Foundations of Reinforcement Learning with Applications in Finance

Ashwin Rao, Tikhon Jelvis

1 Order-Book Trading Algorithms

In this chapter, we venture into the world of Algorithmic Trading and specifically, we cover a couple of problems involving a trading *Order Book* that can be cast as Markov Decision Processes, and hence tackled with Dynamic Programming or Reinforcement Learning. We start the chapter by covering the basics of how trade orders are submitted and executed on an *Order Book*, a structure that allows for efficient transactions between buyers and sellers of a financial asset. Without loss of generality, we refer to the financial asset being traded on the Order Book as a "stock" and the number of units of the asset as "shares." Next we will explain how a large trade can significantly shift the Order Book, a phenomenon known as *Price Impact*. Finally, we will cover the two algorithmic trading problems that can be cast as MDPs. The first problem is Optimal Execution of the sale of a large number of shares of a stock so as to yield the maximum utility of sales proceeds over a finite horizon. This involves breaking up the sale of the shares into appropriate pieces and selling those pieces at the right times so as to achieve the goal of maximizing the utility of sales proceeds. Hence, it is an MDP Control problem where the actions are the number of shares sold at each time step. The second problem is Optimal Market-Making, i.e., the optimal *bid*s (willingness to buy a certain number of shares at a certain price) and *ask*s (willingness to sell a certain number of shares at a certain price) to be submitted on the Order Book. Again, by optimal, we mean maximization of the utility of revenues generated by the market-maker over a finite-horizon (market-makers generate revenue through the spread, i.e. the gap between the bid and ask prices they offer). This is also an MDP Control problem where the actions are the bid and ask prices along with the bid and ask shares at each time step.

For a deeper study on the topics of Order Book, Price Impact, Order Execution, Market-Making (and related topics), we refer you to the comprehensive treatment in Olivier Gueant's book (Gueant 2016).

1.1 Basics of Order Book and Price Impact

Some of the financial literature refers to the Order Book as Limit Order Book (abbreviated as LOB) but we will stick with the lighter language - Order Book, abbreviated as OB. The Order Book is essentially a data structure that facilitates matching stock buyers with stock sellers (i.e., an electronic marketplace). Figure 1.1 depicts a simplified view of an order book. In this order book market, buyers and sellers express their intent to trade by submitting *bid*s (intent to buy) and *ask*s (intent to sell). These expressions of intent to buy or sell are known as Limit Orders (abbreviated as LO). The word "limit" in Limit Order refers to the fact that one is interested in buying only below a certain price level (and likewise, one is interested in selling only above a certain price level). Each LO is comprised of a price P and number of shares N. A bid, i.e., Buy LO (P, N) states willingness to buy N shares at a price less than or equal to P. Likewise, an ask, i.e., a Sell LO (P, N) states willingness to sell N shares at a price greater than or equal to P.

Note that multiple traders might submit LOs with the same price. The order book aggregates the number of shares at each unique price, and the OB data structure is typically

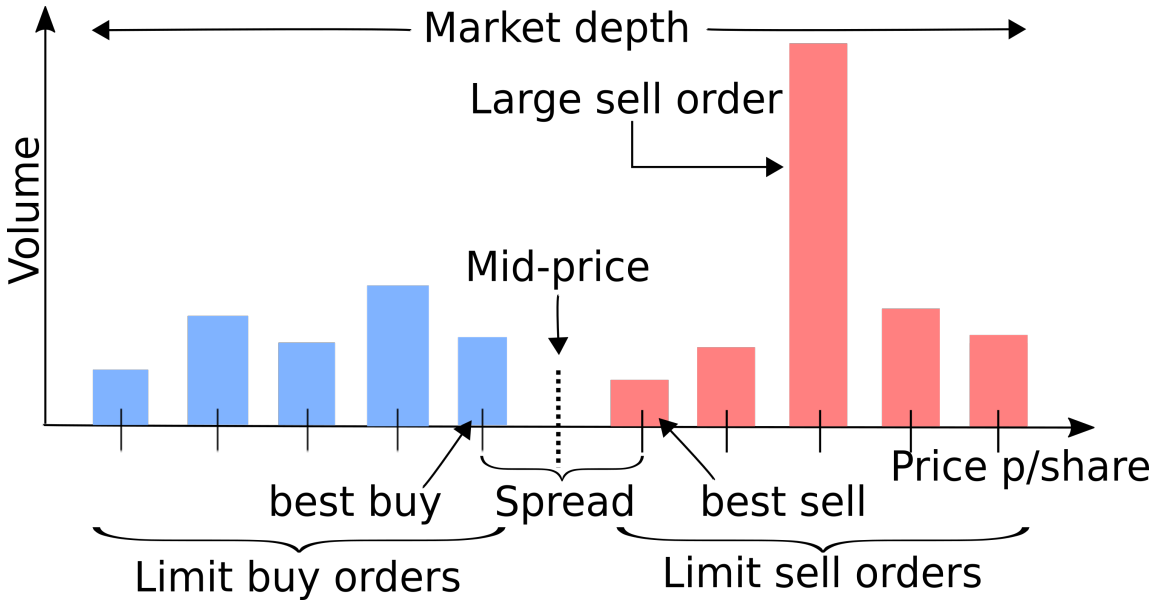

Figure 1.1: Trading Order Book (Image Credit: https://nms.kcl.ac.uk/rll/enrique-miranda/index.html)

presented for trading in the form of this aggregated view. Thus, the OB data structure can be represented as two sorted lists of (Price, Size) pairs:

$$\text{Buy LOs (Bids): } [(P_i^{(b)}, N_i^{(b)}) \mid 0 \leq i < m], P_i^{(b)} > P_j^{(b)} \text{ for } i < j$$

$$\text{Sell LOs (Asks): } [(P_i^{(a)}, N_i^{(a)}) \mid 0 \leq i < n], P_i^{(a)} < P_j^{(a)} \text{ for } i < j$$

Note that the Buy LOs are arranged in descending order and the Sell LOs are arranged in ascending order to signify the fact that the beginning of each list consists of the most important (best-price) LOs.

Now let's learn about some of the standard terminology:

- We refer to $P_0^{(b)}$ as *The Best Bid Price* (lightened to *Best Bid*) to signify that it is the highest offer to buy and hence, the *best* price for a seller to transact with.
- Likewise, we refer to $P_0^{(a)}$ as *The Ask Price* (lightened to *Best Ask*) to signify that it is the lowest offer to sell and hence, the *best* price for a buyer to transact with.
- $\frac{P_0^{(a)}+P_0^{(b)}}{2}$ is refered to as the *The Mid Price* (lightened to *Mid*).
- $P_0^{(a)} - P_0^{(b)}$ is refered to as *The Best Bid-Ask Spread* (lightened to *Spread*).
- $P_{n-1}^{(a)} - P_{m-1}^{(b)}$ is refered to as *The Market Depth* (lightened to *Depth*).

Although an actual real-world trading order book has many other details, we believe this simplified coverage is adequate for the purposes of core understanding of order book trading and to navigate the problems of optimal order execution and optimal market-making. Apart from Limit Orders, traders can express their interest to buy/sell with another type of order - a *Market Order* (abbreviated as MO). A Market Order (MO) states one's intent to buy/sell N shares at the *best possible price*(s) available on the OB at the time of MO submission. So, an LO is keen on price and not so keen on time (willing to wait to get the price one wants) while an MO is keen on time (desire to trade right away) and not so keen on

price (will take whatever the best LO price is on the OB). So now let us understand the actual transactions that occur between LOs and MOs (buy and sell interactions, and how the OB changes as a result of these interactions). Firstly, we note that in normal trading activity, a newly submitted sell LO's price is typically above the price of the best buy LO on the OB. But if a new sell LO's price is less than or equal to the price of the best buy LO's price, we say that the *market has crossed* (to mean that the range of bid prices and the range of ask prices have intersected), which results in an immediate transaction that eats into the OB's Buy LOs.

Precisely, a new Sell LO (P, N) potentially transacts with (and hence, removes) the best Buy LOs on the OB.

$$\text{Removal: } [(P_i^{(b)}, \min(N_i^{(b)}, \max(0, N - \sum_{j=0}^{i-1} N_j^{(b)}))) \mid (i : P_i^{(b)} \geq P)] \tag{1.1}$$

After this removal, it potentially adds the following LO to the asks side of the OB:

$$(P, \max(0, N - \sum_{i:P_i^{(b)} \geq P} N_i^{(b)})) \tag{1.2}$$

Likewise, a new Buy MO (P, N) potentially transacts with (and hence, removes) the best Sell LOs on the OB

$$\text{Removal: } [(P_i^{(a)}, \min(N_i^{(a)}, \max(0, N - \sum_{j=0}^{i-1} N_j^{(a)}))) \mid (i : P_i^{(a)} \leq P)] \tag{1.3}$$

After this removal, it potentially adds the following to the bids side of the OB:

$$(P, \max(0, N - \sum_{i:P_i^{(a)} \leq P} N_i^{(a)})) \tag{1.4}$$

When a Market Order (MO) is submitted, things are simpler. A Sell Market Order of N shares will remove the best Buy LOs on the OB.

$$\text{Removal: } [(P_i^{(b)}, \min(N_i^{(b)}, \max(0, N - \sum_{j=0}^{i-1} N_j^{(b)}))) \mid 0 \leq i < m] \tag{1.5}$$

The sales proceeds for this MO is:

$$\sum_{i=0}^{m-1} P_i^{(b)} \cdot (\min(N_i^{(b)}, \max(0, N - \sum_{j=0}^{i-1} N_j^{(b)}))) \tag{1.6}$$

We note that if N is large, the sales proceeds for this MO can be significantly lower than the best possible sales proceeds ($= N \cdot P_0^{(b)}$), which happens only if $N \leq N_0^{(b)}$. Note also that if N is large, the new Best Bid Price (new value of $P_0^{(b)}$) can be significantly lower than the Best Bid Price before the MO was submitted (because the MO "eats into" a significant volume of Buy LOs on the OB). This "eating into" the Buy LOs on the OB and consequent lowering of the Best Bid Price (and hence, Mid Price) is known as *Price Impact* of an MO (more specifically, as the *Temporary Price Impact* of an MO). We use the word "temporary" because subsequent to this "eating into" the Buy LOs of the OB (and consequent, "hole," ie., large Bid-Ask Spread), market participants will submit "replenishment LOs" (both

Buy LOs and Sell LOs) on the OB. These replenishments LOs would typically mitigate the Bid-Ask Spread and the eventual settlement of the Best Bid/Best Ask/Mid Prices constitutes what we call *Permanent Price Impact* - which refers to the changes in OB Best Bid/Best Ask/Mid prices relative to the corresponding prices before submission of the MO.

Likewise, a Buy Market Order of N shares will remove the best Sell LOs on the OB

$$\text{Removal: } [(P_i^{(a)}, \min(N_i^{(a)}, \max(0, N - \sum_{j=0}^{i-1} N_j^{(a)}))) \mid 0 \leq i < n] \tag{1.7}$$

The purchase bill for this MO is:

$$\sum_{i=0}^{n-1} P_i^{(a)} \cdot (\min(N_i^{(a)}, \max(0, N - \sum_{j=0}^{i-1} N_j^{(a)}))) \tag{1.8}$$

If N is large, the purchase bill for this MO can be significantly higher than the best possible purchase bill ($= N \cdot P_0^{(a)}$), which happens only if $N \leq N_0^{(a)}$. All that we wrote above in terms of Temporary and Permanent Price Impact naturally apply in the opposite direction for a Buy MO.

We refer to all of the above-described OB movements, including both temporary and permanent Price Impacts broadly as *Order Book Dynamics*. There is considerable literature on modeling Order Book Dynamics and some of these models can get fairly complex in order to capture various real-world nuances. Much of this literature is beyond the scope of this book. In this chapter, we will cover a few simple models for how a sell MO will move the OB's *Best Bid Price* (rather than a model for how it will move the entire OB). The model for how a buy MO will move the OB's *Best Ask Price* is naturally identical.

