Numerical Methods in Finance with C++

Maciej J. Capiński and Tomasz Zastawniak

Solutions to Exercises

Chapter 1

1.1. The line

```
int n=3; int i=2;
```

in `Main02.cpp` needs to be changed to

```
int n,i;
cout << "Enter n: "; cin >> n;
cout << "Enter i: "; cin >> i;
```

The details can be found in

`Exe1-1_Main02.cpp`

1.2. Here is a function that interchanges the contents of two variables of type `double`:

```
void interchange(double& x, double& y)
{
   double z;
   z=x;
   x=y;
   y=z;
   return;
}
```

This function can be called by the statement

```
interchange(a,b);
```

where `a`, `b` are variables of type `double`. See

`Exe1-2_Main.cpp`

1.3. The `for` loops in `PriceByCRR()` in `Options01.cpp` can be replaced by

```
int i=0;
while (i<=N)
{
   Price[i]=CallPayoff(S(S0,U,D,N,i),K);
   i++;
}
int n=N-1;
while (n>=0)
{
   int i=0;
   while (i<=n)
   {
      Price[i]=(q*Price[i+1]+(1-q)*Price[i])/(1+R);
      i++;
   }
   n--;
}
```

This can be found in the file

`Exe1-3_Options01.cpp`

1.4. The lines

```
if (N<=0 || K<=0)
{
   cout << "Illegal data ranges" << endl;
   cout << "Terminating program" << endl;
   return 1;
}
```

need to be inserted before `return 0;` in the body of `GetInputData()` in `Options01.cpp`, and the line calling the function in `main()` in `Main05.cpp` should be replaced by

```
if (GetInputData(N,K)==1) return 1;
```

See

`Exe1-4_Options01.cpp`
`Exe1-4_Main05.cpp`

1.5. The `PriceByCRR()` function in `Options01.cpp` can be modified as follows to compute the time 0 price of a European option using the Cox–Ross–Rubinstein formula:

```
double PriceByCRR(double S0, double U, double D,
                  double R, int N, double K)
{
   double p=RiskNeutProb(U,D,R);
   double Price=0;
   for (int n=0; n<=N; n++)
   {
       Price=Price
             +NewtonSymb(N,n)*pow(p,n)*pow(1-p,N-n)
             *CallPayoff(S(S0,U,D,N,n),K)/pow(1+R,N);
   }
   return Price;
}
```

The following function also needs to be added to compute the Newton symbol $\frac{N!}{i!(N-i)!}$:

```
double NewtonSymb(int N, int n)
{
   if (n<0 || n>N) return 0;
   double result=1;
   for (int i=1; i<=n; i++) result=result*(N-n+i)/i;
   return result;
}
```

The complete code is in

`Exe1-5_Options01.cpp`

1.6. The following function uses the bubble sort method to order a sequence of numbers:

```
void bubblesort(double a[], int N)
{
   for (int i=1; i<N; i++)
   {
      for (int j=1; j<=N-i; j++)
      {
         if (a[j-1]>a[j]) interchange(a[j-1],a[j]);
```

```
        }
    }
}
```

This function can be called by

```
bubblesort(Sequence,SequenceLength);
```

where `Sequence` is an array of type `double` and `SequenceLength` a variable of type `int`. See

```
Exe1-6_Main.cpp
```

1.7. The following modification of the function `interchange()` from Exercise 1.2 accepts two pointers to variables of type `double` as arguments instead of having the arguments passed by reference:

```
void interchange(double* px, double* py)
{
   double z;
   z=*px;
   *px=*py;
   *py=z;
   return;
}
```

When `a`, `b` are variables of type `double`, then this function can be called by

```
interchange(&a,&b);
```

In the code for `bubblesort()` in Exercise 1.6 the line calling the `interchange()` function can be replaced by

```
if (a[j-1]>a[j]) interchange(a+j-1,a+j);
```

The full code can be found in

```
Exe1-7_Main.cpp
```

1.8. See

```
Exe1-8_BinModel01.h
Exe1-8_BinModel01.cpp
Exe1-8_Main04.cpp
```

1.9. To include the ability to price digital calls we insert the lines

```
//computing digital call payoff
double DigitalCallPayoff(double z, double K);
```

in `Options03.h` and

```
double DigitalCallPayoff(double z, double K)
{
   if (z>K) return 1.0;
   return 0.0;
}
```

in `Options03.cpp`. We can compute the price of such an option by adding the following lines in the `main()` function in `Main07.cpp`:

```
cout << "Enter digital call option data:" << endl;
GetInputData(N,K);
cout << "European digital call option price =  "
     << PriceByCRR(S0,U,D,R,N,K,DigitalCallPayoff)
     << endl << endl;
```

