Math Notes

Version: 0.1.3

Holden J Bailey

2/3/2023

Introduction

I'm writing a web app that evaluates options strategies using statistical methods. I've settled on behavior that is defined by the flow chart in Figure 1.

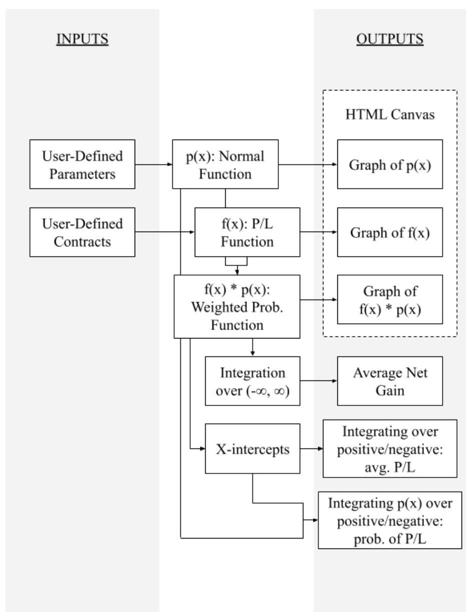

Figure 1: Program Flow

1 Profit-and-Loss Function

I've chosen to implement a graph for users to visualize the underlying security's price upon expiration vs. profit, probability, and profit-weighted probability using an HTML canvas. Since the canvas relies on a draw function, and because these calculations need to happen client-side, *in JavaScript*, I need a way to encapsulate multiple options contracts in a form that can be graphed as a function, multiplied, and the resulting product integrated cheaply and quickly.

This problem therefore reduces to the following:

> **Problem:** Describe a procedure for expressing the profit and loss of a set of options contracts upon expiration as a piece-wise function $f(x) = y$, where x is a price at expiration and y is the net profit or loss from all contracts expiring at that price, given the following assumptions:
>
> a. That the contracts all expire on the same day,
>
> b. That some contracts may share the same strike price,
>
> c. That there may be multiple of the same contract.

1.1 Representation in Memory

The domain of a piece-wise function is chopped up into mutually-exclusive subdomains that determine which expression defines $f(x)$ for a given x. The desired piece-wise function will therefore be represented in memory as a map of lower-bounds to the coefficients of the linear function that defines $f(x)$ above that bound and before the subsequent one (a C_1 and a C_0). Because the profit-loss function of an options contract (and by extension a sum of them) is continuous, it is unimportant whether a given x that occurs on one of the bounds evaluates to the expression before or after it. Finally, any piece-wise function of an expiration price is undefined below 0, making 0 the lowest bound.

```
expressions = {
    0: {"C_1": 0,   "C_0": 10},
    2: {"C_1": 1,   "C_0": 5},
    5: {"C_1": 0,   "C_0": 7},
    7: {"C_1": 5,   "C_0": 5},
    9: {"C_1": 0,   "C_0": 0},
   15: {"C_1": 0.6, "C_0": 13},
    ...
}
```

1.2 Access

The resulting object will have a function for evaluating the profit or loss at a price, x - something like "`ProfitLoss.f(price)`". Calling the function with a negative x will return an error. The function will take the sorted keys from the expression map and iterate over them until it encounters a key greater than x, at which point it will evaluate x using the prior key's expression.

1.3 Generation

At first blush, generation is expected to happen contract-by-contract as the user uses the interface to add, change, and delete contracts. Program behavior can therefore be segregated into one of the three aforementioned actions.

1.3.1 Initial State

As mentioned, all profit-loss functions will, by definition, be non-existent below 0. Holding no contracts is equivalent to non-participation, and therefore the net P-n-L will always be 0. The function should always initialize to a single-expression state: `{0: {"C_1": 0, "C_0": 0}}`.

1.3.2 Adding

The expression builder will supply a function for adding a contract to the expression. It will take parameters for whether the contract is bought or sold, whether it is a put or a call, the premium, strike, and count. The function will need to interpret the contract as a piece-wise function, then apply that function to the expression.

Profit-Loss is given by $f(x, S, P)$, where x is the price upon expiration, S is the strike price, and P is the contract's premium. Depending on the type of contract, the function is equal to the appropriate cell in the table below, plus or minus the premium, as appropriate.

<table>
<tr><th></th><th colspan="2">Call</th><th colspan="2">Put</th><th></th></tr>
<tr><td rowspan="2">Bought</td><td>0</td><td>if $x < S$</td><td>$S - x$</td><td>if $x < S$</td><td rowspan="2">$-P$</td></tr>
<tr><td>$x - S$</td><td>if $x > S$</td><td>0</td><td>if $x > S$</td></tr>
<tr><td rowspan="2">Sold</td><td>0</td><td>if $x < S$</td><td>$x - S$</td><td>if $x < S$</td><td rowspan="2">$+P$</td></tr>
<tr><td>$S - x$</td><td>if $x > S$</td><td>0</td><td>if $x > S$</td></tr>
</table>

Note that by definition (and intuitively), $f(x, S, P) = \pm P$ when $x = S$, with the premium being positive or negative - again - depending on whether the contract was bought or sold.

Note, too, that every contract's C_1 is either 1 or -1.

Knowing how to mathematically-represent contracts lets me approach the procedure for adding a contract:

1. Declare left and right coefficients (C_1 and C_0 for both sides)
2. Iterate over the map by sorted keys; for each key-expression pair:
   - If the key is less than x:
     - Add the left coefficients to the expression
   - If the key is equal to x:
     - Add the right coefficients to the current and all proceeding expressions

- If the key is greater than x:
  - Add a new key-expression pair using x and the preceding expression + the right coefficients
  - Add the right coefficients to the current and all proceeding expressions (*Will adding to the map mid-iteration mess with this procedure?? -Not if I use a copy of the keys*)