NYU | TANDON SCHOOL OF ENGINEERING

# Topic 8
# Monte Carlo Methods for Path-dependent Options

# Overview

- ***Path-dependent Options***
- ***Valuation***
- ***Pricing Error***
- ***Greek Parameters***
- ***Variance Reduction***
- ***Path-dependent Basket Options***

# Path-dependent Options

- ***A money account***:

  *$A(t) = e^{rt}$, where $t \geq 0$ is the time, $r \in R$ is the risk-free rate under continuous compounding.*

- ***A risky asset:***

$$S(t) = S(0)e^{(r-\frac{\sigma^2}{2})t+\sigma W_Q(t)},$$

  – *Where $\sigma \leq R$ is the volatility and $W_Q(t)$ is a Wiener Process under the risk-neutral probability Q. The Wiener process $W_Q$ has independent increments, with $W_Q(t)$ - $W_Q(s)$ having normal distribution $N(0, t-s)$ for any $t > s \geq 0$*

$$S(t_k) = S(t_{k-1})e^{(r-\frac{\sigma^2}{2})(t_k-t_{k-1})+\sigma\sqrt{t_k-t_{k-1}}Z_k},$$

  – *Where $Z_1$, …. $Z_m$ are independent and identically distributed (i.i.d) random variables with distribution $N(0,1)$.*

- ***A path-dependent option:***
  - *Expiry date T*
  - $T_k = (k/m)T$ *for* $k = 1, \ldots, m$.
  - $\boldsymbol{H(T) = h(S(t_1), \ldots, S(t_m))}$, where $h: R^m \to \Re$
  - *The option price can be determined by computing the expected discounted payoff under Q, the risk-neutral probability:* $$H(0) = e^{-rT} E_Q(H(T))$$
  - *The expectation can be computed using Monte Carlo.*
    - *Let* $\hat{Z}_1, \ldots, \hat{Z}_m$ *be a sequence of independent samples of* $Z_1 \ldots Z_m$
    - *We refer to the following sequence as an independent* ***sample path***: $$(\hat{S}(t_1), \ldots, \hat{S}(t_m))$$

– *The Sample Path is defined by:*

$$\hat{S}(t_1) = S(0)e^{(r-\frac{\sigma^2}{2})t_1+\sigma\sqrt{t_1}\hat{Z}_1},$$

$$\hat{S}(t_k) = \hat{S}(t_{k-1})e^{(r-\frac{\sigma^2}{2})(t_k-t_{k-1})+\sigma\sqrt{t_k-t_{k-1}}\hat{Z}_k}, k = 2,\ldots,m$$

– *By the law of large numbers:*

$$E_Q(h(S(t_1),\ldots,S(t_m)) = \lim_{N->\infty}\frac{1}{N}\sum_{i=1}^{N}h(\hat{S}^i(t_1),\ldots,\hat{S}^i(t_m)).$$

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

- ***Arithmetic Asian Call***
  - *A typical example of a path-dependent option.*
  - *The payoff function:*

$$h^{arithmAsiancall}(S_1,\ldots,S_m)=\left(\frac{1}{m}\sum_{k=1}^{m}S_i-K\right)^+$$

## Samples of Random Variables of N(0,1)

- Box-Muller Method:
  - If $U_1$, $U_2$ are independent random variables with uniform distribution on an interval (0,1], then the following random variable has distribution N(0,1):

$$Z = \sqrt{-2\ln(U_1)}\cos(2\pi U_2)$$

- C++ function, rand(), generates uniformly distributed random numbers in the range [0, RAND_MAX].