See

```
Exe1-9_Options03.h
Exe1-9_Options03.cpp
Exe1-9_Main07.cpp
```

1.10. The payoff function for a digital put option can be placed in new files. For details, see

```
Exe1-10_DigitalPut.h
Exe1-10_DigitalPut.cpp
```

The `Main07.cpp` file should be modified by inserting the line

```
#include "Exe1-10_DigitalPut.h"
```

at the beginning of the file, and the lines

```
cout << "Enter digital put option data:" << endl;
GetInputData(N,K);
cout << "European digital put option price =  "
     << PriceByCRR(S0,U,D,R,N,K,DigitalPutPayoff)
     << endl << endl;
```

in the body of the `main()` function. See

```
Exe1-10_Main07.cpp
```

1.11. The strike price `K` can be replaced by an array to contain the parameters of the payoff function. The details can be found in

```
Exe1-11_Options03.h
Exe1-11_Options03.cpp
Exe1-11_Main07.cpp
```

Chapter 2

2.1. The code is implemented and tested by computing the integral $\int_1^2 (x^3 - x^2 + 1)dx$ in

```
Exe2-1_Main.cpp
```

2.2. The code for the `BullSpread` and `BearSpread` classes is given in

```
Exe2-2_Options06.h
Exe2-2_Options06.cpp
```

These classes are used to price a bull spread and a bear spread in

```
Exe2-2_Main10.cpp
```

2.3 See

```
Exe2-3_Main.cpp
```

2.4 The classes to price strangle and butterfly spreads are included in the files

```
Exe2-4_SpreadOptions.h
Exe2-4_SpreadOptions.cpp
```

and used in

```
Exe2-4_Main11.cpp
```

Chapter 3

3.1 The code for computing the replicating strategy for a European option can be found in

```
Exe3-1_Options09.h
Exe3-1_Options09.cpp
Exe3-1_Main14.cpp
```

Note that the strategy as given by (3.3), (3.4) is a predictable process indexed by $n = 1, \ldots, N$. However, in the code the numbering is shifted by 1, with $n = 0, \ldots, N-1$.

3.2 A new class `BSModel` is developed for the Black–Scholes model in

```
Exe3-2_BSModel.h
```

It contains the function `ApproxBinModel()` to set up the approximating binomial model. In the files

```
Exe3-2_BinModel02.h
Exe3-2_BinModel02.cpp
Exe3-2_Options09.h
Exe3-2_Options09.cpp
```

manual data entry has been replaced by constructor functions to initiate the parameter values. In

```
Exe3-2_Main.cpp
```

after setting the values of all parameters, we compute the binomial model `ApproxModel` approximating the Black–Scholes model `Model`, and then compute the price of an American put using the Snell algorithm in this binomial model.

3.3 A function template designed to interchange the contents of two variables of arbitrary type can be designed by adapting the function from Exercise 1.2:

```
template<typename Type>
   void interchange(Type& x, Type& y)
{
   Type z;
   z=x;
   x=y;
   y=z;
   return;
}
```

The code in

`Exe3-3_Main.cpp`

uses this function template with variables of type `double` and `int`.

Chapter 4

4.1 See

`Exe4-1_Solver03.h`
`Exe4-1_Main17.cpp`

4.2 In

`Exe4-2_Main.cpp`

we define a class called `Bond` which contains the `Value()` function to compute the bond price from (4.5) and the `Deriv()` function to compute the derivative of the bond price (4.5) with respect to the yield y. An object of the `Bond` class is then initiated and passed to the solvers to compute the yield.

4.3 See

`Exe4-3_Main.cpp`

4.4 `Option` becomes a template parameter rather than a class. The classes `EurOption` and `AmOption` are disposed of. What remains from them are the `PriceByCRR()` and `PriceBySnell()` functions, which become function templates, and are therefore moved from the `.cpp` file to the header file. There are also a few cosmetic changes, as `N` needs to be moved into the `Call` and `Put` classes. Full details can be found in

`Exe4-4_Options09.h`
`Exe4-4_Options09.cpp`
`Exe4-4_Main14.cpp`

Chapter 5

5.1 Two subclasses `EurCall` and `EurPut` of the `PathDepOption` class are added to the files from Listings 5.3, 5.4. See

