Programming Project: Binomial-model option-valuation

PRASAD CHALASANI
chal@cs.cmu.edu
(412)268-3194(office)

July 31, 1998

1 Introduction

The point of this programming project is to apply your programming (C,C++,Java) and algorithm/data-structure-design skills to certain option-pricing problems in the binomial model. You will also learn the following concepts:

1. Use of *recursive* functions,
2. Use of *dynamic programming* (or backward recursion) to avoid an exponential growth in computation time.
3. Use of linear *interpolation* to estimate a function value during backward recursion.

Use whichever language you're most comfortable with. This initial project may be too easy. If so let me know. If you have any questions, email me or call me at the office.

2 Background: option pricing in the binomial model

The following sub-sections give a watered-down account of the finance concepts you'll need for this assignment. If you're curious and want to learn more, you are encouraged to look up Steven Shreve's lecture notes on Stochastic Calculus and Finance, available at `http://www.cs.cmu.edu/~chal`. The first few chapters cover the binomial model in detail.

2.1 Binomial model

The binomial model is used to compute the "fair price" of an option on an underlying asset, which we will often refer to as the "stock". The inputs for setting up the model are:

1. The **life** T (i.e., time to expiration) of the option under consideration. Typical value: 1.0 year.

2. The number n of equal-length **periods** into which T is to be divided. Each period has length $\Delta t = T/n$. For $k = 0, 1, 2, \ldots, n$, discrete time k will refer to continuous-time $k\Delta t = kT/n$. Typical value: $n = 10, 20, 30, 60$.

3. The continuously-compounded annualized **risk-free interest rate** r. Thus a dollar invested in a risk-free instrument at time 0 would be worth $R = e^{r\Delta t}$ at time Δt years. Typical value: $r = .05$

4. The annual **volatility** σ of the stock price. Typical value: $\sigma = 0.3, 0.1$.

From these inputs, the following parameters are computed:

1. The **up-factor** u and the **down-factor** $1/u$, where

$$u = \exp\left\{\sigma\sqrt{\Delta t}\right\} = \exp\{\sigma\sqrt{T/n}\}. \tag{1}$$

2. The **up-tick probability** p, and down-tick probability $q = 1 - p$, where

$$p = \frac{R - 1/u}{u - 1/u} = \frac{e^{r\Delta t} - 1/u}{u - 1/u}. \tag{2}$$

The stock price at (discrete) time k is denoted S_k. The initial stock price S_0 is non-random and known. The process S_k is specified as follows, for $k = 0, 1, \ldots, n-1$:

$$S_{k+1} = \begin{cases} uS_k & \text{with probability } p, \\ (1/u)S_k & \text{with probability } 1-p. \end{cases}$$

Notice that with the above choice of parameters, the expected value of S_{k+1} given S_k is $S_k R$, i.e.,

$$\mathbf{E}(S_{k+1}|S_k) = RS_k,$$

which means the discounted stock price process is a martingale.

The stock price process can be described by a **binomial tree**, where the initial stock price S_0 is the root node, and each path in the tree represents a possible sequence of stock prices $S_0, S_1, \ldots, S_n$. A price path ω is characterized by a sequence of random variables $X_1, X_2, \ldots, X_n$, where $X_k = 1$ if there is an up-tick at time k and -1 otherwise. That is,

$$S_{k+1} = S_k X_{k+1}, \quad k = 0, 1, \ldots, n-1,$$

and

$$\mathbf{P}[X_k = 1] = p; \quad \mathbf{P}[X_k = -1] = q = 1 - p.$$

We will also use the random variable

$$Y_k = \sum_{i=1}^{k} X_i, \quad k = 1, 2, \ldots, n,$$

which is the number of *up-ticks minus down-ticks* by time k.

2.2 Binomial lattice

Note that the stock price S_k at time k only depends on the *number* of up-ticks H_k that have occured by time k:

$$S_k = S_0 u^{H_k} (1/u)^{k-H_k} = S_0 u^{2H_k - k}.$$

Thus in the binomial tree, several nodes at depth k actually represent the *same* stock price S_k. We use this fact to collapse the binomial tree into a **binomial lattice** (sometimes also called a *recombining binomial tree*). A node numbered (k, i) in the lattice represents a state at time k where $Y_k = i$, i.e. the number of up-ticks minus down-ticks equals i. Thus the root node of the lattice is $(0, 0)$. The two nodes at depth 1 are $(1 - 1)$ and $(1, 1)$. The three nodes at depth 2 are $(2, -2)$, $(2, 0)$, and $(2, 2)$, and so on. In general the nodes at depth k are $(k, -k), (k, -k+2), \ldots, (k, k)$. See Fig. 1 for a picture of a lattice. We will abuse notation slightly, and denote the value of a random variable Z in state (k, i) by $Z(k, i)$. For instance the stock price S in state (k, i) is $S(k, i)$. Note that

$$S(k, i) = S(0, 0) u^i = S_0 u^i. \tag{3}$$

One useful way to think of a lattice node (k, i) is that it represents the *collection of tree paths* reaching depth k from the root, that have i more up-ticks than down-ticks.

