Queens College, CUNY, Department of Computer Science

Computational Finance

CSCI 365 / 765

Fall 2017

Instructor: Dr. Sateesh Mane

November 16, 2017

due Monday November 27, 2017 at 11.59 pm

8 Homework: Binomial model 2

8.1 Outline of work

- Recall the function `binomial_simple()` from Homework 7.
- The function `binomial_simple()` takes many function arguments, which should really be encapsulated in a derivatives class object.
- The function `binomial_simple()` internally allocates and deallocates memory in each function call.
- This is wasteful: the memory allocation for the binomial tree itself depends only on the value of n. If the function is called in a loop, with the same value of n every time, the memory allocation is exactly the same for every function call.
- In this homework, we shall write some C++ classes to do a better job.

8.2 Recapitulation

- Consider the following C++ code, to calculate the fair values of: (i) American put, (ii) European put, (iii) American call, (iv) European call.
- There are totally 8000 function calls and the memory allocation is the same for them all.
- **Complete the code below and run it. Show me your completed code.**

```
double S = 0;
double K = 100.0;
double r = 0.05;
double q = 0.01;
double sigma = 0.5;
double T = 1.0;
double t0 = 0.0;

double FV_Am_put = 0;
double FV_Eur_put = 0;
double FV_Am_call = 0;
double FV_Eur_call = 0;

int n = 100;

double dS = 0.1;
int imax = 2000;
for (int i = 1; i <= imax; ++i) {
  S = i*dS;

  binomial_simple(S, K, r, q, sigma, T, t0, false, true,  n, FV_Am_put);
  binomial_simple(S, K, r, q, sigma, T, t0, false, false, n, FV_Eur_put);
  binomial_simple(S, K, r, q, sigma, T, t0, true,  true,  n, FV_Am_call);
  binomial_simple(S, K, r, q, sigma, T, t0, true,  false, n, FV_Eur_call);

  // print output to file
  outfile << S << "  ";
  outfile << FV_Am_put << "  ";
  outfile << FV_Eur_put << "  ";
  outfile << FV_Am_call << "  ";
  outfile << FV_Eur_call << "  ";
  outfile << std::endl;
}
```

8.3 Derivative base class

- First we encapsulate data in a derivatives class.
- We might as well call it `Derivative`.
- It is an abstract base class and contains virtual functions and a virtual destructor.
- Some of the data items below (such as the risk free interst rate) really don't belong in this class, but never mind for now.
- The stock price S is not included because its value changes every day in the stock market and is not a constant.
- **Write the class below.**
- **As a general policy, all functions with return type `int` return 0 for success and 1 for fail.**
- The virtual function `double TerminalPayoff(double S);` returns the terminal payoff of the derivative for a stock price value of S.
- The virtual function `int ValuationTests(double S, double & V);` is called in a loop by the binomial model. (We shall see this later.) This function checks if the value of V should be updated to the intrinsic value of the drivative.

```
class Derivative
{
public:
  virtual ~Derivative() {}

  virtual double TerminalPayoff(double S) { return 0; }
  virtual int ValuationTests(double S, double & V) { return 0; }

  // data
  double r;
  double q;
  double sigma;
  double T;

protected:
  Derivative() { r = 0; q = 0; sigma = 0; T = 0; }
};
```

8.4 Option derived class

- We define a derived class `Option` to fill in the virtual functions.
- The `Option` class contains additional data items.
  1. The strike price `K` and two Booleans `isCall` and `isAmerican`.
  2. The definitions of all of the should be obvious.
  3. All the data members are public. The calling application will set their values.
- **Write the class below.**
- ***** Write the code for the virtual functions. *****
- **Use the Booleans to write the code for put/call and American/European options.**

```
class Option : public Derivative
{
public:
  Option() { K = 0; isCall = false; isAmerican = false; }
  virtual ~Option() {}

  virtual double TerminalPayoff(double S);
  virtual int ValuationTests(double S, double &V);

  // data
  double K;
  bool isCall;
  bool isAmerican;
};

double Option::TerminalPayoff(double S)
{
  //  *** RETURN TERMINAL PAYOFF FOR PUT OR CALL OPTION ***
}

int Option::ValuationTests(double S, double &V)
{
  //  *** TEST IF THE VALUE OF V SHOULD BE UPDATED TO THE INTRINSIC VALUE ***
}
```

