1

Binomial pricer

In the **binomial model** the prices of assets evolve in discrete time steps $n = 0, 1, 2, \ldots$. There is a **stock** whose price evolves randomly by moving up by a factor $1 + U$ or down by $1 + D$ independently at each time step, starting from the spot price $S(0)$. As a result, the stock price becomes

$$S(n,i) = S(0)(1+U)^i(1+D)^{n-i}$$

at step n and node i in the binomial tree

$$\begin{tikzcd}
 & & & & \cdots \\
 & & & i=3 \arrow[ur] \arrow[dr] & \\
 & & i=2 \arrow[ur] \arrow[dr] & & \cdots \\
 & i=1 \arrow[ur] \arrow[dr] & & i=2 \arrow[ur] \arrow[dr] & \\
i=0 \arrow[ur] \arrow[dr] & & i=1 \arrow[ur] \arrow[dr] & & \cdots \\
 & i=0 \arrow[ur] \arrow[dr] & & i=1 \arrow[ur] \arrow[dr] & \\
 & & i=0 \arrow[ur] \arrow[dr] & & \cdots \\
 & & & i=0 \arrow[ur] \arrow[dr] & \\
 & & & & \cdots \\
n=0 & n=1 & n=2 & n=3 & \cdots
\end{tikzcd} \tag{1.1}$$

where $S(0) > 0$, $U > D > -1$ and $n \geq i \geq 0$. There is also a risk-free security, a **money market account**, growing by a factor $1 + R > 0$ during each time step. The model admits no arbitrage whenever $D < R < U$.

Within the binomial model the price $H(n, i)$ at each time step n and node i of a **European option** with expiry date N and payoff $h(S(N))$ can be computed using the **Cox–Ross–Rubinstein** (**CRR**) **procedure**, which proceeds by backward induction:

- At the expiry date N
$$H(N, i) = h(S(N, i)) \tag{1.2}$$
for each node $i = 0, 1, \ldots, N$.
- If $H(n + 1, i)$ is already known at each node $i = 0, 1, \ldots, n + 1$ for some $n = 0, \ldots, N - 1$, then
$$H(n, i) = \frac{qH(n + 1, i + 1) + (1 - q)H(n + 1, i)}{1 + R} \tag{1.3}$$
for each $i = 0, 1, \ldots, n$.

Here
$$q = \frac{R - D}{U - D}$$
is the **risk-neutral probability**. In particular, for a **call option** the payoff function is
$$h^{\text{call}}(z) = \begin{cases} z - K & \text{if } z > K, \\ 0 & \text{otherwise.} \end{cases} = (z - K)^+$$
and for a **put option** it is
$$h^{\text{put}}(z) = \begin{cases} K - z & \text{if } z < K, \\ 0 & \text{otherwise.} \end{cases} = (K - z)^+$$
for all $z > 0$, where K is the **strike price**. For more details on pricing European options in the binomial model, see [DMFM].

We are going to develop a simple binomial pricer based on these formulae, working through a number of versions of the code, and adding new features at each step.

1.1 Program shell

We start with an almost empty shell that does just one thing: it displays a brief message on the screen and pauses so that the user can read it.

Listing 1.1 Main01.cpp

```
#include <iostream>                                   ❶
using namespace std;                                  ❷

int main()                                            ❸
{
   //display message                                  ❹
   cout << "Hi there" << endl;                        ❺

   //pause program                                    ❹
   char x; cin >> x;                                  ❻

   return 0;                                          ❼
}
```

Let us examine this code line by line.

❶ `#include <iostream>`
tells the compiler to locate and read the **header file** `iostream.h` and to include its contents in the program. This header file is part of the **standard library** and handles input–output operations. It is needed here because we want to display a message on the screen.

❷ **Namespaces** prevent inadvertent name clashes and help to group related names together. All names in the standard C++ library are wrapped in a single namespace `std`. The line
`using namespace std;`
makes this namespace available throughout the file. Without it we would have to tell the compiler explicitly that `cout`, `endl`, `cin` belong to the namespace by writing them as `std::cout`, `std::endl`, `std::cin`. (Try it!)

