Queens College, CUNY, Department of Computer Science

**Computational Finance**

**CSCI 365 / 765**

**Spring 2018**

Instructor: Dr. Sateesh Mane



April 25, 2018

**due Friday April 27, 2018 at 11.59 pm**

# 9 Homework: Binomial model 2

## 9.1 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.

### 9.2 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 << std::setw(16) << S << "  ";
  outfile << std::setw(16) << FV_Am_put << "  ";
  outfile << std::setw(16) << FV_Eur_put << "  ";
  outfile << std::setw(16) << FV_Am_call << "  ";
  outfile << std::setw(16) << FV_Eur_call << "  ";
  outfile << std::endl;
}
```

### 9.3 Input data

- The inputs `S`, `K`, `r`, `q`, etc. are all input data, but they are not all of the same nature.
- We must distinguish between two different types of data.
- There is **indicative data** and `market data`.
- **Indicative data** are parameters which define the characteristics of a financial security (a derivative, in our case).
  1. In the case of an option, the indicative data consists of four parameters: (i) strike, (ii) expiration, (iii) call/put, (iv) American/European.
  2. If the values of any of the above parameters are changed, it becomes a different option.
  3. The values of indicative data do not change because of day to day trading.
  4. The values of indicative data can be stored in a database.
- **Market data** are parameters which change every day, or during the day.
  1. The stock price $S$ and the current time $t_0$ are examples of market data.
  2. The stock price changes all the time because of trading.
  3. The values of market data must be supplied as 'user inputs' to functions to calculate the fair value of an option or other derivative.
  4. If the stock prices changes, the *fair value* of an option will change. However, the strike, expiration, etc. of the option do not change.
  5. Hence the value of the stock price cannot be stored in a database.
  6. (Note that *historical stock prices* are stored in a database. But those numbers are only the values at the close of tradng every day. We shall not consider historical stock prices.

### 9.4 Database class

- We shall not use a real database in this course. That would be too complicated.
- Instead we shall make our own `Database` class and populate it ourselves.
- **Write the class below.**

```
class Database
{
public:
  Database() { r = 0; q = 0; }
  ~Database() {}

  // data
  double r;
  double q;
};
```

- Technically, interest rates change every day, and are really market data.
- However, it is too complicated to create a `YieldCurve` class and interpolate it, etc.
- We shall treat the values of the interest rate $r$ and dividebd yield $q$ are constants and store them in our own '`Database`' class.
- For simplicity, we make all the data members public.
- Otherwise we would have to write 'get' and 'set' class methods which just waste time and have nothing to do with finance.

### 9.5 Tree node class

- We have seen that in a binomial tree, we must store several data values at each mode.
- We require the stock price $S$ at the node, also the derivative value $V$, and the time $t$.
- In the simple binomial model, we stored the data in separate arrays.
- However, with object oriented programming, we can formulate a better design.
- We can collect all the variables at a node and store them in a class object.
- **Write the class below.**

```
class TreeNode
{
public:
  TreeNode() { S = 0; V = 0; t = 0; }
  ~TreeNode() {}

  // data
  double S;
  double V;
  double t;
};
```

- Technically, the time $t$ is the same for all nodes at a given timestep in the binomial tree, hence the above class design is slightly wasteful of memory.
- It is a weak point of this software design.

## 9.6 Derivative base class

- **Write the class below.**
- It is an abstract base class and contains virtual functions and a virtual destructor.
- The constructor is protected so that this class cannot be instantiated directly.
- We shall create derived classes for options, etc. which will inherit from this base class.
- The `Derivative` base class contains virtual functions which will be overridden by the derived classes, which we shall write later.
- All of the class methods are `const`. They all perform calculations (or tests), but they do not change any of the indicative data in the objects.
- Some functions perform validation checks, and the calculations may fail. Hence their function return type is `int` not `void`.
- **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(TreeNode & node)` is called in a loop by the binomial model. (We shall see this later.) This function checks if the value of $V$ in the node should be updated to the intrinsic value of the drivative.
- The `Derivative` base class contains a data member `double T` because all the derivaties we shall study all have an expiration time.
- Nevertheless, this is an inelegant feature of this software design.
- It is better to separate 'data' from 'calculation' and place them in different classes.

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

  virtual double TerminalPayoff(double S) const { return 0; }
  virtual int ValuationTests(TreeNode & node) const { return 0; }

  // data
  double T;

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

### 9.7 Option derived class

- We define a derived class `Option` to override the virtual functions.
- The `Option` class contains additional data items (indicative data).
  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) const;
  virtual int ValuationTests(TreeNode & node) const;

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

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

int Option::ValuationTests(TreeNode & node) const
{
  //  *** UPDATE THE VALUE OF V IN THE NODE TO THE INTRINSIC VALUE (IF NECESSARY) ***
}
```