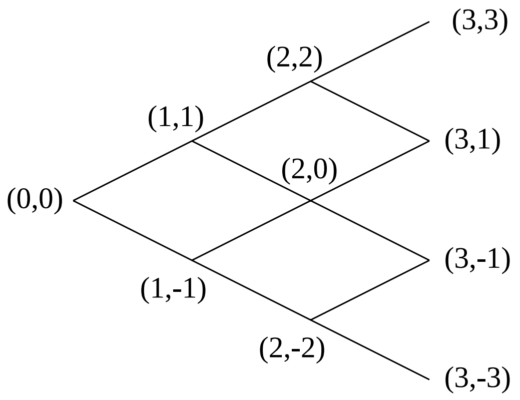

Figure 1: A binomial lattice. A node (k, i) represents a state at time k when the number of up-ticks minus down-ticks is i.

2.3 Options

A **European-style option** which expires at discrete time n gives the owner the right to *exercise* the option at time n. When the owner exercises the option at time n, she receives a payoff G_n. The payoff G_n depends only on the specific stock-price history $S_0, S_1, \ldots, S_n$ that has occurred. For instance, a European **call option** with **strike** K has payoff $G_n = (S_n - K)^+$ where for any real x, $x^+ \equiv \max\{x, 0\}$. Similarly, a European **put option** with strike K has payoff $G_n = (K - S_n)^+$. Note that for these two simple options, the payoff G_n depends only on the *final* stock price S_n. Such options are said to be **path-independent**. By contrast, the payoff of a European-style **Asian call option** with strike K is given by $G_n = (A_n - K)^+$, where A_k is the average stock price from time 0 to time k:

$$A_k = \frac{S_0 + S_1 + \ldots + S_k}{k+1}. \tag{4}$$

The payoff of an Asian option is thus clearly **path-dependent**.

An **American-style** option which expires at discrete time n gives the owner the right to exercise the option at *any* time before the expiration time n. Such an option is characterized by a *payoff process* G_k, $k = 0, 1, \ldots, n$, where G_k depends only on the specific stock-price path $S_0, S_1, \ldots, S_k$ that was realized at the time of exercise. For instance an American call option with strike K that expires at time n has payoff function $G_k = (S_k - K)^+$ for $k = 0, 1, \ldots, n$. Similarly an American-style Asian call option has payoff $G_k = (A_k - K)^+$ for $k = 0, 1, \ldots, n$.

2.4 Pricing options

Denote a typical node in the (non-recombining) binomial tree by v, and its up- and down-successors respectively by v^+ and v^-. Note that a node v at depth k in the tree defines a *specific* path of stock prices $S_0, S_1, \ldots, S_k$. When referring to the value of a process Z_k at a node v, we will drop the subscript on Z, and denote it by $Z(v)$. For instance, the stock price at node v is $S(v)$, and the *average* stock price on the path from the root to v is $A(v)$. We define the special random variable $D(v)$ to be the *depth* of v. Thus, $D(v)$ denotes the time corresponding to the state v. We also write $v \rightarrow w$ to denote that v "leads to" w in the tree, i.e., w is a successor of v. For any node v let $\mathbf{P}(v)$ denote the *probability* of the path from the root to v. Abusing notation slightly, when $v \rightarrow w$ we write $\mathbf{P}(w|v)$ to denote the probability of reaching a node w, starting from v. Clearly,

$$\mathbf{P}(w|v) = \frac{\mathbf{P}(w)}{\mathbf{P}(v)}.$$

The expectation of a random variable Z_k is

$$\mathbf{E}Z_k = \sum_{v:D(v)=k} Z(v)\mathbf{P}(v).$$

Similarly, if $D(v) \leq k$, the *conditional expectation* of a random variable Z_k *given* the current state v, is

$$\mathbf{E}(Z_k|v) = \sum_{w:D(w)=k, v\rightarrow w} Z(w)\mathbf{P}(w|v).$$

Consider a European-style option with payoff function G_n. The **arbitrage-free value** of the option at a depth-k node v is given by

$$V(v) = R^k \mathbf{E}\left[\frac{G_n}{R^n}\middle| v\right] = R^{k-n}\mathbf{E}(G_n|v). \tag{5}$$

From this it is easily seen that for non-terminal nodes v (i.e., $D(v) < n$), $V(v)$ can be recursively written as

$$V(v) = \frac{pV(v^+) + (1-p)V(v^-)}{R}, \tag{6}$$

and for terminal nodes v, the value is simply the payoff:

$$V(v) = G(v). \tag{7}$$

For path-*independent* options, the recursive expressions above remain valid even if v, v^+, v^- represent binomial *lattice* nodes rather than tree nodes.

The valuation of American-style options is complicated by the fact that the owner may use a *strategy* to exercise her option: at any time $k \leq n$, she can decide whether or not to exercise the option, based on the history of the stock price so far. Fortunately, the arbitrage-free value of an American-style option at a node v can be expressed recursively in a manner similar to the European case. Specifically, on the (non-recombining) binomial tree, for non-terminal v,

$$V(v) = \max\left\{G(v),\ \frac{pV(v^+) + (1-p)V(v^-)}{R}\right\}, \tag{8}$$

and for terminal v,

$$V(v) = G(v). \tag{9}$$

The **optimal exercise strategy** (i.e. the one that maximizes the expected discounted payoff) for the owner of the option can be easily defined from the backward-recursive computation: While moving forward on a path from the root node, if the current node is v, exercise immediately if and only if $V(v) = G(v)$, or equivalently,

$$G(v) \geq \frac{pV(v^+) + (1-p)V(v^-)}{R}.$$

Thus, on any tree path, the optimal exercise point is the *earliest* node v where the option payoff $G(v)$ from immediate exercise equals the option value $V(v)$.