Now let's write some code that models how LOs and MOs interact with the OB. We write a class `OrderBook` that represents the Buy and Sell Limit Orders on the Order Book, which are each represented as a sorted sequence of the type `DollarsAndShares`, which is a `dataclass` we created to represent any pair of a dollar amount (`dollar: float`) and number of shares (`shares: int`). Sometimes, we use `DollarsAndShares` to represent an LO (pair of price and shares) as in the case of the sorted lists of Buy and Sell LOs. At other times, we use `DollarsAndShares` to represent the pair of total dollars transacted and total shares transacted when an MO is executed on the OB. The `OrderBook` maintains a price-descending sequence of `PriceSizePairs` for Buy LOs (`descending_bids`) and a price-ascending sequence of `PriceSizePairs` for Sell LOs (`ascending_asks`). We write the basic methods to get the `OrderBook`'s highest bid price (method `bid_price`), lowest ask price (method `ask_price`), mid price (method `mid_price`), spread between the highest bid price and lowest ask price (method `bid_ask_spread`), and market depth (method `market_depth`).

```
@dataclass(frozen=True)
class DollarsAndShares:
    dollars: float
    shares: int

PriceSizePairs = Sequence[DollarsAndShares]

@dataclass(frozen=True)
class OrderBook:
    descending_bids: PriceSizePairs
    ascending_asks: PriceSizePairs

    def bid_price(self) -> float:
        return self.descending_bids[0].dollars
```

```
    def ask_price(self) -> float:
        return self.ascending_asks[0].dollars

    def mid_price(self) -> float:
        return (self.bid_price() + self.ask_price()) / 2

    def bid_ask_spread(self) -> float:
        return self.ask_price() - self.bid_price()

    def market_depth(self) -> float:
        return self.ascending_asks[-1].dollars - \
            self.descending_bids[-1].dollars
```

Next we want to write methods for LOs and MOs to interact with the `OrderBook`. Notice that each of Equation (1.1) (new Sell LO potentially removing some of the beginning of the Buy LOs on the OB), Equation (1.3) (new Buy LO potentially removing some of the beginning of the Sell LOs on the OB), Equation (1.5) (Sell MO removing some of the beginning of the Buy LOs on the OB) and Equation (1.7) (Buy MO removing some of the beginning of the Sell LOs on the OB) all perform a common core function - they "eat into" the most significant LOs (on the opposite side) on the OB. So we first write a `@staticmethod` `eat_book` for this common function.

`eat_book` takes as input a `ps_pairs: PriceSizePairs` (representing one side of the OB) and the number of `shares: int` to buy/sell. Notice `eat_book`'s return type: `Tuple[DollarsAndShares, PriceSizePairs]`. The returned `DollarsAndShares` represents the pair of dollars transacted and the number of shares transacted (with number of shares transacted being less than or equal to the input `shares`). The returned `PriceSizePairs` represents the remainder of `ps_pairs` after the transacted number of shares have eaten into the input `ps_pairs`. `eat_book` first deletes (i.e. "eats up") as much of the *beginning* of the `ps_pairs: PriceSizePairs` data structure as it can (basically matching the input number of shares with an appropriate number of shares at the beginning of the `ps_pairs: PriceSizePairs` input). Note that the returned `PriceSizePairs` is a separate data structure, ensuring the immutability of the input `ps_pairs: PriceSizePairs`.

```
    @staticmethod
    def eat_book(
        ps_pairs: PriceSizePairs,
        shares: int
    ) -> Tuple[DollarsAndShares, PriceSizePairs]:
        rem_shares: int = shares
        dollars: float = 0.
        for i, d_s in enumerate(ps_pairs):
            this_price: float = d_s.dollars
            this_shares: int = d_s.shares
            dollars += this_price * min(rem_shares, this_shares)
            if rem_shares < this_shares:
                return (
                    DollarsAndShares(dollars=dollars, shares=shares),
                    [DollarsAndShares(
                        dollars=this_price,
                        shares=this_shares - rem_shares
                    )] + list(ps_pairs[i+1:])
                )
            else:
                rem_shares -= this_shares

        return (
            DollarsAndShares(dollars=dollars, shares=shares - rem_shares),
            []
        )
```

Now we are ready to write the method `sell_limit_order` which takes Sell LO Price and

Sell LO shares as input. As you can see in the code below, first it potentially removes (if it "crosses") an appropriate number of shares on the Buy LOs side of the OB (using the `@staticmethod eat_book`), and then potentially adds an appropriate number of shares at the Sell LO Price on the Sell LOs side of the OB. `sell_limit_order` returns a pair of `DollarsAndShares` type and `OrderBook` type. The returned `DollarsAndShares` represents the pair of dollars transacted and the number of shares transacted with the Buy LOs side of the OB (with number of shares transacted being less than or equal to the input `shares`). The returned `OrderBook` represents the new OB after potentially eating into the Buy LOs side of the OB and then potentially adding some shares at the Sell LO Price on the Sell LOs side of the OB. Note that the returned `OrderBook` is a newly-created data structure, ensuring the immutability of `self`. We urge you to read the code below carefully as there are many subtle details that are handled in the code.

```
from dataclasses import replace
    def sell_limit_order(self, price: float, shares: int) -> \
            Tuple[DollarsAndShares, OrderBook]:
        index: Optional[int] = next((i for i, d_s
                                     in enumerate(self.descending_bids)
                                     if d_s.dollars < price), None)
        eligible_bids: PriceSizePairs = self.descending_bids \
            if index is None else self.descending_bids[:index]
        ineligible_bids: PriceSizePairs = [] if index is None else \
            self.descending_bids[index:]

        d_s, rem_bids = OrderBook.eat_book(eligible_bids, shares)
        new_bids: PriceSizePairs = list(rem_bids) + list(ineligible_bids)
        rem_shares: int = shares - d_s.shares

        if rem_shares > 0:
            new_asks: List[DollarsAndShares] = list(self.ascending_asks)
            index1: Optional[int] = next((i for i, d_s
                                          in enumerate(new_asks)
                                          if d_s.dollars >= price), None)
            if index1 is None:
                new_asks.append(DollarsAndShares(
                    dollars=price,
                    shares=rem_shares
                ))
            elif new_asks[index1].dollars != price:
                new_asks.insert(index1, DollarsAndShares(
                    dollars=price,
                    shares=rem_shares
                ))
            else:
                new_asks[index1] = DollarsAndShares(
                    dollars=price,
                    shares=new_asks[index1].shares + rem_shares
                )
            return d_s, OrderBook(
                ascending_asks=new_asks,
                descending_bids=new_bids
            )
        else:
            return d_s, replace(
                self,
                descending_bids=new_bids
            )
```

Next, we write the easier method `sell_market_order` which takes as input the number of shares to be sold (as a market order). `sell_market_order` transacts with the appropriate number of shares on the Buy LOs side of the OB (removing those many shares from

the Buy LOs side). It returns a pair of `DollarsAndShares` type and `OrderBook` type. The returned `DollarsAndShares` represents the pair of dollars transacted and the number of shares transacted (with number of shares transacted being less than or equal to the input `shares`). The returned `OrderBook` represents the remainder of the OB after the transacted number of shares have eaten into the Buy LOs side of the OB. Note that the returned `OrderBook` is a newly-created data structure, ensuring the immutability of `self`.

```
    def sell_market_order(
        self,
        shares: int
    ) -> Tuple[DollarsAndShares, OrderBook]:
        d_s, rem_bids = OrderBook.eat_book(
            self.descending_bids,
            shares
        )
        return (d_s, replace(self, descending_bids=rem_bids))
```

We won't list the methods `buy_limit_order` and `buy_market_order` here as they are completely analogous (you can find the entire code for `OrderBook` in the file rl/chapter9/order_book.py). Now let us test out this code by creating a sample `OrderBook` and submitting some LOs and MOs to interact with the `OrderBook`.

```
bids: PriceSizePairs = [DollarsAndShares(
    dollars=x,
    shares=poisson(100. - (100 - x) * 10)
) for x in range(100, 90, -1)]
asks: PriceSizePairs = [DollarsAndShares(
    dollars=x,
    shares=poisson(100. - (x - 105) * 10)
) for x in range(105, 115, 1)]
ob0: OrderBook = OrderBook(descending_bids=bids, ascending_asks=asks)
```

The above code creates an `OrderBook` in the price range [91, 114] with a bid-ask spread of 5. Figure 1.2 depicts this `OrderBook` visually.

Let's submit a Sell LO that says we'd like to sell 40 shares as long as the transacted price is greater than or equal to 107. Our Sell LO should simply get added to the Sell LOs side of the OB.

```
d_s1, ob1 = ob0.sell_limit_order(107, 40)
```

The new `OrderBook` `ob1` has 40 more shares at the price level of 107, as depicted in Figure 1.3.

Now let's submit a Sell MO that says we'd like to sell 120 shares at the "best price." Our Sell MO should transact with 120 shares at "best prices" of 100 and 99 as well (since the OB does not have enough Buy LO shares at the price of 100).

```
d_s2, ob2 = ob1.sell_market_order(120)
```

The new `OrderBook` `ob2` has 120 less shares on the Buy LOs side of the OB, as depicted in Figure 1.4.

Now let's submit a Buy LO that says we'd like to buy 80 shares as long as the transacted price is less than or equal to 100. Our Buy LO should get added to the Buy LOs side of the OB.

```
d_s3, ob3 = ob2.buy_limit_order(100, 80)
```

