Delta Hedging

Delta Hedging

- Show how C++ can be used to test effectiveness of delta hedging
- Exercises give lots of examples of how to use object-oriented programming to enhance this example.

Overview

At time 0, a trader sells a European call option on the stock with strike K and maturity T to a customer at the Black–Scholes price. This means that in exchange for the price P, the trader is committed to paying the customer the amount

$$\max\{S_T - K, 0\}$$

at time T.

The trader's strategy is to delta hedge this liability. They delta hedge at N discrete time steps. So each time step has length $\delta t = \frac{T}{N}$.

Initial cashflows

We write b_i for the Trader's bank balance at each time point i. At time point 0 the trader puts

$$b_0 = P - \Delta_0 S_0 \tag{1}$$

into their risk-free account and invests the remainder of the principal, $\Delta_0 S_0$ in stock.

Cashflows at time i

- Accumulate interest.
- Rebalance portfolio. They wish to own a total of Δ_i stocks. They currently hold Δ_{i-1} units. They must buy the difference.

$$b_i = e^{r\delta t} b_{i-1} - (\Delta_i - \Delta_{i-1}) S_{i\delta t}. \tag{2}$$

Cashflows at maturity

- Accumulate interest.
- Sell stock that was used for hedging.
- Pay the customer if required.

$$b_N = e^{r\delta t} b_{N-1} + \Delta_{N-1} S_T - \max\{S - K, 0\}. \tag{3}$$

Member variables of Hedging Simulator

We write a class HedgingSimulator with these variables

```
private:
    /*  The option that has been written */
    std::shared_ptr<CallOption> toHedge;
    /*  The model used to simulate stock prices */
    std::shared_ptr<BlackScholesModel>
        simulationModel;
    /* The model used to compute prices and deltas */
    std::shared_ptr<BlackScholesModel> pricingModel;
    /*  The number of steps to use */
    int nSteps;
```

Comments

- Store data using `shared_ptr`. This is essential if we want to be able to store subclasses.
- Use `shared_ptr` as the default option to reference other classes.
- We have a pricing model and a simulation model so we can see what happens if they are different.
- We have getters and setters for these, and a default constructor.

runSimulations

The interesting method is runSimulations. Returns a vector of profits and losses.

```
    std::vector<double> runSimulations(
            int nSimulations ) const;
```

‘

Helper methods

```
    /*  Run a simulation and compute
        the profit and loss */
    double runSimulation() const;
    /*  How much should we charge the customer */
    double chooseCharge( double stockPrice ) const;
    /*  Hoe much stock should we hold */
    double selectStockQuantity(
        double date,
        double stockPrice ) const;
```

`runSimulation` does all the work. The other methods make the code easier to read.

Cashflows at time 0

```
double HedgingSimulator::runSimulation() const {
    double T = toHedge->getMaturity();
    double S0 = simulationModel->stockPrice;
    vector<double> pricePath =
        simulationModel->generatePricePath(T, nSteps);

    double dt = T / nSteps;
    double charge = chooseCharge(S0);
    double stockQuantity = selectStockQuantity(0, S0);
    double bankBalance = charge - stockQuantity*S0;
```

Cashflows at time i

```
    for (int i = 0; i< nSteps-1; i++) {
        double balanceWithInterest = bankBalance *
            exp(simulationModel->riskFreeRate*dt);
        double S = pricePath[i];
        double date = dt*(i + 1);
        double newStockQuantity =
            selectStockQuantity(date, S);
        double costs =
            (newStockQuantity - stockQuantity)*S;
        bankBalance = balanceWithInterest - costs;
        stockQuantity = newStockQuantity;
    }
```

Cashflows at maturity

```
    double balanceWithInterest = bankBalance *
        exp(simulationModel->riskFreeRate*dt);
    double S = pricePath[nSteps - 1];
    double stockValue = stockQuantity*S;
    double payout = toHedge->payoff(S);
    return balanceWithInterest + stockValue - payout;
}
```

Implementing selectStockQuantity

```
double HedgingSimulator::selectStockQuantity(
        double date,
        double stockPrice) const {
    // create a copy of the pricing model
    BlackScholesModel pm = *pricingModel;
    pm.stockPrice = stockPrice;
    pm.date = date;
    return toHedge->delta(pm);
}
```

Note that we are taking a copy of the pricing model. So, we change its stock price and date to reflect the simulation.

Implementing chooseCharge

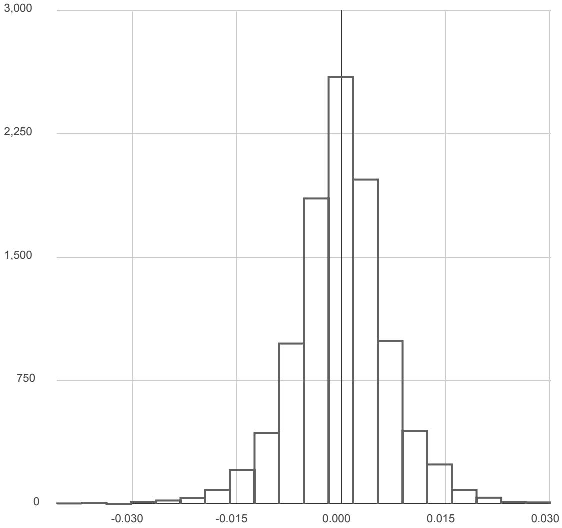
```
double HedgingSimulator::chooseCharge(
                        double stockPrice) const {
    // create a copy of the pricing model
    BlackScholesModel pm = *pricingModel;
    pm.stockPrice = stockPrice;
    return toHedge->price(pm);
}
```

Computing delta

```
double CallOption::delta(
    const BlackScholesModel& bsm) const {
    double S = bsm.stockPrice;
    double K = getStrike();
    double sigma = bsm.volatility;
    double r = bsm.riskFreeRate;
    double T = getMaturity() - bsm.date;

    double numerator = log(S / K) + (r + sigma*sigma*0.5)*T;
    double denominator = sigma * sqrt(T);
    double d1 = numerator / denominator;
    return normcdf(d1);
}
```

Results



Summary

- We have developed a C++ trading simulator to test the effectiveness of the delta hedging strategy. It backs up the Black–Scholes theory, but also shows that in discrete time it is not a risk-free strategy.
- The exercises show how object-orientated programming techniques can be used to make our trading simulator extremely versatile.