**Spring 2021 FRE-GY 6883 Financial Computing (Tuesday)**

Last Name: \_\_\_\_\_\_\_\_\_Wang\_\_\_\_\_\_\_\_\_\_\_\_First Name: \_\_\_\_\_\_\_\_Zhiheng\_\_\_\_\_\_\_\_\_NYUID:\_\_\_ \_N10802389\_\_\_\_\_\_\_

**Q1: Implements the following Swap() function to swap two values. Your implementation must satisfy the main function invoking it.**

|  |
| --- |
| **Implement Swap() function** |
| **#include <iostream>**  **using namespace std;**  template<class Type>  void Swap(Type &int1, Type\* ptr2)  {  Type tmp = int1;  int1 = \*ptr2;  \*ptr2 = tmp;  } |
| **The main function** |
| int main()  {  int a = 1, b = -2;  Swap<int>(a, &b);  cout << "After swap " << "a = " << a << " and b = " << b << endl;  double x = 1.2, y = -2.3;  Swap<double>(x, &y);  cout << "After swap " << "x = " << x << " and y = " << y << endl;  return 0;  }  /\*  After swap a = -2 and b = 1  After swap x = -2.3 and y = 1.2  \*/ |

**Q2-Q3 are based on the following classes and independent functions (you could not use any class or function not specified for Q2 to Q3).**

