number generation capability. Once we have created an initial set of standard normal draws, we simply have to pass that to an instance of CorrelatedSND (in this line: CorrelatedSND csnd(rho, &snd_normal_draws);) and then call random_draws(..) to create the correlated stream. Finally, we output both sets of values:

```
3.56692, 1.40915
3.28529, 1.67139
0.192324, 0.512374
-0.723522, 0.992231
1.10093, 1.14815
0.217484, -0.211253
-2.22963, -1.94287
-1.06868, -0.500967
-0.35082, -0.0884041
0.806425, 0.326177
-0.168485, -0.242706
-1.3742, 0.752414
0.131154, -0.632282
0.59425, 0.311842
-0.449029, 0.129012
-2.37823, -0.469604
0.0431789, -0.52855
0.891999, 1.0677
0.564585, 0.825356
1.26432, -0.653957
-1.21881, -0.521325
-0.511385, -0.881099
-0.43555, 1.23216
0.93222, 0.237333
-0.0973298, 1.02387
-0.569741, 0.33579
-1.7985, -1.52262
-1.2402, 0.211848
-1.26264, -0.490981
-0.39984, 0.150902
```

There are plenty of extensions we could make to this code. The obvious two are encapsulating the random number generation and converting it to use an efficient Cholesky Decomposition

implementation. Now that we have correlated streams, we can also implement the Heston Model in Monte Carlo.

Up until this point we have priced all of our options under the assumption that the volatility, σ, of the underlying asset has been constant over the lifetime of the option. In reality financial markets do not behave this way. Assets exist under *market regimes* where their volatility can vary signficantly during different time periods. The 2007-2008 financial crisis and the May Flash Crash of 2010 are good examples of periods of intense market volatility.

Thus a natural extension of the Black Scholes model is to consider a non-constant volatility. Steven Heston formulated a model that not only considered a time-dependent volatility, but also introduced a stochastic (i.e. non-deterministic) component as well. This is the famous **Heston model for stochastic volatility**.

In this chapter we will outline the mathematical model and use a discretisation technique known as Full Truncation Euler Discretisation, coupled with Monte Carlo simulation, in order to price a European vanilla call option with C++. As with the majority of the models implemented on QuantStart, the code is object-oriented, allowing us to "plug-in" other option types (such as Path-Dependent Asians) with minimal changes.

16.5 Mathematical Model

The Black Scholes model uses a stochastic differential equation with a geometric Brownian motion to model the dynamics of the asset path. It is given by:

$$dS_t = \mu S_t dt + \sigma S_t dW_t^S \tag{16.10}$$

S_t is the price of the underlying asset at time t, μ is the (constant) drift of the asset, σ is the (constant) volatility of the underlying and dW_t^S is a Weiner process (i.e. a random walk).

The Heston model extends this by introducing a second stochastic differential equation to represent the "path" of the volatility of the underlying over the lifetime of the option. The SDE for the variance is given by a Cox-Ingersoll-Ross process:

$$dS_t = \mu S_t dt + \sqrt{\nu_t} S_t dW_t^S \tag{16.11}$$

$$d\nu_t = \kappa(\theta - \nu_t)dt + \xi\sqrt{\nu_t} dW_t^\nu \tag{16.12}$$

Where:

- μ is the drift of the asset
- θ is the *expected value* of ν_t, i.e. the long run average price variance
- κ is the rate of mean reversion of ν_t to the long run average θ
- ξ is the "vol of vol", i.e. the variance of ν_t

Note that none of the parameters have any time-dependence. Extensions of the Heston model generally allow the values to become piecewise constant.

In order for $\nu_t > 0$, the Feller condition must be satisfied:

$$2\kappa\theta > \xi^2 \tag{16.13}$$

In addition, the model enforces that the two separate Weiner processes making up the randomness are in fact correlated, with instantaneous constant correlation ρ:

$$dW_t^S dW_t^\nu = \rho dt \tag{16.14}$$

16.6 Euler Discretisation

Given that the SDE for the asset path is now dependent (in a temporal manner) upon the solution of the second volatility SDE, it is necessary to simulate the volatility process first and then utilise this "volatility path" in order to simulate the asset path. In the case of the original Black Scholes SDE it is possible to use Ito's Lemma to directly solve for S_t. However, we are unable to utilise that procedure here and must use a *numerical approximation* in order to obtain both paths. The method utilised is known as **Euler Discretisation**.

