

Queens College, CUNY, Department of Computer Science

Computational Finance

CSCI 365 / 765

Spring 2018

Instructor: Dr. Sateesh Mane

© Sateesh R. Mane 2018

April 16, 2018

due Sunday April 22, 2018 at 11.59 pm

7 Homework: Binomial model

7.1 Function signature

- Let us write a simple (but working) C++ function to implement the binomial model.
- The input arguments are
 1. The stock price S .
 2. The strike price K .
 3. The risk-free interest rate r . (We shall use decimal, not percent.)
 4. The continuous dividend yield q . (We shall use decimal, not percent.)
 5. The stock volatility σ . (We shall use decimal, not percent.)
 6. The expiration time T . (Measured in years.)
 7. The current time t_0 . (Measured in years.)
 8. **boolean** “call” (true for a call, false for a put).
 9. **boolean** “American” (true for an American option, false for European).
 10. **int** n , the number of timesteps ($n \geq 1$).
 11. **reference to double, output** the option fair value V .
- The function signature is

```
int binomial_simple(double S,  
    double K,  
    double r,  
    double q,  
    double sigma,  
    double T,  
    double t0,  
    bool call,  
    bool American,  
    int n,  
    double & V);
```

- **The return type is int because we shall perform validation checks.**

7.2 Validation tests

- Write the following validation checks at the top of the function:
- If $n < 1$ return 1 (fail).
- If $S \leq 0$ return 1 (fail).
- If $T \leq t_0$ return 1 (fail).
- If $\sigma \leq 0.0$ return 1 (fail).

7.3 Parameters

- Next calculate the following parameters (all are of type `double`):

```
double dt = (T-t0)/double(n);  
double df = exp(-r*dt);  
double growth = exp((r-q)*dt);  
double u = exp(sigma*sqrt(dt));  
double d = 1.0/u;  
  
double p_prob = (growth - d)/(u-d);  
double q_prob = 1.0 - p_prob;
```

- Perform some more validation tests:
- If $p_{\text{prob}} < 0.0$ return 1 (fail).
- If $p_{\text{prob}} > 1.0$ return 1 (fail).

7.4 Allocate memory/set up arrays

- You can implement this differently.
- You can make use of STL vectors, etc. It may be a better implementation.
- The essential goal is to have a two dimensional array for the (i) stock nodes, (ii) option nodes.
- Note the “ $n + 1$ ” because n timesteps requires arrays of length $n + 1$.
- Here is an implementation using standard C++ arrays:

```
// allocate memory
double **stock_nodes = new double*[n+1];
double **option_nodes = new double*[n+1];

for (i = 0; i <= n; ++i) {
    stock_nodes[i] = new double[n+1];
    option_nodes[i] = new double[n+1];

    S_tmp = stock_nodes[i];
    V_tmp = option_nodes[i];
    for (j = 0; j <= n; ++j) {
        S_tmp[j] = 0;
        V_tmp[j] = 0;
    }
}
```

- If you employ C++ arrays as above, remember to deallocate memory at the end.

```
// deallocate memory
for (i = 0; i <= n; ++i) {
    delete [] stock_nodes[i];
    delete [] option_nodes[i];
}
delete [] stock_nodes;
delete [] option_nodes;
```

- Note that the memory allocation must be performed AFTER the validation checks all pass, else you will have a memory leak (or complicated code).

7.5 Set up stock prices in nodes

- Now we must fill the stock price nodes with the appropriate stock prices.
- Note that the arrays are rectangular, whereas the binomial tree is triangular.
- Hence we are allocating too much memory by a factor of 2, but never mind for now.
- **You can do this differently, if you use STL, etc.**
- I declare pointers “S_tmp” and “V_tmp” to help out, but you can do it differently.
- Fill the first node `stock_nodes[0][0]`.

```
S_tmp = stock_nodes[0];  
S_tmp[0] = S;
```

- Fill the remaining (relevant) nodes.
- We use i to index the time steps and j to index the price steps.

```
for (i = 1; i <= n; ++i) {  
    double * prev = stock_nodes[i-1];  
    S_tmp = stock_nodes[i];  
    S_tmp[0] = prev[0] * d;  
    for (j = 1; j <= n; ++j) {  
        S_tmp[j] = S_tmp[j-1]*u*u;  
    }  
}
```