Again, for path-*independent* options, the same expressions can be used on the binomial lattice.

3 Programming problem: using recursive functions

3.1 American put option.

■ This problem asks you to write a C/C++/Java program to compute the (arbitrage-free) value of a simple American-style put option on a stock (defined in Section 2.3). Use a recursive function, and do *not* explicitly construct the binomial tree. Recall that the payoff of a put option with strike K when exercised at time k is $G_k = (K - S_k)^+$. Equivalently, when exercised at a node v of the binomial tree, the payoff is

$$G(v) = [K - S(v)]^+.$$

The inputs to the program are:

1. Expiration time T (years) of the option,
2. Number of time-divisions n,
3. Risk-free annual interest rate r,
4. Stock volatility σ,
5. Initial stock price S_0 (dollars),
6. Strike price K (dollars).

Hints: First compute the binomial tree parameters u (up-factor) and p (up-probability), using (1) and (2). Next, we would like to use the expressions (8) and (9) to define a recursive function $V(v)$ to compute the option value at a node v. How would you represent a node v? Convince yourself that, since a put option is path-independent, it suffices to represent node v by its depth $k = D(v)$ and the stock price $S(v)$. Given $v = (k, S)$, the representation for v^+ and v^- is easily computed (how?). Thus your recursive function would look like $V(k, S)$, and should return the option value at a node of depth k at which the stock price is S. In other words, $V(k, S)$ is the option value in the state at time k in which the stock price is S. Clearly the option value at time 0 (which is what we need to ultimately compute) is then obtained by invoking $V(0, S_0)$.

3.2 Time complexity.

■ Run the program with $n = 10$ time-divisions and see how long it takes. Based on this, estimate how long the program would take with $n = 60$ time-divisions. To answer this, you'll need to understand the *time complexity* $f(n)$ of the algorithm: this means that the algorithm takes time proportional to $f(n)$ for some function. What is the function $f(n)$ in this case?

3.3 Asian call option.

■ Modify your code to compute the value of an American-style *Asian call option* on a stock, with strike K. As mentioned above, the payoff of this option at a binomial-tree node v is

$$G(v) = [A(v) - K]^+,$$

where $A(v)$ is the stock-price average on the path from the root to v (defined in (4)). The inputs to this program are the same as before. Again, do not explicitly construct the binomial tree; use a recursive function.

Hints. Since an Asian option is path-dependent, we will need to represent a node v by more than just the depth k and the stock price $S(v)$. Convince yourself that it suffices to represent a node v by (k, S, A) where k is the depth $D(v)$, $S = S(v)$, and A is $A(v)$, the average stock price on the path from the root to v. Given $v = (k, S, A)$ you can easily compute the representation for v^+ and v^- (how?). Thus your recursive function would look like $V(k, S, A)$, and should return the value of the option in the state at time k where the stock price is S, *and* the stock-price-average so far is A. The time-0 value of the option is then $V(0, S_0, S_0)$.

4 Programming problem: using dynamic programming on the lattice

4.1 American put option.

As the number of time-divisions n increases, the time taken by the above recursive pricing function increases rapidly. (You should have answered exactly how fast in the previous section). For a simple put option, however, we saw that the option value at a node $v = (k, S)$ depends only on k and S, i.e., the option is path-independent. In particular, at every depth-k node w with $S(w) = S$, the option value $V(w)$ is the *same.* Thus the above recursive-function approach is very wasteful: the function $V(k, S)$ in invoked *every* time a depth k node v is reached with $S(v) = S$. We can avoid this waste if we represent the tree as a binomial *lattice*, which as explained in Section 2.2 collapses all nodes at a given depth that have the same stock price. In other words, the lattice representation does not lose any information, as far as pricing this option is concerned. As mentioned in Section 2.4, for path-independent options, the recursive expressions (8) and (9) remain valid on the lattice.

■ How would you write a program to take advantage of this fact? An easy way to do this is to explicitly create a lattice using say a two-dimensional array $V[][]$ so that $V[k][i]$ stores the value of the option at lattice node (k, i) (recall that i is the number of up-ticks minus down-ticks, which uniquely determines the stock price at the lattice node). Start from the end of the lattice, i.e., $k = n$, and set the option value $V[n][i]$ for $i = -n, \ldots, n$, to be equal to the immediate payoff (this is expression (9)). Then continue backward on the lattice, computing for each k, the values $V[k][i]$, $i = -k, \ldots, k$, (using (8)) and so on until you obtain $V[0][0]$, which is the time-0 option value. This type of procedure is called **dynamic programming.** Write the code to compute the option value as outlined above.

4.2 Time complexity

■ How long does your program take for $n = 10$? What about $n = 60$? What is the time complexity of your algorithm?

4.3 Optimal exercise strategy

■ Modify your code so that it prints out the optimal exercise strategy for the given American put. In particular, print out the lattice node coordinates (k, i) where it is optimal to exercise the option immediately. Recall that on any path in the lattice, the optimal exercise point is the earliest node v where $V(v) = G(v)$.

5 Asian call option: Hull-White interpolation method

Unfortunately, an Asian option is path-dependent, so the above lattice approach cannot be used to compute the value faster. In fact, in general the average stock prices $A(v)$ for *all* depth-k nodes v of the tree are different from each other! So at depth k there are 2^k possible stock price averages. Thus, unlike in the case of an American put option, there is no wasted computation in the recursive function approach above; each specific invocation $V(k, S, A)$ is made only once. Consequently, the valuation of Asian options (European- or American-style) has been a very hard problem in finance. However, numerous *approximate* algorithms have been proposed, and you will implement one such recent algorithm of Hull and White [3].