|  |
| --- |
| BinModel.h |
| #pragma once  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;  } |
| **Q2: Complete the implementation of constructors and destructor for class Option:** |
| Options.h |
| #pragma once  #include "BinModel.h"  #include <vector>  using namespace std;  class Option  {  protected:  int N;  char\* pName;  vector<double> K;  public:  **// Complete the implementation of the follow two constructors**  Option(int N\_, double K\_, const char \* pName\_) : N(N\_)  {  int i;  for(i = 0 ; pName\_[i]!='\0'; ++i);  pName = new char[i+1];  \*pName = \*pName\_;  K.push\_back(K\_);  }  Option(int N\_, double K1\_, double K2\_, const char \*pName\_) : N(N\_)  {  int i;  for(i = 0 ; pName\_[i]!='\0'; ++i);  pName = new char[i+1];  \*pName = \*pName\_;  K.push\_back(K1\_);  K.push\_back(K2\_);    }  int GetN() const { return N; }  const char \* GetName() const { return pName; }  virtual double Payoff(double z) const = 0;  virtual ~Option() = 0;  };  class Call : public Option  {  public:  Call(int N\_, double K\_, const char \* pName\_) :Option(N\_, K\_, pName\_) {}  ~Call() {}  double Payoff(double z) const;  };  class DigitCall : public Call  {  public:  DigitCall(int N\_, double K\_, const char \*pName\_) :Call(N\_, K\_, pName\_) { }  ~DigitCall() { }  double Payoff(double z) const;  };  class Put : public Option  {  public:  Put(int N\_, double K\_, const char\* pName\_) :Option(N\_, K\_, pName\_) {}  ~Put() {}  double Payoff(double z) const;  };  class DigitPut : public Put  {  public:  DigitPut(int N\_, double K\_, const char\* pName\_) :Put(N\_, K\_, pName\_) { }  ~DigitPut() { }  double Payoff(double z) const;  };  class BullSpread : public Option  {  public:  BullSpread(int N\_, double K1\_, double K2\_, const char\* pName\_) :Option(N\_, K1\_, K2\_, pName\_) { }  ~BullSpread() { }  double Payoff(double z) const;  };  class BearSpread : public Option  {  public:  BearSpread(int N\_, double K1\_, double K2\_, const char \* pName\_) :Option(N\_, K1\_, K2\_, pName\_) { }  ~BearSpread() { }  double Payoff(double z) const;  };  class StrangleOpt : public Option  {  public:  StrangleOpt(int N\_, double K1\_, double K2\_, const char \* pName\_) :Option(N\_, K1\_, K2\_, pName\_) { }  ~StrangleOpt() {}  double Payoff(double z) const;  };  class ButterflyOpt : public Option  {  public:  ButterflyOpt(int N\_, double K1\_, double K2\_, const char \* pName\_) :Option(N\_, K1\_, K2\_, pName\_) { }  ~ButterflyOpt() {}  double Payoff(double z) const;  };  class DoubDigitOpt : public Option  {  public:  DoubDigitOpt(int N\_, double K1\_, double K2\_, const char \*pName\_) :Option(N\_, K1\_, K2\_, pName\_) { }  ~DoubDigitOpt() { }  double Payoff(double z) const;  };  class OptionCalculation  {  public:  double PriceByCRR(const BinModel& rModel, const Option \* pOption);  double PriceBySnell(const BinModel& rModel, const Option \*pOption);  }; |
| Options.cpp |
| #include "BinModel.h"  #include "Options.h"  **// Complete the implementation of the following destructor**  Option::~Option()  {  delete pName;  pName = NULL;    }  double Call::Payoff(double z) const  {  if (z > K[0] ) return z - K[0];  return 0.0;  }  double DigitCall::Payoff(double z) const  {  if (z > K[0]) return 1.0;  return 0.0;  }  double Put::Payoff(double z) const  {  if (z < K[0]) return K[0] - z;  return 0.0;  }  double DigitPut::Payoff(double z) const  {  if (z < K[0]) return 1.0;  return 0.0;  }  double BullSpread::Payoff(double z) const  {  if (K[1] <= z) return K[1] - K[0];  else if (K[0] < z) return z - K[0];  return 0.0;  }  double BearSpread::Payoff(double z) const  {  if (K[1] <= z) return 0.0;  else if (K[0] < z) return K[1] - z;  return K[1] - K[0];  }  double StrangleOpt::Payoff(double z) const  {  if (z <= K[0]) return K[0] - z;  else if (K[1] < z) return z - K[1];  return 