The volatility path will be discretised into constant-increment time steps of Δt, with the updated volatility, ν_{i+1} given as an *explicit* function of ν_i:

$$\nu_{i+1} = \nu_i + \kappa(\theta - \nu_i)\Delta t + \xi\sqrt{\nu_i}\Delta W_{i+1}^\nu \tag{16.15}$$

However, since this is a finite discretisation of a continuous process, it is possible to introduce

discretisation errors where ν_{i+1} can become negative. This is not a "physical" situation and so is a direct consequence of the numerical approximation. In order to handle negative values, we need to modify the above formula to include methods of eliminating negative values for subsequent iterations of the volatility path. Thus we introduce three new functions f_1, f_2, f_3, which lead to three separate "schemes" for how to handle the negative volatility values:

$$\nu_{i+1} = f_1(\nu_i) + \kappa(\theta - f_2(\nu_i))\Delta t + \xi\sqrt{f_3(\nu_i)}\Delta W^{\nu}_{i+1} \tag{16.16}$$

The three separate schemes are listed below:

Scheme	f_1	f_2	f_3
Reflection	$\|x\|$	$\|x\|$	$\|x\|$
Partial Truncation	x	x	x
Full Truncation	x	x^+	x^+

Where $x^+ = \max(x, 0)$.

The literature tends to suggest that the Full Truncation method is the "best" and so this is what we will utilise here. The Full Truncation scheme discretisation equation for the volatility path will thus be given by:

$$\nu_{i+1} = \nu_i + \kappa(\theta - \nu_i^+)\Delta t + \xi\sqrt{\nu_i^+}\Delta W^{\nu}_{i+1} \tag{16.17}$$

In order to simulate ΔW^{ν}_{i+1}, we can make use of the fact that since it is a Brownian motion, $W^{\nu}_{i+1} - W^{\nu}_i$ is normally distributed with variance Δt and that the distribution of $W^{\nu}_{i+1} - W^{\nu}_i$ is independent of i. This means it can be replaced with $\sqrt{\Delta t}N(0,1)$, where $N(0,1)$ is a random draw from the standard normal distribution.

We will return to the question of how to calculate the W^{ν}_i terms in the next section. Assuming we have the ability to do so, we are able to simulate the price of the asset path with the following discretisation:

$$S_{i+1} = S_i \exp\left(\mu - \frac{1}{2}v_i^+\right)\Delta t + \sqrt{v_i^+}\sqrt{\Delta t}\Delta W^{S}_{i+1} \tag{16.18}$$

As with the Full Truncation mechanism outlined above, the volatility term appearing in the asset SDE discretisation has also been truncated and so ν_i is replaced by ν_i^+.

16.6.1 Correlated Asset Paths

The next major issue that we need to look at is how to generate the W_i^ν and W_i^S terms for the volatility path and the asset path respectively, such that they remain correlated with correlation ρ, as prescribed via the mathematical model. This is exactly what is necessary here.

Once we have two uniform random draw vectors it is possible to use the StandardNormalDistribution class outlined in the chapter on statistical distributions to create two new vectors containing standard normal random draws - exactly what we need for the volatility and asset path simulation!

16.6.2 Monte Carlo Algorithm

In order to price a European vanilla call option under the Heston stochastic volatility model, we will need to generate many asset paths and then calculate the risk-free discounted average pay-off. This will be our option price.

The algorithm that we will follow to calculate the full options price is as follows:

1. Choose number of asset simulations for Monte Carlo and number of intervals to discretise asset/volatility paths over

2. For each Monte Carlo simulation, generate two uniform random number vectors, with the second correlated to the first

3. Use the statistics distribution class to convert these vectors into two new vectors containing standard normal draws

4. For each time-step in the discretisation of the vol path, calculate the next volatility value from the normal draw vector

5. For each time-step in the discretisation of the asset path, calculate the next asset value from the vol path vector and normal draw vector

6. For each Monte Carlo simulation, store the pay-off of the European call option

7. Take the mean of these pay-offs and then discount via the risk-free rate to produce an option price, under risk-neutral pricing.