Figure 1.2: Starting Order Book

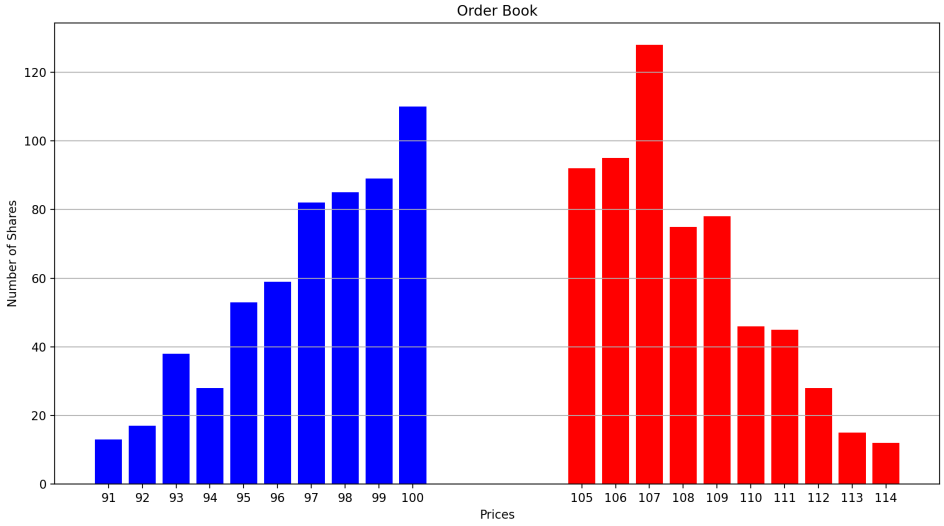

Figure 1.3: Order Book after Sell LO

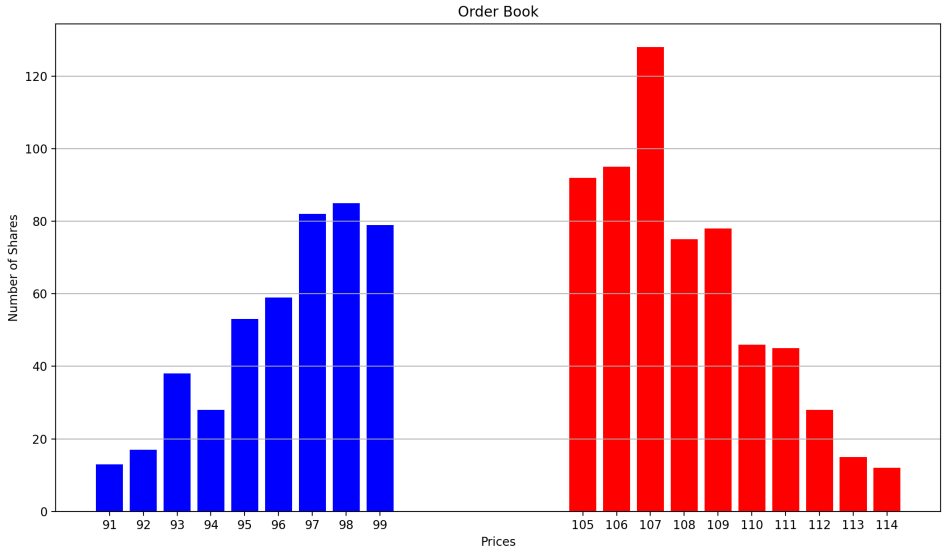

Figure 1.4: Order Book after Sell MO

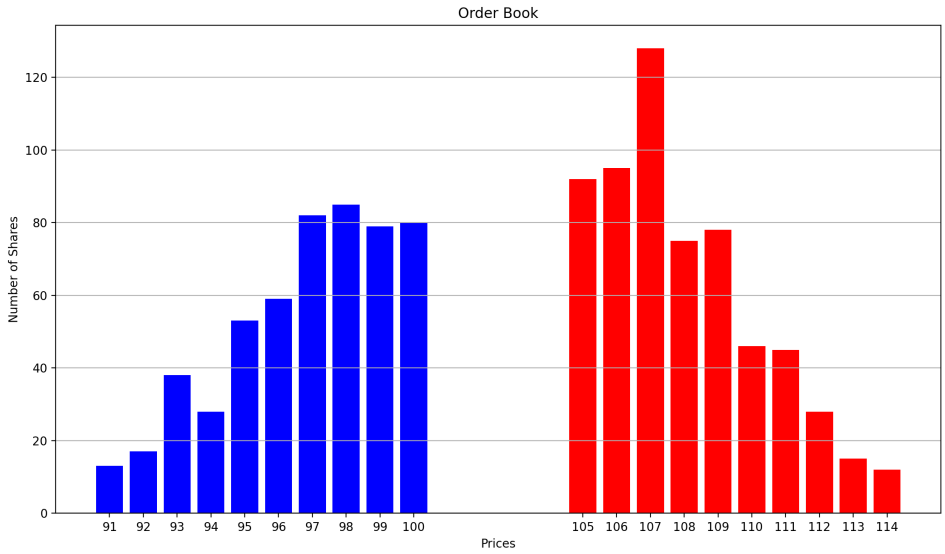

Figure 1.5: Order Book after Buy LO

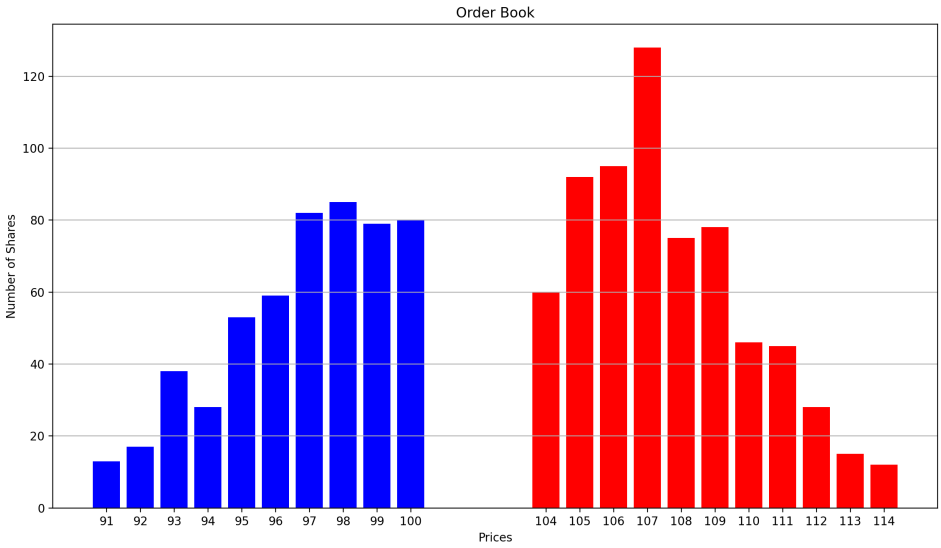

Figure 1.6: Order Book after 2nd Sell LO

The new `OrderBook` `ob3` has re-introduced a Buy LO at the price level of 100 (now with 80 shares), as depicted in Figure 1.5.

Now let's submit a Sell LO that says we'd like to sell 60 shares as long as the transacted price is greater than or equal to 104. Our Sell LO should get added to the Sell LOs side of the OB.

```
d_s4, ob4 = ob3.sell_limit_order(104, 60)
```

The new `OrderBook` `ob4` has introduced a Sell LO at a price of 104 with 60 shares, as depicted in Figure 1.6.

Now let's submit a Buy MO that says we'd like to buy 150 shares at the "best price." Our Buy MO should transact with 150 shares at "best prices" on the Sell LOs side of the OB.

```
d_s5, ob5 = ob4.buy_market_order(150)
```

The new `OrderBook` `ob5` has 150 less shares on the Sell LOs side of the OB, wiping out all the shares at the price level of 104 and almost wiping out all the shares at the price level of 105, as depicted in Figure 1.7.

This has served as a good test of our code (transactions working as we'd like) and we encourage you to write more code of this sort to interact with the `OrderBook`, and to produce graphs of evolution of the `OrderBook` as this will help develop stronger intuition and internalize the concepts we've learnt above. All of the above code is in the file rl/chapter9/order_book.py.

Now we are ready to get started with the problem of Optimal Execution of a large-sized Market Order.

1.2 Optimal Execution of a Market Order

Imagine the following problem: You are a trader in a stock and your boss has instructed that you exit from trading in this stock because this stock doesn't meet your company's

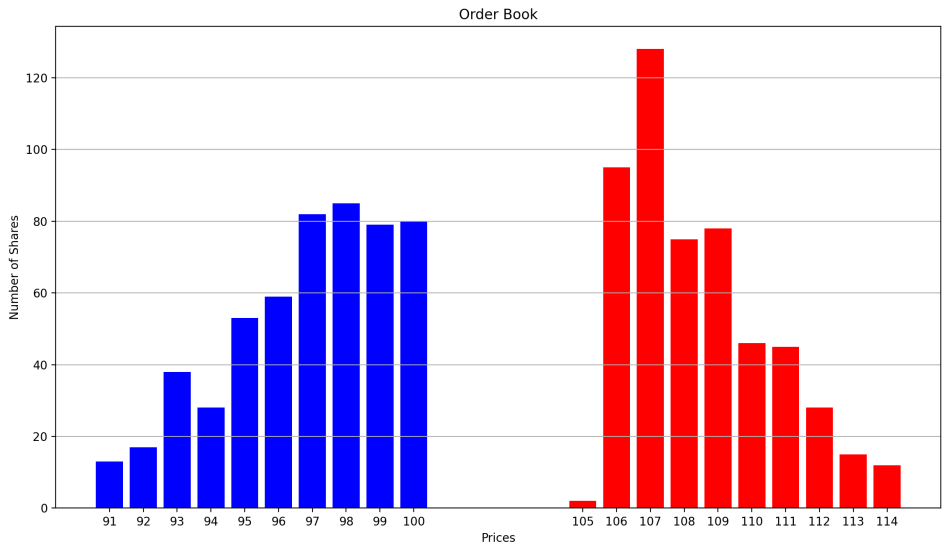

Figure 1.7: Order Book after Buy MO

new investment objectives. You have to sell all of the N shares you own in this stock in the next T hours, but you have been instructed to accomplish the sale by submitting only Market Orders (not allowed to submit any Limit Orders because of the uncertainty in the time of execution of the sale with a Limit Order). You can submit sell market orders (of any size) at the start of each hour - so you have T opportunities to submit market orders of any size. Your goal is to maximize the Expected Total Utility of sales proceeds for all N shares over the T hours. Your task is to break up N into T appropriate chunks to maximize the Expected Total Utility objective. If you attempt to sell the N shares too fast (i.e., too many in the first few hours), as we've learnt above, each (MO) sale will eat a lot into the Buy LOs on the OB (Temporary Price Impact) which would result in transacting at prices below the best price (Best Bid Price). Moreover, you risk moving the *Best Bid Price* on the OB significantly lower (Permanent Price Impact) that would affect the sales proceeds for the next few sales you'd make. On the other hand, if you sell the N shares too slow (i.e., too few in the first few hours), you might transact at good prices but then you risk running out of time, which means you will have to dump a lot of shares with time running out which in turn would mean transacting at prices below the best price. Moreover, selling too slow exposes you to more uncertainty in market price movements over a longer time period, and more uncertainty in sales proceeds means the Expected Utility objective gets hurt. Thus, the precise timing and sizes in the breakup of shares is vital. You will need to have an estimate of the Temporary and Permanent Price Impact of your Market Orders, which can help you identify the appropriate number of shares to sell at the start of each hour.

Unsurprisingly, we can model this problem as a Market Decision Process control problem where the actions at each time step (each hour, in this case) are the number of shares sold at the time step and the rewards are the Utility of sales proceeds at each time step. To keep things simple and intuitive, we shall model *Price Impact* of Market Orders in terms of their effect on the *Best Bid Price* (rather than in terms of their effect on the entire OB). In other words, we won't be modeling the entire OB Price Dynamics, just the Best Bid Price

Dynamics. We shall refer to the OB activity of an MO immediately "eating into the Buy LOs" (and hence, potentially transacting at prices lower than the best price) as the *Temporary* Price Impact. As mentioned earlier, this is followed by subsequent replenishment of both Buy and Sell LOs on the OB (stabilizing the OB) - we refer to any eventual (end of the hour) lowering of the Best Bid Price (relative to the Best Bid Price before the MO was submitted) as the *Permanent* Price Impact. Modeling the temporary and permanent Price Impacts separately helps us in deciding on the optimal actions (optimal shares to be sold at the start of each hour).

Now we develop some formalism to describe this problem precisely. As mentioned earlier, we make a number of simplifying assumptions in modeling the OB Dynamics for ease of articulation (without diluting the most important concepts). We index discrete time by $t = 0, 1, \ldots, T$. We denote P_t as the Best Bid Price on the OB at the start of time step t (for all $t = 0, 1, \ldots, T$) and N_t as the number of shares sold at time step t for all $t = 0, 1, \ldots, T-1$. We denote the number of shares remaining to be sold at the start of time step t as R_t for all $t = 0, 1, \ldots, T$. Therefore,

$$R_t = N - \sum_{i=0}^{t-1} N_i \text{ for all } t = 0, 1, \ldots, T$$

Note that:

$$R_0 = N$$

$$R_{t+1} = R_t - N_t \text{ for all } t = 0, 1, \ldots, T-1$$

Also note that we need to sell everything by time $t = T$ and so:

$$N_{T-1} = R_{T-1} \Rightarrow R_T = 0$$

The model of Best Bid Price Dynamics from one time step to the next is given by:

$$P_{t+1} = f_t(P_t, N_t, \epsilon_t) \text{ for all } t = 0, 1, \ldots, T-1$$

where f_t is an arbitrary function incorporating:

- The Permanent Price Impact of selling N_t shares.
- The Price-Impact-independent market-movement of the Best Bid Price from time t to time $t+1$.
- Noise ϵ_t, a source of randomness in Best Bid Price movements.