# Recipe for generating sample paths

1. ***Generate two integers $K_1$ and $K_2$ using rand(), and rescale the result to lie in (0,1]. Then computing***

$$\hat{U}_l = \frac{k_l + 1}{RAND\_MAX + 1}, l = 1, and 2.$$

2. ***Compute a sample of Z:***

$$\hat{Z} = \sqrt{-2\ln(\hat{U}_1)}\cos(2\pi U_2)$$

3. ***Repeat step 1 and 2 m-times to obtain $\hat{Z}_1, \ldots, \hat{Z}_m$. Then compute*** $(\hat{S}(t_1), \ldots, \hat{S}(t_m))$

# MCModel.h

```
#pragma once
#include <vector>
#include <cstdlib>
#include <ctime>
using namespace std;
namespace fre {
    typedef vector<double> SamplePath;
   class MCModel
   {
   private:
      double S0, r, sigma;
   public:
      MCModel() :S0(0.0), r(0.0), sigma(0.0) {}
      MCModel(double S0_, double r_, double sigma_) :S0(S0_), r(r_), sigma(sigma_)
      {
         srand((unsigned)time(NULL));
      }
      void GenerateSamplePath(double T, int m, SamplePath& S) const;
```

# MCModel.h (continue)

```
        double GetS0() const { return S0; }
        double GetR() const { return r; }
        double GetSigma() const { return sigma; }
        void SetS0(double S0_) { S0 = S0_; }
        void SetR(double r_) { r = r_; }
        void SetSigma(double sigma_) { sigma = sigma_; }
    };
}
```

# MCModel.cpp

```
#include "MCModel.h"
#include <cmath>
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);
    }
    void MCModel::GenerateSamplePath(double T, int m, SamplePath& S) const
    {
        double St = S0;
        for (int k = 0; k < m; k++)
        {
            S[k] = St * exp((r - sigma * sigma * 0.5) * (T / m) + sigma * sqrt(T / m) * Gauss());
            St = S[k];
        }
    }
}
```

- Notes:
  - ***SamplePath*** is a vector of numbers of type double.
  - ***Class MCModel*** stores the parameters S(0), r and σ.
  - The pseudo-random number generator ***srand()*** is initialized using the argument passed as seed. For every different seed value used in a call to ***srand()***, the pseudo-random number generator can be expected to generate a different succession of results in the subsequent calls to ***rand()***. *www.cplusplus.com/reference/cstdlib/srand*
  - The call to ***time(NULL)*** returns the current calendar time (seconds since Jan 1, 1970), is used as the distinctive seed value for the random number generator ***srand().***
  - C++ does not have the constant $\pi$. We use $\pi$ = 4arctan(1) to generate it.
  - Independent ***Gauss()*** generates a sample of Z.
  - The member function ***GenerateSamplePath()*** generates a sample path referenced as ***S.*** In other words, the sample path ***S*** will be populated with m price points on one path.

## PathDepOption.h

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

## PathDepOption.h (continue)

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

## PathDepOption.cpp

```
#include "PathDepOption.h"
#include <cmath>
namespace fre {
   double PathDepOption::PriceByMC(const MCModel& Model, long N)
   {
      double H = 0.0, Hsq = 0.0, Heps = 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 = exp(-Model.GetR() * T) * H;
      return Price;
   }
```

## PathDepOption.cpp

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

Notes:

- ***PriceByMC()*** is the main pricing function, which can be shared by different types of path-dependent options.
- ***Payoff()*** is a pure virtual function. So PriceByMC() uses different implementation of Payoff(), depending on the derived classes of PathDepOption. The SamplePath variable, ***S***, is passed to the ***Payoff()*** function by reference to const.
- The class ***ArthmAsianCall*** is used for pricing arithmetic Asian calls. The virtual function Payoff() function is overridden in this class.
- The overridden virtual function ***Payoff()*** is invoked in the ***PriceByMC()*** via ***Polymorphism.***

## Notes (continue):

- Every object in C++ has access to its own address through an important pointer called ***this*** pointer. The ***this*** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking member functions:
  - **H = (i * H + Payoff(S)) / (i + 1.0) ⬄**
  - **H = (i * H + this->Payoff(S)) / (i + 1.0);**

## Main01.cpp

```
#include <iostream>
#include "PathDepOption.h"
using namespace std;
using namespace fre;
int main()
{
   double S0=100.0, r=0.03, sigma=0.2;
   MCModel Model(S0,r,sigma);
   double T =1.0/12.0, K=100.0;
   int m=30;
   ArthmAsianCall Option(T,K,m);
   long N=30000;
   Option.PriceByMC(Model,N);
   cout << "Arithmetic Asian Call by direct Monte Carlo = " << Option.GetPrice() << endl;
   return 0;
}
/*
Arithmetic Asian Call by direct Monte Carlo = 1.42557
*/
```

- Notes:
  - An object ***Model*** is initialized as an instance of ***MCModel.***
  - The object ***Option*** is initialized as an object of ***ArthmAsianCall***
  - Calling ***Option.PriceByMC(Model,N)*** executes the pricing function from the ***PathDepOption*** class.
  - What is the data type of ***this*** pointer? What is the address the ***this*** pointer is associated with?