`Exe5-1_PathDepOption01.h`
`Exe5-1_PathDepOption01.cpp`

These classes are put to work in

`Exe5-1_Main19.cpp`

5.2 The function

`PathDepOption::PriceByMC()`

from Listing 5.9 is expanded to include the computation of γ. See

`Exe5-2_Main21.cpp`
`Exe5-2_PathDepOption03.h`
`Exe5-2_PathDepOption03.cpp`

5.3 Based on a sample path $(\hat{S}(t_1), \ldots, \hat{S}(t_m))$, we can compute a vector $Z = (Z_1, \ldots, Z_m)$ satisfying

$$\hat{S}(t_1) = S(0)e^{\left(r-\frac{\sigma^2}{2}\right)t_1+\sigma\sqrt{t_1}Z_1}$$
$$\hat{S}(t_i) = \hat{S}(t_{i-1})\, e^{\left(r-\frac{\sigma^2}{2}\right)(t_i-t_{i-1})+\sigma\sqrt{t_i-t_{i-1}}Z_i} \qquad \text{for } i = 2, \ldots, m.$$

It is given by

$$Z_1 = \frac{\ln\frac{\hat{S}(t_1)}{S(0)} - \left(r-\frac{\sigma^2}{2}\right)t_1}{\sigma\sqrt{t_1}},$$
$$Z_i = \frac{\ln\frac{\hat{S}(t_i)}{\hat{S}(t_{i-1})} - \left(r-\frac{\sigma^2}{2}\right)(t_i-t_{i-1})}{\sigma\sqrt{t_i-t_{i-1}}} \qquad \text{for } i = 2, \ldots, m,$$

and computed by `GetZ()` from the `Exe5-3_PathDepOption03.cpp` file.

The function `Rescale()` from the `Exe5-3_PathDepOption03.cpp` file computes a rescaled sample path

$$\left(\bar{S}_1, \ldots, \bar{S}_m\right),$$

given by

$$\bar{S}_1 = \bar{S}_1(\bar{S}(0), \bar{r}, \bar{\sigma}, \bar{\tau}) = \bar{S}(0)e^{\left(\bar{r}-\frac{\bar{\sigma}^2}{2}\right)\sqrt{\bar{\tau}}+\bar{\sigma}\sqrt{\bar{\tau}}Z_1},$$
$$\bar{S}_i = \bar{S}_i(\bar{S}(0), \bar{r}, \bar{\sigma}, \bar{\tau}) = \bar{S}_{i-1}e^{\left(\bar{r}-\frac{\bar{\sigma}^2}{2}\right)\sqrt{\bar{\tau}}+\bar{\sigma}\sqrt{\bar{\tau}}Z_i} \qquad \text{for } i = 2, \ldots, m.$$

We can use such rescaled paths to compute the Greeks using the same method as for δ in Listing 5.9. See

`Exe5-3_Main21.cpp`
`Exe5-3_PathDepOption03.h`
`Exe5-3_PathDepOption03.cpp`

5.4 See

`Exe5-4_Main.cpp`

5.5 The `EurCall` class from Listing 4.7 is expanded to include a function `DeltaByBSFormula()` which computes

$$\delta = N(d_+).$$

A function `DeltaByBSFormula()` is added to the `GmtrAsianCall` class from Listing 5.12. It computes $\delta = N(d_+^{a,b})\frac{a}{S(0)}$. The $N(d_+^{a,b})$ is computed using the `DeltaByBSFormula()` function from the `EurCall` class.

Finally, a line

```
delta = VarRedOpt.delta
          + CVOption.DeltaByBSFormula(Model);
```

is added to the `PriceByVarRedMC()` function from Listing 5.11. It computes the δ of the option using

$$\delta_H = \delta_{H-G} + \delta_G.$$

See

`Exe5-5_EurCall.h`
`Exe5-5_EurCall.cpp`
`Exe5-5_GmtrAsianCall.h`
`Exe5-5_GmtrAsianCall.cpp`
`Exe5-5_Main22.cpp`
`Exe5-5_PathDepOption04.h`
`Exe5-5_PathDepOption04.cpp`

5.6 The files

`Exe5-6_BarrierCall.h`
`Exe5-6_BarrierCall.cpp`

contain a subclass `BarrierCall` of the `PathDepOption` class for the payoff

$$h^{\text{barrier call}}(z_1, \ldots, z_m) = \mathbf{1}_{\{\max_{k=1,\ldots,m} z_k \leq L\}} (z_m - K)^+ .$$

The `EurCall` class from Listing 4.8 is made a subclass of the `PathDepOption` class and equipped with a payoff on a sample path. The `EurCall` can thus play the role of a control variate for the `BarrierCall`. See

`Exe5-6_PathDepOption04.h`
`Exe5-6_PathDepOption04.cpp`

```
Exe5-6_BarrierCall.h
Exe5-6_BarrierCall.cpp
Exe5-6_EurCall.h
Exe5-6_EurCall.cpp
Exe5-6_Main22.cpp
```