The sales proceeds from the sale at time step t, for all $t = 0, 1, \ldots, T-1$, is defined as:

$$N_t \cdot Q_t = N_t \cdot (P_t - g_t(P_t, N_t))$$

where g_t is a function modeling the Temporary Price Impact (i.e., the N_t MO "eating into" the Buy LOs on the OB). Q_t should be interpreted as the average Buy LO price transacted against by the N_t MO at time t.

Lastly, we denote the Utility (of Sales Proceeds) function as $U(\cdot)$.

As mentioned previously, solving for the optimal number of shares to be sold at each time step can be modeled as a discrete-time finite-horizon Markov Decision Process, which we describe below in terms of the order of MDP activity at each time step $t = 0, 1, \ldots, T-1$ (the MDP horizon is time T meaning all states at time T are terminal states). We follow the notational style of finite-horizon MDPs that should now be familiar from previous chapters.

Order of Events at time step t for all $t = 0, 1, \ldots, T-1$:

- Observe *State* $s_t := (P_t, R_t) \in \mathcal{S}_t$
- Perform *Action* $a_t := N_t \in \mathcal{A}_t$
- Receive *Reward* $r_{t+1} := U(N_t \cdot Q_t) = U(N_t \cdot (P_t - g_t(P_t, N_t)))$
- Experience Price Dynamics $P_{t+1} = f_t(P_t, N_t, \epsilon_t)$ and set $R_{t+1} = R_t - N_t$ so as to obtain the next state $s_{t+1} = (P_{t+1}, R_{t+1}) \in \mathcal{S}_{t+1}$.

Note that we have intentionally not specified if P_t, R_t, N_t are integers or real numbers, or if constrained to be non-negative etc. Those precise specifications will be customized to the nuances/constraints of the specific Optimal Order Execution problem we'd be solving. Be default, we shall assume that $P_t \in \mathbb{R}^+$ and $N_t, R_t \in \mathbb{Z}_{\geq 0}$ (as these represent realistic trading situations), although we do consider special cases later in the chapter where $P_t, R_t \in \mathbb{R}$ (unconstrained real numbers for analytical tractability).

The goal is to find the Optimal Policy $\pi^* = (\pi_0^*, \pi_1^*, \ldots, \pi_{T-1}^*)$ (defined as $\pi_t^*((P_t, R_t)) = N_t^*$ that maximizes:

$$\mathbb{E}[\sum_{t=0}^{T-1} \gamma^t \cdot U(N_t \cdot Q_t)]$$

where γ is the discount factor to account for the fact that future utility of sales proceeds can be modeled to be less valuable than today's.

Now let us write some code to solve this MDP. We write a class `OptimalOrderExecution` which models a fairly generic MDP for Optimal Order Execution as described above, and solves the Control problem with Approximate Value Iteration using the backward induction algorithm that we implemented in Chapter **??**. Let us start by taking a look at the attributes (inputs) in `OptimalOrderExecution`:

- `shares` refers to the total number of shares N to be sold over T time steps.
- `time_steps` refers to the number of time steps T.
- `avg_exec_price_diff` refers to the time-sequenced functions g_t that return the reduction in the average price obtained by the Market Order at time t due to eating into the Buy LOs. g_t takes as input the type `PriceAndShares` that represents a pair of `price: float` and `shares: int` (in this case, the `price` is P_t and the `shares` is the MO size N_t at time t). As explained earlier, the sales proceeds at time t is: $N_t \cdot (P_t - g_t(P_t, N_t))$.
- `price_dynamics` refers to the time-sequenced functions f_t that represent the price dynamics: $P_{t+1} \sim f_t(P_t, N_t)$. f_t outputs a probability distribution of prices for P_{t+1}.
- `utility_func` refers to the Utility of Sales Proceeds function, incorporating any risk-aversion.
- `discount_factor` refers to the discount factor γ.
- `func_approx` refers to the `ValueFunctionApprox` type to be used to approximate the Value Function for each time step (since we are doing backward induction).
- `initial_price_distribution` refers to the probability distribution of prices P_0 at time 0, which is used to generate the samples of states at each of the time steps (needed in the approximate backward induction algorithm).

```
from rl.approximate_dynamic_programming import ValueFunctionApprox

@dataclass(frozen=True)
class PriceAndShares:
    price: float
    shares: int

@dataclass(frozen=True)
class OptimalOrderExecution:
    shares: int
```

```
time_steps: int
avg_exec_price_diff: Sequence[Callable[[PriceAndShares], float]]
price_dynamics: Sequence[Callable[[PriceAndShares], Distribution[float]]]
utility_func: Callable[[float], float]
discount_factor: float
func_approx: ValueFunctionApprox[PriceAndShares]
initial_price_distribution: Distribution[float]
```

The two key things we need to perform the backward induction are:

- A method `get_mdp` that given a time step t, produces the `MarkovDecisionProcess` object representing the transitions from time t to time $t+1$. The class `OptimalExecutionMDP` within `get_mdp` implements the abstract methods `step` and `actions` of the abstract class `MarkovDecisionProcess`. The code should be fairly self-explanatory - just a couple of things to point out here. Firstly, the input `p_r: NonTerminal[PriceAndShares]` to the `step` method represents the state (P_t, R_t) at time t, and the variable `p_s: PriceAndShares` represents the pair of (P_t, N_t), which serves as input to `avg_exec_price_diff` and `price_dynamics` (attributes of `OptimalOrderExecution`). Secondly, note that the `actions` method returns an `Iterator` on a single `int` at time $t = T-1$ because of the constraint $N_{T-1} = R_{T-1}$.
- A method `get_states_distribution` that returns the probability distribution of states (P_t, R_t) at time t (of type `SampledDistribution[NonTerminal[PriceAndShares]]`). The code here is similar to the `get_states_distribiution` method of `AssetAllocDiscrete` in Chapter **??** (essentially, walking forward from time 0 to time t by sampling from the state-transition probability distribution and also sampling from uniform choices over all actions at each time step).

```
def get_mdp(self, t: int) -> MarkovDecisionProcess[PriceAndShares, int]:
    utility_f: Callable[[float], float] = self.utility_func
    price_diff: Sequence[Callable[[PriceAndShares], float]] = \
        self.avg_exec_price_diff
    dynamics: Sequence[Callable[[PriceAndShares], Distribution[float]]] = \
        self.price_dynamics
    steps: int = self.time_steps

    class OptimalExecutionMDP(MarkovDecisionProcess[PriceAndShares, int]):

        def step(
            self,
            p_r: NonTerminal[PriceAndShares],
            sell: int
        ) -> SampledDistribution[Tuple[State[PriceAndShares],
                                       float]]:

            def sr_sampler_func(
                p_r=p_r,
                sell=sell
            ) -> Tuple[State[PriceAndShares], float]:
                p_s: PriceAndShares = PriceAndShares(
                    price=p_r.state.price,
                    shares=sell
                )
                next_price: float = dynamics[t](p_s).sample()
                next_rem: int = p_r.state.shares - sell
                next_state: PriceAndShares = PriceAndShares(
                    price=next_price,
                    shares=next_rem
                )
                reward: float = utility_f(
                    sell * (p_r.state.price - price_diff[t](p_s))
```

```
                )
                return (NonTerminal(next_state), reward)

            return SampledDistribution(
                sampler=sr_sampler_func,
                expectation_samples=100
            )

        def actions(self, p_s: NonTerminal[PriceAndShares]) -> \
                Iterator[int]:
            if t == steps - 1:
                return iter([p_s.state.shares])
            else:
                return iter(range(p_s.state.shares + 1))

    return OptimalExecutionMDP()

def get_states_distribution(self, t: int) -> \
        SampledDistribution[NonTerminal[PriceAndShares]]:

    def states_sampler_func() -> NonTerminal[PriceAndShares]:
        price: float = self.initial_price_distribution.sample()
        rem: int = self.shares
        for i in range(t):
            sell: int = Choose(range(rem + 1)).sample()
            price = self.price_dynamics[i](PriceAndShares(
                price=price,
                shares=rem
            )).sample()
            rem -= sell
        return NonTerminal(PriceAndShares(
            price=price,
            shares=rem
        ))

    return SampledDistribution(states_sampler_func)
```

Finally, we produce the Optimal Value Function and Optimal Policy for each time step with the following method `backward_induction_vf_and_pi`:

```
from rl.approximate_dynamic_programming import back_opt_vf_and_policy

def backward_induction_vf_and_pi(
    self
) -> Iterator[Tuple[ValueFunctionApprox[PriceAndShares],
                    DeterministicPolicy[PriceAndShares, int]]]:

    mdp_f0_mu_triples: Sequence[Tuple[
        MarkovDecisionProcess[PriceAndShares, int],
        ValueFunctionApprox[PriceAndShares],
        SampledDistribution[NonTerminal[PriceAndShares]]
    ]] = [(
        self.get_mdp(i),
        self.func_approx,
        self.get_states_distribution(i)
    ) for i in range(self.time_steps)]

    num_state_samples: int = 10000
    error_tolerance: float = 1e-6

    return back_opt_vf_and_policy(
        mdp_f0_mu_triples=mdp_f0_mu_triples,
        gamma=self.discount_factor,
        num_state_samples=num_state_samples,
        error_tolerance=error_tolerance
    )
```

The above code is in the file rl/chapter9/optimal_order_execution.py. We encourage you to create a few different instances of `OptimalOrderExecution` by varying it's inputs (try

different temporary and permanent price impact functions, different utility functions, impose a few constraints etc.). Note that the above code has been written with an educational motivation rather than an efficient-computation motivation, so the convergence of the backward induction ADP algorithm is going to be slow. How do we know that the above code is correct? Well, we need to create a simple special case that yields a closed-form solution that we can compare the Optimal Value Function and Optimal Policy produced by `OptimalOrderExecution` against. This will be the subject of the following subsection.

1.2.1 Simple Linear Price Impact Model with no Risk-Aversion

Now we consider a special case of the above-described MDP - a simple linear Price Impact model with no risk-aversion. Furthermore, for analytical tractability, we assume N, N_t, P_t are all unconstrained continuous-valued (i.e., taking values $\in \mathbb{R}$).

We assume simple linear price dynamics as follows:

$$P_{t+1} = f_t(P_t, N_t, \epsilon) = P_t - \alpha \cdot N_t + \epsilon_t$$

where $\alpha \in \mathbb{R}$ and ϵ_t for all $t = 0, 1, \ldots, T-1$ are independent and identically distributed (i.i.d.) with $\mathbb{E}[\epsilon_t | N_t, P_t] = 0$. Therefore, the Permanent Price Impact (as an Expectation) is $\alpha \cdot N_t$.

As for the Temporary Price Impact, we know that g_t needs to be a non-decreasing function of N_t. We assume a simple linear form for g_t as follows:

$$g_t(P_t, N_t) = \beta \cdot N_t \text{ for all } t = 0, 1, \ldots, T-1$$

for some constant $\beta \in \mathbb{R}_{\geq 0}$. So, $Q_t = P_t - \beta N_t$. As mentioned above, we assume no risk-aversion, i.e., the Utility function $U(\cdot)$ is assumed to be the identity function. Also, we assume that the MDP discount factor $\gamma = 1$.

Note that all of these assumptions are far too simplistic and hence, an unrealistic model of the real-world, but starting with this simple model helps build good intuition and enables us to develop more realistic models by incrementally adding complexity/nuances from this simple base model.

As ever, in order to solve the Control problem, we define the Optimal Value Function and invoke the Bellman Optimality Equation. We shall use the standard notation for discrete-time finite-horizon MDPs that we are now very familiar with.

