Generating Correlated Brownian Motions

When pricing options we need a model for the evolution of the underlying asset. The model used is a Geometric Brownian Motion, which can be described by the following *stochastic differential equation*

$$dS_t = \mu S_t \; dt + \sigma S_t \; dW_t$$

where μ is the expected annual return of the underlying asset, σ is the annualized volatility, and W_t is a Brownian Motion. This *stochastic differential equation* has the following solution

$$S_{t+\Delta t} = S_t \exp\left(\left(\mu - \frac{\sigma^2}{2}\right)\Delta t + \sigma\sqrt{\Delta t}\; W_t\right)$$

which is the equation used in the implementation of any model that requires the simulation of the future behavior of an asset.

In this article however, we will consider the simulation of several correlated assets. This is an important topic in quantitative finance, as it can be applied to simulating assets held in a portfolio which are dependent on one another to determine the underlying risk of the portfolio, or to pricing assets - such as Spread Options - which are path dependent as well as dependent on two underlying assets which are correlated. The pricing of spread options will be considered in the following article. Suppose we have a correlation matrix, denoted C

$$C = \begin{bmatrix} \rho_{1,1} & \rho_{1,2} & \cdots & \rho_{1,n} \\ \rho_{2,1} & \rho_{2,2} & \cdots & \rho_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \rho_{n,1} & \rho_{n,2} & \cdots & \rho_{n,n} \end{bmatrix} = \begin{bmatrix} 1 & \rho_{1,2} & \cdots & \rho_{1,n} \\ \rho_{2,1} & 1 & \cdots & \rho_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \rho_{n,1} & \rho_{n,2} & \cdots & 1 \end{bmatrix}$$

where each ρ_{ij} represents the correlation of the ith and jth asset. Since the correlation of each asset with itself is 1, the correlation matrix has ones on the diagonal. It is clear that C is symmetric and positive definite. The following is the procedure for generating correlated random variables is:

1. Perform Cholesky Decomposition of correlation matrix C to obtain upper triangular matrix L^T.
2. Generate random vector $\mathbf{X} \sim N(0, I)$.
3. Obtain a correlated random vector $\mathbf{Z} = \mathbf{X}L^T$.

This procedure is used in the simulation of correlated asset paths. Below is the C++ implementation of the simulation of correlated asset price paths.

`Correlated Brownian Motions`

```
//This function generates correlated price paths
//between a specified number of assets.
//
//S0: initial price vector
//mu: mean vector
//sigma: standard deviation vector
//Corr: correlation matrix
//T: time (in years)
//numAssets: number of assets
//steps: number of days
//sims: number of iterations

#include<iostream>
#include <fstream>
#include <iomanip>
#include "Cholesky.h"
#include "matrixOperations.h"
#include "stdNormalRV.h"

using namespace std;

vector<vector<double> > multiBrownianMotion(vector<double>
S0, vector<double> mu,
vector<double> sigma, vector<vector<double> >  Corr, int T,
int numAssets, int steps, int sims)
{
double dt = 1.0/252;
vector<vector<double> > nudt(steps*sims, vector<double>
(numAssets));
vector<vector<double> > diagSig(numAssets,
vector<double> (numAssets));
vector<vector<double> >  vPrices(steps*sims,
vector<double> (numAssets));
vector<vector<double> > C(steps*sims, vector<double>
(numAssets));
vector<vector<double> > randnMatrix(steps*sims,
vector<double> (numAssets));
vector<vector<double> > Z(steps*sims, vector<double>
(numAssets));

//sigma*sigma/2
vector<double> sigSquaredHalf = scalVecDiv(elemVecMult
(sigma, sigma, numAssets), 2, numAssets);

//nu = mu - sigma*sigma/2
vector<double> nu = elemVecSub(mu, sigSquaredHalf,
```

```
numAssets);

//Cholesky decomposition of correlation matrix
//to produce upper triangular matrix
vector<vector<double> > Y = transpose(cholesky(Corr),
numAssets, numAssets);

//populate matrix of size steps by numAssets with nu*dt
for (int i = 0; i < steps*sims; i++)
{
for (int j = 0; j < numAssets; j++)
nudt[i][j] = nu[j]*dt;
}

//convert vector sigma into diagonal matrix with zeros on
//off-diagonals
diagSig = diag(sigma, numAssets);

ofstream myfile;
    myfile.open ("pricesMatrix.txt");

/*******************************************************
vvv Multidimensional Brownian Motion Simulation Begins vvv
********************************************************/

srand((unsigned)time(0));


   for (int i = 0; i < steps*sims; i++)
   {
   for (int j = 0; j < numAssets; j++)
  //populate random matrix
  randnMatrix[i][j] = random_normal();
   }

   //correlate the random variables
   Z = matrixMult(randnMatrix, Y, steps*sims, numAssets,
   numAssets);

   C = matrixMult(Z, diagSig, steps*sims, numAssets,
   numAssets);

   for (int i = 0; i < steps*sims-1; i++)
   {
   for (int j = 0; j < numAssets; j++)
   {
```

```
   if (i==0 || (i+1)%252==0)
   //initialize new paths with initial asset prices
   vPrices[i][j] = S0[j];
   else
   //generate full paths
       vPrices[i][j] = exp(nudt[i][j] + C[i][j]*sqrt(dt))
        *vPrices[i-1][j];
   }
   }

return vPrices;

/*********************************************************
^^Multidimensional Brownian Motion Simulation Concludes^^
*********************************************************/
}
```