- **Test your function.**
- **Call the function “as is” from a main program, with a small value of n , such as 1,2,3 and print the values of i , j and the stock price at the node (i,j) .**
- **Verify that the stock prices are correct at every node.**

7.6 Terminal payoff

- Now we begin the valuation process.
- We fill the option nodes at $i = n$ with the terminal payoff.

```
i = n;
S_tmp = stock_nodes[i];
V_tmp = option_nodes[i];
for (j = 0; j <= n; ++j) {
    double intrinsic = 0;
    ...
    V_tmp[j] = intrinsic;
}
```

- The value of “intrinsic” depends on the boolean “call” (true for call, false for put).
- For a call option, if $S_tmp[j] > K$ then $intrinsic = S_tmp[j] - K$.
- For a put option, if $S_tmp[j] < K$ then $intrinsic = K - S_tmp[j]$.
- **Test your function.**
- **Call the function “as is” from a main program, with smalls value of n and print the values of the stock price and option terminal value.**
- **Verify that the option terminal values are correct at every node for $i = n$.**

7.7 Main valuation loop

- Now we begin the main valuation loop.
- **Time: we loop backwards through values of i from $i = n - 1$ to $i = 0$.**
- **Stock: for each value of i we loop through values of j from $j = 0$ to $j = i$.**
- ***** You should check that you understand why the loop limit is $j = 0$ to $j = i$. *****
- The loops are as follows:

```
for (i = n-1; i >= 0; --i) {  
    ...  
    for (j = 0; j <= i; ++j) {  
        ...  
    }  
}
```

- **I use pointers to help out. You can do it differently using STL, etc.**
- We calculate the discounted expected values using the formula in the lectures

```
for (i = n-1; i >= 0; --i) {  
    S_tmp = stock_nodes[i];  
    V_tmp = option_nodes[i];  
  
    double * V_next = option_nodes[i+1];  
    for (j = 0; j <= i; ++j) {  
        V_tmp[j] = df*(p_prob*V_next[j+1] + q_prob*V_next[j]);  
  
        // early exercise test  
        if (American) {  
            ...  
        }  
    }  
}
```

- **The boolean “American” indicates if a test is required for early exercise.**
- ***** You *** should know how to implement the relevant tests and write the code.**

7.8 Option fair value

- This is easy. It is just the value at the $(0,0)$ option node.

```
// option fair value
i = 0;
V_tmp = option_nodes[i];
V = V_tmp[0];
```


7.9 Memory deallocation

- This should be the entire function. Remember to release any allocated memory.
- Return with 0 (success).

7.10 Tests

- Write a main program to call your function with the following inputs
 1. $S = 100$
 2. $K = 100$
 3. $r = 0.1$
 4. $q = 0.0$
 5. $\sigma = 0.5$
 6. $T = 0.3$
 7. $t_0 = 0.0$
 8. $n = 3$
- Call the function for four cases (by setting the booleans) American/European, call/put.
- **Verify that you obtain the same option values as in Lecture 17a.**

$$c \simeq 13.1588, \quad (7.10.1a)$$

$$p \simeq 10.2034, \quad (7.10.1b)$$

$$C \simeq 13.1588, \quad (7.10.1c)$$

$$P \simeq 10.4549. \quad (7.10.1d)$$

- The values of c and C are equal because if the stock does not pay dividends, then the fair values of an American and European call are equal.
- However the American put has a higher fair value than the European put.

7.11 New calculations

- **Now let us perform new calculations!**
- Set $r = q = 0.1$.
- Set $T = 1$.
- Set $n = 100$.
- The other input values can remain the same $S = K = 100$ etc.
- Call the function for four cases (by setting the booleans) American/European, call/put.
- **If you have done your work correctly, you should find that $C = P$ and $c = p$.**
- Additional tests:
 1. Change the values of T and σ .
 2. Change the values of S and K **but keep $S = K$.**
 3. Change the values of r and q **but keep $r = q$.**
- **If you have done your work correctly, you should find that $C = P$ and $c = p$ in all cases.**