Denote the Value Function for policy π at time t (for all $t = 0, 1, \ldots T-1$) as:

$$V_t^\pi((P_t, R_t)) = \mathbb{E}_\pi[\sum_{i=t}^{T-1} N_i \cdot (P_i - \beta \cdot N_i) | (P_t, R_t)]$$

Denote the Optimal Value Function at time t (for all $t = 0, 1, \ldots, T-1$) as:

$$V_t^*((P_t, R_t)) = \max_\pi V_t^\pi((P_t, R_t))$$

The Optimal Value Function satisfies the finite-horizon Bellman Optimality Equation for all $t = 0, 1, \ldots, T-2$, as follows:

$$V_t^*((P_t, R_t)) = \max_{N_t} \{N_t \cdot (P_t - \beta \cdot N_t) + \mathbb{E}[V_{t+1}^*((P_{t+1}, R_{t+1}))]\}$$

and

$$V^*_{T-1}((P_{T-1}, R_{T-1})) = N_{T-1} \cdot (P_{T-1} - \beta \cdot N_{T-1}) = R_{T-1} \cdot (P_{T-1} - \beta \cdot R_{T-1})$$

From the above, we can infer:

$$V^*_{T-2}((P_{T-2}, R_{T-2})) = \max_{N_{T-2}} \{N_{T-2} \cdot (P_{T-2} - \beta \cdot N_{T-2}) + \mathbb{E}[R_{T-1} \cdot (P_{T-1} - \beta \cdot R_{T-1})]\}$$

$$= \max_{N_{T-2}} \{N_{T-2} \cdot (P_{T-2} - \beta \cdot N_{T-2}) + \mathbb{E}[(R_{T-2} - N_{T-2})(P_{T-1} - \beta \cdot (R_{T-2} - N_{T-2}))]\}$$

$$= \max_{N_{T-2}} \{N_{T-2} \cdot (P_{T-2} - \beta \cdot N_{T-2}) + (R_{T-2} - N_{T-2}) \cdot (P_{T-2} - \alpha \cdot N_{T-2} - \beta \cdot (R_{T-2} - N_{T-2}))\}$$

This simplifies to:

$$V^*_{T-2}((P_{T-2}, R_{T-2})) = \max_{N_{T-2}} \{R_{T-2} \cdot P_{T-2} - \beta \cdot R^2_{T-2} + (\alpha - 2\beta)(N^2_{T-2} - N_{T-2} \cdot R_{T-2})\} \quad (1.9)$$

For the case $\alpha \geq 2\beta$, noting that $N_{T-2} \leq R_{T-2}$, we have the trivial solution:

$$N^*_{T-2} = 0 \text{ or } N^*_{T-2} = R_{T-2}$$

Substituting either of these two values for N^*_{T-2} in the right-hand-side of Equation (1.9) gives:

$$V^*_{T-2}((P_{T-2}, R_{T-2})) = R_{T-2} \cdot (P_{T-2} - \beta \cdot R_{T-2})$$

Continuing backwards in time in this manner (for the case $\alpha \geq 2\beta$) gives:

$$N^*_t = 0 \text{ or } N^*_t = R_t \text{ for all } t = 0, 1, \ldots, T-1$$

$$V^*_t((P_t, R_t)) = R_t \cdot (P_t - \beta \cdot R_t) \text{ for all } t = 0, 1, \ldots, T-1$$

So the solution for the case $\alpha \geq 2\beta$ is to sell all N shares at any one of the time steps $t = 0, 1, \ldots, T-1$ (and none in the other time steps), and the Optimal Expected Total Sale Proceeds is $N \cdot (P_0 - \beta \cdot N)$

For the case $\alpha < 2\beta$, differentiating the term inside the max in Equation (1.9) with respect to N_{T-2}, and setting it to 0 gives:

$$(\alpha - 2\beta) \cdot (2N^*_{T-2} - R_{T-2}) = 0 \Rightarrow N^*_{T-2} = \frac{R_{T-2}}{2}$$

Substituting this solution for N^*_{T-2} in Equation (1.9) gives:

$$V^*_{T-2}((P_{T-2}, R_{T-2})) = R_{T-2} \cdot P_{T-2} - R^2_{T-2} \cdot (\frac{\alpha + 2\beta}{4})$$

Continuing backwards in time in this manner gives:

$$N^*_t = \frac{R_t}{T-t} \text{ for all } t = 0, 1, \ldots, T-1$$

$$V^*_t((P_t, R_t)) = R_t \cdot P_t - \frac{R^2_t}{2} \cdot (\frac{2\beta + \alpha \cdot (T-t-1)}{T-t}) \text{ for all } t = 0, 1, \ldots, T-1$$

Rolling forward in time, we see that $N_t^* = \frac{N}{T}$, i.e., splitting the N shares uniformly across the T time steps. Hence, the Optimal Policy is a constant deterministic function (i.e., independent of the *State*). Note that a uniform split makes intuitive sense because Price Impact and Market Movement are both linear and additive, and don't interact. This optimization is essentially equivalent to minimizing $\sum_{t=1}^{T} N_t^2$ with the constraint: $\sum_{t=1}^{T} N_t = N$. The Optimal Expected Total Sales Proceeds is equal to:

$$N \cdot P_0 - \frac{N^2}{2} \cdot (\alpha + \frac{2\beta - \alpha}{T})$$

Implementation Shortfall is the technical term used to refer to the reduction in Total Sales Proceeds relative to the maximum possible sales proceeds ($= N \cdot P_0$). So, in this simple linear model, the Implementation Shortfall from Price Impact is $\frac{N^2}{2} \cdot (\alpha + \frac{2\beta - \alpha}{T})$. Note that the Implementation Shortfall is non-zero even if one had infinite time available ($T \to \infty$) for the case of $\alpha > 0$. If Price Impact were purely temporary ($\alpha = 0$, i.e., Price fully snapped back), then the Implementation Shortfall is zero if one had infinite time available.

So now let's customize the class `OptimalOrderExecution` to this simple linear price impact model, and compare the Optimal Value Function and Optimal Policy produced by `OptimalOrderExecution` against the above-derived closed-form solutions. We write code below to create an instance of `OptimalOrderExecution` with time steps $T = 5$, total number of shares to be sold $N = 100$, linear temporary price impact with $\alpha = 0.03$, linear permanent price impact with $\beta = 0.03$, utility function as the identity function (no risk-aversion), and discount factor $\gamma = 1$. We set the standard deviation for the price dynamics probability distribution to 0 to speed up the calculation. Since we know the closed-form solution for the Optimal Value Function, we provide some assistance to `OptimalOrderExecution` by setting up a linear function approximation with two features: $P_t \cdot R_t$ and R_t^2. The task of `OptimalOrderExecution` is to infer the correct coefficients of these features for each time step. If the coefficients match that of the closed-form solution, it provides a great degree of confidence that our code is working correctly.

```
num_shares: int = 100
num_time_steps: int = 5
alpha: float = 0.03
beta: float = 0.05
init_price_mean: float = 100.0
init_price_stdev: float = 10.0

price_diff = [lambda p_s: beta * p_s.shares for _ in range(num_time_steps)]
dynamics = [lambda p_s: Gaussian(
    mu=p_s.price - alpha * p_s.shares,
    sigma=0.
) for _ in range(num_time_steps)]
ffs = [
    lambda p_s: p_s.state.price * p_s.state.shares,
    lambda p_s: float(p_s.state.shares * p_s.state.shares)
]
fa: FunctionApprox = LinearFunctionApprox.create(feature_functions=ffs)
init_price_distrib: Gaussian = Gaussian(
    mu=init_price_mean,
    sigma=init_price_stdev
```

```
)

ooe: OptimalOrderExecution = OptimalOrderExecution(
    shares=num_shares,
    time_steps=num_time_steps,
    avg_exec_price_diff=price_diff,
    price_dynamics=dynamics,
    utility_func=lambda x: x,
    discount_factor=1,
    func_approx=fa,
    initial_price_distribution=init_price_distrib
)
it_vf: Iterator[Tuple[ValueFunctionApprox[PriceAndShares],
                      DeterministicPolicy[PriceAndShares, int]]] = \
    ooe.backward_induction_vf_and_pi()
```

Next we evaluate this Optimal Value Function and Optimal Policy on a particular state for all time steps, and compare that against the closed-form solution. The state we use for evaluation is as follows:

```
state: PriceAndShares = PriceAndShares(
    price=init_price_mean,
    shares=num_shares
)
```

The code to evaluate the obtained Optimal Value Function and Optimal Policy on the above state is as follows:

```
for t, (vf, pol) in enumerate(it_vf):
    print(f"Time {t:d}")
    print()
    opt_sale: int = pol.action_for(state)
    val: float = vf(NonTerminal(state))
    print(f"Optimal Sales = {opt_sale:d}, Opt Val = {val:.3f}")
    print()
    print("Optimal Weights below:")
    print(vf.weights.weights)
    print()
```

With 100,000 state samples for each time step and only 10 state transition samples (since the standard deviation of ϵ is set to be very small), this prints the following:

```
Time 0

Optimal Sales = 20, Opt Val = 9779.976

Optimal Weights below:
[ 0.9999948  -0.02199718]

Time 1
```

```
Optimal Sales = 20, Opt Val = 9762.479

Optimal Weights below:
[ 0.9999935  -0.02374564]

Time 2

Optimal Sales = 20, Opt Val = 9733.324

Optimal Weights below:
[ 0.99999335 -0.02666098]

Time 3

Optimal Sales = 20, Opt Val = 9675.013

Optimal Weights below:
[ 0.99999316 -0.03249182]

Time 4

Optimal Sales = 20, Opt Val = 9500.000

Optimal Weights below:
[ 1.   -0.05]
```

Now let's compare these results against the closed-form solution.

```
for t in range(num_time_steps):
    print(f"Time {t:d}")
    print()
    left: int = num_time_steps - t
    opt_sale_anal: float = num_shares / num_time_steps
    wt1: float = 1
    wt2: float = -(2 * beta + alpha * (left - 1)) / (2 * left)
    val_anal: float = wt1 * state.price * state.shares + \
        wt2 * state.shares * state.shares

    print(f"Optimal Sales = {opt_sale_anal:.3f}, Opt Val = {val_anal:.3f}")
    print(f"Weight1 = {wt1:.3f}")
    print(f"Weight2 = {wt2:.3f}")
    print()
```

This prints the following:

```
Time 0

Optimal Sales = 20.000, Opt Val = 9780.000
Weight1 = 1.000
```

```
Weight2 = -0.022

Time 1

Optimal Sales = 20.000, Opt Val = 9762.500
Weight1 = 1.000
Weight2 = -0.024

Time 2

Optimal Sales = 20.000, Opt Val = 9733.333
Weight1 = 1.000
Weight2 = -0.027

Time 3

Optimal Sales = 20.000, Opt Val = 9675.000
Weight1 = 1.000
Weight2 = -0.033

Time 4

Optimal Sales = 20.000, Opt Val = 9500.000
Weight1 = 1.000
Weight2 = -0.050
```

We need to point out here that the general case of optimal order execution involving modeling of the entire Order Book's dynamics will have to deal with a large state space. This means the ADP algorithm will suffer from the curse of dimensionality, which means we will need to employ RL algorithms.

1.2.2 Paper by Bertsimas and Lo on Optimal Order Execution

A paper by Bertsimas and Lo on Optimal Order Execution (Bertsimas and Lo 1998) considered a special case of the simple Linear Impact model we sketched above. Specifically, they assumed no risk-aversion (Utility function is identity function) and assumed that the Permanent Price Impact parameter α is equal to the Temporary Price Impact Parameter β. In the same paper, Bertsimas and Lo then extended this Linear Impact Model to include dependence on a serially-correlated variable X_t as follows:

$$P_{t+1} = P_t - (\beta \cdot N_t + \theta \cdot X_t) + \epsilon_t$$

$$X_{t+1} = \rho \cdot X_t + \eta_t$$

$$Q_t = P_t - (\beta \cdot N_t + \theta \cdot X_t)$$

where ϵ_t and η_t are each independent and identically distributed random variables with mean zero for all $t = 0, 1, \ldots, T-1$, ϵ_t and η_t are also independent of each other for all $t = 0, 1, \ldots, T-1$. X_t can be thought of as a market factor affecting P_t linearly. Applying the finite-horizon Bellman Optimality Equation on the Optimal Value Function (and the

same backward-recursive approach as before) yields:

$$N_t^* = \frac{R_t}{T-t} + h(t, \beta, \theta, \rho) \cdot X_t$$

$$V_t^*((P_t, R_t, X_t)) = R_t \cdot P_t - (\text{quadratic in } (R_t, X_t) + \text{ constant})$$

Essentially, the serial-correlation predictability ($\rho \neq 0$) alters the uniform-split strategy.