0.0;  }  double ButterflyOpt::Payoff(double z) const  {  if (K[0] < z && z <= (K[0] + K[1]) / 2) return z - K[0];  else if ((K[0] + K[1]) / 2 < z && z <= K[1]) return K[1] - z;  return 0.0;  }  double DoubDigitOpt::Payoff(double z) const  {  if (K[0] < z && z < K[1]) return 1.0;  return 0.0;  }  double OptionCalculation::PriceByCRR(const BinModel& binModel, const Option \*pOption)  {  double q = binModel.RiskNeutProb();  int N = pOption->GetN();  vector<double> optionPrices(N+1);  for (int i = 0; i <= N; i++)  {  optionPrices[i] = pOption->Payoff(binModel.CalculateAssetPrice(N, i));  }  for (int n = N - 1; n >= 0; n--)  {  for (int i = 0; i <= n; i++)  {  optionPrices[i] = (q \* optionPrices[i + 1] + (1 - q) \* optionPrices[i]) / (1 + binModel.GetR());  }  }  return optionPrices[0];  }  double OptionCalculation::PriceBySnell(const BinModel& binModel, const Option \*pOption)  {  double q = binModel.RiskNeutProb();  int N = pOption->GetN();  vector<double> optionPrices(N + 1);  double ContVal;  for (int i = 0; i <= N; i++)  {  optionPrices[i] = pOption->Payoff(binModel.CalculateAssetPrice(N, i));  }  for (int n = N - 1; n >= 0; n--)  {  for (int i = 0; i <= n; i++)  {  ContVal = (q \* optionPrices[i + 1] + (1 - q) \* optionPrices[i]) / (1 + binModel.GetR());  optionPrices[i] = pOption->Payoff(binModel.CalculateAssetPrice(n, i));  if (ContVal > optionPrices[i])  {  optionPrices[i] = ContVal;  }  }  }  return optionPrices[0];  } |
| **Q3: Complete the implementation of the rest tasks for the main function. The output from the main function is listed after the function.** |
| #include <iostream>  #include <iomanip>  #include "BinModel.h"  #include "Options.h"  using namespace std;  int main()  {  int N = 8;  double S0 = 106.00;  double U = 0.15125, D = -0.13138, R = 0.00545;  double K = 100.00;  double K1 = 100, K2 = 110;  BinModel Model(S0, U, D, R);  Option\* pOptions[9];  OptionCalculation optionCalculation;  double optionPrice = 0;  pOptions[0] = new Call(N, K, "Call");  pOptions[1] = new Put(N, K, "Put");  pOptions[2] = new DigitCall(N, K, "DigitCall");  pOptions[3] = new DigitPut(N, K, "DigitPut");  pOptions[4] = new BullSpread(N, K1, K2, "BullSpread");  pOptions[5] = new BearSpread(N, K1, K2, "BearSpread");  pOptions[6] = new StrangleOpt(N, K1, K2, "Strangle");  pOptions[7] = new ButterflyOpt(N, K1, K2, "Butterfly");  pOptions[8] = new DoubDigitOpt(N, K1, K2, "DoubleDigit");    **// Complete the rest of tasks for the main function**  cout << fixed << setprecision(2) <<"European Call Option price = " << optionCalculation.PriceByCRR(Model,pOptions[0]) << endl;  cout << "European Put Option price = " << optionCalculation.PriceByCRR(Model,pOptions[1]) << endl;  cout << "European DigitCall Option price = " << optionCalculation.PriceByCRR(Model,pOptions[2]) << endl;  cout << "European DigitPut Option price = " << optionCalculation.PriceByCRR(Model,pOptions[3]) << endl;  cout << "European BullSpread Option price = " << optionCalculation.PriceByCRR(Model,pOptions[4]) << endl;  cout << "European BearSpread Option price = " << optionCalculation.PriceByCRR(Model,pOptions[5]) << endl;  cout << "European Strangle Option price = " << optionCalculation.PriceByCRR(Model,pOptions[6]) << endl;  cout << "European Butterfly Option price = " << optionCalculation.PriceByCRR(Model,pOptions[7]) << endl;  cout << "European DoubleDigit Option price = " << optionCalculation.PriceByCRR(Model,pOptions[8]) << endl;  return 0;  }  /\*  European Call Option price = 21.68  European Put Option price = 11.43  European DigitCall Option price = 0.58  European DigitPut Option price = 0.38  European BullSpread Option price = 4.72  European BearSpread Option price = 4.86  European Strangle Option price = 28.39  European Butterfly Option price = 1.04  European DoubleDigit Option price = 0.26  \*/ |

