NYU | TANDON SCHOOL OF ENGINEERING

Topic 8
Enhancing Monte Carlo Framework for Pricing Basket Options

Path-dependent Basket Options

- In the multi-dimensional Black-Scholes model: we have d stocks whose t time prices: $S_1(t), \ldots S_d(t)$:

$$S(t) = \begin{pmatrix} S_1(t) \\ . \\ . \\ . \\ S_d(t) \end{pmatrix}$$

- For the vector of prices at time t, under the risk-neutral probability:

$$S_j(t) = S_j(0)\exp((r - \frac{\sigma_j^2}{2})t + \sum_{l=1}^{d} c_{jl} W_l(t))$$

  where $W_1(t),\ldots, W_d(t)$ are independent Wiener processes under the neutral probability Q, $\boldsymbol{C}=(c_{jl})^d_{j,l=1}$ is a d x d matrix

- $\sigma_1,\ldots, \sigma_d \in \Re$:

$$\sigma_j = \sqrt{c_{j1}^2 + \ldots + c_{jd}^2},\ j = 1,\ldots d$$

- A path-dependent option with payoff at time T of the form: $H(T)=h(\mathbf{S}(t_1),\ldots,\mathbf{S}(t_m))$, where $t_k=k/m*T$ for k = 1,…,m, and where h is a payoff function:

$$h : \underbrace{R^d \times \ldots \times R^d}_{m} \to R$$

- Arithmetic Asian Basket Call:

$$H(T) = \left(\sum_{j=1}^{d} \left(\frac{1}{m} \sum_{k=1}^{m} S_j(t_k) \right) - K \right)^+$$

- For two vectors **v, w** ϵ $\Re$,

$$vw = \begin{pmatrix} v_1 w_1 \\ \ldots \\ v_d w_d \end{pmatrix}, \exp(v) = \begin{pmatrix} e^{v_1} \\ \ldots \\ e^{v_d} \end{pmatrix}$$

- Let $Z_1,\ldots,Z_d$ are i.i.d random variables with N(0,1)

$$Z = \begin{pmatrix} Z_1 \\ \ldots \\ Z_d \end{pmatrix}$$

- Let ***Z**₁,…,**Z**ₘ* be a sequence of i.i.d random variables with N(0,1)
- Let

$$\sigma = \begin{pmatrix} \sigma_1 \\ \dots \\ \sigma_d \end{pmatrix}$$

- For k=1,…,m, the price vector **S**(t_k):

$$\mathbf{S}(t_k) = \mathbf{S}(t_{k-1}) \exp\left(\left(r - \frac{1}{2}\sigma\sigma\right)(t_k - t_{k-1}) + \sqrt{t_k - t_{k-1}}\,\mathbf{C}\mathbf{Z}_k\right)$$

- Let $\hat{\mathbf{Z}}_1,\ldots\hat{\mathbf{Z}}_m$ be a sequence of independent samples of $\mathbf{Z}_1,\ldots,\mathbf{Z}_m$. A sample path $(\hat{\mathbf{S}}(t_1),\ldots\hat{\mathbf{S}}(t_m))$ taking:

$$\hat{\mathbf{S}}(t_1) = \mathbf{S}(0)\exp((r - \frac{1}{2}\sigma\sigma)t_1 + \sqrt{t_1}\mathbf{C}\mathbf{Z}_k$$

$$\hat{\mathbf{S}}(t_k) = \hat{\mathbf{S}}(t_{k-1})\exp((r - \frac{1}{2}ss)(t_k - t_{k-1}) + \sqrt{t_k - t_{k-1}}\mathbf{C}\hat{\mathbf{Z}}_k)$$

For k = 2,…,m

- Let $(\hat{\mathbf{S}}^i(t_1),\ldots\hat{\mathbf{S}}^i(t_m))$ for i = 1,…,N, be a sequence of independent sample paths, the option price H(0):

$$H(0) \approx \hat{H}_N(0) = e^{-rT}\frac{1}{N}\sum_{i=1}^{N} h(\hat{\mathbf{S}}^i(t_1),\ldots\hat{\mathbf{S}}^i(t_m))$$

Matrix.h

```
#pragma once
#include <vector>
#include <iostream>
using namespace std;
namespace fre {
    typedef vector<double> Vector;
    typedef vector<Vector> Matrix;
    Vector operator*(const Matrix& C, const Vector& V);
    Vector operator*(const double& a, const Vector& V);
    Vector operator*(const Vector& V, const Vector& W);
    Vector operator+(const double& a, const Vector& V);
    Vector operator+(const Vector& V, const Vector& W);
    Vector exp(const Vector& V);
    double operator^(const Vector& V, const Vector& W); // scalar operator
    ostream& operator<<(ostream& out, Vector& V);   // Overload cout for Vector
    ostream& operator<<(ostream& out, Matrix& W);  // Overload cout for Matrix
}
```