In the same paper, Bertsimas and Lo presented a more realistic model called *Linear-Percentage Temporary* (abbreviated as LPT) Price Impact model, whose salient features include:

- Geometric random walk: consistent with real data, and avoids non-positive prices.
- Fractional Price Impact $\frac{g_t(P_t, N_t)}{P_t}$ doesn't depend on P_t (this is validated by real data).
- Purely Temporary Price Impact, i.e., the price P_t snaps back after the Temporary Price Impact (no Permanent effect of Market Orders on future prices).

The specific model is:

$$P_{t+1} = P_t \cdot e^{Z_t}$$

$$X_{t+1} = \rho \cdot X_t + \eta_t$$

$$Q_t = P_t \cdot (1 - \beta \cdot N_t - \theta \cdot X_t)$$

where Z_t are independent and identically distributed random variables with mean μ_Z and variance σ_Z^2, η_t are independent and identically distributed random variables with mean zero for all $t = 0, 1, \ldots, T-1$, Z_t and η_t are independent of each other for all $t = 0, 1, \ldots, T-1$. X_t can be thought of as a market factor affecting P_t multiplicatively. With the same derivation methodology as before, we get the solution:

$$N_t^* = c_t^{(1)} + c_t^{(2)} R_t + c_t^{(3)} X_t$$

$$V_t^*((P_t, R_t, X_t)) = e^{\mu_Z + \frac{\sigma_Z^2}{2}} \cdot P_t \cdot (c_t^{(4)} + c_t^{(5)} R_t + c_t^{(6)} X_t + c_t^{(7)} R_t^2 + c_t^{(8)} X_t^2 + c_t^{(9)} R_t X_t)$$

where $c_t^{(k)}, 1 \leq k \leq 9$, are constants (independent of P_t, R_t, X_t).

As an exercise, we recommend implementing the above (LPT) model by customizing `OptimalOrderExecution` to compare the obtained Optimal Value Function and Optimal Policy against the closed-form solution (you can find the exact expressions for the $c_t^{(k)}$ coefficients in the Bertsimas and Lo paper).

1.2.3 Incorporating Risk-Aversion and Real-World Considerations

Bertsimas and Lo ignored risk-aversion for the purpose of analytical tractability. Although there was value in obtaining closed-form solutions, ignoring risk-aversion makes their model unrealistic. We have discussed in detail in Chapter **??** about the fact that traders are wary of the risk of uncertain revenues and would be willing to trade away some expected revenues for lower variance of revenues. This calls for incorporating risk-aversion in the maximization objective. Almgren and Chriss wrote an important paper (Almgren and Chriss 2000) where they work in this Risk-Aversion framework. They consider our simple linear price impact model and incorporate risk-aversion by maximizing $E[Y] - \lambda \cdot Var[Y]$ where Y is the total (uncertain) sales proceeds $\sum_{t=0}^{T-1} N_t \cdot Q_t$ and λ controls the degree of

risk-aversion. The incorporation of risk-aversion affects the time-trajectory of N_t^*. Clearly, if $\lambda = 0$, we get the usual uniform-split strategy: $N_t^* = \frac{N}{T}$. The other extreme assumption is to minimize $Var[Y]$ which yields: $N_0^* = N$ (sell everything immediately because the only thing we want to avoid is uncertainty of sales proceeds). In their paper, Almgren and Chriss go on to derive the *Efficient Frontier* for this problem (analogous to the Efficient Frontier Portfolio Theory we outline in Appendix **??**). They also derive solutions for specific utility functions.

To model a real-world trading situation, the first step is to start with the MDP we described earlier with an appropriate model for the price dynamics $f_t(\cdot)$ and the temporary price impact $g_t(\cdot)$ (incorporating potential time-heterogeneity, non-linear price dynamics and non-linear impact). The `OptimalOrderExecution` class we wrote above allows us to incorporate all of the above. We can also model various real-world "frictions" such as discrete prices, discrete number of shares, constraints on prices and number of shares, as well as trading fees. To make the model truer to reality and more sophisticated, we can introduce various market factors in the *State* which would invariably lead to bloating of the State Space. We would also need to capture *Cross-Asset Market Impact*. As a further step, we could represent the entire Order Book (or a compact summary of the size/shape of the Order book) as part of the state, which leads to further bloating of the state space. All of this makes ADP infeasible and one would need to employ Reinforcement Learning algorithms. More importantly, we'd need to write a realistic Order Book Dynamics simulator capturing all of the above real-world considerations that an RL algorithm would learn from. There are a lot of practical and technical details involved in writing a real-world simulator and we won't be covering those details in this book. It suffices for here to say that the simulator would essentially be a sampling model that has learnt the Order Book Dynamics from market data (supervised learning of the Order Book Dynamics). Using such a simulator and with a deep learning-based function approximation of the Value Function, we can solve a practical Optimal Order Execution problem with Reinforcement Learning. We refer you to a couple of papers for further reading on this:

- Paper by Nevmyvaka, Feng, Kearns in 2006 (Nevmyvaka, Feng, and Kearns 2006)
- Paper by Vyetrenko and Xu in 2019 (Vyetrenko and Xu 2019)

Designing real-world simulators for Order Book Dynamics and using Reinforcement Learning for Optimal Order Execution is an exciting area for future research as well as engineering design. We hope this section has provided sufficient foundations for you to dig into this topic further.

1.3 Optimal Market-Making

Now we move on to the second problem of this chapter involving trading on an Order Book - the problem of Optimal Market-Making. A market-maker is a company/individual which/who regularly quotes bid and ask prices in a financial asset (which, without loss of generality, we will refer to as a "stock"). The market-maker typically holds some inventory in the stock, always looking to buy at one's quoted bid price and sell at one's quoted ask price, thus looking to make money off the spread between one's quoted ask price and one's quoted bid price. The business of a market-maker is similar to that of a car dealer who maintains an inventory of cars and who will offer purchase and sales prices, looking to make a profit off the price spread and ensuring that the inventory of cars doesn't get too big. In this section, we consider the business of a market-maker who quotes one's

bid prices by submitting Buy LOs on an OB and quotes one's ask prices by submitting Sell LOs on the OB. Market-makers are known as *liquidity providers* in the market because they make shares of the stock available for trading on the OB (both on the buy side and sell side). In general, anyone who submits LOs can be thought of as a market liquidity provider. Likewise, anyone who submits MOs can be thought of as a market *liquidity taker* (because an MO takes shares out of the volume that was made available for trading on the OB).

There is typically a fairly complex interplay between liquidity providers (including market-makers) and liquidity takers. Modeling OB dynamics is about modeling this complex interplay, predicting arrivals of MOs and LOs, in response to market events and in response to observed activity on the OB. In this section, we view the OB from the perspective of a single market-maker who aims to make money with Buy/Sell LOs of appropriate bid-ask spread and with appropriate volume of shares (specified in their submitted LOs). The market-maker is likely to be successful if she can do a good job of forecasting OB Dynamics and dynamically adjusting her Buy/Sell LOs on the OB. The goal of the market-maker is to maximize one's *Utility of Gains* at the end of a suitable horizon of time.

The core intuition in the decision of how to set the price and shares in the market-maker's Buy and Sell LOs is as follows: If the market-maker's bid-ask spread is too narrow, they will have more frequent transactions but smaller gains per transaction (more likelihood of their LOs being transacted against by an MO or an opposite-side LO). On the other hand, if the market-maker's bid-ask spread is too wide, they will have less frequent transactions but larger gains per transaction (less likelihood of their LOs being transacted against by an MO or an opposite-side LO). Also of great importance is the fact that a market-maker needs to carefully manage potentially large inventory buildup (either on the long side or the short side) so as to avoid scenarios of consequent unfavorable forced liquidation upon reaching the horizon time. Inventory buildup can occur if the market participants consistently transact against mostly one side of the market-maker's submitted LOs. With this high-level intuition, let us make these concepts of market-making precise. We start by developing some notation to help articulate the problem of Optimal Market-Making clearly. We will re-use some of the notation and terminology we had developed for the problem of Optimal Order Execution. As ever, for ease of exposition, we will simplify the setting for the Optimal Market-Making problem.

Assume there are a finite number of time steps indexed by $t = 0, 1, \ldots, T$. Assume the market-maker always shows a bid price and ask price (at each time t) along with the associated bid shares and ask shares on the OB. Also assume, for ease of exposition, that the market-maker can add or remove bid/ask shares from the OB *costlessly*. We use the following notation:

- Denote $W_t \in \mathbb{R}$ as the market-maker's trading account value at time t.
- Denote $I_t \in \mathbb{Z}$ as the market-maker's inventory of shares at time t (assume $I_0 = 0$). Note that the inventory can be positive or negative (negative means the market-maker is short a certain number of shares).
- Denote $S_t \in \mathbb{R}^+$ as the OB Mid Price at time t (assume a stochastic process for S_t).
- Denote $P_t^{(b)} \in \mathbb{R}^+$ as the market-maker's Bid Price at time t.
- Denote $N_t^{(b)} \in \mathbb{Z}^+$ as the market-maker's Bid Shares at time t.
- Denote $P_t^{(a)} \in \mathbb{R}^+$ as the market-maker's Ask Price at time t.
- Denote $N_t^{(a)} \in \mathbb{Z}^+$ as the market-maker's Ask Shares at time t.
- We refer to $\delta_t^{(b)} = S_t - P_t^{(b)}$ as the market-maker's Bid Spread (relative to OB Mid).

- We refer to $\delta_t^{(a)} = P_t^{(a)} - S_t$ as the market-maker's Ask Spread (relative to OB Mid).
- We refer to $\delta_t^{(b)} + \delta_t^{(a)} = P_t^{(a)} - P_t^{(b)}$ as the market-maker's Bid-Ask Spread.
- Random variable $X_t^{(b)} \in \mathbb{Z}_{\geq 0}$ refers to the total number of market-maker's Bid Shares that have been transacted against (by MOs or by Sell LOs) up to time t ($X_t^{(b)}$ is often refered to as the cumulative "hits" up to time t, as in "the market-maker's buy offer has been *hit*").
- Random variable $X_t^{(a)} \in \mathbb{Z}_{\geq 0}$ refers to the total number of market-maker's Ask Shares that have been transacted against (by MOs or by Buy LOs) up to time t ($X_t^{(a)}$ is often refered to as the cumulative "lifts" up to time t, as in "the market-maker's sell offer has been *lifted*").

