Option Pricing in OCaml: An introduction

This article serves as an introduction to financial option contracts and demonstrates different mathematical models one can use to determine their fair price. Snippets of the functional OCaml language are used to give the reader concrete demonstrations of how the math can be applied to a working algorithm.

Introduction to options:
Options are a type of financial contract that gives one party the option (but not the obligation) to buy/sell an asset at a specified price-point (called the "strike price") at an agreed upon time in the future. This future time is also known as the “maturity" of the option. "Call" options provide the buyer the right to buy an asset at the strike price, while "put" options give the option buyer the right to sell the asset to the other party. For this right, the option buyer pays a premium. The methods by which a fair price can be determined are explored in the topic of the next few sections.

Options fall under a group of financial securities known as derivatives, because their value is directly related to an underlying asset such as a stock or commodity. The price of this asset at maturity, along with the contract’s strike price, ultimately determines the payoff of the option. As an example, consider a buyer, Bob, who purchases the right to buy 1 share of company FooBar exactly 1 year from now for $100 from his broker. If, in one year from now, the price of FooBar is $110, then Bob will opt to exercise the option and buy the stock for $10 cheaper than the market price, thus profiting $10. If instead, the stock price at maturity is $90, then Bob would choose not to exercise the option at all, and his gain in wealth would be $0, instead of $10. A function to determine the payoff of both call and put options can be written as follows:

```
let option_payoff spot strike opt_type =
  match opt_type with
    | Call ->
      max 0. (spot -. strike)
    | Put ->
      max 0. (strike -. spot)
```

An aside: Reasons for using options (optional section)
The two most common uses of options in the trading world are to hedge against risk and to increase leverage.

The Single Period Binomial Model:
The single period binomial model is used to represent the price of an asset S, over a single discrete time period. A single period means that there are only two moments in time: initial time, t and the final time, T. In this model the asset, S is a random variable with two possible final states (both of which are linearly related to its initial state). The stock's initial price at time t, is S(t), and the two possible prices at the final period, T are S(t) * u or S(t) * d, where u and d are constant factors. The factors u and d are based on the asset’s volatility, and are chosen so that they closely approximate the volatility in the continuous Black-Scholes model. OCaml code demonstrating their calculation using the stock’s variance, and the length of a single period, delta_t is as follows:

```
let calculate_factors variance delta_t =
  let u = exp (sqrt (variance *. delta_t)) in
  (u, 1. /. u)
```

The above code shows that u and d are inverses of each other—in other words, the resulting model of potential future profits is recombinant. This is a useful property, as it results in fewer possible states when the number of discrete time periods is increased from 1 to infinitely many.

No-arbitrage and complete markets

Before further discussion of the use of the current model can take place, it is imperative that two assumptions are discussed. The first is the rule of no arbitrage, which states that under perfect market conditions, arbitrage does not exist. This means there is no positive probability of making a profit without the risk of loss. A simple, modern example of arbitrage can be seen on different cryptocurrency exchanges. At the height of the bubble, coins were trading for $10,000 on one exchange, and trading for $9,950 on another exchange. A smart trader would have been able to buy bitcoin for a lower price on the one exchange, and then immediately sell it for a $50 profit on the other exchange. The assumption of no arbitrage leads to the second important concept of the pricing model: complete markets. A market is complete if any contingent claims (in this case, options) can be "replicated" with a portfolio. A portfolio that replicates an option is one that matches the payoff of the claim in all possible states. Since it is possible to replicate an option with a portfolio of primary assets (stocks and bonds), then the rule of no arbitrage makes clear that the cost of forming that portfolio is the cost, C of the option.

Proving option price via. no-arbitrage

A portfolio consisting of a long position on stock S (following the binomial model), and a short position on an option of the same stock, is considered risk-less when the wealth-gain is identical regardless of whether S increases to Su or decreases to Sd. The amount of shares required for this to occur can be calculated using a system of equations. Since this portfolio is risk-less and since the rules of no-arbitrage apply, the final value of the portfolio must be equal to the initial value of the portfolio, increased by the risk free rate. In other words, if the initial amount of money was put into a bank and allowed to grow at the risk-free rate, it would equal the value of the portfolio at maturity, because both the bank and the portfolio contain 0 risk and therefore should have the same cost. It can then be shown that the cost of an option at the initial time is equal to the expected value of the option payoff at the final time, discounted by the risk free rate.