8.5 Binomial model

8.5.1 Declaration of class

- The binomial model should also be made into a class `BinomialModel`.
- The value of n will determine how much memory to allocate.
- The implementation below uses C++ pointers and arrays.
- ***** You do not need to use C++ arrays. You can use STL vectors, etc. *****
- The memory allocation and deallocation is internal to the model and is private.

```
class BinomialModel
{
public:
  BinomialModel(int n);
  ~BinomialModel();

  int FairValue(int n, Derivative * p_derivative, double S, double t0, double & V);

private:
  // methods
  void Clear();
  int Allocate(int n);

  // data
  int n_tree;
  double **stock_nodes;
  double **derivative_nodes;
};
```

8.5.2 Constructor destructor and memory release

- The implementation below uses C++ pointers and arrays.
- ***** You do not need to use C++ arrays. You can use STL vectors, etc. *****
- **Write the function `Clear()` to release allocated memory.**
- ***** Make sure your class functions do not have a memory leak. *****

```
BinomialModel::BinomialModel(int n)
{
  n_tree = 0;
  stock_nodes = 0;
  derivative_nodes = 0;
  Allocate(n);
}

BinomialModel::~BinomialModel()
{
  Clear();
}

void BinomialModel::Clear()
{
// *** WRITE THE FUNCTION TO RELEASE ALLOCATED MEMORY ***
}
```

8.5.3 Memory allocation

- The implementation below uses C++ pointers and arrays.
- ***** You do not need to use C++ arrays. You can use STL vectors, etc. *****
- Now we come to the key feature of allocating memory for the binomial tree.
  1. Suppose the binomial model is called for the first time.
  2. The number of timesteps is $n = 100$.
  3. Hence memory for a tree with 100 timesteps is allocated.
  4. Suppose the binomial model is called again, *but with a smaller value of* n, say $n = 99$.
  5. We do ***not*** need to deallocate the old tree and allocate new memory. The previous tree which was allocated (for $n = 100$ steps) has enough storage to value a derivative using $n = 99$ steps.
  6. However, suppose the binomial model is called with a *larger value of* n, say $n = 101$.
  7. Now we must deallocate the old tree and allocate new memory for a new, larger tree.
- **Hence `Allocate(int n)` should deallocate the old tree and allocate new memory only if `n > n_tree`.**
- The function `Allocate(int n)` should call `Clear()` to deallocate memory.
- **Write the function `Allocate(int n)`.**
- **Return 0 on success, return 1 if the memory allocation fails.**
- ***** Make sure your class functions do not have a memory leak. *****

```
int BinomialModel::Allocate(int n)
{
  if (n <= n_tree) return 0;

  // deallocate old tree
  Clear();

  // allocate memory
  n_tree = n;

  // *** WRITE THE FUNCTION TO ALLOCATE NEW MEMORY ***
}
```

8.5.4 Valuation of derivative Part 1

- Finally we write the public function `FairValue(...)` to calculate the fair value of a derivative.
- The function `FairValue(...)` is essentially a copy of `binomial_simple(...)`.
- Initialize `V=0.0`.
- Validate the input data. Return 1 (fail) if $n < 1$ or $S \leq 0$ or `p_derivative == NULL` or `p_derivative->T <= t0` or `p_derivative->sigma <= 0.0`.
- Calculate the parameters. Get the values of `r`, `q`, `T` and `sigma` from `p_derivative`.

```
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;
```

- Validation check: return 1 (fail) if `p_prob < 0.0` or `p_prob > 1.0`.
- Call `Allocate(n)` to allocate memory for the binomial tree.
- Populate the elements of `stock_node` with the appropriate stock prices. See `binomial_simple(...)`.
- Populate the elements of `derivative_nodes` at step `i = n` with the terminal payoff.

```
i = n;
S_tmp = stock_nodes[i];
V_tmp = derivative_nodes[i];
for (j = 0; j <= n; ++j) {
  V_tmp[j] = p_derivative->TerminalPayoff(S_tmp[j]);
}
```

- ***** IMPORTANT *** Explain why we MUST use `n` and NOT `n_tree`.**
- We call the virtual function `TerminalPayoff(...)` of the derivative class.
- The calculation of the terminal payoff belongs in the derivative class. In this way we can use a binomial model object to value many different types of equity derivatives.