To describe the Hull-White method, it is best to consider the lattice-based American put algorithm you implemented in Section 4.1. There, we defined a two-dimensional array $V[][]$ where $V[k][i]$ stores the option value at lattice node (k, i). This does not work for Asian options because each tree-path represented by a lattice node (k, i) has a different average-stock price A_k, which we could number $a_1, a_2, \dots, a_m$. However in principle we *could* extend this to Asian calls using a *three* dimensional array $V[][][]$ where $V[k][i][j]$ stores the option value at time k when there have been i more up-ticks than down-ticks, *and* the average stock price so far is a_j. Obviously this method would be just as slow as the recursive-function approach.

The main idea of the Hull-White approximate algorithm is this: Don't compute the option value at every possible value of the stock-price-average at lattice node (k, i); rather, compute it only for certain *special* values of the average stock price, of the form

$$S_0 e^{mh},$$

for a specific *grid size* h (such as 0.01) and different integers m (which could be negative). Specifically, use a 3D array $V[][][]$, where $V[k][i][j]$ represents the (American-style) Asian call option value at the state at time k where the up-ticks minus down-ticks is i, and the average stock price [1] is S_0e^{jh}. For each lattice node (k, i), a certain range of possible values of the third index j must be considered. We let a_{ki} be the smallest index considered at node (k, i) and b_{ki} be the largest. The range (a_{ki}, b_{ki}) must be chosen so that all possible stock-price-averages that are considered at node (k, i) lie between $S_0 \exp\{a_{ki}h\}$ and $S_0 \exp\{b_{ki}h\}$. Thus a_{ki} must be chosen to be the biggest integer m such that

$$S_0 \exp\{mh\} \leq \text{smallest possible average at } (k, i)$$

and b_{ki} must be chosen to be the smallest integer m such that

$$S_0 \exp\{mh\} \geq \text{biggest possible average at } (k, i).$$

You can use a conservative estimate for the right-hand-sides of the inequalities above. For instance (this is Pankaj Mody's idea) the "smallest possible average at (k, i)", is clearly no smaller than the smallest possible stock price on any path reaching (k, i), which is

$$S_0 u^{(i-k)/2} = S_0 \exp\{\sigma\sqrt{T/n}(i-k)/2\}.$$

(See Fig. 1 and consider paths reaching $(3, 1)$ to convince yourself of this). Similarly the "biggest possible average at (k, i)", is clearly no bigger than the biggest possible stock price on a path reaching (k, i), which is

$$S_0 u^{(k+i)/2} = S_0 \exp\{\sigma\sqrt{T/n}(k+i)/2\}.$$

Thus you can use

$$a_{ki} = \left\lfloor \sigma\sqrt{T/n}\frac{i-k}{2h} \right\rfloor,$$

and

$$b_{ki} = \left\lceil \sigma\sqrt{T/n}\frac{k+i}{2h} \right\rceil.$$

The function $\lfloor x \rfloor$ is computed by the C++ function `floor(x)`, which returns the biggest integer $\leq x$. The function $\lceil x \rceil$ is computed by `ceil(x)`, which returns the smallest integer $\geq x$.

[1] There may actually be no state with this specific average stock price, but this is immaterial to the algorithm.

The first stage in the Hull-White algorithm is to compute the above ranges (a_{ki}, b_{ki}) for each lattice node (k, i) and set up the appropriate 3D array $V[][][]$ to accomodate these ranges.

Using the above data structure, let us apply the recursive expressions (8) and (9) to compute the $V[k][i][j]$ values. Note that we can set $V[n][i][j]$ to be simply $(S_0 e^{jh} - K)^+$ (this is the option value at time n when the average stock price is $S_0 e^{jh}$).

Now suppose we have computed the entries $V[g][][]$ for all $g = k+1, k+2, \ldots, n$. Let us consider the computation of a specific entry $V[k][i][j]$. This represents the (approximate) option value of the option in a state at depth k where the stock price is $S_0 u^i$, and the average stock price so far is $S_0 e^{jh}$. Let us denote this state by (v) (this is essentially a tree node), so that $S(v) = S_0 u^i$ and $A(v) = S_0 e^{jh}$. You may think that we can now simply use the recursive expression (9) to compute $V[k][i][j]$. But life is not so simple. To use the recursion (9) you must first compute the S and A values at nodes v^+ and v^- (you already did this for the recursive-function approach of Section 3.3); denote these values as $S(v^+)$, $A(v^+)$, etc. For instance,

$$S(v^+) = uS(v); \qquad A(v^+) = \frac{(k+1)A(v) + uS(v)}{k+2}.$$

Now to use the recursion (9), you also need the values $V(v^+)$ and $V(v^-)$. Consider for example $V(v^+)$, which is the option value at lattice node $(k+1, i+1)$ on a path where the arithmetic average so far is $A(v^+)$. However, at this lattice node you have only computed $V[k+1][i+1][]$ for certain *special* values of A_{k+1}, namely for those of the form $S_0 e^{jh}$ for all integers j in the range $(a_{k+1,i+1}, b_{k+1,i+1})$. The arithmetic average $A(v^+)$ will *not* in general be of the form $S_0 e^{sh}$ for *integer* s.

