**Fall 2020 FRE-GY 6883 Financial Computing Midterm Exam**

Last Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_First Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_NYUID:\_\_\_ \_\_\_ \_\_\_ \_\_\_\_\_\_\_\_\_\_

**Q1: Implements the overloaded Swap() functions to be used in the main() function to produce the expected results shown as the following. Your implementation must satisfy the main function invoking it.**

|  |
| --- |
| **Implement Swap () function** |
| #include <iostream>  using namespace std;  void Swap(int\* x, int\* y)  {  int temp;  temp = \*x;  \*x = \*y;  \*y = temp;  }  void Swap(double& x, double& y)  {  double temp = x;  x = y;  y = temp;  } |
| int main(void)  {  int iA = 1, iB = 2;  double dA = 10.12, dB = 20.23;  cout << "before swap: " << " iA= " << iA << " iB= " << iB << endl;  Swap(&iA, &iB);  cout << "after swap:" << " iA= " << iA << " iB= " << iB << endl;  cout << "before swap: " << " dA= " << dA << " dB= " << dB << endl;  Swap(dA, dB);  cout << "after swap:" << " dA= " << dA << " dB= " << dB << endl;    return 0;  }  /\*  before swap: iA= 1 iB= 2  after swap: iA= 2 iB= 1  before swap: dA= 10.12 dB= 20.23  after swap: dA= 20.23 dB= 10.12  \*/ |