**Q4: The binomial model can be employed to approximate the Black-Scholes model. One of several possible approximation schemes is the following. Divide the time interval [0, T] into *N* steps of length *h = T/N*, and set the parameters of the binomial model to be**

**where σ is the volatility and *r* is the continuously compounded interest rate in the Black-Scholes model.**

**Create the class BSModel as a derived class of BinModel to complete the following implementation of computing the approximate price for an American call option in the Black-Scholes model by means of this binomial tree approximation.**

|  |
| --- |
| **BSModel.h** |
| #pragma once  #include <cmath>  using namespace std;  class BinModel  {  protected:  double S0;  double U;  double D;  double R;  public:  BinModel()  {  S0 = 0; U = 0; D = 0; R = 0;  }  virtual ~BinModel() {}  double RiskNeutProb();  double S(int n, int i);  double GetR() { return R; }  };  **// Complete the implementation of BSModel as a derived class of BinModel**  class BSModel : public BinModel { public:  double r, sigma, h;  BSModel(double S0\_, double r\_, double sigma\_, double h\_) : r(r\_),sigma(sigma\_),h(h\_){  S0 = S0\_;  U = exp(sigma \* sqrt(h)) - 1;  D = 1 / (1 + U) - 1;  R = exp(r \* h) - 1;  }  ~BSModel(){} }; |
| **BSModel.cpp** |
| #include "BSModel.h"  #include <cmath>  using namespace std;  double BinModel::RiskNeutProb()  {  return (R-D)/(U-D);  }  double BinModel::S(int n, int i)  {  return S0\*pow(1+U,i)\*pow(1+D,n-i);  } |
| **BinLattice02.h** |
| #pragma once  #include <iostream>  #include <iomanip>  #include <vector>  using namespace std;  template<typename Type> class BinLattice  {  private:  int N;  vector< vector<Type> > Lattice;  public:  void SetN(int N\_)  {  N=N\_;  Lattice.resize(N+1);  for(int n=0; n<=N; n++) Lattice[n].resize(n+1);  }  void SetNode(int n, int i, Type x)  {Lattice[n][i]=x;}  Type GetNode(int n, int i)  {return Lattice[n][i];}  void Display()  {  cout << setiosflags(ios::fixed)  << setprecision(3);  for(int n=0; n<=N; n++)  {  for(int i=0; i<=n; i++)  cout << setw(7) << GetNode(n,i);  cout << endl;  }  cout << endl;  }  }; |
| **Option09.h** |
| #pragma once  #include "BinLattice02.h"  #include "BSModel.h"  class Option  {  private:  int N; //steps to expiry  public:  void SetN(int N\_){N=N\_;}  int GetN(){return N;}  virtual double Payoff(double z)=0;  };  class EurOption: public virtual Option  {  public:  //pricing European option  double PriceByCRR(BinModel & Model);  };  class AmOption: public virtual Option  {  public:  //pricing American option  double PriceBySnell(BinModel & Model,  BinLattice<double>& PriceTree,  BinLattice<bool>& StoppingTree);  };  class Call: public EurOption, public AmOption  {  private:  double K; //strike price  public:  Call(int N\_, double K\_){SetN(N\_); K=K\_;}  double Payoff(double z);  };  class Put: public EurOption, public AmOption  {  private:  double K; //strike price  public:  Put(int N\_, double K\_){SetN(N\_); K=K\_;}  double Payoff(double z);  }; |
| **Option09.cpp** |
| #include "Options09.h"  double EurOption::PriceByCRR(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];  }  double AmOption::PriceBySnell(BinModel & Model,  BinLattice<double>& PriceTree,  BinLattice<bool>& StoppingTree)  {  double q=Model.RiskNeutProb();  int N=GetN();  PriceTree.SetN(N);  StoppingTree.SetN(N);  double ContVal;  for (int i=0; i<=N; i++)  {  PriceTree.SetNode(N,i,Payoff(Model.S(N,i)));  StoppingTree.SetNode(N,i,1);  }  for (int n=N-1; n>=0; n--)  {  for (int i=0; i<=n; i++)  {  ContVal=(q\*PriceTree.GetNode(n+1,i+1)+(1-q)\*PriceTree.GetNode(n+1,i))/(1+Model.GetR());  PriceTree.SetNode(n,i,Payoff(Model.S(n,i)));  StoppingTree.SetNode(n,i,1);  if (ContVal>PriceTree.GetNode(n,i))  {  PriceTree.SetNode(n,i,ContVal);  StoppingTree.SetNode(n,i,0);  }  else if (PriceTree.GetNode(n,i)==0.0)  {  StoppingTree.SetNode(n,i,0);  }  }  }  return PriceTree.GetNode(0,0);  }  double Call::Payoff(double z)  {  if (z>K) return z-K;  return 0.0;  }  double Put::Payoff(double z)  {  if (z<K) return K-z;  return 0.0;  } |
| **main.cpp** |
| #include "BSModel.h"  #include "Options09.h"  #include <iostream>  using namespace std;  int main()  {  double S0=106.0;  double r=0.058;  double sigma=0.46;  double T=(double)9/12;  double K=100.0;  int N=8;  cout << setiosflags(ios::fixed) << setprecision(5);  double h = T / N;  BSModel Model(S0,r,sigma,h);    Call Option(N,K);  BinLattice<double> PriceTree;  BinLattice<bool> StoppingTree;  Option.PriceBySnell(Model,PriceTree,StoppingTree);  cout << "American call option price = " << PriceTree.GetNode(0,0) << endl;  return 0;  }  /\*  American call option price = 21.68171  \*/ |