This leads to the **interpolation** idea. Instead of using the *exact* values of $V(v^+)$ and $V(v^-)$ in the backward recursion (9), we will use approximations $\overline{V}(v^+)$ and $\overline{V}(v^-)$ respectively. Hull-White use linear interpolation to compute these approximate values. Consider for instance $\overline{V}(v^+)$. From the way we chose the ranges (a_{ki}, b_{ki}), there must exist some s in the range $(a_{k+1,i+1}, b_{k+1,i+1})$ such that

$$x_1 = S_0 e^{sh} \;\le\; x = A(v^+) \;\le\; x_2 = S_0 e^{(s+1)h}. \qquad (10)$$

We have already computed the option values $y_1 = V[k+1][i+1][s]$ and $y_2 = V[k+1][i+1][s+1]$. We therefore approximate the value $V(v^+)$ by the interpolated value

$$\overline{V}(v^+) = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x - x_1).$$

We finally compute $V(v) = V[k][i][j]$ using expression (9) with $V(v^+)$ and $V(v^-)$ on the right hand side replaced by their approximations $\overline{V}(v^+)$ and $\overline{V}(v^-)$ respectively. In this way we eventually compute $V[0][0][0]$, which is the time-0 value of the American-style Asian call option.

■ Write the complete code to implement the Hull-White method described above. In addition to the usual inputs to this code, we now have an additional input: the grid-size h that determines the "granularity" of the interpolation.

■ For small n (less than 15 or so), compare the answers you get from the Hull-White method using various grid sizes h (use for example $h = 0.1$ and lower), with the exact values using the recursive-function method of Section 3.3. Also see how much improvement in accuracy you get by reducing the grid size h. Are the Hull-White answers consistently above or below the exact answers? (It can be shown that the Hull-White approximation is in fact an *upper-bound* on the exact price) Run the Hull-White algorithm for $h = 0.05$ and $n = 50$, and see how long it takes. Compare this with the length of time you estimated the recursive function-based program would take (you probably will not want to wait for the answer using that program!).

6 Black-Derman-Toy: A Binomial Term-structure Model

As before, let T be the time horizon, and let n be the number of time-divisions, so that $\Delta t = T/n$. The short-term interest rate, or **short-rate**, at time k, is defined as the risk-free interest rate that holds from (discrete) time k to time $k+1$. In other words, if r is the short-rate at time k, then investing one dollar in a risk-free instrument at time k will result in a payoff of $\exp\{r\Delta t\}$ at time $k+1$. In the models of the previous sections, we assumed that the short-rate at all times k is a fixed constant. We will now consider a model where the short-rate is random, and follows a binomial process somewhat like the stock-price process introduced earlier. We will for the moment ignore the model for the stock price process.

6.1 The model

The short-rate model we will describe is called the **Black-Derman-Toy** (BDT) model [1, 4]. The model is described in terms of the binomial *lattice* introduced in Section 2.2. Recall the notation introduced in Section 2.4, where a node v in the binomial lattice represents a certain state, and $Z(v)$ denotes the value of a random variable Z in state v. Recall also that $D(v)$ is the depth of v, i.e., the discrete time corresponding to v. Finally, recall that v^+ and v^- are the up- and down-successors of v.

For any *non-terminal* node v, let $r(v)$ be the short-rate that holds in state v from time $D(v)$ to time $D(v)+1$. To illustrate, for $v = (3,-1)$, $r(v) = 0.1$ means that if the state at time 3 is $(3,-1)$, then the short rate from time 3 to time 4 is 10%. This means one dollar invested in a risk-free instrument at time 3 in state $(3,-1)$ will result in a guaranteed payoff of $\exp\{r(v)\Delta t\}$ at time 4. In particular, for the starting node $v_0 = (0,0)$, $r(v_0)$ is the *fixed, known, non-random* short-rate at time 0. (This is analogous to knowing the initial stock price S_0 in the stock price binomial model.) The BDT short-rate model has two sets of parameters:

$$a_0, a_1, \ldots, a_{n-1},$$

and

$$b_0, b_1, \ldots, b_{n-1}.$$

For $0 \le k < n$, the short-rate $r(v)$ at lattice node $v = (k,i)$ is given by

$$r(v) \equiv r(k,i) = a_k (b_k)^i, \quad k \ge 1. \tag{11}$$

Notice the similarity of this model to the stock-price model (3): if $a_k = r(0,0)$ (the initial short-rate) for all k, and $b_k = u$ for all k, the short-rate model would be exactly the same as the stock price model.

The short-rate $r(v)$ is closely related to the **one-period discount factor** $f(v)$ in state v:

$$f(k,i) = \exp\{-r(k,i)\Delta t\}.$$

This means that if the current state is $v = (k,i)$, then one dollar at time $k+1$ is worth $f(k,i)$ in the current state. Finally, the probability on every branch in the lattice is assumed to be 0.5.

6.2 Pricing short-rate securities using the model

Consider an arbitrary European-style security with expiration n, whose payoff $G(v)$ depends only on the short-rate $r(v)$. Such a security is clearly *path-independent*. Recall that a European-style security can be exercised only at expiration. The BDT model can be used to price such an option. As you might expect, the pricing of a short-rate derivative in the short-rate model is exactly analogous to the pricing of a stock-price derivative in the earlier stock-price model. In that earlier model, the derivative's payoff depended on the stock price; in the present case, the payoff of the security depends only on the short-rate. Indeed, the **aribtrage-free value** $V(v)$ at node v of such a security is defined as the *expectation of the discounted payoff*, over paths from v to level n in the lattice.