**Q2-Q3 are based on the following classes and independent functions.**

|  |
| --- |
| **Q2: Complete the implementation of constructor and destructor for base class EurOption:** |
| BinModel.h |
| class BinModel  {  private:  double S0;  double U;  double D;  double R;  public:  BinModel() : S0(0), U(0), D(0), R(0) {}  BinModel(double S0\_, double U\_, double D\_, double R\_) : S0(S0\_), U(U\_), D(D\_), R(R\_) {}  BinModel(const BinModel & binModel) : S0(binModel.S0), U(binModel.U), D(binModel.D), R(binModel.R) {}  ~BinModel() {}  double GetS0() const { return S0; }  double GetU() const { return U; }  double GetD() const { return D; }  double GetR() const { return R; }  double RiskNeutProb() const;  double CalculateAssetPrice(int n, int i) const;  void UpdateBinModel(double S0\_, double U\_, double D\_, double R\_);  }; |
| BinModel.cpp |
| #include "BinModel.h"  #include <cmath>  double BinModel::RiskNeutProb() const  { return (R-D)/(U-D);  }  double BinModel::CalculateAssetPrice(int n, int i) const  { return S0\*pow(1+U, i)\*pow(1+D, n-i);  }  void BinModel::UpdateBinModel(double S0\_, double U\_, double D\_, double R\_)  { S0 = S0\_;  U = U\_;  D = D\_;  R = R\_;  } |
| Options.h |
| #include <vector>  #include "BinModel.h"  using namespace std;  class EurOption  {  protected:  int N;  char\* name;  public:  EurOption(int N\_, char const \*name\_);  virtual ~EurOption();  virtual void GetInputData() = 0;  virtual double Payoff(double z) = 0;  double PriceByCRR(BinModel Model);  char\* GetName() { return name; }  };  class Call : public EurOption  {  private:  double K;  public:  Call(int N\_, char const\* name\_);  ~Call() {}  void GetInputData();  double Payoff(double z);  };  class Put : public EurOption  {  private:  double K;  public:  Put(int N\_, char const\* name\_);  ~Put() {}  void GetInputData();  double Payoff(double z);  };  class BullSpread : public EurOption  {  private:  double K1, K2;  public:  BullSpread(int N\_, char const\* name\_);  ~BullSpread() {}  void GetInputData();  double Payoff(double z);  };  class BearSpread : public EurOption  {  private:  double K1, K2;  public:  BearSpread(int N\_, char const\* name\_);  ~BearSpread() {}  void GetInputData();  double Payoff(double z);  }; |
| Options.cpp |
| #include "Options.h"  #include "BinModel.h"  #include <iostream>  #include <cmath>  #include <string.h>  using namespace std;  **// Complete the implementation of constructor**  EurOption::EurOption(int N\_, char const \*name\_):N(N\_)  {name = new char[strlen(name\_) + 1];  strcpy(name, name\_);  }  **// Complete the implementation of destructor**  EurOption::~EurOption()  {  if (name != NULL)  delete[] name;  name = NULL;  }  double EurOption::PriceByCRR(BinModel Model)  {  double q = Model.RiskNeutProb();  vector<double> Price(N + 1);  for (int i = 0; i <= N; i++)  {  Price[i] = Payoff(Model.CalculateAssetPrice(N, i));  }  for (int n = N - 1; n >= 0; n--)  {  for (int i = 0; i <= n; i++)  Price[i] = (q \* Price[i + 1] + (1 - q) \* Price[i]) / (1 + Model.GetR());  }  return Price[0];  }  Call::Call(int N\_, char const\* name\_):EurOption(N\_, name\_),K(0)  {  }  void Call::GetInputData()  {  cout << "Enter strike price: ";  cin >> K;  }  double Call::Payoff(double z)  {  if (z > K) return z - K;  return 0.0;  }  Put::Put(int N\_, char const\* name\_):EurOption(N\_, name\_),K(0)  {  }  void Put::GetInputData()  {  cout << "Enter strike price: ";  cin >> K;  }  double Put::Payoff(double z)  {  if (z < K) return K - z;  return 0.0;  }  BullSpread::BullSpread(int N\_, char const\* name\_):EurOption(N\_, name\_),K1(0),K2(0)  {  }  void BullSpread::GetInputData()  {  cout << "Enter strike price 1: ";  cin >> K1;  cout << "Enter strike price 2: ";  cin >> K2;  }  double BullSpread::Payoff(double z)  {  if (K2 <= z) return K2 - K1;  else if (K1 < z) return z - K1;  return 0.0;  }  BearSpread::BearSpread(int N\_, char const\* name\_):EurOption(N\_, name\_),K1(0),K2(0)  {  }  void BearSpread::GetInputData()  {  cout << "Enter strike price 1: ";  cin >> K1;  cout << "Enter strike price 2: ";  cin >> K2;  cout << endl;  }  double BearSpread::Payoff(double z)  {  if (K2 <= z) return 0.0;  else if (K1 < z) return K2 - z;  return K2 - K1;  } |
| **Q3: Implement the independent function OptionCalculation() to be invoked by the main function and produce the expected result as the following.** |
| #include "BinModel.h"  #include "Options.h"  #include <iostream>  #include <cmath>  using namespace std;  **// Complete the implementation of independent function OptionCalculation**  **void OptionCalculation(T & Option, BinModel& Model)**  **{**  **Option.GetInputData();**  **cout<< "Option Price" << Option.PriceByCRR(Model) << endl;**  **}**  int main()  {  double S0 = 106.00, K = 100.00;  double U = 0.15125, D = -0.13138, R = 0.00545;  BinModel Model(S0, U, D, R);  int N = 8;  char name[10];  memset(name, '\0', 10);  strcpy(name, "Call");  Call Option1(N, name);  OptionCalculation<Call>(Option1, Model);  memset(name, '\0', 10);  strcpy(name, "Put");  Put Option2(N, name);  OptionCalculation<Put>(Option2, Model);  memset(name, '\0', 10);  BullSpread Option3(N, "BullSpread");  OptionCalculation<BullSpread>(Option3, Model);  memset(name, '\0', 10);  BearSpread Option4(N, "BearSpread");  OptionCalculation<BearSpread>(Option4, Model);  return 0;  }  /\*  Enter strike price: 100  Option Call price = 21.6811  Enter strike price: 100  Option Put price = 11.4261  Enter strike price 1: 100  Enter strike price 2: 110  Option BullSpread price = 4.71584  Enter strike price 1: 100  Enter strike price 2: 110  Option BearSpread price = 4.85866  \*/ |