5.7 It is enough to modify the function `PriceByBSFormula()` from the `GmtrAsianCall` class from Listings 5.12–5.13. The method will give correct results, provided that m is large enough for the Riemann sums to approximate the integrals. See

```
Exe5-7_Main22.cpp
Exe5-7_GmtrAsianCall.cpp
Exe5-7_GmtrAsianCall.h
```

5.8 We can use the code from Listings 5.6–5.7. Apart form including appropriate header files, no changes are required. See

```
Exe5-8_PathDepOption05.cpp
Exe5-8_PathDepOption05.h
Exe5-8_Main23.cpp
```

5.9 A new function `Rescale()`, which computes

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

is added to the code from Listing 5.20. The `PriceByMC()` function from the file

```
Exe5-9_PathDepOption05.cpp
```

is analogous to `PriceByMC()` from Listing 5.9. See

```
Exe5-9_Main23.cpp
Exe5-9_PathDepOption05.cpp
Exe5-9_PathDepOption05.h
```

5.10 It is enough to add a subclass `EurBasketCall` of the `PathDepOption` class. For details see

```
Exe5-10_Main23.cpp
```

`Exe5-10_EurBasket.h`
`Exe5-10_EurBasket.cpp`

5.11 A new class `SumOfCalls` is created in

`Exe5-11_SumOfCalls.h`
`Exe5-11_SumOfCalls.cpp`

to act as a control variate. In the `PriceByBSFormula()` function we compute the price of the control variate using a linear combination of prices of European calls of `EurCall` class.

The files

`Exe5-11_PathDepOption05.cpp`
`Exe5-11_PathDepOption05.h`

are identical to Listings 5.10–5.11, with the exception of including different header files. See also:

`Exe5-11_EurBasket.h`
`Exe5-11_EurBasket.cpp`
`Exe5-11_Main23.cpp`

Chapter 6

6.1 By put–call parity

$$h_u^{\text{call}}(t) - h_u^{\text{put}}(t) = z_u - Ke^{-r(T-t)},$$
$$h_l^{\text{call}}(t) - h_l^{\text{put}}(t) = z_l - Ke^{-r(T-t)}.$$

Substituting (6.7) and (6.8) from the book gives $h_u^{\text{call}}(t) = z_u - Ke^{-r(T-t)}$ and $h_l^{\text{call}}(t) = z_l$, respectively.

6.2 See

`Exe6-2_CallOption.cpp`
`Exe6-2_CallOption.h`

where a subclass `Call` of the `Option` class is added. This class is put to work in

`Exe6-2_Main24.cpp`

6.3 We compute

$$x(t,z) = \frac{\partial u}{\partial z}(t,z) \approx \frac{u(t,z+\Delta z) - u(t,z+\Delta z)}{2\Delta z},$$
$$y(t,z) = u(t,z) - x(t,z)z.$$

It is enough to add two functions:

```
double x(FDMethod* Method, double t, double S)
{
   return (Method->v(t,S+Method->dx)
      -Method->v(t,S-Method->dx))
       /(2.0*Method->dx);
}

double y(FDMethod* Method, double t, double S)
{
   return Method->v(t,S)-x(Method,t,S)*S;
}
```

to the `main()` function from Listing 6.10. See

`Exe6-3_Main24.cpp`

6.4 We add the function

```
int testStability(BSEq &BSPDE,Call &EuropeanCall, int
imax, int jmax)
```

to the `main()` function from Listing 6.10. It tests the stability of the explicit finite difference method by comparing the price with the price obtained from the Black–Scholes formula. See

`Exe6-4_CallOption.h.cpp`
`Exe6-4_CallOption.cpp`
`Exe6-4_Main24.cpp`

6.5 We add a subclass `ImplicitMethod` of the `ImplicitScheme` class in the file

`Exe6-5_ImplicitMethod.h`

See also

`Exe6-5_Main.cpp`

6.6 Substituting

$$\frac{\partial v(t_{i-\lambda}, x_j)}{\partial t} \approx \frac{v_{i,j} - v_{i-1,j}}{\Delta t},$$
$$\frac{\partial v(t_{i-\lambda}, x_j)}{\partial x} \approx \lambda \frac{v_{i-1,j+1} - v_{i-1,j-1}}{2\Delta x} + (1-\lambda)\frac{v_{i,j+1} - v_{i,j-1}}{2\Delta x},$$
$$\frac{\partial^2 v(t_{i-\lambda}, x_j)}{\partial x^2} \approx \lambda \frac{v_{i-1,j+1} - 2v_{i-1,j} + v_{i-1,j-1}}{\Delta x^2} + (1-\lambda)\frac{v_{i,j+1} - 2v_{i,j} + v_{i,j-1}}{\Delta x^2},$$
$$v(t_{i-\lambda}, x_j) \approx \lambda v_{i-1,j} + (1-\lambda)\, v_{i,j}.$$