We will now present a C++ implementation of this algorithm using a mixture of new code and prior classes written in prior chapters.

16.7 C++ Implementation

We are going to take an object-oriented approach and break the calculation domain into various re-usable classes. In particular we will split the calculation into the following objects:

- PayOff - This class represents an option pay-off object. We have discussed it at length on QuantStart.
- Option - This class holds the parameters associated with the *term sheet* of the European option, as well as the risk-free rate. It requires a PayOff instance.
- StandardNormalDistribution - This class allows us to create standard normal random draw values from a uniform distribution or random draws.
- CorrelatedSND - This class takes two standard normal random draws and correlates the second with the first by a correlation factor ρ.
- HestonEuler - This class accepts Heston model parameters and then performs a Full Truncation of the Heston model, generating both a volatility path and a subequent asset path.

We will now discuss the classes individually.

16.7.1 PayOff Class

The PayOff class won't be discussed in any great detail within this chapter as it is described fully in previous chapters. The PayOff class is a functor and as such is *callable*.

16.7.2 Option Class

The Option class is straightforward. It simply contains a set of public members for the option *term sheet* parameters (strike K, time to maturity T) as well as the (constant) risk-free rate r. The class also takes a pointer to a PayOff object, making it straightforward to "swap out" another pay-off (such as that for a Put option).

The listing for option.h follows:

```
#ifndef __OPTION_H
#define __OPTION_H
```

```
#include "payoff.h"

class Option {
 public:
  PayOff* pay_off;
  double K;
  double r;
  double T;

  Option(double _K, double _r,
         double _T, PayOff* _pay_off);

  virtual ~Option();
};

#endif
```

The listing for option.cpp follows:

```
#ifndef __OPTION_CPP
#define __OPTION_CPP

#include "option.h"

Option::Option(double _K, double _r,
               double _T, PayOff* _pay_off) :
  K(_K), r(_r), T(_T), pay_off(_pay_off) {}

Option::~Option() {}

#endif
```

As can be seen from the above listings, the class doesn't do much beyond storing some data members and exposing them.

16.7.3 Statistics and CorrelatedSND Classes

The StandardNormalDistribution and CorrelatedSND classes are described in detail within the chapter on statistical distributions and in the sections above so we will not go into detail here.

16.7.4 HestonEuler Class

The HestonEuler class is designed to accept the parameters of the Heston Model - in this case κ, θ, ξ and ρ - and then calculate both the volatility and asset price paths. As such there are private data members for these parameters, as well as a pointer member representing the option itself. There are two calculation methods designed to accept the normal draw vectors and produce the respective volatility or asset spot paths.

The listing for heston_mc.h follows:

```
#ifndef __HESTON_MC_H
#define __HESTON_MC_H

#include <cmath>
#include <vector>
#include "option.h"

// The HestonEuler class stores the necessary information
// for creating the volatility and spot paths based on the
// Heston Stochastic Volatility model.
class HestonEuler {
 private:
  Option* pOption;
  double kappa;
  double theta;
  double xi;
  double rho;

 public:
  HestonEuler(Option* _pOption,
              double _kappa, double _theta,
              double _xi, double _rho);
  virtual ~HestonEuler();
```

```
  // Calculate the volatility path
  void calc_vol_path(const std::vector<double>& vol_draws,
                     std::vector<double>& vol_path);

  // Calculate the asset price path
  void calc_spot_path(const std::vector<double>& spot_draws,
                      const std::vector<double>& vol_path,
                      std::vector<double>& spot_path);
};

#endif
```