**Q4-Q5 are based on the following implantation of DefInt class.**

|  |
| --- |
| **Q4: Complete the implementation of the public member functions for DefIntCalculation, which will be used in main() to calculate the numerical approximation based on Trapezoid and Simpson.** |
| #include <iostream>  using namespace std;  class DefInt  {  protected:  int N;  double a, b;  double (\*f)(double x);  public:  DefInt(int N\_, double a\_, double b\_, double (\*f\_)(double x)) :N(N\_), a(a\_), b(b\_), f(f\_) {}  virtual ~DefInt() {}  virtual double ByNumApproximation(void) = 0;  };  class Trapezoid : public DefInt  {  private:  double ByTrapezoid(void);  public:  Trapezoid(int N\_, double a\_, double b\_, double(\*f\_)(double x)) :DefInt(N\_, a\_, b\_, f\_) {}  ~Trapezoid() {}  double ByNumApproximation(void)  {  return ByTrapezoid();  }  };  class Simpson : public DefInt  {  private:  double BySimpson(void);  public:  Simpson(int N\_, double a\_, double b\_, double (\*f\_)(double x)) :DefInt(N\_, a\_, b\_, f\_) {}  ~Simpson() {}  double ByNumApproximation(void)  {  return BySimpson();  }  };  double Trapezoid::ByTrapezoid(void)  {  cout << "ByTrapezoid: ";  double h = (b - a) / N;  double Result = 0.5 \* f(a);  for (int n = 1; n < N; n++) Result += f(a + n \* h);  Result += 0.5 \* f(b);  return Result \* h;  }  double Simpson::BySimpson(void)  {  cout << "BySimpson: ";  double h = (b - a) / N;  double Result = f(a);  for (int n = 1; n < N; n++)  Result += 4 \* f(a + n \* h - 0.5 \* h) + 2 \* f(a + n \* h);  Result += 4 \* f(b - 0.5 \* h) + f(b);  return Result \* h / 6;  }  **// Complete the implementation of a constructor and the member function**  **// CalculateDefInt() of the following class DefIntCalculation.**  **// The constructor and the member function will be used in the main function.**  class DefIntCalculation  {  private:  DefInt\* ptr;  public:  **// Implement a constructor**  **DefIntCalculation(DefInt \* ptr\_)**  **{**  **ptr = ptr\_;**  **}**  **// Complete the implementation of the following member function**  **double CalculateDefInt (void)**  {  return ptr->ByNumApproximation();  }  }; |
| **Q5. Create objects for DefIntCalculation and use its member function CalculateDefInt to calculate numerical approximation for defInt1 and defInt2. You are not allowed to directly invoke the member function ByNumApproximation in main() function.** |
| double f(double x) { return x \* x \* x - x \* x + 1; }  int main()  {  double a = 1.0;  double b = 2.0;  int N = 1000;  Trapezoid defInt1(N, a, b, f);  Simpson defInt2(N, a, b, f);  return 0;  }  /\*  ByTrapezoid: 2.41667  BySimpson: 2.41667  \*/ |