With this notation in place, we can write the trading account balance equation for all $t = 0, 1, \ldots, T-1$ as follows:

$$W_{t+1} = W_t + P_t^{(a)} \cdot (X_{t+1}^{(a)} - X_t^{(a)}) - P_t^{(b)} \cdot (X_{t+1}^{(b)} - X_t^{(b)}) \tag{1.10}$$

Note that since the inventory I_0 at time 0 is equal to 0, the inventory I_t at time t is given by the equation:

$$I_t = X_t^{(b)} - X_t^{(a)}$$

The market-maker's goal is to maximize (for an appropriately shaped concave utility function $U(\cdot)$) the sum of the trading account value at time T and the value of the inventory of shares held at time T, i.e., we maximize:

$$\mathbb{E}[U(W_T + I_T \cdot S_T)]$$

As we alluded to earlier, this problem can be cast as a discrete-time finite-horizon Markov Decision Process (with discount factor $\gamma = 1$). Following the usual notation for discrete-time finite-horizon MDPs, the order of activity for the MDP at each time step $t = 0, 1, \ldots, T-1$ is as follows:

- Observe *State* $(S_t, W_t, I_t) \in \mathcal{S}_t$.
- Perform *Action* $(P_t^{(b)}, N_t^{(b)}, P_t^{(a)}, N_t^{(a)}) \in \mathcal{A}_t$.
- Random number of bid shares hit at time step t (this is equal to $X_{t+1}^{(b)} - X_t^{(b)}$).
- Random number of ask shares lifted at time step t (this is equal to $X_{t+1}^{(a)} - X_t^{(a)}$),
- Update of W_t to W_{t+1}.
- Update of I_t to I_{t+1}.
- Stochastic evolution of S_t to S_{t+1}.
- Receive *Reward* R_{t+1}, where

$$R_{t+1} := \begin{cases} 0 & \text{for } 1 \leq t+1 \leq T-1 \\ U(W_T + I_T \cdot S_T) & \text{for } t+1 = T \end{cases}$$

The goal is to find an *Optimal Policy* $\pi^* = (\pi_0^*, \pi_1^*, \ldots, \pi_{T-1}^*)$, where

$$\pi_t^*((S_t, W_t, I_t)) = (P_t^{(b)}, N_t^{(b)}, P_t^{(a)}, N_t^{(a)})$$

that maximizes:

$$\mathbb{E}[\sum_{t=1}^{T} R_t] = \mathbb{E}[R_T] = \mathbb{E}[U(W_T + I_T \cdot S_T)]$$

1.3.1 Avellaneda-Stoikov Continuous-Time Formulation

A landmark paper by Avellaneda and Stoikov (Avellaneda and Stoikov 2008) formulated this optimal market-making problem in it's continuous-time version. Their formulation is conducive to analytical tractability and they came up with a simple, clean and intuitive solution. In this subsection, we go over their formulation and in the next subsection, we show the derivation of their solution. We adapt our discrete-time notation above to their continuous-time setting.

$[(X_t^{(b)}|0 \leq t < T]$ and $[X_t^{(a)}|0 \leq t < T]$ are assumed to be continuous-time *Poisson processes* with the *hit rate per unit of time* and the *lift rate per unit of time* denoted as $\lambda_t^{(b)}$ and $\lambda_t^{(a)}$ respectively. Hence, we can write the following:

$$dX_t^{(b)} \sim Poisson(\lambda_t^{(b)} \cdot dt)$$
$$dX_t^{(a)} \sim Poisson(\lambda_t^{(a)} \cdot dt)$$
$$\lambda_t^{(b)} = f^{(b)}(\delta_t^{(b)})$$
$$\lambda_t^{(a)} = f^{(a)}(\delta_t^{(a)})$$

for decreasing functions $f^{(b)}(\cdot)$ and $f^{(a)}(\cdot)$.

$$dW_t = P_t^{(a)} \cdot dX_t^{(a)} - P_t^{(b)} \cdot dX_t^{(b)}$$
$$I_t = X_t^{(b)} - X_t^{(a)} \text{ (note: } I_0 = 0)$$

Since infinitesimal Poisson random variables $dX_t^{(b)}$ (shares hit in time interval from t to $t + dt$) and $dX_t^{(a)}$ (shares lifted in time interval from t to $t + dt$) are Bernoulli random variables (shares hit/lifted within time interval of duration dt will be 0 or 1), $N_t^{(b)}$ and $N_t^{(a)}$ (number of shares in the submitted LOs for the infinitesimal time interval from t to $t + dt$) can be assumed to be 1.

This simplifies the *Action* at time t to be just the pair:

$$(\delta_t^{(b)}, \delta_t^{(a)})$$

OB Mid Price Dynamics is assumed to be scaled brownian motion:

$$dS_t = \sigma \cdot dz_t$$

for some $\sigma \in \mathbb{R}^+$.

The Utility function is assumed to be: $U(x) = -e^{-\gamma x}$ where $\gamma > 0$ is the risk-aversion parameter (this Utility function is essentially the CARA Utility function devoid of associated constants).

1.3.2 Solving the Avellaneda-Stoikov Formulation

The following solution is as presented in the Avellaneda-Stoikov paper. We can express the Avellaneda-Stoikov continuous-time formulation as a Hamilton-Jacobi-Bellman (HJB) formulation (note: for reference, the general HJB formulation is covered in Appendix **??**).

We denote the Optimal Value function as $V^*(t, S_t, W_t, I_t)$. Note that unlike Section **??** in Chapter **??** where we denoted the Optimal Value Function as a time-indexed sequence

$V_t^*(\cdot)$, here we make t an explicit functional argument of V^* and each of S_t, W_t, I_t also as separate functional arguments of V^* (instead of the typical approach of making the state, as a tuple, a single functional argument). This is because in the continuous-time setting, we are interested in the time-differential of the Optimal Value Function and we also want to represent the dependency of the Optimal Value Function on each of S_t, W_t, I_t as explicit separate dependencies. Appendix **??** provides the derivation of the general HJB formulation (Equation (**??**) in Appendix **??**) - this general HJB Equation specializes here to the following:

$$\max_{\delta_t^{(b)}, \delta_t^{(a)}} \mathbb{E}[dV^*(t, S_t, W_t, I_t)] = 0 \text{ for } t < T$$

$$V^*(T, S_T, W_T, I_T) = -e^{-\gamma \cdot (W_T + I_T \cdot S_T)}$$

An infinitesimal change dV^* to $V^*(t, S_t, W_t, I_t)$ is comprised of 3 components:

- Due to pure movement in time (dependence of V^* on t).
- Due to randomness in OB Mid-Price (dependence of V^* on S_t).
- Due to randomness in hitting/lifting the market-maker's Bid/Ask (dependence of V^* on $\lambda_t^{(b)}$ and $\lambda_t^{(a)}$). Note that the probability of being hit in interval from t to $t + dt$ is $\lambda_t^{(b)} \cdot dt$ and probability of being lifted in interval from t to $t + dt$ is $\lambda_t^{(a)} \cdot dt$, upon which the trading account value W_t changes appropriately and the inventory I_t increments/decrements by 1.

With this, we can expand $dV^*(t, S_t, W_t, I_t)$ and rewrite HJB as:

$$\begin{aligned}
\max_{\delta_t^{(b)}, \delta_t^{(a)}} \{ & \frac{\partial V^*}{\partial t} \cdot dt + \mathbb{E}[\sigma \cdot \frac{\partial V^*}{\partial S_t} \cdot dz_t + \frac{\sigma^2}{2} \cdot \frac{\partial^2 V^*}{\partial S_t^2} \cdot (dz_t)^2] \\
& + \lambda_t^{(b)} \cdot dt \cdot V^*(t, S_t, W_t - S_t + \delta_t^{(b)}, I_t + 1) \\
& + \lambda_t^{(a)} \cdot dt \cdot V^*(t, S_t, W_t + S_t + \delta_t^{(a)}, I_t - 1) \\
& + (1 - \lambda_t^{(b)} \cdot dt - \lambda_t^{(a)} \cdot dt) \cdot V^*(t, S_t, W_t, I_t) \\
& - V^*(t, S_t, W_t, I_t)\} = 0
\end{aligned}$$

Next, we want to convert the HJB Equation to a Partial Differential Equation (PDE). We can simplify the above HJB equation with a few observations:

- $\mathbb{E}[dz_t] = 0$.
- $\mathbb{E}[(dz_t)^2] = dt$.
- Organize the terms involving $\lambda_t^{(b)}$ and $\lambda_t^{(a)}$ better with some algebra.
- Divide throughout by dt.

$$\begin{aligned}
\max_{\delta_t^{(b)}, \delta_t^{(a)}} \{ & \frac{\partial V^*}{\partial t} + \frac{\sigma^2}{2} \cdot \frac{\partial^2 V^*}{\partial S_t^2} \\
& + \lambda_t^{(b)} \cdot (V^*(t, S_t, W_t - S_t + \delta_t^{(b)}, I_t + 1) - V^*(t, S_t, W_t, I_t)) \\
& + \lambda_t^{(a)} \cdot (V^*(t, S_t, W_t + S_t + \delta_t^{(a)}, I_t - 1) - V^*(t, S_t, W_t, I_t))\} = 0
\end{aligned}$$

Next, note that $\lambda_t^{(b)} = f^{(b)}(\delta_t^{(b)})$ and $\lambda_t^{(a)} = f^{(a)}(\delta_t^{(a)})$, and apply the max only on the relevant terms:

$$\begin{aligned}
&\frac{\partial V^*}{\partial t} + \frac{\sigma^2}{2} \cdot \frac{\partial^2 V^*}{\partial S_t^2} \\
&+ \max_{\delta_t^{(b)}} \{ f^{(b)}(\delta_t^{(b)}) \cdot (V^*(t, S_t, W_t - S_t + \delta_t^{(b)}, I_t + 1) - V^*(t, S_t, W_t, I_t)) \} \\
&+ \max_{\delta_t^{(a)}} \{ f^{(a)}(\delta_t^{(a)}) \cdot (V^*(t, S_t, W_t + S_t + \delta_t^{(a)}, I_t - 1) - V^*(t, S_t, W_t, I_t)) \} = 0
\end{aligned}$$

This combines with the boundary condition:

$$V^*(T, S_T, W_T, I_T) = -e^{-\gamma \cdot (W_T + I_T \cdot S_T)}$$

Next, we make an *educated guess* for the functional form of $V^*(t, S_t, W_t, I_t)$:

$$V^*(t, S_t, W_t, I_t) = -e^{-\gamma \cdot (W_t + \theta(t, S_t, I_t))} \tag{1.11}$$

to reduce the problem to a Partial Differential Equation (PDE) in terms of $\theta(t, S_t, I_t)$. Substituting this guessed functional form into the above PDE for $V^*(t, S_t, W_t, I_t)$ gives:

$$\begin{aligned}
&\frac{\partial \theta}{\partial t} + \frac{\sigma^2}{2} \cdot (\frac{\partial^2 \theta}{\partial S_t^2} - \gamma \cdot (\frac{\partial \theta}{\partial S_t})^2) \\
&+ \max_{\delta_t^{(b)}} \{ \frac{f^{(b)}(\delta_t^{(b)})}{\gamma} \cdot (1 - e^{-\gamma \cdot (\delta_t^{(b)} - S_t + \theta(t, S_t, I_t + 1) - \theta(t, S_t, I_t))}) \} \\
&+ \max_{\delta_t^{(a)}} \{ \frac{f^{(a)}(\delta_t^{(a)})}{\gamma} \cdot (1 - e^{-\gamma \cdot (\delta_t^{(a)} + S_t + \theta(t, S_t, I_t - 1) - \theta(t, S_t, I_t))}) \} = 0
\end{aligned}$$

The boundary condition is:

$$\theta(T, S_T, I_T) = I_T \cdot S_T$$

It turns out that $\theta(t, S_t, I_t + 1) - \theta(t, S_t, I_t)$ and $\theta(t, S_t, I_t) - \theta(t, S_t, I_t - 1)$ are equal to financially meaningful quantities known as *Indifference Bid and Ask Prices*.

Indifference Bid Price $Q^{(b)}(t, S_t, I_t)$ is defined as follows:

$$V^*(t, S_t, W_t - Q^{(b)}(t, S_t, I_t), I_t + 1) = V^*(t, S_t, W_t, I_t) \tag{1.12}$$

$Q^{(b)}(t, S_t, I_t)$ is the price to buy a single share with a *guarantee of immediate purchase* that results in the Optimum Expected Utility staying unchanged.

Likewise, Indifference Ask Price $Q^{(a)}(t, S_t, I_t)$ is defined as follows:

$$V^*(t, S_t, W_t + Q^{(a)}(t, S_t, I_t), I_t - 1) = V^*(t, S_t, W_t, I_t) \tag{1.13}$$

$Q^{(a)}(t, S_t, I_t)$ is the price to sell a single share with a *guarantee of immediate sale* that results in the Optimum Expected Utility staying unchanged.