❸ `int main()`
is the entry point for the C++ program, which starts by executing the first line enclosed within the curly brackets `{` and `}`. Every C++ program must contain exactly one such entry point.

❹ The lines starting with `//` are **comments**. When `//` is encountered anywhere in a line of code, everything to the right of `//` in that line is ignored by the C++ compiler.

❺ When the line of code
`cout << "Hi there" << endl;`
is executed, the **output operator** `<<` sends the text enclosed in

quotation marks to the standard output `cout`, which in the majority of operating systems will be a screen window. This is followed by `endl`, with the effect of writing a new line.

❻ Depending on the system used, the window in which the program is executed may be closed automatically when the program terminates, and this may happen too quickly for the user to read the message. The statements
`char x; cin >> x;`
pause the program until the user enters a character from the keyboard. They may be unnecessary on systems that keep the window open after the program terminates. These statements will be omitted in subsequent versions of the program, but can be inserted if pausing is desirable.

This also shows how to enter input from the keyboard. First, `char x;` declares `x` to be a **variable** (in fact a location in computer memory) to hold a single character. Then `cin >> x;` instructs the program to wait for input from the keyboard (the user needs to type a character and press *Enter*) and to store the input at `x`. Here `>>` is the **input operator**. We do not really need `x` or anything stored in there, but use it merely as a simple technique to pause the program.

❼ `return 0;`
is finally executed. It terminates the program and returns value `0` to tell the operating system that the program has run successfully. Returning a non-zero value would indicate failure.

1.2 Entering data

We are ready to place some useful code inside the shell to read and check the input data. For good measure, we also compute a couple of things, namely the risk-neutral probability and the stock price at a given time step *n* and node *i*.

Listing 1.2 Main02.cpp

```
#include <iostream>
#include <cmath>                                         ❶
using namespace std;

int main()
{
   double S0,U,D,R;                                      ❷
   //entering data
   cout << "Enter S0: "; cin >> S0;                                  ❸
   cout << "Enter U:  "; cin >> U;
   cout << "Enter D:  "; cin >> D;
   cout << "Enter R:  "; cin >> R;
   cout << endl;

   //making sure that 0<S0, -1<D<U, -1<R
   if (S0<=0.0 || U<=-1.0 || D<=-1.0 || U<=D                         ❹
                                    || R<=-1.0)
   {
      cout << "Illegal data ranges" << endl;
      cout << "Terminating program" << endl;
      return 1;
   }

   //checking for arbitrage
   if (R>=U || R<=D)                                                 ❺
   {
      cout << "Arbitrage exists" << endl;
      cout << "Terminating program" << endl;
      return 1;
   }

   cout << "Input data checked" << endl;
   cout << "There is no arbitrage" << endl << endl;

   //compute risk-neutral probability
   cout << "q = " << (R-D)/(U-D) << endl;                            ❻

   //compute stock price at node n=3,i=2
   int n=3; int i=2;
   cout << "n = " << n << endl;
   cout << "i = " << i << endl;
   cout << "S(n,i) = " << S0*pow(1+U,i)*pow(1+D,n-i)                 ❼
                       << endl;

   return 0;
}
```

We focus our attention on the new features in this piece of code.

❶ `#include <cmath>`
loads the header file `cmath.h` so we can use various mathematical functions defined in the standard library. We need it in order to have the power function `pow()` later in the code.

❷ `double S0,U,D,R;`
declares some variables of type `double` to store floating point input data for the spot price $S(0)$, the up and down returns U, D and the risk-free return R.

In C++ the **type** of every variable must be declared before the variable is used. Apart from `double` there are many other variable types. For example, variables of type `int` are used to store integer numbers. In Listing 1.1 we have already seen a variable of type `char` to hold a single character.