**Q6-Q7 are based on the following implantation of AmOption class.**

|  |
| --- |
| **Q6: Complete the implementation of the member function PriceBySnell().** |
| BinModel02.h |
| #ifndef BinModel02\_h  #define BinModel02\_h  class BinModel  {  private:  double S0;  double U;  double D;  double R;  public:  BinModel(double S0\_, double U\_, double D\_, double R\_) : S0(S0\_), U(U\_), D(D\_), R(R\_) {}  double GetS0() const { return S0; }  double GetU() const { return U; }  double GetD() const { return D; }  double GetR() const { return R; }  double RiskNeutProb() const;  double S(int n, int i) const;  };  #endif |
| BinModel02.cpp |
| #include "BinModel02.h"  #include <iostream>  #include <cmath>  using namespace std;  double BinModel::RiskNeutProb() const  { return (R-D)/(U-D);  }  double BinModel::S(int n, int i) const  { return S0\*pow(1+U,i)\*pow(1+D,n-i);  } |
| Option08.h |
| #ifndef Options08\_h  #define Options08\_h  #include "BinModel02.h"  enum OptionType { European, American };  class Option  {protected:  int N;  OptionType type;  public:  void SetN(int N\_){N=N\_;}  int GetN(){return N;}  void SetType(OptionType type\_) { type = type\_; }  OptionType GetType() { return type; }  virtual ~Option() {}  virtual void GetInputData() = 0;  virtual double Payoff(double z)=0;  virtual double GetPrice(const BinModel & model) = 0;  };  class EurOption: public virtual Option  {public:  double PriceByCRR(const BinModel & Model);  };  class AmOption: public virtual Option  {public:  double PriceBySnell(const BinModel & Model);  };  class Call: public EurOption, public AmOption  {private:  double K;  public:  void SetK(double K\_){K=K\_;}  void GetInputData();  double Payoff(double z);  double GetPrice(const BinModel & Model);  };  class Put: public EurOption, public AmOption  {private:  double K;  public:  void SetK(double K\_){K=K\_;}  void GetInputData();  double Payoff(double z);  double GetPrice(const BinModel& Mode);  };  #endif |
| Option08.cpp |
| #include "Options08.h"  #include "BinModel02.h"  #include <iostream>  #include <cmath>  #include <vector>  using namespace std;  double EurOption::PriceByCRR(const BinModel & Model)  {  double q=Model.RiskNeutProb();  int N=GetN();  vector<double> Price(N+1);  for (int i=0; i<=N; i++)  {  Price[i]=Payoff(Model.S(N,i));  }  for (int n=N-1; n>=0; n--)  {  for (int i=0; i<=n; i++)  {  Price[i]=(q\*Price[i+1]+(1-q)\*Price[i])/(1+Model.GetR());  }  }  return Price[0];  }  **// Complete the member function PriceBySnell() for AmOption**  **double AmOption::PriceBySnell(const BinModel & Model)**  **{**  **}**  double Call::Payoff(double z)  {  if (z>K) return z-K;  return 0.0;  }  void Call::GetInputData()  {  cout << "Enter call option strike price: ";  cin >> K;  }  double Call::GetPrice(const BinModel & Model)  {  if (type == European)  return PriceByCRR(Model);  else  return PriceBySnell(Model);  }  double Put::Payoff(double z)  {  if (z<K) return K-z;  return 0.0;  }  void Put::GetInputData()  {  cout << "Enter put option strike price: ";  cin >> K;  }  double Put::GetPrice(const BinModel& Model)  {  if (type == European)  return PriceByCRR(Model);  else  return PriceBySnell(Model);  } |
| **Q7: Complete the logic inside the for loop to calculate the price for American Call and Put. You are requested to use the iterator for optionMap to complete the task. You are not allowed to use map key directly to access options. You are not allowed to create any object inside the for loop either.** |
| Main.cpp |
| #include "BinModel02.h"  #include "Options08.h"  #include <iostream>  #include <string>  #include <map>  using namespace std;  int main()  {  double S0 = 106.00, K = 100.00;  double U = 0.15125, D = -0.13138, R = 0.00545;  BinModel Model(S0, U, D, R);    int N = 8;  map<string, Option\*> optionMap;  optionMap["Call"] = new Call();  optionMap["Put"] = new Put();  map<string, Option\*>::iterator itr ;  // Complete implementation of the for loop to produce the following output  **for (itr = optionMap.begin(); itr != optionMap.end(); itr++)**  **{**  **itr->second->SetN(N);**  **itr->second->GetInputData();**  **itr->second->SetType(American);**  **itr->first;**  **cout << itr->second->GetPrice(Model) << endl;**  **}**  return 0;  }  /\*  Enter call option strike price: 100  American Call Option Price 21.6811  Enter put option strike price: 100  American Put Option Price 11.724  \*/ |

**Q8. Implement the following overloaded operator for Matrix class, to satisfy the main function.**

|  |
| --- |
| #include<iostream>  #include<stdlib.h>  #define MAX 3  using namespace std;  class Matrix  { private:  int Mat[MAX][MAX];  public:  Matrix() {  for(int i=0;i<MAX;i++)  for(int j=0;j<MAX;j++)  Mat[i][j]=0; }  Matrix(int a) {  for(int i=0;i<MAX;i++)  for(int j=0;j<MAX;j++)  Mat[i][j]=a++; }  ~Matrix() { }  // **Implement a member function for overloading the addition operator for Matrix**  **Matrix operator+ ( const Matrix & V)**  **{**  **Matrix m3;**  **for (int i = 0; i < MAX; i++)**  **for (int j = 0; j < MAX; j++)**  **m3.Mat[i][j] = Mat[i][j]+ V.Mat[i][j];**  **return m3;**  **}**  void Display() {  for(int i=0;i<MAX;i++)  { for(int j=0;j<MAX;j++)  cout << Mat[i][j] << " ";  cout << endl;  }  cout << endl; }  };  int main()  { Matrix m1(1), m2(2), m3;  **m3 = m1 + m2; // Using overloading addition operator for Matrix**  cout << "Matrix m1 is: " << endl;  m1.Display();  cout << "Matrix m2 is: " << endl;  m2.Display();  cout << "Matrix m1 + m2 is: " << endl;  m3.Display();  return 0;  }  /\*  Matrix m1 is:  1 2 3  4 5 6  7 8 9  Matrix m2 is:  2 3 4  5 6 7  8 9 10  Matrix m1 + m2 is:  3 5 7  9 11 13  15 17 19  \*/} |