into

$$\frac{\partial v(t,x)}{\partial t} = a(t,x)\frac{\partial^2 v(t,x)}{\partial x^2} + b(t,x)\frac{\partial v(t,x)}{\partial x} + c(t,x)v(t,x) + d(t,x)$$

and rearranging the terms, gives a difference equation

$$E_{i,j}v_{i-1,j-1} + F_{i,j}v_{i-1,j} + G_{i,j}v_{i-1,j+1} = A_{i,j}v_{i,j-1} + B_{i,j}v_{i,j} + C_{i,j}v_{i,j+1} + D_{i,j},$$

with

$$\begin{aligned}
A_{i,j} &= \tfrac{(1-\lambda)\Delta t}{\Delta x}\left(\tfrac{b_{i-\lambda,j}}{2} - \tfrac{a_{i-\lambda,j}}{\Delta x}\right), & E_{i,j} &= -\tfrac{\lambda\Delta t}{\Delta x}\left(\tfrac{b_{i-\lambda,j}}{2} - \tfrac{a_{i-\lambda,j}}{\Delta x}\right),\\
B_{i,j} &= 1 + (1-\lambda)\,\Delta t\left(\tfrac{2a_{i-\lambda,j}}{\Delta x^2} - c_{i-\lambda,j}\right), & F_{i,j} &= 1 - \lambda\Delta t\left(\tfrac{2a_{i-\lambda,j}}{\Delta x^2} - c_{i-\lambda,j}\right),\\
C_{i,j} &= -\tfrac{\Delta t(1-\lambda)}{\Delta x}\left(\tfrac{b_{i-\lambda,j}}{2} + \tfrac{a_{i-\lambda,j}}{\Delta x}\right), & G_{i,j} &= \tfrac{\lambda\Delta t}{\Delta x}\left(\tfrac{\Delta t b_{i-\lambda,j}}{2} + \tfrac{a_{i-\lambda,j}}{\Delta x}\right),
\end{aligned}$$

$$D_{i,j} = -\Delta t d_{i-\lambda,j}.$$

We add a subclass `WeightImplicit` of the `ImplicitScheme` class in the file

`Exe6-6_WeightImplicit.h`

See also

`Exe6-6_Main25.cpp`

6.7 The inverse change of coordinates to

$$\begin{aligned}
z = Z(x) &= \frac{Lx}{1-x},\\
v = V(z,u) &= \frac{u}{z+L},
\end{aligned}$$

is given by

$$\begin{aligned}
X(z) &= \frac{z}{L+z},\\
U(x,v) &= \frac{Lv}{1-x}.
\end{aligned}$$

We can take

$$x_l = X(z_l), \qquad x_u = X(z_u),$$

and consider the following boundary conditions:

$$\begin{aligned}
f(x) &= V(Z(x), h(Z(x))),\\
f_l(t) &= V(z_l, h_l(t)),\\
f_u(t) &= V(z_u, h_u(t)).
\end{aligned}$$

Once we have solved the equation for v numerically, we can compute

$$u(t,z) = U(X(z), v(t, X(z))).$$

See

`Exe6-7_FiniteDomainEq.h`
`Exe6-7_FiniteDomainEq.cpp`
`Exe6-7_Main.cpp`

6.8 In the file `Exe6-8_HeatEqLCP.h` we introduce a class `HeatEqLCP` with the free boundary condition

$$g(t,x) = V(t, h(Z(t,x)).$$

In the `main()` function in `Exe6-8_Main27.cpp` we need to remember to choose $z_l > 0$ so that z_l lies in the domain of $X(0,z)$. After we solve the LCP for the heat equation for $v(t,x)$, we can compute $u(t,z)$ using

$$u(t,z) = U(t, v(t, X(t,z))).$$

See

`Exe6-8_Main27.cpp`
`Exe6-8_HeatEqLCP.h`

6.9 It is enough to add the function

```
int ExercisePolicy(FDMethod* PtrMethod,LCP* PtrLCP,double
t,double S)
{
   if (PtrMethod->v(t,S) > PtrLCP->g(t,S)) return 0;
   return 1;
}
```

in the main file from Listing 6.23. If the function returns 1, then we should exercise the American option.

See

`Exe6-9_Main27.cpp`