8.5.5 Valuation of derivative Part 2

- The main valuation loop is copied from `binomial_simple(...)`.
- However, the valuation tests are performed by the derivative object.

```
// valuation loop
for (i = n-1; i >= 0; --i) {
  S_tmp = stock_nodes[i];
  V_tmp = derivative_nodes[i];
  double * V_next = derivative_nodes[i+1];
  for (j = 0; j <= i; ++j) {
    V_tmp[j] = df*(p_prob*V_next[j+1] + q_prob*V_next[j]);
    p_derivative->ValuationTests(S_tmp[j], V_tmp[j]);  // VALUATION TESTS
  }
}
```

- ***** IMPORTANT *** Explain why we MUST use `n` and NOT `n_tree`.**
- We call the virtual function `ValuationTests(...)` of the derivative class.
- The valuation tests belong in the derivative class. In this way we can use a binomial model object to value many different types of equity derivatives.
- **Set the value of `V` and exit.**

```
// derivative fair value
V_tmp = derivative_nodes[0];
V = V_tmp[0];

return 0;
```

8.5.6 Valuation of derivative Part 3

Write the complete function `FairValue(...)`.

```
int BinomialModel::FairValue(int n,
                             Derivative * p_derivative,
                             double S, double t0, double & V)
{
  int rc = 0;

  V = 0;

  // validation checks
  ...

  // declaration of local variables (I use S_tmp and V_tmp)
  ...

  // calculate parameters
  ..

  // more validation checks
  ...

  // allocate memory if required (call Allocate(n))
  ...

  // set up stock prices in tree
  ...

  // set terminal payoff  (call virtual function in derivative class to calculate payoff)
  ...

  // valuation loop  (call virtual function in derivative class for valuation tests)
  ...

  // option fair value
  V_tmp = derivative_nodes[0];
  V = V_tmp[0];

  return 0;
}
```

8.6 Calling application

I shall run the following function to test your code.
I shall also perform other tests (memory allocation, etc.

```
int binomial_test()
{
  int rc = 0;

  // output file
  std::ofstream ofs("output.txt");

  double S = 100;
  double K = 100;
  double r = 0.05;
  double q = 0.01;
  double sigma = 0.5;
  double T = 1.0;
  double t0 = 0;

  Option Eur_put;
  Eur_put.r = r;
  Eur_put.q = q;
  Eur_put.sigma = sigma;
  Eur_put.T = T;
  Eur_put.K = K;
  Eur_put.isCall = false;
  Eur_put.isAmerican = false;

  Option Am_put;
  Am_put.r = r;
  Am_put.q = q;
  Am_put.sigma = sigma;
  Am_put.T = T;
  Am_put.K = K;
  Am_put.isCall = false;
  Am_put.isAmerican = true;

  Option Eur_call;
  Eur_call.r = r;
  Eur_call.q = q;
  Eur_call.sigma = sigma;
  Eur_call.T = T;
  Eur_call.K = K;
  Eur_call.isCall = true;
  Eur_call.isAmerican = false;
```

```
  Option Am_call;
  Am_call.r = r;
  Am_call.q = q;
  Am_call.sigma = sigma;
  Am_call.T = T;
  Am_call.K = K;
  Am_call.isCall = true;
  Am_call.isAmerican = true;

  double FV_Am_put = 0;
  double FV_Eur_put = 0;
  double FV_Am_call = 0;
  double FV_Eur_call = 0;

  int n = 100;
  BinomialModel binom(n);

  double dS = 0.1;
  int imax = 2000;
  int i;
  for (i = 1; i <= imax; ++i) {
    S = i*dS;

    rc = binom.FairValue(n, &Am_put,   S, t0, FV_Am_put);
    rc = binom.FairValue(n, &Eur_put,  S, t0, FV_Eur_put);
    rc = binom.FairValue(n, &Am_call,  S, t0, FV_Am_call);
    rc = binom.FairValue(n, &Eur_call, S, t0, FV_Eur_call);

    ofs << std::setw(16) << S << "  ";
    ofs << std::setw(16) << FV_Am_put << "  ";
    ofs << std::setw(16) << FV_Eur_put << "  ";
    ofs << std::setw(16) << FV_Am_call << "  ";
    ofs << std::setw(16) << FV_Eur_call << "  ";
    ofs << std::endl;
  }
  return 0;
}
```