We now make this more precise. For any path π in the lattice, let $\mathbf{P}(\pi)$ denote its **probability**, which in this case is simply $\frac{1}{2}$ to the power of the length (i.e. number of edges) of the path. Let $F(\pi)$ be the **discount** on a path π, defined as the product of the one-period discount factors along the path. Abusing notation, define the **payoff** $G(\pi)$ on a path π to be the payoff at the node at the end of the path. For instance, in Fig. 1, for the path

$$\pi = (0,0)(1,1)(2,0),$$

the payoff $G(\pi) = G(2,0)$, the probability $\mathbf{P}(\pi) = (\frac{1}{2})^2 = 1/4$, and the discount $F(\pi) = f(0,0)f(1,1)$. Note that we do *not* include $f(2,0)$ in $F(\pi)$ since $f(2,0)$ applies for the period from time 2 to time 3, which is not covered by the path π (it only goes up to time 2).

Now we can state our definition of $V(v)$ more precisely: it is the sum over all paths π from v to level n, of the product $\mathbf{P}(\pi)F(\pi)G(\pi)$; this is exactly the expected discounted payoff. For example, in the lattice of Fig. 1, consider a European option expiring at time 2, whose payoff function G is

$$G(2,2) = 0, \quad G(2,0) = 1, \quad G(2,-2) = 2.$$

Suppose $f(v) = .9$ at all nodes v. Then to compute the value $V(0,0)$ of this security at node $(0,0)$, notice that there are four paths from $(0,0)$ to level 2 (the expiration time):

up-up, up-down, down-up, down-down,

and the probability on each path is $1/4$. Therefore,

$$\begin{aligned}
V(0,0) &= \frac{1}{4}f(0,0)f(1,1)G(2,2) + \frac{1}{4}f(0,0)f(1,1)G(2,0) \\
&\quad + \frac{1}{4}f(0,0)f(1,-1)G(2,0) + \frac{1}{4}f(0,0)f(1,-1)G(2,-2) \\
&= \frac{1}{4}(0.9)^2 + \frac{1}{4}(0.9)^2 + \frac{1}{4}(0.9)^2 2 \\
&= 0.81
\end{aligned}$$

From this you can easily see that the value $V(v)$ of a European short-rate option expiring at n is given by the *backward induction*

$$V(v) = \begin{cases} G(v) & \text{if the depth } D(v) \text{ is } n, \\ \frac{1}{2}f(v)\left(V(v^+) + V(v^-)\right) & \text{otherwise} \end{cases} \tag{12}$$

which is exactly analogous to expressions (6) and (7) for stock options.
■ Write code using the above backward induction to price a European call option on the short rate, with strike rate K. This is analogous to a call option on the stock price. The payoff $G(v)$ of such an option at node $v = (k, i)$ is given by

$$G(v) = (r(v) - K)^{+} = \left[a_k (b_k)^i - K\right]^{+}.$$

The input parameters to your code are the option expiration time T, the number of time-divisions n, the strike K, and the initial short-rate $r(0, 0)$. Assume that the BDT parameters $b_i = 1.01$ for all i, and $a_i = r(0, 0)$ for all i. Your code should use the fast dynamic programming approach you implemented in Section 4.1 (i.e., do *not* use a recursive function).

6.3 Green's function and forward induction

We will now consider an alternative, **forward-inductive** way to compute the value of a European short-rate option. First we will need the concept of a **pure-state security**. For a lattice node (k, i), a pure-state security $s(k, i)$ is simply a (European) short-rate security expiring at time k that pays off \$ 1 *only at* (k, i) and 0 everywhere else.

Now for a lattice node (k, i), **Green's function** [2, 4] $H(k, i)$ is defined as the time-0 value of the pure-state security $s(k, i)$. For instance, in Fig. 1, $H(2, 0)$ is the time-0 value of the pure-state security $s(2, 0)$, i.e., a European short-rate option expiring at time 2 that pays \$ 1 at node $(2, 0)$ and zero at all others. Using the definition of "value" given in the previous sub-section, we compute $H(2, 0)$ as follows: We take the sum over all paths π from $(0, 0)$ to $(2, 0)$, of the product $\mathbf{P}(\pi)F(\pi)$:

$$H(2,0) = \frac{1}{4} f(0,0) f(1,1) \; + \; \frac{1}{4} f(0,0) f(1,-1)$$

Note that $H(k, i)$ can be computed by a simple *forward induction:*

$$\begin{aligned} H(k+1, i) &= \frac{1}{2}\left[H(k, i-1) f(k, i-1) + H(k, i+1) f(k, i+1)\right] \quad \text{if } |i| \le k-1, \\ &= \frac{1}{2} H(k, i-1) f(k, i-1) \quad \text{if } i = k+1, \qquad (13) \\ &= \frac{1}{2} H(k, i+1) f(k, i+1) \quad \text{if } i = -k-1, \end{aligned}$$

The initial condition is $H(0, 0) = 1$.

Note that an arbitrary short-rate security expiring at time n with payoff $G(v)$ can be viewed as a combination of pure-state securities. For instance, recall the example considered before of a European short-rate security expiring at time 2, with payoffs given by

$$G(2,2) = 0, \quad G(2,0) = 1, \quad G(2,-2) = 2.$$

This can be viewed as a combination of 0 pure-state securities $s(2,2)$, one pure-state security $s(2,0)$ and 2 pure-state securities $s(2,-2)$. Naturally the time-0 value of the combined security is simply the sum of the time-0 values of the appropriate multiple of the individual pure-state securities. In this example, the value of the combined security is the sum of the value of the pure-state security $s(2,0)$ (which is $H(2,0)$) plus 2 times the value of $s(2,-2)$ (which is $H(2,-2)$).