## 9.8 Binomial model

### 9.8.1 Declaration of class

- The binomial model should also be made into a class `BinomialTree`.
- 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.
- The market data ($S$, $\sigma$, $t_0$) are supplied as inputs.
- The indicatives data is contained in the derivative objects (also some data from the database).
- The `Derivative` base class is an example of **polymorphism.**
  1. The binomial model operates exclusively with the `Derivative` base class.
  2. The binomial model does not know ***and should not know*** about any derived classes for options, etc.
  3. With this software design, we write only one binomial model class, and it can be used to perform calculations for multiple equity derivatives.
  4. **Note that both `p_derivative` and `p_db` are `const` pointers.**
  5. The binomial model performs calculations, but does not change internal data in the input objects.

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

  int FairValue(int n, const Derivative * p_derivative, const Database * p_db,
                double S, double sigma, double t0, double & FV);

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

  // data
  int n_tree;
  TreeNode **tree_nodes;
};
```

### 9.8.2 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. *****

```
BinomialTree::BinomialTree(int n)
{
  n_tree = 0;
  tree_nodes = 0;
  Allocate(n);
}

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

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

### 9.8.3 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 BinomialTree::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 ***

  //return 0 for success and 1 for fail
}
```

### 9.8.4 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 `FV=0.0`.
- **Validate the input data.** Return 1 (fail) if $n < 1$ or $S \leq 0$ or `p_derivative == NULL` or `p_db == NULL` or `p_derivative->T <= t0` or `sigma <= 0.0`.
- Calculate the parameters. Get the values of `r`, `q`, `T` and `sigma` from various sources.

```
double dt = (p_derivative->T-t0)/double(n);
double df = exp(-p_db->r*dt);
double growth = exp((p_db->r - p_db->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 `tree_nodes` with the appropriate stock prices and times. See `binomial_simple(...)`.

```
TreeNode * node_tmp = tree_nodes[0];
node_tmp[0].S = S;
node_tmp[0].t = t0;

for (i = 1; i <= n; ++i) {
  double t = t0 + i*dt;
  TreeNode * prev = tree_nodes[i-1];
  node_tmp = tree_nodes[i];

  // *** FILL IN THE REST OF THE CODE ***
}
```

- Set the value of $V$ in `tree_nodes` at step `i = n` with the terminal payoff.

```
i = n;
node_tmp = tree_nodes[i];
for (j = 0; j <= n; ++j) {
  node_tmp[j].V = p_derivative->TerminalPayoff(node_tmp[j].S);
}
```

- ***** 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.

### 9.8.5 Valuation of derivative Part 2

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

```
// valuation loop
for (i = n-1; i >= 0; --i) {
  node_tmp = tree_nodes[i];
  TreeNode * node_next = tree_nodes[i+1];
  for (j = 0; j <= i; ++j) {
    node_tmp[j].V = df*(p_prob*node_next[j+1].V + q_prob*node_next[j].V);

    p_derivative->ValuationTests(node_tmp[j]);
  }
}
```

- ***** 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 `FV` and exit.**

```
// derivative fair value
node_tmp = tree_nodes[0];
FV = node_tmp[0].V;

return 0;
```

- **We do not deallocate memory in this function.**
- **Deallocation is done by the destructor.**

### 9.8.6 Valuation of derivative Part 3

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

```
int BinomialTree::FairValue(int n, const Derivative * p_derivative, const Database * p_db,
                            double S, double sigma, double t0, double & FV)
{
  int rc = 0;

  FV = 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 and times in tree nodes
  ...

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

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

  // derivative fair value
  node_tmp = tree_nodes[0];
  FV = node_tmp[0].V;

  return 0;
}
```

### 9.9 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;

  std::ofstream ofs("output.txt");
  ofs.precision(6);

  double r = 0.05;
  double q = 0.02;
  double T = 1.0;
  double t0 = 0;

  Database db;
  db.r = r;
  db.q = q;

  double S = 100;
  double K = 100;
  double sigma = 0.5;

  Option Eur_put;
  Eur_put.K = K;
  Eur_put.T = T;
  Eur_put.isCall = false;
  Eur_put.isAmerican = false;

  Option Am_put;
  Am_put.K = K;
  Am_put.T = T;
  Am_put.isCall = false;
  Am_put.isAmerican = true;

  Option Eur_call;
  Eur_call.K = K;
  Eur_call.T = T;
  Eur_call.isCall = true;
  Eur_call.isAmerican = false;

  Option Am_call;
  Am_call.K = K;
  Am_call.T = T;
```

```
  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;
  BinomialTree binom(n);

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

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

    if (n > 100)
      n = 99;
    else
      n = 101;

    ofs << std::setw(6) << i << "  ";
    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;
  }
}
```