Matrix.cpp

```
#include "Matrix.h"
#include <cmath>
using namespace std;
namespace fre {
    Vector operator*(const Matrix& C, const Vector& V)
    {
        int d = (int)C.size();
        Vector W(d);
        for (int j = 0; j < d; j++)
        {
            W[j] = 0.0;
            for (int l = 0; l < d; l++) W[j] = W[j] + C[j][l] * V[l];
        }
        return W;
    }
```

Matrix.cpp (continue)

```
Vector operator*(const double& a, const Vector& V)
{
   int d = (int)V.size();
   Vector U(d);
   for (int j = 0; j < d; j++) U[j] = a * V[j];
   return U;
}

Vector operator*(const Vector& V, const Vector& W)
{
   int d = (int)V.size();
   Vector U(d);
   for (int j = 0; j < d; j++) U[j] = V[j] * W[j];
   return U;
}
```

Matrix.cpp (continue)

```
Vector operator+(const Vector& V, const Vector& W)
{
   int d = (int)V.size();
   Vector U(d);
   for (int j = 0; j < d; j++) U[j] = V[j] + W[j];
   return U;
}

Vector operator+(const double& a, const Vector& V)
{
   int d = (int)V.size();
   Vector U(d);
   for (int j = 0; j < d; j++) U[j] = a + V[j];
   return U;
}
```

Matrix.cpp (continue)

```
Vector exp(const Vector& V)
{
   int d = (int)V.size();
   Vector U(d);
   for (int j = 0; j < d; j++) U[j] = std::exp(V[j]);
   return U;
}

double operator^(const Vector& V, const Vector& W)
{
   double sum = 0.0;
   int d = (int)V.size();
   for (int j = 0; j < d; j++) sum = sum + V[j] * W[j];
   return sum;
}
```

Matrix.cpp (continue)

```
// overload cout for vector, cout every element in the vector
ostream& operator<<(ostream& out, Vector& V)
{
    for (Vector::iterator itr = V.begin(); itr != V.end(); itr++)
        out << *itr << "   ";
    out << endl;
    return out;
}
ostream& operator<<(ostream& out, Matrix& W)
{
    for (Matrix::iterator itr = W.begin(); itr != W.end(); itr++)
        out << *itr;    // Use ostream & operator<<(ostream & out, Vector & V)
    out << endl;
    return out;
}
}
```

Notes

- ***Vector*** is a STL vector of double
- ***Matrix*** is a STL vector of ***Vector***
- *Declare various operations on vectors and matrices such as:*
  - ***Vector operator*(const Matrix& C,const Vector& V);***
  - *For multiplying a matrix by a vector: W = C * V, the result is vector.*
  - ***Operator Overloading:*** Customizes the C++ operators for operands of user-defined types.
  - Matrix and Vector are passed to overloaded operator functions by reference.
  - The keyword ***const*** to ensure the operators do not change the parameters passed by reference.

MCModel02.h

```
#pragma once
#include "Matrix.h"
namespace fre {
   typedef vector<Vector> SamplePath;
   class MCModel
   {
   private:
      Vector S0, sigma;
      Matrix C;
      double r;
   public:
      MCModel(Vector S0_, double r_, Matrix C_);
      void GenerateSamplePath(double T, int m, SamplePath& S) const;
      Vector GetS0() const { return S0; }
      Vector GetSigma() const { return sigma; }
      Matrix GetC() const { return C; }
      double GetR() const { return r; }
      void SetS0(const Vector & S0_) { S0 = S0_; }
      void SetSigma(const Vector& sigma_) { sigma = sigma_; }
      void SetC(const Matrix & C_) { C = C_; }
      void SetR(double r_) { r = r_; }
   };
}
```

MCModel02.cpp

```
#include "MCModel02.h"
#include <cmath>
#include <cstdlib>
#include <ctime>
namespace fre {
   const double pi = 4.0 * atan(1.0);
   double Gauss()
   {
      double U1 = (rand() + 1.0) / (RAND_MAX + 1.0);
      double U2 = (rand() + 1.0) / (RAND_MAX + 1.0);
      return sqrt(-2.0 * log(U1)) * cos(2.0 * pi * U2);
   }

   Vector Gauss(int d)
   {
      Vector Z(d);
      for (int j = 0; j < d; j++) Z[j] = Gauss();
      return Z;
   }
```

MCModel02.cpp (continue)

```
MCModel::MCModel(Vector S0_, double r_, Matrix C_)
{
   S0 = S0_; r = r_; C = C_; srand(time(NULL));
   int d = S0.size();
   sigma.resize(d);
   for (int j = 0; j < d; j++) sigma[j] = sqrt(C[j] ^ C[j]);        // compute σj
}

void MCModel::GenerateSamplePath(double T, int m, SamplePath& S) const
{
   Vector St = S0;
   int d = S0.size();
   for (int k = 0; k < m; k++)
   {
      S[k] = St * exp((T / m) * (r + (-0.5) * sigma * sigma) + sqrt(T / m) * (C * Gauss(d)));
      St = S[k];
   }
}
}
```