The listing for heston_mc.cpp follows:

```
#ifndef __HESTON_MC_CPP
#define __HESTON_MC_CPP

#include "heston_mc.h"

// HestonEuler
// ===========

HestonEuler::HestonEuler(Option* _pOption,
                         double _kappa, double _theta,
                         double _xi, double _rho) :
  pOption(_pOption), kappa(_kappa), theta(_theta), xi(_xi), rho(_rho) {}

HestonEuler::~HestonEuler() {}

void HestonEuler::calc_vol_path(const std::vector<double>& vol_draws,
                                std::vector<double>& vol_path) {
  size_t vec_size = vol_draws.size();
  double dt = pOption->T/static_cast<double>(vec_size);

  // Iterate through the correlated random draws vector and
  // use the 'Full Truncation' scheme to create the volatility path
  for (int i=1; i<vec_size; i++) {
    double v_max = std::max(vol_path[i-1], 0.0);
```

```
    vol_path[i] = vol_path[i-1] + kappa * dt * (theta - v_max) +
      xi * sqrt(v_max * dt) * vol_draws[i-1];
  }
}

void HestonEuler::calc_spot_path(const std::vector<double>& spot_draws,
                                 const std::vector<double>& vol_path,
                                 std::vector<double>& spot_path) {
  size_t vec_size = spot_draws.size();
  double dt = pOption->T/static_cast<double>(vec_size);

  // Create the spot price path making use of the volatility
  // path. Uses a similar Euler Truncation method to the vol path.
  for (int i=1; i<vec_size; i++) {
    double v_max = std::max(vol_path[i-1], 0.0);
    spot_path[i] = spot_path[i-1] * exp( (pOption->r - 0.5*v_max)*dt +
        sqrt(v_max*dt)*spot_draws[i-1]);
  }
}

#endif
```

The calc_vol_path method takes references to a const vector of normal draws and a vector to store the volatility path. It calculates the Δt value (as dt), based on the option maturity time. Then, the stochastic simulation of the volatility path is carried out by means of the Full Truncation Euler Discretisation, outlined in the mathematical treatment above. Notice that ν_i^+ is precalculated, for efficiency reasons.

The calc_spot_path method is similar to the calc_vol_path method, with the exception that it accepts another vector, vol_path that contains the volatility path values at each time increment. The risk-free rate r is obtained from the option pointer and, once again, ν_i^+ is precalculated. Note that all vectors are passed by reference in order to reduce unnecessary copying.

16.7.5 Main Program

This is where it all comes together. There are two components to this listing: The generate_normal_correlation_paths function and the main function. The former is designed to handle the "boilerplate" code of generating the necessary uniform random draw vectors and then utilising the CorrelatedSND object

to produce correlated standard normal distribution random draw vectors.

I wanted to keep this entire example of the Heston model tractable, so I have simply used the C++ built-in rand function to produce the uniform standard draws. However, in a production environment a Mersenne Twister uniform number generator (or something even more sophisticated) would be used to produce high-quality pseudo-random numbers. The output of the function is to calculate the values for the spot_normals and cor_normals vectors, which are used by the asset spot path and the volatility path respectively.

The main function begins by defining the parameters of the simulation, including the Monte Carlo values and those necessary for the option and Heston model. The actual parameter values are those give in the paper by Broadie and Kaya[3]. The next task is to create the pointers to the PayOff and Option classes, as well as the HestonEuler instance itself.

After declaration of the various vectors used to hold the path values, a basic Monte Carlo loop is created. For each asset simulation, the new correlated values are generated, leading to the calculation of the vol path and the asset spot path. The option pay-off is calculated for each path and added to the total, which is then subsequently averaged and discounted via the risk-free rate. The option price is output to the terminal and finally the pointers are deleted.

