Queens College, CUNY, Department of Computer Science

Computational Finance

CSCI 365 / 765

Fall 2017

Instructor: Dr. Sateesh Mane

November 19, 2017

due Monday December 4, 2017 at 11.59 pm

9 Homework: Binomial model 3

9.1 Outline of work

- We shall extend the binomial model to calculate the **implied volatility.**
- This is analogous to the "`price_from_yield`" function for a bond.
- Given a market price (the "target" price) for a derivative, we find the value of the volatility such that the theoretical fair value equals the target price.
- The calculation requires an iterative numerical algorithm.
- We shall employ the bisection algorithm that was also used for `price_from_yield`.
- We shall make use of the C++ classes introduced in the previous homework assignment.
- We shall add an `ImpliedVolatility` function to the `BinomialModel` class.
- There are *no* changes required in the derivative classes.

9.2 Review of `BinomialModel`

- Our `BinomialModel` class now looks like this (see below).
- We wish to add an `ImpliedVolatility` function to it.

```
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 **value_nodes;
};
```

9.3 Weak point in software design

- We now encounter a weak point in the software design I have given you.
- We wish to iterate to calculate the value of the volatilty that will make the fair value of the derivative match the target price.
  1. However, the volatility is a data member "`sigma`" of the `Derivative` class.
  2. *If we change the value of the volatility in an iterative loop, it will mess up the value of* `sigma` *in the* `Derivative` *class object, which is bad.*
- Hence we must save the original value of `sigma`. We must not lose it.
- The correct way to solve the problem is to design a better software architecture, but that is too complicated.
- Therefore my proposed solution is to write *two* new class methods.
  1. One function is a "wrapper" which saves the value of `sigma` and restores it at the end.
  2. The second function contains the real iteration loop.

9.4 New class functions

- Add two new functions to the `BinomialModel` class as shown below.
- One function is public and the other is private.
- **Write the function declarations below.**

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

  int FairValue(int n, Derivative * p_derivative, double S, double t0, double & V);
  int ImpliedVolatility(int n, Derivative * p_derivative, double S, double t0,
                        double target, double & implied_vol, int & num_iter);

private:
  // methods
  void Clear();
  int Allocate(int n);
  int ImpliedVolatilityPrivate(int n, Derivative * p_derivative, double S, double t0,
                               double target, double & implied_vol, int & num_iter);

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

9.5 Public function

- The code for the public function is shown below.
- **Write the public function given below.**

```
int BinomialModel::ImpliedVolatility(int n, Derivative * p_derivative, double S, double t0,
                                     double target, double & implied_vol, int & num_iter)
{
  int rc = 0;
  const double saved_vol = p_derivative->sigma;
  rc = ImpliedVolatilityPrivate(n, p_derivative, S, t0, target, implied_vol, num_iter);
  p_derivative->sigma = saved_vol;
  return rc;
}
```

9.6 Private function

9.6.1 Summary

- The code for the private function will be the subject of this homework assignment.
- The function signature is as follows.

```
int BinomialModel::ImpliedVolatilityPrivate(int n, Derivative * p_derivative,
                                            double S, double t0, double target,
                                            double & implied_vol, int & num_iter);
```

- Review the "`yield_from_price`" function from Homework 1a.
- Many of the same ideas will be employed below.
- The procedure is basically the same as in `yield_from_price`.

  1. As opposed to `yield_from_price`, let us internally set a tolerannce of 10^{-4}. We do this because implied volatility is a computationally expensive function and we do not want users to set unnecessatily small tolerances.

     ```
     const double tol = 1.0e-4;
     ```

  2. Perform some validation tests.
  3. Set a low value for the volatility `sigma_low`. Calculate the fair value `FV_low`.
     If `abs(FV_low − target) <= tol`, the answer is within the tolerance. Exit (success).
  4. Set a high value for the volatility `sigma_high`. Calculate the fair value `FV_high`.
     If `abs(FV_high − target) <= tol`, the answer is within the tolerance. Exit (success).
  5. Test if the target value lies between `FV_low` and `FV_high`. If not, then we have not bracketed a solution, hence we exit (fail).
  6. Run the main bisection loop `for (i = 0; i < max_iter; ++i)`.
  7. Set `sigma = 0.5*(sigma_low + sigma_high)` (and remember that "`sigma`" is really a data member of the `Derivative` class). Call `FairValue` and compute a value `FV`.
  8. Test if `abs(FV − target) <= tol`, if yes then the iteration has converged and we exit (success).
  9. Else check if `FV` and `FV_low` are both on the same side as `target`. If yes, then update `FV_low = FV`, else update `FV_high = FV`.
  10. Also check if `abs(sigma_high − sigma_low) <= tol`. If yes, then the iteration has converged and we exit (success).
  11. If the loop has not converged after `max_iter` steps, exit (fail).