# Assignment 1:

- ***Derive a class from the base class, PathDepOption, to compute the prices of European call and put options.***

# Pricing Error

- The price estimator $\hat{H}_N(0)$ depends on a sample, and hence contains an error .
- Unbiased estimator of the standard error of $\hat{H}_N(0)$

$$\hat{H}^i(T) = h(\hat{S}^i(t_1),\ldots,\hat{S}^i(t_m)), i = 1,\ldots,N$$

$$\hat{s}_N = \sqrt{\frac{1}{(N-1)} \sum_{i=1}^{N} (e^{-rT} \hat{H}^i(T) - \hat{H}_N(0))^2}$$

$$\hat{\sigma}_N = \frac{e^{-rT}}{\sqrt{N-1}} \sqrt{\frac{1}{N}\sum_{i=1}^{N} \hat{H}^i(T)^2 - \left(\frac{1}{N}\sum_{k=1}^{N} \hat{H}^i(T)\right)^2}$$

$$e^{rT}\sqrt{N-1}\hat{\sigma}_N \approx \sqrt{E(H(T)^2)-E(H(T))^2} = \sqrt{Var(H(T))}$$

$$\hat{\sigma}_N \approx \frac{e^{-rT}}{\sqrt{N-1}}\sqrt{Var(H(T))}$$

- $\hat{\sigma}_N$ converges to zero as we increase N, but this convergence is slow. For example

$$\hat{\sigma}_{100N} \approx \frac{1}{10}\hat{\sigma}_N$$

  - To reduce the error by one decimal point we need to make about 100 times more simulations.

## PathDepOption.h

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

## PathDepOption.h (Continue)

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

## PathDepOption.cpp

```
#include "PathDepOption.h"
#include <cmath>
namespace fre {
   double PathDepOption::PriceByMC(const MCModel& Model, long N)
   {   double H = 0.0, Hsq = 0.0, Heps = 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);
         Hsq = (i * Hsq + pow(Payoff(S), 2.0)) / (i + 1.0);
      }
      Price = exp(-Model.GetR() * T) * H;
      PricingError = exp(-Model.GetR() * T) * sqrt(Hsq - H * H) / sqrt(N - 1.0);
      return Price;
   }
   double ArthmAsianCall::Payoff(const SamplePath& S) const
   {   double Ave = 0.0;
      for (int k = 0; k < m; k++) Ave = (k * Ave + S[k]) / (k + 1.0);
      if (Ave < K) return 0.0;
      return Ave - K;
   }
}
```

- Notes:
  - A new member variable, ***PricingError***, is added to class ***PathDepOption.*** After executing ***Option.PriceByMC(Model, N)***, the price and price error will be stored inside of ***Option***.
  - ***Hsq*** is used to compute $\frac{1}{N}\sum_{i=1}^{N}\hat{H}^{i}(T)^{2}$

## Main02.cpp

```
#include <iostream>
#include "PathDepOption.h"
using namespace std;
using namespace fre;
int main()
{
   double S0=100.0, r=0.03, sigma=0.2;
   MCModel Model(S0,r,sigma);
   double T =1.0/12.0, K=100.0;
   int m=30;
   ArthmAsianCall Option(T,K,m);
   long N=30000;
   Option.PriceByMC(Model,N);
   cout << "Arithmetic Asian Call by direct Monte Carlo = " << Option.GetPrice() << endl
        << "Pricing Error = " << Option.GetPricingError() << endl;
   return 0;
}
/* Arithmetic Asian Call by direct Monte Carlo = 1.41748
   Pricing Error = 0.01197   */
```