In general, the time-0 value of a European short-rate security expiring at time $(n+1)$ with payoff function $G(v)$ is given by

$$V(0,0) = \sum_{i=-n-1}^{n+1} H(n+1,i)G(n+1,i) \tag{14}$$

In fact $V(0,0)$ can also be written as

$$V(0,0) = \sum_{i=-n}^{n} H(n,i)f(n,i)\frac{1}{2}[G(n+1,i+1) + G(n+1,i-1)] \tag{15}$$

■ Write code to price the short-rate call option of the previous subsection, but this time by *forward induction* using the above Green's function approach (14). Use an array `H[][]` to store the Green's function values. First compute the Green's function values by forward induction using (13). Then compute the option value using (14). Assume the same input parameters and BDT-parameters as in the previous subsection.

6.4 Pricing bonds

What do we really mean when we say that the "risk-free rate of interest" for a short period of time Δt is r? This means there is a certain risk-less instrument with the property that that if you invest \$ 1 in this instrument at time 0, then you will receive a payoff of $e^{r\Delta t}$ dollars at time at time Δt. A government bond is an example of a risk-less instrument. More precisely, a **zero-coupon bond** (also informally called a "zero," or a "pure-discount bond") maturing at discrete time k is an instrument that pays 1 dollar at discrete time time k (which is continuous time $k\Delta t$). Unlike a call option, the payoff from a bond at maturity is fixed and not uncertain. This is why a bond is called a *risk-less* instrument, or a *fixed-income* instrument. Of course, to own a bond, one must pay a price, and we now consider how to compute this price.

Consider an $(m + 1)$-maturity bond. This is simply a European short-rate security that expires at time $(m + 1)$ and pays exactly \$ 1 at *every* lattice-node at level $m + 1$, i.e., $G(v) = 1$ for every node v at level $m + 1$. Therefore, in terms of Green's function, we can write the time-0 value of the $(m + 1)$-maturity bond, written B_{m+1}, in two different ways (see (14) and (15)):

$$B_{m+1} = \sum_{i=-m-1}^{m+1} H(m+1, i) \tag{16}$$

$$= \sum_{i=-m}^{m} H(m, i) f(m, i) \tag{17}$$

The second form will be useful for us later, when we fit the parameters of the short-rate model so that the bond prices computed from it match their known market prices.

■ Modify the code of the last section to compute the time-0 value of a bond maturing in time T. Input parameters to your code are the same as in the previous subsection (except that you don't need the strike K in this case). Assume the same BDT parameters as in the previous subsection. Price the bond using the forward-inductive approach using Green's function (expression 16). Use an array $H[][]$ to store the Green's function values. First fill in the array up to $H[n][]$ and then use expression 16 to compute the bond value B_n.

■ Return to the backward-induction approach you implemenemented in Section 6.2 using (12). Use that approach to price a T-maturity bond at *every* node v of the lattice. Notice that to start the backward induction, at the terminal nodes v you must use $V(v) = G(v) = 1$, since a bond is worth exactly 1 dollar when it matures. The rest of your code would look just like the code for the short-rate call option you implemented in that Section. (You will need the bond values $V(v)$ at every lattice node for the next section).

6.5 Pricing bond options

In the last exercise of the previous section you computed the value $V(v)$ (which we will here denote by $B(v)$) of a T-maturity bond at every lattice node. Now we can define call and put options on bonds exactly like the corresponding options on stocks; the only difference is that the underlying variable for bond options is the bond price rather than the stock price. Specifically, for $m < n$, an m-period European-style **call option** with strike K on an n-maturity bond can only be exercised at time m, and the payoff at a level-m node v is given by

$$G(v) = (B(v) - K)^+.$$

Notice that there are two different maturities involved here: the maturity of the bond, which is longer than the maturity of the option on the bond.

In the BDT model, bond options are valued using the same backward-recursion approach based on (12) that you implemented in the last Section and in Section 6.2. To compute the option price, you will start at the end of the lattice (level n) and compute the bond values $B(v)$ at each node v by backward induction (as you did in the last Section), until you reach level m. At levels below m you will no longer need the bond values. At level m, you will also compute the terminal values of the option. In particular, at a node v at level m, from Eq. (12), you will compute the option value $V(v)$ as

$$V(v) = G(v) = (B(v) - K)^+.$$

At a node v at a level smaller than m, you will use the second equation in (12) to compute the option value, and finally arrive at $V(0, 0)$.

■ Write code to price a European call-option with strike K expiring at time $T/2$ on a bond that matures at time T. Use $m = \lfloor n/2 \rfloor$ as the expiration time for the option (this roughly represents continuous-time $T/2$). Your input parameters should be T, n, K as before. Assume the same BDT model as before.

6.6 Fitting the model to current bond yields

So far we just assumed that the parameter-sets $\{a_i\}$ and $\{b_i\}$ came "out of the blue". Of course, for the model to be useful, these parameters should be chosen so that they are consistent with the current market data. We now describe how this is done.