9.6.2 Validation tests

- Because the volatility is a data member of the `Derivative` class, the code is slightly different (and clumsier) than in `yield_from_price`.
- First set the tolerance and the maximum number of iterations internally.

```
const double tol = 1.0e-4;
const int max_iter = 100;
```

- Initialize `implied_vol` and `num_iter` both to zero.

```
implied_vol = 0;
num_iter = 0;
```

- We can use a low volatility of 1% and a high volatility of 300%.

```
double sigma_low = 0.01;   // 1%
double sigma_high = 3.0;   // 300%
double FV_low = 0;
double FV_high = 0;
double FV = 0;
```

- The validation test for `sigma_low` looks like this.

```
p_derivative->sigma = sigma_low;
FairValue(n, p_derivative, S, t0, FV_low);
double diff_FV_low = FV_low - target;
if (fabs(diff_FV_low) <= tol) {
  implied_vol = p_derivative->sigma;
  return 0;
}
```

- **Write the corresponding test for `sigma_high`.**
- Now test to see if `target` lies between `FV_low` and `FV_high`. This will be the case if `diff_FV_low` and `diff_FV_high` have opposite signs, i.e. `diff_FV_low * diff_FV_high < 0`. Hence if `diff_FV_low * diff_FV_high > 0` then exit (fail).

```
if (diff_FV_low * diff_FV_high > 0) {
  implied_vol = 0;
  return 1; // fail
}
```

- Note that in `yield_from_price` I did not make use of the variables `diff_FV_low` and `diff_FV_high`. This was a mistake or weak point I made in `yield_from_price`. The code would have been simpler if we had computed and used `diff_FV_low` and `diff_FV_high`. We shall do so here.
- If we have come this far, it is time to begin the main iteration loop.

9.6.3 Main iteration loop

- Set up an iteration loop.

```
for (i = 0; i < max_iter; ++i) {
    ...
}
```

- In the loop, set the value of `p_derivative->sigma` and calculate the value of `FV` by calling `FairValue`.

```
p_derivative->sigma = 0.5*(sigma_low + sigma_high);
FairValue(n, p_derivative, S, t0, FV);
double diff_FV = FV - target;
```

- Test if `abs(diff_FV) <= tol`. If yes, then we have converged.
  Set `implied_vol = p_derivative->sigma` and `num_iter = i` and exit with a return value of 0 (success).
- Test if `(diff_FV_low * diff_FV) > 0`. If yes, then `FV_low` and `FV` are both on the same side of `target`. Update `sigma_low = p_derivative->sigma`.
- Else update `sigma_high = p_derivative->sigma`.
- Test if `abs(sigma_low - sigma_high) <= tol`. If yes, then we have converged.
  Set `implied_vol = p_derivative->sigma` and `num_iter = i` and exit with a return value of 0 (success).
- If we have come this far, continue with the iteration loop.
- If we exit the iteration loop without converging, do the following and exit (fail).

```
num_iter = max_iter;
implied_vol = 0;
return 1;
```

- If you have done your work correctly, you will observe a great similarity to `yield_from_price`.

9.7 Tests

- For a given volatility, an American option has a higher fair value than the corresponding European option.

  1. Hence if you call `ImpliedVolatility` with the same target price for an American and a European option, the implied volatility of the American option will be $\leq$ the implied volatility of the European option.
  2. ***Do you understand why?***

- The fair value of an option increases as the volatility increases. Hence if you increase the target price, the implied volatility should increase.

- Use put–call parity. Choose a volatility σ_0. Calculate the fair value of a European put option $p(\sigma_0)$. Set a target value target $= p(\sigma_0) + Se^{-q(T-t_0)} - Ke^{-r(T-t_0)}$. Calculate the implied volatility of a European call option with this target price. The implied volatility should be close to σ_0.

9.8 Weak points in the software design

- Recall the rational option pricing inequalities

$$0 \le c, C \le S\,, \tag{9.8.1a}$$

$$0 \le P \le K\,, \tag{9.8.1b}$$

$$0 \le p \le \mathrm{PV}(K)\,. \tag{9.8.1c}$$

- Recall also that the value of an American option must be $\ge$ intrinsic value.

$$C \ge \max(S - K, 0)\,, \tag{9.8.2a}$$

$$P \ge \max(K - S, 0)\,. \tag{9.8.2b}$$

- The current software design does not test for these inequalities.
- If the target is too high, or if the target is too low (below intrinsic value for American options), then we know immediately that the implied volatility calculation will not converge. Our current software design does not test for these inequalities.
- However, the above inequalities apply only to options. There are different inequalities for other derivatives.
- Hence we would have to write a new virtual function, say `RationalPricingTests(...)`, to test if the target price violated any rational pricing inequalities.
- In principle, we can do this.
- However, it is late in the semester, and you are busy preparing for finals for many other courses.
- Hence I shall not ask you to write virtual functions to test for invalid target prices.