Queens College, CUNY, Department of Computer Science
Computational Finance
CSCI 365 / 765
Spring 2018
Instructor: Dr. Sateesh Mane

April 17, 2018

due Friday May 4, 2018 at 11.59 pm

10 Homework: Binomial model 3

10.1 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 `BinomialTree` class.
- There are *no changes* required in the derivative classes.
- This is a demonstration of encapsulation: we have separated the functionality.
- We can extend the binomial model without having to modify the derivative class(es).

10.2 Review of `BinomialModel`

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

```
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;
};
```

10.3 New class function

- Add a new public method to the `BinomialTree` class as shown below.
- **Write the function declaration below.**

```
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);

  int ImpliedVolatility(int n, const Derivative * p_derivative, const Database * p_db,
                        double S, double t0, double target,
                        double & implied_vol, int & num_iter);

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

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

10.4 Implied volatility

10.4.1 Summary

- The function signature is as follows.

```
int BinomialTree::ImpliedVolatility(int n,
                                    const Derivative * p_derivative,
                                    const Database * p_db,
                                    double S, double t0, double target,
                                    double & implied_vol, int & num_iter);
```

- Review the calculation of the 'yield from price' for a bond.
- Many of the same ideas of data validaton and iteration will be employed below.
- The calculation procedure is basically the same the 'yield from bond price' calculation.
- As opposed to the bond calculation, we shall internally set some limits. We do this because implied volatility is a computationally expensive function and we do not want users to set unnecessarily small tolerances, etc.

```
const double tol = 1.0e-6;
const int max_iter = 100;
```

- Initialize `implied_vol = 0` and `num_iter = 0`.
- **Perform some validation tests.**
  1. Set a low value for the volatility `sigma_low = 0.01` (= 1%).
     (a) Call `FairValue` using the input `sigma_low` to calculate a fair value `FV_low`.
         `FairValue(n, p_derivative, p_db, S, sigma_low, t0, FV_low);`
     (b) Define `double diff_FV_low = FV_low - target;`
     (c) If `std::abs(diff_FV_low) <= tol`, the answer is within the tolerance.
     (d) Set `implied_vol = sigma_low` and 'return 0' and exit (success).
  2. Next set a high value for the volatility `sigma_high = 2.0` (= 200%).
     (a) Call `FairValue` using the input `sigma_high` to calculate a fair value `FV_high`.
     (b) Define `double diff_FV_high = FV_high - target;`
     (c) If `std::abs(diff_FV_high) <= tol`, the answer is within the tolerance.
     (d) Set `implied_vol = sigma_high` and 'return 0' and exit (success).
- Test if the target value lies between `FV_low` and `FV_high`.
  1. If `diff_FV_low * diff_FV_high > 0` then we have not bracketed a solution.
  2. Set `implied_vol = 0` and 'return 1' and exit (fail).

- If we have come this far, it is safe to run the main bisection loop.
- We can use `num_iter` itself as the loop counter.

```
for (num_iter = 1; num_iter < max_iter; ++num_iter)
```

- In the loop, set `double sigma = 0.5*(sigma_low + sigma_high)`.
  1. Call `FairValue` using `sigma` and compute a value `FV`.
  2. Define `double diff_FV = FV - target;`
  3. Test if `std::abs(diff_FV) <= tol`.
     (a) If yes, then the iteration has converged.
     (b) Set `implied_vol = sigma` and 'return 0' and exit (success).
  4. Else check if `FV` and `FV_low` are both on the same side as `target`.
     (a) Test if `diff_FV_low * diff_FV > 0`.
     (b) If yes, then set `sigma_low = sigma`.
     (c) If no, then set `sigma_high = sigma`.
  5. Next check if `std::abs(sigma_high − sigma_low) <= tol`.
     (a) If yes, then the iteration has converged.
     (b) Set `implied_vol = sigma` and 'return 0' and exit (success).
- If we exit the loop after `max_iter` steps, then the iteration has failed.
- Set `implied_vol = 0` and `num_iter = max_iter` and 'return 1' and exit (fail).
- If you have done your work correctly, you will observe a great similarity to the '`yield from price`' calculation.

10.5 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 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.

10.6 Weak points in the software design

- Recall the rational option pricing inequalities

$$0 \le c, C \le S\,, \tag{10.6.1a}$$
$$0 \le P \le K\,, \tag{10.6.1b}$$
$$0 \le p \le \mathrm{PV}(K)\,. \tag{10.6.1c}$$

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

$$C \ge \max(S - K, 0)\,, \tag{10.6.2a}$$
$$P \ge \max(K - S, 0)\,. \tag{10.6.2b}$$

- 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, 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.