This function utilizes the auxiliary functions discussed in the Matrix Operations article as well as the following functions

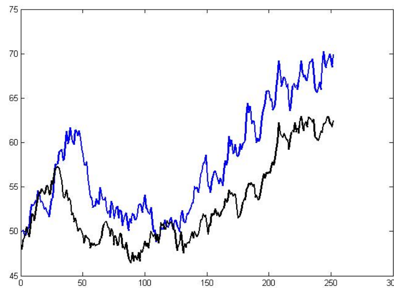
```
//Vector division by scalar
vector<double> scalVecDiv(vector<double> V, double scalar, int n)
{
vector<double> ans(n);
for (int i=0; i<n; i++)
{
ans[i] = V[i] / scalar;
}

return ans;
}
```

```
//Element by element vector multiplication between 2 vectors
//of same size
vector<double> elemVecMult(vector<double> V, vector<double> W, int n)
{
vector<double> ans(n);
for (int i=0; i<n; i++)
{
ans[i] = V[i] * W[i];
}
```

```
return ans;
}



//Element by element vector subtraction between 2 vectors
//of same size
vector<double> elemVecSub(vector<double> V, vector<double> W, int n)
{
vector<double> ans(n);
for (int i=0; i<n; i++)
{
ans[i] = V[i] - W[i];
}

return ans;
}



//Takes in vector and writes its entires on the diagonal of a matrix
//with zeros elsewhere
vector<vector<double> > diag(vector<double> V, int n)
{
vector<vector<double> > ans(n, vector<double> (n));

for (int j=0; j< n; j++)
{
for (int i=0; i<n; i++)
{
ans[i][j] = 0;
}
}

for (int j=0; j< n; j++)
{
for (int i=0; i<n; i++)
{
ans[i][i] = V[i];
}
}

return ans;
}
```

Let's consider an example with two stocks over an interval of 252 days - approximately an entire year of trading days. Suppose the first has an initial price of $S_0 = \$50$, mean $\mu_1 = 0.13$, and volatility $\sigma_1 = 0.25$. Suppose the second has an initial price of $S_0 = \$48$, mean $\mu_2 = 0.16$, and volatility $\sigma_2 = 0.20$. Let

$$C = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}$$



The above is the plot of a single simulation of the two stocks with the correlation structure defined by C. Here is a program that uses this function to generate the price paths of the aforementioned stocks.

```
#include<iostream>
#include<vector>
#include <iomanip>
#include "MultivariateBrownianMotion.h"

using namespace std;

int main( )
{
int numAssets = 2;
int days_ = 252;
int time_ = 1;
int sims = 1;
vector<double> sigma(numAssets);
vector<double> mu(numAssets);
vector<double> S0(numAssets);
vector<vector<double> >  Corr(numAssets, vector<double> (numAssets));
vector<vector<double> > ans(days_*sims, vector<double> (numAssets));

//initial price vector
```

```
S0[0] = 50; S0[1] = 48;

//vector of individual volatilities
sigma[0] = 0.25; sigma[1] = 0.2;

//vector of expected returns
mu[0] = 0.13; mu[1] = 0.16;

//correlation matrix
Corr[0][0] = 1; Corr[0][1] = 0.5;
Corr[1][0] = 0.5; Corr[1][1] = 1;

ofstream myfile;
    myfile.open ("pricesMatrix.txt");

ans = multiBrownianMotion(S0, mu, sigma, Corr, time_, numAssets, days_, sims);


   for (int i = 0; i < days_*sims-1; i++)
   {
   for (int j = 0; j < numAssets; j++)
   {

   cout <<left<<setw(15)<< ans[i][j];
   myfile << ans[i][j] << "\t";
   }
   cout << endl;
   myfile << endl;
   }

system("PAUSE");
return 0;
}
```