❸ `cout << "Enter S0: "; cin >> S0;`
displays a message prompting the user to enter the spot price $S(0)$ and handles the input. There are similar lines of code for entering U, D, R.

❹
```
if (S0<=0.0 || U<=-1.0 || D<=-1.0 || U<=D
                                   || R<=-1.0)
```
comes next to verify the integrity of input data, ensuring that $0 < S(0)$, $-1 < D < U$ and $-1 < R$. Here `||` is the **logical OR operator**, and `<=` is the **logical inequality operator**, which checks whether or not the inequality $\leq$ holds between two numbers. If the condition inside the round brackets `(` and `)` is satisfied, the lines inside the curly brackets `{` and `}` after the **`if` statement** will be executed, displaying a warning message and terminating the program with return value `1`, which indicates failure. If the condition inside the round brackets is not satisfied, the program skips the lines inside the curly brackets and moves on to the following line of code.

❺ `if (R>=U || R<=D)`
similarly checks for the lack of arbitrage, that is, verifies whether or not $D < R < U$. Once these checks are completed successfully, messages to that effect are displayed and the program proceeds to the next line.

❻ `cout << "q = " << (R-D)/(U-D) << endl;`
computes and displays the risk-neutral probability $q = \frac{R-D}{U-D}$.

❼
```
cout << "S(n,i) = " << S0*pow(1+U,i)*pow(1+D,n-i)
                    << endl;
```
computes and displays the stock price $S(n,i) = S(0)(1+U)^i(1+D)^{n-i}$ at time step n and node i. The variables `n` and `i` are first declared to be of type `int`, initiated with values 3 and 2, respectively, and displayed, just to illustrate how things work.

Exercise 1.1 Tweak the code in `Main02.cpp` so the user can enter n and i from the keyboard.

1.3 Functions

The various parts of the program perform distinct tasks such as inputting and verifying data, computing the risk-neutral probability, or computing the stock price at a given node of the binomial tree. It is good programming practice to arrange such tasks into separate functions, which is what we do next.

Listing 1.3 Main03.cpp

```
#include <iostream>
#include <cmath>
using namespace std;

//computing risk-neutral probability
double RiskNeutProb(double U, double D, double R)                    ❶
{
   return (R-D)/(U-D);
}

//computing the stock price at node n,i
double S(double S0, double U, double D, int n, int i)                ❷
{
   return S0*pow(1+U,i)*pow(1+D,n-i);
}

//inputting, displaying and checking model data
int GetInputData(double& S0,                                         ❸
                 double& U, double& D, double& R)
{
   //entering data
   cout << "Enter S0: "; cin >> S0;
   cout << "Enter U:  "; cin >> U;
   cout << "Enter D:  "; cin >> D;
   cout << "Enter R:  "; cin >> R;
   cout << endl;

   //making sure that 0<S0, -1<D<U, -1<R
   if (S0<=0.0 || U<=-1.0 || D<=-1.0 || U<=D
                                     || R<=-1.0)
   {
      cout << "Illegal data ranges" << endl;
      cout << "Terminating program" << endl;
      return 1;
   }

   //checking for arbitrage
   if (R>=U || R<=D)
   {
      cout << "Arbitrage exists" << endl;
      cout << "Terminating program" << endl;
      return 1;
   }

   cout << "Input data checked" << endl;
   cout << "There is no arbitrage" << endl << endl;

   return 0;
}

int main()                                                        ❹
{
   double S0,U,D,R;

   if (GetInputData(S0,U,D,R)==1) return 1;                      ❺

   //compute risk-neutral probability
   cout << "q = " << RiskNeutProb(U,D,R) << endl;                ❻

   //compute stock price at node n=3,i=2
   int n=3; int i=2;
   cout << "n = " << n << endl;
   cout << "i = " << i << endl;
   cout << "S(n,i) = " << S(S0,U,D,n,i) << endl;                 ❼

   return 0;
}
```

The program works almost exactly as before, but much of the code has been moved into functions. Here is a breakdown of the new features.