$C = e^{-rT} * [p * g(S_u) + (1-p) * g(S_d)]$, where $p = \frac{e^{rT} - d}{u - d}$

Note:

The function g(x) is the option payoff function.

The code for discounting a price by the risk-free interest rate is:

```
let discount_price price rate delta_t =
  price *. (exp@@(-1. *. rate *. delta_t))
```

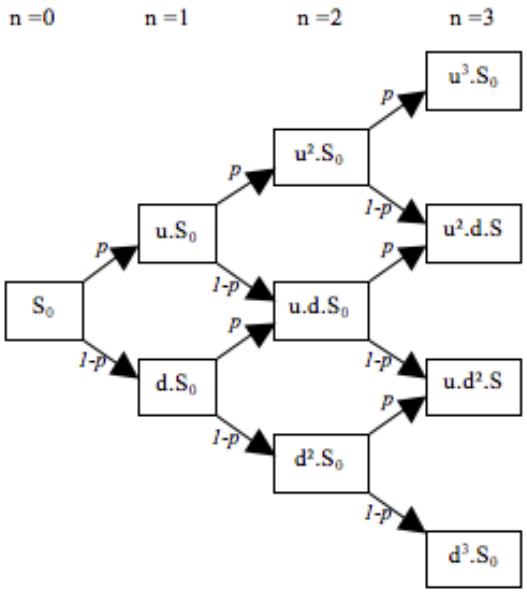
And the code for the calculating the probability of the stock going up and down is:

```
let martingale_probability rate ~up ~down ~delta_t =
  (exp (rate *. delta_t) -. down) /. (up -. down)
```

Revisiting the binomial model

In the above code the name of the function used to calculate the probabilities is "martingale_probability". A martingale is a process (sequence of random variables) in which the value at some time, t, equals the expectation of the value at time t+1. It is important to note that the calculated probabilities are not real-world probabilities, but rather risk-neutral probabilities. This assumption (that the option price does not take the risk profile of investors into account) allows the binomial model to use the probabilities calculated prior to price the option instead of the real-world probabilities.

The single period binomial model can be extended to a multi period model by replacing the random variable, S, with a random walk. This creates a tree-like structure that can be used to recursively calculate the price of an option.



$$p = \frac{e^{rt/n} - d}{u - d}$$

$$u = e^{\sigma\sqrt{t/n}}$$

$$d = e^{-\sigma\sqrt{t/n}}$$

HTTPS://EN.WIKIPEDIA.ORG/WIKI/BINOMIAL_OPTIONS_PRICING_MODEL#/MEDIA/FILE:ARBRE_BINOMIAL_OPTIONS_REELLES.PNG

This is achieved by starting with the future-most nodes and using the formula for discounted expectation of cost to "reverse engineer" the cost of the option in the prior state. Some OCaml code used to do this can be seen below:

```
let staggered_map lst ~f =
  let rec aux accum = function
    | [] | [_] -> List.rev accum
    | a::b::tl ->
      aux ((f a b)::accum) (b::tl)
  in
  aux [] lst
```

The staggered_map is a simple utility function that maps a binary function to each element in a list and the item in front of it. This results in a list that is smaller by one item (because the last element in the list has no corresponding item in front of it). In the code, each possible state for a given time period is represented in a list. The staggered_map function can then be used with a function calculating the discounted expectation of two different states in order to determine the cost of the option at the previous time.

```
let rec price_tree maturity_values ~prob_up ~rate ~delta_t =
  match maturity_values with
  | [] | [_] ->
    List.hd_exn maturity_values
  | maturity_values ->
    let level_prices = price_level maturity_values ~prob_up ~rate ~delta_t in
    price_tree level_prices ~prob_up ~rate ~delta_t
```