PathDepOption02.h

```
#pragma once
#include "MCModel02.h"
namespace fre {
   class PathDepOption
   {
   protected:
      double T;
      double K;
      double Price;
      int m;
   public:
      PathDepOption(double T_, double K_, int m_) : Price(0.0), T(T_), K(K_), m(m_) {}
      double PriceByMC(MCModel& Model, long N);
      virtual ~PathDepOption() {}
      virtual double Payoff(const SamplePath& S) const = 0;
   };
```

PathDepOption02.h (continue)

```
    class ArthmAsianCall : public PathDepOption
    {
    public:
        ArthmAsianCall(double T_, double K_, int m_) : PathDepOption(T_, K_, m_) {}
        double Payoff(const SamplePath& S) const;
    };
}
```

PathDepOption02.cpp

```
#include "PathDepOption02.h"
#include <cmath>
namespace fre {
   double PathDepOption::PriceByMC(MCModel& Model, long N)
   {
      double H = 0.0;
      SamplePath S(m);
      for (long i = 0; i < N; i++)
      {
         Model.GenerateSamplePath(T, m, S);
         H = (i * H + Payoff(S)) / (i + 1.0);
      }
      Price = std::exp(-Model.GetR() * T) * H;
      return Price;
   }
```

PathDepOption02.cpp (continue)

```
    double ArthmAsianCall::Payoff(const SamplePath& S) const
    {
        double Ave = 0.0;
        int d = S[0].size();
        Vector one(d);
        for (int i = 0; i < d; i++) one[i] = 1.0;
        for (int k = 0; k < m; k++)
        {
            Ave = (k * Ave + (one ^ S[k])) / (k + 1.0);
        }
        if (Ave < K) return 0.0;
        return Ave - K;
    }
}
```

Main05.cpp

```
#include <iostream>
#include "PathDepOption02.h"
using namespace std;
using namespace fre;
int main()
{
   int d=3;
   Vector S0(d);
   S0[0]=40.0;
   S0[1]=60.0;
   S0[2]=100.0;
   double r=0.03;
   Matrix C(d);
   for(int i=0;i<d;i++) C[i].resize(d);
   C[0][0] =  0.1;  C[0][1] = -0.1;  C[0][2] = 0.0;
   C[1][0] = -0.1;  C[1][1] =  0.2;  C[1][2] = 0.0;
   C[2][0] =  0.0;  C[2][1] =  0.0;  C[2][2] = 0.3;
   MCModel Model(S0,r,C);
```

Main05.cpp (continue)

```
  double T=1.0/12.0, K=200.0;
  int m=30;
  ArthmAsianCall Option(T,K,m);

  long N=3000;
  cout << "Arithmetic Basket Call Price = " << Option.PriceByMC(Model, N) << endl;

  return 0;
}

// Arithmetic Basket Call Price = 2.20446
```

Notes:

- The basket option has just 3 underlying asset, d=3.
- The dxd matrix C is initialized with some values.
- The 3-dimensional model is an object called Model.
- The object for an arithmetic Asian basket call option is created, and execute the pricing function, PriceByMC().

Greeks for basket options

- Let $u(\mathbf{S}(0))=u(S_1(0),\dots, S_d(0))$ denote the price of the option dependent on the underlying asset prices.
- Assume $u(z_1,\dots, z_d)$ is differentiable. Then

$$\delta_j = \frac{\partial u}{\partial z_j}(S_1(0),\dots S_d(0)),\ j = 1,\dots,d$$

- We write

$$(1+\varepsilon_j) = \begin{pmatrix} 1 \\ \dots \\ 1 \\ 1+\varepsilon \\ 1 \\ \dots \\ 1 \end{pmatrix}$$

for a vector, where 1+ε is the jth coordinate.

- For k =1, ..., m

$$(1+\varepsilon_j)\hat{\mathbf{S}}(t_k) = \begin{pmatrix} \hat{S}_1(t_k) \\ \dots \\ \hat{S}_{j-1}(t_k) \\ (1+\varepsilon)\hat{S}_j(t_k) \\ \hat{S}_{j+1}(t_k) \\ \dots \\ \hat{S}_d(t_k) \end{pmatrix}$$

$$H_{\varepsilon,j,N}(0) = e^{-rT}\frac{1}{N}\sum_{i=1}^{N} h((1+\varepsilon_j)\hat{\mathbf{S}}^i(t_1),\dots,(1+\varepsilon_j)\hat{\mathbf{S}}^i(t_m))$$

- For small ε:

$$\delta_j \approx \frac{u((1+\varepsilon_j)\mathbf{S}(0)) - u(\mathbf{S}(0))}{\varepsilon S_j(0)}$$

$$\delta_j \approx \hat{\delta}_j = \frac{\hat{H}_{\varepsilon_j, N}(0) - \hat{H}_N(0)}{\varepsilon S_j(0)}$$

Practice Question

- Compute the deltas using the formula we discussed in the class. Be sure to use same samples while computing $\hat{\delta}_1, \ldots, \hat{\delta}_d$

References

- Numerical Methods in Finance with C++ (Mastering Mathematical Finance), by Maciej J. Capinski and Tomasz Zastawniak, Cambridge University Press, 2012, ISBN-10: 0521177162