**Q5-Q7 are based on the following classes and independent functions (you could not use any class or function not specified for Q5 to Q7).**

|  |
| --- |
| **Functions.h:** |
| #pragma once  class Function  {  public:  virtual double Value(double x) = 0;  virtual double Deriv(double x) = 0;  };  double F1(double x);  double DF1(double x); |
| **Functions.cpp:** |
| #include "Functions.h"  double F1(double x) { return x \* x - 2; }  double DF1(double x) { return 2 \* x; } |
| **NonlinerSolver.h:** |
| #pragma once  #include "Functions.h"  class NonlinearSolver  {  private:  double Tgt;  double LEnd;  double REnd;  double Acc;  double Guess;  public:  NonlinearSolver(double Tgt\_, double LEnd\_, double REnd\_, double Acc\_, double Guess\_)  {  Tgt = Tgt\_;  LEnd = LEnd\_;  REnd = REnd\_;  Acc = Acc\_;  Guess = Guess\_;  }  virtual ~NonlinearSolver() {}  double SolveByBisect(double (\*Fct)(double x));  double SolveByNR(double (\*Fct)(double x), double (\*DFct)(double x));  double SolveByBisect(Function\* Fct);  double SolveByNR(Function\* Fct);  void UpdateSolver(double Tgt\_, double LEnd\_, double REnd\_, double Acc\_, double Guess\_)  {  Tgt = Tgt\_;  LEnd = LEnd\_;  REnd = REnd\_;  Acc = Acc\_;  Guess = Guess\_;  }  }; |
| **NonlinerSolver.cpp:** |
| #include "NonlinearSolver.h"  double NonlinearSolver::SolveByBisect(double (\*Fct)(double x))  {  double left = LEnd, right = REnd, mid = (left + right) / 2;  double y\_left = Fct(left) - Tgt, y\_mid = Fct(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(mid) - Tgt;  }  return mid;  }  double NonlinearSolver::SolveByNR(double (\*Fct)(double x), double (\*DFct)(double x))  {  double x\_prev = Guess;  double x\_next = x\_prev - (Fct(x\_prev) - Tgt) / DFct(x\_prev);  while (x\_next - x\_prev > Acc || x\_prev - x\_next > Acc)  {  x\_prev = x\_next;  x\_next = x\_prev - (Fct(x\_prev) - Tgt) / DFct(x\_prev);  }  return x\_next;  }  double NonlinearSolver::SolveByBisect(Function\* Fct)  {  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 NonlinearSolver::SolveByNR(Function\* Fct)  {  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;  } |
| **Q5: Complete the implementation of the following Bond.h, so the class Bond can be used in the following main function.** |
| **Bond.h** |
| #pragma once  #include <vector>  #include "Functions.h"  using namespace std;  class Bond : public Function{ private:  double F;  int T;  double yield;  vector<double> C;  vector<double> t; public:  Bond(double F\_, double T\_, vector<double>& C\_, const vector<double> t\_):F(F\_),T(T\_),yield(0.0){  for (unsigned int n = 0; n < C\_.size(); n++){  C.push\_back(C\_.at(n));  t.push\_back(t\_.at(n));  }  }  double Value(double y);  double Deriv(double y); }; |
| **Bond.cpp** |
| #include "Bond.h"  #include <cmath>  double Bond::Value(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 Bond::Deriv(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;  } |