Here is the listing for main.cpp:

```
#include <iostream>

#include "payoff.h"
#include "option.h"
#include "correlated_snd.h"
#include "heston_mc.h"

void generate_normal_correlation_paths(double rho,
    std::vector<double>& spot_normals, std::vector<double>& cor_normals) {
  unsigned vals = spot_normals.size();

  // Create the Standard Normal Distribution and random draw vectors
  StandardNormalDistribution snd;
  std::vector<double> snd_uniform_draws(vals, 0.0);

  // Simple random number generation method based on RAND
  for (int i=0; i<snd_uniform_draws.size(); i++) {
    snd_uniform_draws[i] = rand() / static_cast<double>(RAND_MAX);
```

```
  }

  // Create standard normal random draws
  snd.random_draws(snd_uniform_draws, spot_normals);

  // Create the correlated standard normal distribution
  CorrelatedSND csnd(rho, &spot_normals);
  std::vector<double> csnd_uniform_draws(vals, 0.0);

  // Uniform generation for the correlated SND
  for (int i=0; i<csnd_uniform_draws.size(); i++) {
    csnd_uniform_draws[i] = rand() / static_cast<double>(RAND_MAX);
  }

  // Now create the -correlated- standard normal draw series
  csnd.random_draws(csnd_uniform_draws, cor_normals);
}

int main(int argc, char **argv) {
  // First we create the parameter list
  // Note that you could easily modify this code to input the parameters
  // either from the command line or via a file
  unsigned num_sims = 100000;   // Number of simulated asset paths
  unsigned num_intervals = 1000;  // Number of intervals for the asset path
       to be sampled

  double S_0 = 100.0;     // Initial spot price
  double K = 100.0;       // Strike price
  double r = 0.0319;      // Risk-free rate
  double v_0 = 0.010201; // Initial volatility
  double T = 1.00;        // One year until expiry

  double rho = -0.7;      // Correlation of asset and volatility
  double kappa = 6.21;    // Mean-reversion rate
  double theta = 0.019;   // Long run average volatility
  double xi = 0.61;       // "Vol of vol"
```

```
    // Create the PayOff, Option and Heston objects
    PayOff* pPayOffCall = new PayOffCall(K);
    Option* pOption = new Option(K, r, T, pPayOffCall);
    HestonEuler hest_euler(pOption, kappa, theta, xi, rho);

    // Create the spot and vol initial normal and price paths
    std::vector<double> spot_draws(num_intervals, 0.0); // Vector of initial
        spot normal draws
    std::vector<double> vol_draws(num_intervals, 0.0); // Vector of initial
        correlated vol normal draws
    std::vector<double> spot_prices(num_intervals, S_0); // Vector of
        initial spot prices
    std::vector<double> vol_prices(num_intervals, v_0); // Vector of
        initial vol prices

    // Monte Carlo options pricing
    double payoff_sum = 0.0;
    for (unsigned i=0; i<num_sims; i++) {
      std::cout << "Calculating path " << i+1 << " of " << num_sims << std::
          endl;
      generate_normal_correlation_paths(rho, spot_draws, vol_draws);
      hest_euler.calc_vol_path(vol_draws, vol_prices);
      hest_euler.calc_spot_path(spot_draws, vol_prices, spot_prices);
      payoff_sum += pOption->pay_off->operator()(spot_prices[num_intervals
          -1]);
    }
    double option_price = (payoff_sum / static_cast<double>(num_sims)) * exp
        (-r*T);
    std::cout << "Option Price: " << option_price << std::endl;

    // Free memory
    delete pOption;
    delete pPayOffCall;

    return 0;
}
```

For completeness, I have included the *makefile* utilised on my MacBook Air, running Mac OSX 10.7.4:

```
heston: main.cpp heston_mc.o correlated_snd.o statistics.o option.o payoff.
    o
    clang++ -o heston main.cpp heston_mc.o correlated_snd.o statistics.o
        option.o payoff.o -arch x86_64

heston_mc.o: heston_mc.cpp option.o
    clang++ -c heston_mc.cpp option.o -arch x86_64

correlated_snd.o: correlated_snd.cpp statistics.o
    clang++ -c correlated_snd.cpp statistics.o -arch x86_64

statistics.o: statistics.cpp
    clang++ -c statistics.cpp -arch x86_64

option.o: option.cpp payoff.o
    clang++ -c option.cpp payoff.o -arch x86_64

payoff.o: payoff.cpp
    clang++ -c payoff.cpp -arch x86_64
```

Here is the output of the program:

```
..
..
Calculating path 99997 of 100000
Calculating path 99998 of 100000
Calculating path 99999 of 100000
Calculating path 100000 of 100000
Option Price: 6.81982
```

The exact option price is 6.8061, as reported by Broadie and Kaya[3]. It can be made somewhat more accurate by increasing the number of asset paths and discretisation intervals.

There are a few extensions that could be made at this stage. One is to allow the various "schemes" to be implemented, rather than hard-coded as above. Another is to introduce time-dependence into the parameters. The next step after creating a model of this type is to actually calibrate to a set of market data such that the parameters may be determined.