## Greek Parameters

- Let u: $\Re$->$\Re$ be a function such that *H(0) = u(S(0)).*
- Assume that u(z) is differentiable,

  the Greek parameter ***delta*** is defined as:

$$\delta = \frac{du}{dz}(S(0))$$

- To compute $\delta$ we use the fact that for sufficiently small $\varepsilon$:

$$\frac{du}{dz}(S(0)) \approx \frac{u((1+\varepsilon)S(0)) - u(S(0))}{\varepsilon S(0)}$$

$$u((1+\varepsilon)S(0)) = e^{-rT}E_Q(h((1+\varepsilon)(S(t_1),\dots,S(t_m))))$$

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

$$\delta \approx \hat{\delta} = \frac{\hat{H}_{\varepsilon,N}(0) - \hat{H}_N(0)}{\varepsilon S(0)}$$

## PathDepOption.h

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

## PathDepOption.h (continue)

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

## PathDepOption.cpp

```
#include "PathDepOption.h"
#include <cmath>
namespace fre {
   double ArthmAsianCall::Payoff(const SamplePath& S) const
   {
      double Ave = 0.0;
      for (int k = 0; k < m; k++) Ave = (k * Ave + S[k]) / (k + 1.0);
      if (Ave < K) return 0.0;
      return Ave - K;
   }

   void Rescale(SamplePath& S, double x)
   {  int m = S.size();
      for (int j = 0; j < m; j++) S[j] = x * S[j];
   }
```

## PathDepOption.cpp (Continue)

```
double PathDepOption::PriceByMC(const MCModel& Model, long N, double epsilon)
{   double H = 0.0, Hsq = 0.0, Heps = 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);
        Hsq = (i * Hsq + pow(Payoff(S), 2.0)) / (i + 1.0);
        Rescale(S, 1.0 + epsilon);
        Heps = (i * Heps + Payoff(S)) / (i + 1.0);
    }
    Price = exp(-Model.GetR() * T) * H;
    PricingError = exp(-Model.GetR() * T) * sqrt(Hsq - H * H) / sqrt(N - 1.0);
    delta = exp(-Model.GetR() * T) * (Heps - H) / (Model.GetS0() * epsilon);
    return Price;
  }
}
```

## Main03.cpp

```
#include <iostream>
#include "PathDepOption.h"
using namespace std;
using namespace fre;
int main()
{  double S0=100.0, r=0.03, sigma=0.2;
   MCModel Model(S0,r,sigma);
   double T =1.0/12.0, K=100.0;
   int m=30;
   ArthmAsianCall Option(T,K,m);
   long N=30000;
   double epsilon =0.001;
   Option.PriceByMC(Model,N,epsilon);
   cout << "Arithmetic Asian Call by direct Monte Carlo = " << Option.GetPrice() << endl
      << "Error = " << Option.GetPricingError() << endl  << "delta = " << Option.GetDelta() << endl;
   return 0;
}
/*  Arithmetic Asian Call by direct Monte Carlo = 1.42246
Error = 0.0119462
delta = 0.52592  */
```

## Notes:

- We add $\delta$ as a member of class PathDepOption
- We add $\varepsilon$ as a variable of the function PriceByMC().
- ***Rescale()*** function multiplies a sample path by a number.
- ***Heps*** is used to compute, $\frac{1}{N}\sum_{i=1}^{N} h((1+\varepsilon)(\hat{S}^i(t_1),\ldots,\hat{S}^i(t_m)))$ hence we rescale the sample path.
- Note we use the ***same*** sample path for the computation of the price, pricing error, and $\delta$.
  - ***What is the benefits doing that?***

## Assignment 2:

- Using the fact that

$$\frac{d^2u}{dz^2}(S(0)) \approx \frac{u((1+\varepsilon)S(0)) - 2u(S(0)) + u((1-\varepsilon)S(0))}{(\varepsilon S(0))^2}$$

expand the code to compute the Greek parameter ***gamma***.

$$\gamma = \frac{d^2u}{dz^2}(S(0))$$

## Variance Reduction

- Employs an alternative estimator:
  - Unbiased
  - More deterministic
  - Yields a smaller variance without increasing the number of simulation
- Method: Control Variates
  - Suppose to compute E(x) using Monte Carlo
  - Assume a random variable ***Y close to X*** and E(Y) = y
  - E(X) = E(X-Y)+y
  - Using Monte Carlo to compute E(X-Y). Y is a ***Control Variate*** for X.