**Q9. Complete the definition of member functions of the Intermediary class to connect the following Bond class and the class Function in Solver02 to satisfy all the function calls in the main function.**

|  |  |
| --- | --- |
| #include "Solver02.h"  #include <vector>  #include <cmath>  #include <iostream>  #include <iomanip>  using namespace std;  class Bond  { protected:  double F; //face value  double T; //maturity time  vector<double> C; //coupons  vector<double> t; //coupon times  public:  Bond(double F\_, double T\_, vector<double>& C\_, vector<double>& t\_) {F=F\_; T=T\_; C=C\_; t=t\_;}  double BondValue(double y) {  double P=0;  for (unsigned int n=0; n<C.size(); n++)  P+=C[n]\*exp(-y\*t[n]);  P+=F\*exp(-y\*T);  return P;  }  double BondDeriv(double y) {  double D=0;  for (unsigned int n=0; n<C.size(); n++)  D+=-C[n]\*t[n]\*exp(-y\*t[n]);  D+=-F\*T\*exp(-y\*T);  return D;  }  };  **class Intermediary : public Bond, public Function**  **{**  **public:**  **Intermediary(double F\_, double T\_, vector<double>& C\_, vector<double>& t\_);**  **double Value(double y);**  **double Deriv(double y);**  **};**  **// Complete the definition of the member functions for the above Intermediary class**  **Intermediary::Intermediary(double F\_, double T\_, vector<double>& C\_, vector<double>& t\_):Bond(F\_, T\_, C\_, t\_){}**  **double Intermediary::Value(double y)**  **{**  **return BondValue(y);**  **}**  **double Intermediary::Deriv(double y)**  **{**  **return BondDeriv(y);**  **}** | |
| int main()  {  double F=100.0; //face value  double T=3.0; //maturity time  vector<double> C; //coupons  C.push\_back(1.2); C.push\_back(1.2); C.push\_back(1.2);  vector<double> t; //coupon times  t.push\_back(1.0); t.push\_back(2.0); t.push\_back(3.0);  **Intermediary MyBond(F,T,C,t);**  double P=98.56;  double Acc=0.0001;  double y;  cout << setiosflags(ios::fixed) << setprecision(4);  cout << "F = " << F << endl;  cout << "T = " << T << endl;  cout << "coupons: " << endl;  for (unsigned int n=0; n<C.size(); n++)  cout << "C" << n << " = " << C[n] << " " <<endl;  cout << "tenors: " << endl;  for (unsigned int n=0; n<t.size(); n++)  cout << "T" << n << " = " << t[n] << " " <<endl;  cout << "P = " << P << endl << endl;  double LEnd=0.0;  double REnd=1.0;  **y=SolveByBisect(&MyBond,P,LEnd,REnd,Acc);**  cout << "Yield by bisection method: " << y << endl;  double Guess=0.2;  **y=SolveByNR(&MyBond,P,Guess,Acc);**  cout << "Yield by Newton-Raphson method: " << y << endl;  return 0;  } | | |
| Solver02.h |  | |
| #ifndef Solver02\_h  #define Solver02\_h  class Function  { public:  virtual double Value(double x)=0;  virtual double Deriv(double x)=0;  };  double SolveByBisect(Function\* Fct, double Tgt, double LEnd, double REnd, double Acc)  {  double left=LEnd, right=REnd, mid=(left+right)/2;  double y\_left=Fct->Value(left)-Tgt,  y\_mid=Fct->Value(mid)-Tgt;  while (mid-left>Acc)  {  if ((y\_left>0 && y\_mid>0)||(y\_left<0 && y\_mid<0))  { left=mid;  y\_left=y\_mid;  }  else right=mid;  mid=(left+right)/2;  y\_mid=Fct->Value(mid)-Tgt;  }  return mid;  } | double SolveByNR(Function\* Fct, double Tgt, double Guess, double Acc)  {  double x\_prev=Guess;  double x\_next=x\_prev  -(Fct->Value(x\_prev)-Tgt)/Fct->Deriv(x\_prev);  while (x\_next-x\_prev>Acc || x\_prev-x\_next>Acc)  {  x\_prev=x\_next;  x\_next=x\_prev  - (Fct->Value(x\_prev)-Tgt)/Fct->Deriv(x\_prev);  }  return x\_next;  } | |