The **yield** Y_m of an m-maturity bond (at time 0) is defined as the effective constant interest-rate that would result in the same growth as an m-maturity bond. In other words, Y_m satisfies

$$\frac{1}{B_m} = \exp\{Y_m m\Delta t\}, \tag{18}$$

or

$$Y_m = -\frac{1}{m\Delta t} \ln B_m.$$

(Recall that B_m is the time-0 price or value of the m-maturity bond). The **current term-structure** of interest rates is a specification of the current yields

$$Y_1, Y_2, \ldots, Y_n,$$

of bonds maturing at (discrete) times $1, 2, \ldots, n$. Note that from the above equations, this is equivalent to specifying the current prices of these bonds.

In this sub-section we will consider how to fit the BDT model parameters $\{a_i\}$ to the current term-structure, given the parameters $\{b_i\}$. In other words, we would like to pick the $\{a_i\}$ parameters so that the bond-prices (or equivalently, bond-yields) match the current term-structure given by the $\{Y_i\}$. This is done by the following algorithm:

1. Initially set $a_0 = Y_1$, and set the Green's function values $H(0,0) = 1$,

   $$H(1,-1) = H(1,1) = (0.5)f(0,0) = (0.5)\exp\{-Y_1\Delta t\},$$

   and let $k = 1$. In the code, you will have an array `H[][]` to store the Green's function values.

2. Compute the a_k value as follows. At this stage, the $H(k,i)$ values are known for all $i = -k, \ldots, k$. From the input data Y_{k+1} is known, which means B_{k+1} is known (expression (18)). From expression (17), B_{k+1} must satisfy

   $$\begin{aligned} B_{k+1} &= \sum_{i=-k}^{k} H(k,i)f(k,i) \\ &= \sum_{i=-k}^{k} H(k,i)\exp\{-r(k,i)\Delta t\} \\ &= \sum_{i=-k}^{k} H(k,i)\exp\{-a_k(b_k)^i\Delta t\} \qquad (19) \end{aligned}$$

   Since the $\{b_i\}$ parameters are known, the only unknown above is a_k, so call the expression (19) $g(a_k)$, to emphasize that it is a function of a_k. You therefore need to find a_k so that $B_{k+1} = g(a_k)$, i.e., you must solve the (non-linear) equation $\Psi(x) = 0$ where $\Psi(x) \equiv g(x) - B_{k+1}$. You can use the standard Newton-Raphson (see Appendix) method, with initial guess $x_0 = a_{k-1}$, to solve this equation for x; this solution will be the a_k value.

3. Compute $H(k+1,i)$ for $i = -k-1, \ldots, k+1$ using the forward induction (13), with

   $$f(k,i) = \exp\{-r(k,i)\Delta t\} = \exp\{-a_k(b_k)^i\Delta t\}.$$

(In this expression, use the a_k value just computed.)

4. Increment k by 1. If $k < n$, return to step 2; otherwise, stop.

■ (a) Implement the above algorithm to find the BDT parameters $a_0, a_1, a_2, \ldots, a_{n-1}$ so that they fit the current term-structure. Assume that $b_i = 1.01$ for all i, and that the current term-structure is given by the bond prices $B_i = (0.99)^i$, for $i = 1, 2, \ldots, n$. The input parameters to your code are T and n. Given any T, n as input, your code should output $a_0, a_1, \ldots, a_{n-1}$.
■ (b) Use your $\{a_i\}$ values to compute the bond prices B_i, $i = 1, 2, \ldots, n$, using the approach you used in Subsection (6.4). These should match the specified values $(0.99)^i$. Note that you can use the Green's function values that you just computed in the algorithm above, and use expression (16) to compute the bond prices.
■ (c) Use your $\{a_i\}$ values to price a call option on the short-rate that expires at time $n - 1$ with strike K (this was described in Subsection 6.2), using the approach used in Subsection 6.3. Again note that you can use the Green's function values that you just computed in part (a), and use expression (14) to compute the option value.

References

[1] F. Black, E. Derman, and W. Toy. A one-factor model of interest rates and its applications to treasury bond options. *Financial analysts journal*, pages 33–39, February 1990.

[2] D. Duffie. *Dynamic Asset Pricing Theory*. Princeton University Press, 2 edition, 1996.

[3] J. Hull and A. White. Efficient procedures for valuing european and american path-dependent options. *Journal of Derivatives*, 1:21–31, 1993.

[4] F. Jamshidian. Forward induction and construction of yield curve diffusion models. *Journal of fixed income*, pages 62–74, June 1991.

APPENDIX

A The Newton-Raphson method to solve single-variable equations

If you have code to compute a one-variable function $f(x)$, the Newton-Raphson method can be used to find the value of x for which $f(x) = 0$. The algorithm is as follows. Let $\alpha > 0$ be the solution accuracy desired (use $\alpha = 10^{-6}$), and let $\delta > 0$ be an "infinitesimal" quantity (use $\delta = 10^{-10}$). Let MAX be the maximum number of iterations by which the algorithm should succeed.

1. Set $x = x_0$, where x_0 is a "good guess" for the solution. Let $k = 1$.
2. Set
$$x' = x - \frac{f(x)}{f'(x)},$$
where $f'(x)$ is the derivative of f w.r.t. x, and is computed by the approximation
$$f'(x) \simeq \frac{f(x+\delta) - f(x)}{\delta}.$$
3. Increment k by 1, and set $x = x'$.
4. If $|f(x)| < \alpha$, STOP (x is the desired solution).
5. Otherwise, if $k > MAX$ STOP and print FAILURE (since no solution has been found within MAX iterations).
6. Otherwise, return to step 2.