❶ `double RiskNeutProb(double U, double D, double R)` tells the compiler that we are defining a **function** `RiskNeutProb()` that takes three arguments of type `double` and returns a value of type `double`. The body of this function, enclosed within curly brackets, consists of a single line
`return (R-D)/(U-D);`
which computes the value of the expression and returns it to whatever part of the program is going to call this function.

When we need to compute the risk-neutral probability, we can write `RiskNeutProb(U,D,R)` instead of `(R-D)/(U-D)`. It will hardly save us any typing, but a meaningful name for the function improves readability. Moreover, if we decide to change anything in the expression, perhaps to fix a bug, to improve efficiency or to add some extra functionality, we can do it all in one place, which can be very useful if the risk-neutral probability needs to be evaluated in several different places in the code.

❷ The next function

```
double S(double S0, double U, double D, int n, int i)
```

is similar. It takes three arguments of type `double` and two of type `int`, and returns a value of type `double`, computed using the appropriate formula for the stock price at a given node.

When an argument is passed to a function as above, for example as in `double S0`, it is said to be **passed by value**. It is important to understand what happens in this case. When the function is called, a copy of that variable is made in a separate location in computer memory. The function can see and change the copy, but has no access to the original variable. On the other hand, the calling program cannot see the copy (which is in fact destroyed when the function returns control to the calling program) and only has access to the original variable.

In many situations it is important to know that a function has not altered, perhaps inadvertently, the original value of a variable passed to it. For example, it would not do if the function `RiskNeutProb()` inadvertently changed the value of `U`, which is then used again elsewhere in the program to compute the stock price. Passing a parameter by value guarantees that this cannot happen. On the other hand, making copies of variables consumes time, possibly a consideration when calls to the same function are made a large number of times.

❸ When defining a function to handle input data we face a problem because a function in C++ can return only a single value, but we want to enter four numbers as inputs, and the function will need to pass all of them to the program. To deal with this, in

```
int GetInputData(double& S0,
                          double& U, double& D, double& R)
```

the arguments are **passed by reference**, which is indicated by `&`. In this case a single copy of the variable in computer memory is shared by the function and the calling program. The function can see and alter the shared variable. Any changes made by the function to this variable remain available to the calling program when the function returns control. This is exactly what we need to pass on the values of the input data.

There is an older and now much out of favour method of achieving a similar result by passing a pointer to a variable. It will be covered in Section 1.6.

The code for inputting and verifying data is largely the same as in Listing 1.2, but now it is placed inside the function. In addition, the function returns a value of type `int`, which is used to indicate to the calling program whether or not inputting data has been successful.

❹ The body of `main()` has been streamlined and made more readable. Much of the code has been moved into functions and replaced by calls to these functions:

- ❺ `if (GetInputData(S0,U,D,R)==1) return 1;` takes care of inputting and verifying the data, checks if this has been successful, and terminates the program if not.

  Note that `==` is the **logical equality operator**, returning a true value if the expressions on either side are equal, and false otherwise. Do not confuse it with `=`, the **assignment operator**.
- ❻ `RiskNeutProb(U,D,R)` computes the risk-neutral probability.
- ❼ `S(S0,U,D,n,i)` computes the stock price at the given node.

Incidentally, `int main()` indicates that `main()` is also a function, which returns a value of type `int` and takes no arguments, as shown by the empty brackets `()`.

Exercise 1.2 Write a function called `interchange()` that interchanges the contents of two variables of type `double`, which are to be passed to the function by reference.

1.4 Separate compilation

If the program uses several functions, which may also be used by other programs, it is advisable to place the functions into a separate file, which is what we do next.

Listing 1.4 BinModel01.cpp

```
#include <iostream>
#include <cmath>
using namespace std;

double RiskNeutProb(double U, double D, double R)
{
   return (R-D)/(U-D);
}
```