**Q10. Complete the member functions PriceByBSFormula() and DeltaByBSFormula() of the GmtrAsianCall for the following program in order to calculate option price and delta for Arithmetic Asian Call based on Variance Reduction Monto Carlo procedure.**

The delta of the geometric Asian call could be calculated as:

****

|  |
| --- |
| **MCModel.h** |
| #pragma once  #include <vector>  #include <cstdlib>  #include <ctime>  using namespace std;  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;  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>  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];  }  } |
| **EurCall.h** |
| #ifndef EurCall\_h  #define EurCall\_h  class EurCall  {  public:  double T, K;  EurCall(double T\_, double K\_){T=T\_; K=K\_;}  double d\_plus(double S0, double sigma, double r);  double d\_minus(double S0, double sigma, double r);  double PriceByBSFormula(double S0, double sigma, double r);  double VegaByBSFormula(double S0, double sigma, double r);  double DeltaByBSFormula(double S0, double sigma, double r);  };  #endif |
| **PathDepOption.h** |
| #pragma once  #include "MCModel.h"  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 double Payoff(SamplePath& S) = 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; }  virtual double DeltaByBSFormula(const MCModel & Model) { return 0.0; }  double GetT() { return T; }  double GetPrice() { return Price; }  double GetPricingError() { return PricingError; }  double GetDelta() { return delta; }  };  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(SamplePath& S)  {  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(SamplePath& S);  }; |
| **GmtrAsianCall.h** |
| #include "EurCall.h"  #include "PathDepOption.h"  #include "MCModel.h"  class GmtrAsianCall : public PathDepOption  {  private:  double a, b;  void Calculate\_a\_b(const MCModel& Model);  public:  GmtrAsianCall(double T\_, double K\_, int m\_) :PathDepOption(T\_, K\_, m\_), a(0.0), b(0.0) {}  double Payoff(SamplePath& S);  double PriceByBSFormula(const MCModel & Model);  double DeltaByBSFormula(const MCModel & Model);  }; |
| **EurCall.cpp** |
| #include "EurCall.h"  #include <cmath>  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);  }  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));  } |
| **PathDepOption.cpp** |
| #include "PathDepOption.h"  #include "MCModel.h"  #include <cmath>  void Rescale(SamplePath& S, double epsilon)  {  int m=S.size();  for (int j=0; j<m; j++) S[j] = (1.0+epsilon)\*S[j];  }  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,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;  }  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);  delta = VarRedOpt.delta + CVOption.DeltaByBSFormula(Model);  PricingError = VarRedOpt.PricingError;  return Price;  }  double ArthmAsianCall::Payoff(SamplePath& S)  {  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;  } |
| **main.cpp** |
| #include <iostream>  #include "PathDepOption.h"  #include "GmtrAsianCall.h"  using namespace std;  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  << "delta = " << Option.GetDelta() << endl << endl;  Option.PriceByMC(Model,N,epsilon);  cout << "Price by direct MC = " << Option.GetPrice() << endl  << "Error = " << Option.GetPricingError() << endl  << "delta = " << Option.GetDelta() << endl;  return 0;  }  /\*  Arithmetic call price = 1.42595  Error = 0.000137061  delta = 0.520043  Price by direct MC = 1.42307  Error = 0.0120812  delta = 0.52239  \*/ |
| **GmtrAsianCall.cpp** |
| #include "GmtrAsianCall.h"  double GmtrAsianCall::Payoff(SamplePath& S)  {  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;  }  void GmtrAsianCall::Calculate\_a\_b(const MCModel& Model)  {  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));  b = Model.GetSigma() \* sqrt((m + 1.0) \* (2.0 \* m + 1.0) / (6.0 \* m \* m));  }  **// Complete the implementation of the member function, PriceByBSFormula**  double GmtrAsianCall::PriceByBSFormula(const MCModel& Model)  {  Calculate\_a\_b(Model);  **EurCall G(T, K);**  **Price = G.PriceByBSFormula(a, b, Model.GetR());**  **return Price;**  }  **// Complete the implementation of the member function, DeltaByBSFormula**  double GmtrAsianCall::DeltaByBSFormula(const MCModel& Model)  {  Calculate\_a\_b(Model);  EurCall G(T, K);  delta = G.DeltaByBSFormula(a, b, Model.GetR() \* a / Model.GetS0());  return delta;  } |