| **Q6: Complete the implement of EurCall.h to create the class used in main function for calculating the implied volatility.** |
| **EurCall.h** |
| #pragma once  #include "NonlinearSolver.h"  class EurCall  {  private:  double T, K;  public:  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);  };  class Intermediary : public EurCall{ private:  double S0;  double r; public:  Intermediary(double S0\_, double r\_, double T\_, double K\_) : EurCall(T\_,K\_){  S0=S0\_;  r=r\_;  }  double Value(double sigma);  double Deriv(double sigma); }; |
| **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 Intermediary::Value(double sigma)  {  return PriceByBSFormula(S0, sigma, r);  }  double Intermediary::Deriv(double sigma)  {  return VegaByBSFormula(S0, sigma, r);  } |
| **Q7: Complete the implementation of main function. The output from the main function is listed after the function. Your implementation must provide the same result as the listed output.**  **You could not use any other class or function except the ones declared and defined above. In addition, you are not allowed to create any new object in the main function.** |
| **main.cpp** |
| #include <iostream>  #include "Functions.h"  #include "NonlinearSolver.h"  #include "Bond.h"  #include "EurCall.h"  using namespace std;  int main()  {  double Acc1 = 0.001;  double LEnd1 = 0.0, REnd1 = 2.0;  double Tgt1 = 0.0;  double Guess1 = 1.0;  NonlinearSolver solver(Tgt1, LEnd1, REnd1, Acc1, Guess1);  **// Complete the missing codes in the main function**  **// to created expected results list as the following:**  double BondF = 100.0;  double BondT = 3.0;  vector<double> C;  C.push\_back(1.2); C.push\_back(1.2); C.push\_back(1.2);  vector<double> t;  t.push\_back(1.0); t.push\_back(2.0); t.push\_back(3.0);  Bond MyBond(BondF, BondT, C, t);  double Tgt2 = 98.56;  double Acc2 = 0.0001;  double LEnd2 = 0.0, REnd2 = 1.0;  double Guess2 = 0.2;  double S0 = 100.0;  double r = 0.1;  double CallT = 1.0;  double K = 100.0;  Intermediary Call(S0, r, CallT, K);  double Acc3 = 0.001;  double LEnd3 = 0.01, REnd3 = 1.0;  double Tgt3 = 12.56;  double Guess3 = 0.23;  solver.UpdateSolver(Tgt2,LEnd2,REnd2,Acc2,Guess2); solver.UpdateSolver(Tgt3,LEnd3,REnd3,Acc3,Guess3);  cout << "Root of F1 by Bisect: " << solver.SolveByBisect(F1) << endl; cout << "Root of F1 by Newton-Raphson: " << solver.SolveByNR(F1,DF1) << endl; cout << "Yield by Bisection method: " << solver.SolveByBisect(&MyBond) << endl; cout << "Yield by Newton-Raphson method: " << solver.SolveByNR(&MyBond) << endl; cout << "Implied Volatility by Bisect: " << solver.SolveByBisect(&Call) << endl; cout << "Implied Volatility by Newton-Raphson: " << solver.SolveByNR(&Call) << endl;  return 0;  }  /\*  Root of F1 by Bisect: 1.41504  Root of F1 by Newton-Raphson: 1.41421  Yield by Bisection method: 0.0167847  Yield by Newton-Raphson method: 0.0168215  Implied Volatility by Bisect: 0.179189  Implied Volatility by Newton-Raphson: 0.17839  \*/ |