- To find the price of a path-dependent option with a payoff function: $R^m \to \Re$ using Control Variate method:
  - Suppose an option with a payoff function g: $R^m \to \Re$, which is close to ***h:***

    $\boldsymbol{G(T) = g(S(t_1),\ldots, S(t_m))}$ as the control variate for

    $\boldsymbol{H(T) = h(S(t_1),\ldots, S(t_m))}$
  - Assume G(0) could be computed analytically:

$$H(0) = e^{-rT} E_Q(H(T) - G(T)) + G(0)$$

## PathDepOption.h

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

## PathDepOption.h (continue)

```
class DifferenceOfOptions : public PathDepOption
{
private:
   PathDepOption* Ptr1;
   PathDepOption* Ptr2;
public:
   DifferenceOfOptions(double T_, double K_, int m_, PathDepOption* Ptr1_, PathDepOption* Ptr2_) :
                              PathDepOption(T_, K_, m_), Ptr1(Ptr1_), Ptr2(Ptr2_)
   { }
   double Payoff(const SamplePath& S) const
   {
      return Ptr1->Payoff(S) - Ptr2->Payoff(S);
   }
};
class ArthmAsianCall : public PathDepOption
{
public:
   ArthmAsianCall(double T_, double K_, int m_) : PathDepOption(T_, K_, m_) {}
   double Payoff(const SamplePath& S) const;
};
}
```

## PathDepOption.cpp

```
#include "PathDepOption.h"
#include "EurCall.h"
#include <cmath>
namespace fre {
    void Rescale(SamplePath& S, double x)
    {
        int m = S.size();
        for (int j = 0; j < m; j++) S[j] = x * S[j];
    }
```

## PathDepOption.cpp (Continue)

```
double PathDepOption::PriceByMC(const MCModel& Model, long N, double epsilon)
{
   double H = 0.0, Hsq = 0.0, Heps = 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);
      Hsq = (i * Hsq + pow(Payoff(S), 2.0)) / (i + 1.0);
      Rescale(S, 1.0 + epsilon);
      Heps = (i * Heps + Payoff(S)) / (i + 1.0);
   }
   Price = exp(-Model.GetR() * T) * H;
   PricingError = exp(-Model.GetR() * T) * sqrt(Hsq - H * H) / sqrt(N - 1.0);
   delta = exp(-Model.GetR() * T) * (Heps - H) / (Model.GetS0() * epsilon);
   return Price;
}
```