**Q8. Overload the operation addition for class Rectangle to add two Rectangle objects shown in the following main function.**

|  |
| --- |
| #include <iostream>  using namespace std;  class Rectangle  {  private:  int length, width;  public:  Rectangle() :length(0), width(0) {}  Rectangle(int length\_, int width\_) :length(length\_), width(width\_) {}  Rectangle(const Rectangle& R) :length(R.length), width(R.width) {}  ~Rectangle() {}  int GetLength() const  {  return length;  }  int GetWidth() const  {  return width;  }  int Area()  {  return length \* width;  }  };  **// overload addition operator to add lengths of two rectangles and**  **// widths of two rectangles to create a new rectangle**  Rectangle operator+(const Rectangle& rec1, const Rectangle& rec2) {  return Rectangle(rec1.GetLength() + rec2.GetLength(),rec1.GetWidth()+rec2.GetWidth()); }  int main(void)  {  Rectangle rect1(5, 10), rect2(15, 20);  Rectangle rect3 = rect1 + rect2;  cout << "The length of rect3 = " << rect3.GetLength() << ", the width of rect3 = " << rect3.GetWidth() << endl;  cout << "Area of rect3 = " << rect3.Area() << endl;  return 0;  }  /\*  The length of rect3 = 20, the width of rect3 = 30  Area of rect3 = 600  \*/ |

**Q9.** **Complete the following implementation for using PriceByVarRedMC for calculating option price and delta for Arithmetic Asian Call.**

|  |
| --- |
| **EurCall.h** |
| #pragma once  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);  }; |
| **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));  } |
| **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];  }  } |
| **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 ~PathDepOption() {}  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);  }; |
| **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;  } |
| **Complete the implementation of class GmtrAsianCall shown as the following. You must use pEurCall in the class member functions for your calculation.** |
| **GmtrAsianCall.h:** |
| #pragma once  #include "EurCall.h"  #include "PathDepOption.h"  #include "MCModel.h"  class GmtrAsianCall : public PathDepOption  {  private:  double a, b;  void Calculate\_a\_b(const MCModel& Model);  EurCall\* pEurCall;  public:  GmtrAsianCall(double T\_, double K\_, int m\_): PathDepOption(T\_, K\_, m\_) {}  ~GmtrAsianCall() {  delete[] pEurCall;  pEurCall = NULL; }  double Payoff(SamplePath& S);  double PriceByBSFormula(const MCModel & Model);  double DeltaByBSFormula(const MCModel & Model);  }; |
| **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));  }  double GmtrAsianCall::PriceByBSFormula(const MCModel & Model){  Calculate\_a\_b(Model);  Price = pEurCall->PriceByBSFormula(a,b,Model.GetR());  return Price; }  double GmtrAsianCall::DeltaByBSFormula(const MCModel & Model){  Calculate\_a\_b(Model);  delta = pEurCall->DeltaByBSFormula(a,b,(Model.GetR()\*a)/Model.GetS0());  return delta; } |
| **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  \*/ |

**Q10. Complete the following implementation of calculating of delta and gamma based on Monto Carlo Simulation.**









|  |
| --- |
| void Rescale(SamplePath& S, double x)  {  int m=S.size();  for (int j=0; j<m; j++) S[j] = x\*S[j];  }  double PathDepOption::PriceByMC(BSModel & Model, long N, double epsilon)  {  double H=0.0, Hsq=0.0, Heps=0.0, Hmeps=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);  **// Complete the implementation of Heps and Hmeps**  Heps = (i / (i + 1.0)) \* Heps + Payoff(S) / (i + 1.0);  Rescale(S, (1.0 - epsilon) / (1.0 + epsilon));  Hgam = (i / (i + 1.0)) \* Hgam + Payoff(S) / (i + 1.0);  }  Price = exp(-Model.r\*T)\*H;  PricingError = exp(-Model.r\*T)\*sqrt(Hsq-H\*H)/sqrt(N-1.0);  **// Complete the implementation of delta and gamma.**  **// Assume delta and gamma are already declared.**  delta = exp(-Model.r \* T) \* (Heps - H) / (Model.S0 \* epsilon);  gamma = exp(-Model.r \* T) \* (Heps + Hgam - 2 \* H) / pow(Model.S0 \* epsilon, 2);    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;  } |