## PathDepOption.cpp (Continue)

```
double PathDepOption::PriceByVarRedMC(const MCModel& Model, long N,
                                      PathDepOption& CVOption, double epsilon)
{
   DifferenceOfOptions VarRedOpt(T, K, m, this, &CVOption);
   Price = VarRedOpt.PriceByMC(Model, N, epsilon) + CVOption.PriceByBSFormula(Model);
   PricingError = VarRedOpt.PricingError;
   return Price;
}

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

## Notes:

- ***double PriceByVarRedMC(BSModel Model, long N, PathDepOption& CVOption, double epsilon)*** is the pricing function computing H(0) using ***CVOption*** as control variate.
- ***virtual double PriceByBSFormula(BSModel Model) {return 0.0;}*** is the function computing G(0) from the Black-Scholes formula. The virtual function returns 0.0 to allow a derived class without implementing PriceByBSFormula().
  - ***Can this virtual function be pure virtual?***
- The class ***DifferenceOfOptions*** combines two options with payoffs H(T) and G(T), and creates an option with payoff H(T)-G(T). Pointers ***Ptr1*** and ***Ptr2*** refer to options with payoffs H(T) and G(T). The payoff is equal to H(T)-G(T).

- The keyword ***this*** is a pointer holding the address to a class object from which its member function is invoked:
  - *ArthmAsianCall Option(T,K,m);*
  - *Option.PriceByVarRedMC(Model,N,CVOption);*
  - *DifferenceOfOptions VarRedOpt(T,m,***this***,&CVOption);*
- In ***PriceByVarRedMC()***, $e^{-rT}E_Q(H(T)-G(T))$ is computing using Monte Carlo by ***VarRedOpt.PriceByMC().*** G(0) is computed in ***CVOption.PriceByBSFormula().***
- Since G(0) is computed analytically, the only source of error from the Monte Carlo computation is $e^{-rT}E_Q(H(T)-G(T))$

## Geometric Asian Option

- Choose a control variate and implement its PriceByBSFormula function:
  - Choose a geometric Asian call option with payoff function:

$$g(z_1,\ldots z_m) = h^{GeomAsianCall}(z_1,\ldots,z_m) = \left(\sqrt[m]{\prod_{k=1}^{m} Z_k} - K\right)^+$$

$$G(T) = h^{GeomAsianCall}(S(t_1),\ldots S(t_m)) = \left(ae^{\left(r-\frac{b^2}{2}\right)T + b\sqrt{T}Z} - K\right)^+$$

$$a = e^{-rT} S(0) \exp\left( \frac{(m+1)T}{2m} \left( r + \frac{\sigma^2}{2} \left( \frac{(2m+1)}{3m} - 1 \right) \right) \right)$$

$$b = \sigma \sqrt{\frac{(m+1)(2m+1)}{6m^2}}$$

- Where Z has the standard normal distribution N(0,1) and a, b ϵ $\Re$ are constants.
- G(T) can be regarded as a European call option on an asset with Time 0 price ***a*** and volatility ***b:***

***G(0) = C(a, K, T, b, r)***

## EurCall.h

```
#pragma once
namespace fre {
   class EurCall
   {
   private:
      double T, K;
      double d_plus(double S0, double sigma, double r);
      double d_minus(double S0, double sigma, double r);
   public:
      EurCall(double T_, double K_) : T(T_), K(K_) {}
      double PriceByBSFormula(double S0, double sigma, double r);
      double VegaByBSFormula(double S0, double sigma, double r);
      double DeltaByBSFormula(double S0, double sigma, double r);
   };
}
```

## EurCall.cpp

```
#include "EurCall.h"
#include <cmath>
namespace fre {
   double N(double x)
   {   double gamma = 0.2316419;     double a1 = 0.319381530;
      double a2 = -0.356563782;   double a3 = 1.781477937;
      double a4 = -1.821255978;   double a5 = 1.330274429;
      double pi = 4.0 * atan(1.0); double k = 1.0 / (1.0 + gamma * x);
      if (x >= 0.0)
      {       return 1.0 - ((((a5 * k + a4) * k + a3) * k + a2) * k + a1) * k * exp(-x * x / 2.0) / sqrt(2.0 * pi);
      }
      else return 1.0 - N(-x);
   }
   double EurCall::d_plus(double S0, double sigma, double r)
   {     return (log(S0 / K) + (r + 0.5 * pow(sigma, 2.0)) * T) / (sigma * sqrt(T));
   }
   double EurCall::d_minus(double S0, double sigma, double r)
   {     return d_plus(S0, sigma, r) - sigma * sqrt(T);
   }
```

## EurCall.cpp (continue)

```
double EurCall::PriceByBSFormula(double S0, double sigma, double r)
{
   return S0 * N(d_plus(S0, sigma, r)) - K * exp(-r * T) * N(d_minus(S0, sigma, r));
}

double EurCall::VegaByBSFormula(double S0, double sigma, double r)
{
   double pi = 4.0 * atan(1.0);
   return S0 * exp(-d_plus(S0, sigma, r) * d_plus(S0, sigma, r) / 2) * sqrt(T) / sqrt(2.0 * pi);
}

double EurCall::DeltaByBSFormula(double S0, double sigma, double r)
{
   return N(d_plus(S0, sigma, r));
}
}
```

# GmtrAsianCall.h

```
#pragma once
#include "PathDepOption.h"
namespace fre {
   class GmtrAsianCall : public PathDepOption
   {
   public:
      GmtrAsianCall(double T_, double K_, int m_) : PathDepOption(T_, K_, m_) {}
      double Payoff(const SamplePath& S) const;
      double PriceByBSFormula(const MCModel& Model);
   };
}
```

## GmtrAsianCall.cpp

```
#include "GmtrAsianCall.h"
#include "EurCall.h"
#include <cmath>
namespace fre {
   double GmtrAsianCall::Payoff(const SamplePath& S) const
   {
      double Prod = 1.0;
      for (int i = 0; i < m; i++)
      {
         Prod = Prod * S[i];
      }
      if (pow(Prod, 1.0 / m) < K) return 0.0;
      return pow(Prod, 1.0 / m) - K;
   }
```

# GmtrAsianCall.cpp (continue)

```
    double GmtrAsianCall::PriceByBSFormula(const MCModel& Model)
    {
      double a = exp(-Model.GetR() * T) * Model.GetS0() * exp((m + 1.0) * T / (2.0 * m) *
        (Model.GetR() + Model.GetSigma() * Model.GetSigma() * ((2.0 * m + 1.0) / (3.0 * m) - 1.0) / 2.0));
      double b = Model.GetSigma() * sqrt((m + 1.0) * (2.0 * m + 1.0) / (6.0 * m * m));
      EurCall G(T, K);
      Price = G.PriceByBSFormula(a, b, Model.GetR());
      return Price;
    }
}
```

# Main04.cpp

```
#include <iostream>
#include "PathDepOption.h"
#include "GmtrAsianCall.h"
using namespace std;
using namespace fre;
int main()
{
   double S0=100.0, r=0.03, sigma=0.2;
   MCModel Model(S0,r,sigma);
   double T =1.0/12.0, K=100.0;
   int m=30;
   ArthmAsianCall Option(T,K,m);
   GmtrAsianCall  CVOption(T,K,m);
   long N=30000;
   double epsilon =0.001;
   Option.PriceByVarRedMC(Model,N,CVOption,epsilon);
   cout << "Arithmetic call price = " << Option.GetPrice() << endl
        << "Error = " << Option.GetPricingError() << endl << endl;
```

# Main04.cpp (continue)

```
   Option.PriceByMC(Model,N,epsilon);
   cout << "Price by direct MC = " << Option.GetPrice() << endl
        << "Error = " << Option.GetPricingError() << endl << endl;

   return 0;
}
/*
Arithmetic call price = 1.42588
Error = 0.000136653

Price by direct MC = 1.42856
Error = 0.0120379

*/
```

**Notes:**

- ***Option*** is the arithmetic Asian call we are going to price. ***Cvoption*** is the control variate. We price ***Option*** using the variance reduction technique.
- We also price ***Option*** with the standard Monte Carlo method and display the error for comparison. In our example, the standard error of ***PriceByVarRedMC()*** is about 100 times smaller than the standard error of ***PriceByMC()***.
- ***The error has been reduced without increasing N.***

## Using Control Variates for Greeks

- Let $u_H(z)$ and $u_G(z)$ denote functions such that $H(0) = u_H(S(0))$ and $G(0)= u_G(S(0))$

$$\delta_H = \frac{du_H}{dz}(S(0)), \delta_G = \frac{du_G}{dz}(S(0)), \delta_{H-G} = \frac{d(u_H - u_G)}{dz}(S(0)).$$

- Suppose that we know how to compute $\delta_G$ analytically.
- Since $\delta_H = \delta_{H\text{-}G} + \delta_G$, we can compute $\delta_{H\text{-}G}$ using Monte Carlo and thus obtain $\delta_H$.

# Assignment 3:

- Expand the code we learned for variance reduction to compute $\delta$ for an arithmetic Asian call. Use the fact that the $\delta$ of the geometric Asian call is $N(d_+^{a,b})\dfrac{a}{S(0)}$, *where*

$$d_+^{a,b} = \frac{\ln\dfrac{a}{K} + (r + \dfrac{b^2}{2})T}{b\sqrt{T}}$$

# References


- Numerical Methods in Finance with C++ (Mastering Mathematical Finance), by Maciej J. Capinski and Tomasz Zastawniak, Cambridge University Press, 2012, ISBN-10: 0521177162
- *Financial Instrument Pricing Using C++, Daniel J. Duffy, ISBN 0470855096, Wiley, 2004.*