For convenience, we abbreviate $Q^{(b)}(t, S_t, I_t)$ as $Q_t^{(b)}$ and $Q^{(a)}(t, S_t, I_t)$ as $Q_t^{(a)}$. Next, we express $V^*(t, S_t, W_t - Q_t^{(b)}, I_t + 1) = V^*(t, S_t, W_t, I_t)$ in terms of θ:

$$-e^{-\gamma \cdot (W_t - Q_t^{(b)} + \theta(t, S_t, I_t + 1))} = -e^{-\gamma \cdot (W_t + \theta(t, S_t, I_t))}$$

$$\Rightarrow Q_t^{(b)} = \theta(t, S_t, I_t + 1) - \theta(t, S_t, I_t) \tag{1.14}$$

Likewise for $Q_t^{(a)}$, we get:

$$Q_t^{(a)} = \theta(t, S_t, I_t) - \theta(t, S_t, I_t - 1) \tag{1.15}$$

Using Equations (1.14) and (1.15), bring $Q_t^{(b)}$ and $Q_t^{(a)}$ in the PDE for θ:

$$\frac{\partial \theta}{\partial t} + \frac{\sigma^2}{2} \cdot (\frac{\partial^2 \theta}{\partial S_t^2} - \gamma \cdot (\frac{\partial \theta}{\partial S_t})^2) + \max_{\delta_t^{(b)}} g(\delta_t^{(b)}) + \max_{\delta_t^{(a)}} h(\delta_t^{(b)}) = 0$$

$$\text{where } g(\delta_t^{(b)}) = \frac{f^{(b)}(\delta_t^{(b)})}{\gamma} \cdot (1 - e^{-\gamma \cdot (\delta_t^{(b)} - S_t + Q_t^{(b)})})$$

$$\text{and } h(\delta_t^{(a)}) = \frac{f^{(a)}(\delta_t^{(a)})}{\gamma} \cdot (1 - e^{-\gamma \cdot (\delta_t^{(a)} + S_t - Q_t^{(a)})})$$

To maximize $g(\delta_t^{(b)})$, differentiate g with respect to $\delta_t^{(b)}$ and set to 0:

$$e^{-\gamma \cdot (\delta_t^{(b)^*} - S_t + Q_t^{(b)})} \cdot (\gamma \cdot f^{(b)}(\delta_t^{(b)^*}) - \frac{\partial f^{(b)}}{\partial \delta_t^{(b)}}(\delta_t^{(b)^*})) + \frac{\partial f^{(b)}}{\partial \delta_t^{(b)}}(\delta_t^{(b)^*}) = 0$$

$$\Rightarrow \delta_t^{(b)^*} = S_t - P_t^{(b)^*} = S_t - Q_t^{(b)} + \frac{1}{\gamma} \cdot \log(1 - \gamma \cdot \frac{f^{(b)}(\delta_t^{(b)^*})}{\frac{\partial f^{(b)}}{\partial \delta_t^{(b)}}(\delta_t^{(b)^*})}) \tag{1.16}$$

To maximize $h(\delta_t^{(a)})$, differentiate h with respect to $\delta_t^{(a)}$ and set to 0:

$$e^{-\gamma \cdot (\delta_t^{(a)^*} + S_t - Q_t^{(a)})} \cdot (\gamma \cdot f^{(a)}(\delta_t^{(a)^*}) - \frac{\partial f^{(a)}}{\partial \delta_t^{(a)}}(\delta_t^{(a)^*})) + \frac{\partial f^{(a)}}{\partial \delta_t^{(a)}}(\delta_t^{(a)^*}) = 0$$

$$\Rightarrow \delta_t^{(a)^*} = P_t^{(a)^*} - S_t = Q_t^{(a)} - S_t + \frac{1}{\gamma} \cdot \log(1 - \gamma \cdot \frac{f^{(a)}(\delta_t^{(a)^*})}{\frac{\partial f^{(a)}}{\partial \delta_t^{(a)}}(\delta_t^{(a)^*})}) \tag{1.17}$$

Equations (1.16) and (1.17) are implicit equations for $\delta_t^{(b)^*}$ and $\delta_t^{(a)^*}$ respectively.
Now let us write the PDE in terms of the Optimal Bid and Ask Spreads:

$$\begin{aligned}
&\frac{\partial \theta}{\partial t} + \frac{\sigma^2}{2} \cdot (\frac{\partial^2 \theta}{\partial S_t^2} - \gamma \cdot (\frac{\partial \theta}{\partial S_t})^2) \\
&+ \frac{f^{(b)}(\delta_t^{(b)^*})}{\gamma} \cdot (1 - e^{-\gamma \cdot (\delta_t^{(b)^*} - S_t + \theta(t, S_t, I_t + 1) - \theta(t, S_t, I_t))}) \\
&+ \frac{f^{(a)}(\delta_t^{(a)^*})}{\gamma} \cdot (1 - e^{-\gamma \cdot (\delta_t^{(a)^*} + S_t + \theta(t, S_t, I_t - 1) - \theta(t, S_t, I_t))}) = 0 \\
&\text{with boundary condition: } \theta(T, S_T, I_T) = I_T \cdot S_T
\end{aligned} \tag{1.18}$$

How do we go about solving this? Here are the steps:

- First we solve PDE (1.18) for θ in terms of $\delta_t^{(b)^*}$ and $\delta_t^{(a)^*}$. In general, this would be a numerical PDE solution.

- Using Equations (1.14) and (1.15), and using the above-obtained θ in terms of ${\delta_t^{(b)}}^*$ and ${\delta_t^{(a)}}^*$, we get $Q_t^{(b)}$ and $Q_t^{(a)}$ in terms of ${\delta_t^{(b)}}^*$ and ${\delta_t^{(a)}}^*$.
- Then we substitute the above-obtained $Q_t^{(b)}$ and $Q_t^{(a)}$ (in terms of ${\delta_t^{(b)}}^*$ and ${\delta_t^{(a)}}^*$) in Equations (1.16) and (1.17).
- Finally, we solve the implicit equations for ${\delta_t^{(b)}}^*$ and ${\delta_t^{(a)}}^*$ (in general, numerically).

This completes the (numerical) solution to the Avellaneda-Stoikov continuous-time formulation for the Optimal Market-Making problem. Having been through all the heavy equations above, let's now spend some time on building intuition.

Define the *Indifference Mid Price* $Q_t^{(m)} = \frac{Q_t^{(b)} + Q_t^{(a)}}{2}$. To develop intuition for Indifference Prices, consider a simple case where the market-maker doesn't supply any bids or asks after time t. This means the trading account value W_T at time T must be the same as the trading account value at time t and the inventory I_T at time T must be the same as the inventory I_t at time t. This implies:

$$V^*(t, S_t, W_t, I_t) = \mathbb{E}[-e^{-\gamma \cdot (W_t + I_t \cdot S_T)}]$$

The process $dS_t = \sigma \cdot dz_t$ implies that $S_T \sim \mathcal{N}(S_t, \sigma^2 \cdot (T - t))$, and hence:

$$V^*(t, S_t, W_t, I_t) = -e^{-\gamma \cdot (W_t + I_t \cdot S_t - \frac{\gamma \cdot I_t^2 \cdot \sigma^2 \cdot (T-t)}{2})}$$

Hence,

$$V^*(t, S_t, W_t - Q_t^{(b)}, I_t + 1) = -e^{-\gamma \cdot (W_t - Q_t^{(b)} + (I_t+1) \cdot S_t - \frac{\gamma \cdot (I_t+1)^2 \cdot \sigma^2 \cdot (T-t)}{2})}$$

But from Equation (1.12), we know that:

$$V^*(t, S_t, W_t, I_t) = V^*(t, S_t, W_t - Q_t^{(b)}, I_t + 1)$$

Therefore,

$$-e^{-\gamma \cdot (W_t + I_t \cdot S_t - \frac{\gamma \cdot I_t^2 \cdot \sigma^2 \cdot (T-t)}{2})} = -e^{-\gamma \cdot (W_t - Q_t^{(b)} + (I_t+1) \cdot S_t - \frac{\gamma \cdot (I_t+1)^2 \cdot \sigma^2 \cdot (T-t)}{2})}$$

This implies:

$$Q_t^{(b)} = S_t - (2I_t + 1) \cdot \frac{\gamma \cdot \sigma^2 \cdot (T - t)}{2}$$

Likewise, we can derive:

$$Q_t^{(a)} = S_t - (2I_t - 1) \cdot \frac{\gamma \cdot \sigma^2 \cdot (T - t)}{2}$$

The formulas for the Indifference Mid Price and the Indifference Bid-Ask Price Spread are as follows:

$$Q_t^{(m)} = S_t - I_t \cdot \gamma \cdot \sigma^2 \cdot (T - t)$$
$$Q_t^{(a)} - Q_t^{(b)} = \gamma \cdot \sigma^2 \cdot (T - t)$$

These results for the simple case of no-market-making-after-time-t serve as approximations for our problem of optimal market-making. Think of $Q_t^{(m)}$ as a *pseudo mid price* for

the market-maker, an adjustment to the OB mid price S_t that takes into account the magnitude and sign of I_t. If the market-maker is long inventory ($I_t > 0$), then $Q_t^{(m)} < S_t$, which makes intuitive sense since the market-maker is interested in reducing her risk of inventory buildup and so, would be be more inclined to sell than buy, leading her to show bid and ask prices whose average is lower than the OB mid price S_t. Likewise, if the market-maker is short inventory ($I_t < 0$), then $Q_t^{(m)} > S_t$ indicating inclination to buy rather than sell.

Armed with this intuition, we come back to optimal market-making, observing from Equations (1.16) and (1.17):

$$P_t^{(b)^*} < Q_t^{(b)} < Q_t^{(m)} < Q_t^{(a)} < P_t^{(a)^*}$$

Visualize this ascending sequence of prices $[P_t^{(b)^*}, Q_t^{(b)}, Q_t^{(m)}, Q_t^{(a)}, P_t^{(a)^*}]$ as jointly sliding up/down (relative to OB mid price S_t) as a function of the inventory I_t's magnitude and sign, and perceive $P_t^{(b)^*}, P_t^{(a)^*}$ in terms of their spreads to the *pseudo mid price* $Q_t^{(m)}$:

$$Q_t^{(b)} - P_t^{(m)^*} = \frac{Q_t^{(b)} + Q_t^{(a)}}{2} + \frac{1}{\gamma} \cdot \log\left(1 - \gamma \cdot \frac{f^{(b)}(\delta_t^{(b)^*})}{\frac{\partial f^{(b)}}{\partial \delta_t^{(b)}}(\delta_t^{(b)^*})}\right)$$

$$P_t^{(a)^*} - Q_t^{(m)} = \frac{Q_t^{(b)} + Q_t^{(a)}}{2} + \frac{1}{\gamma} \cdot \log\left(1 - \gamma \cdot \frac{f^{(a)}(\delta_t^{(a)^*})}{\frac{\partial f^{(a)}}{\partial \delta_t^{(a)}}(\delta_t^{(a)^*})}\right)$$

1.3.3 Analytical Approximation to the Solution to Avellaneda-Stoikov Formulation

The PDE (1.18) we derived above for θ and the associated implicit Equations (1.16) and (1.17) for $\delta_t^{(b)^*}, \delta_t^{(a)^*}$ are messy. So we make some assumptions, simplify, and derive analytical approximations(as presented in the Avellaneda-Stoikov paper). We start by assuming a fairly standard functional form for $f^{(b)}$ and $f^{(a)}$:

$$f^{(b)}(\delta) = f^{(a)}(\delta) = c \cdot e^{-k \cdot \delta}$$

This reduces Equations (1.16) and (1.17) to:

$$\delta_t^{(b)^*} = S_t - Q_t^{(b)} + \frac{1}{\gamma} \cdot \log\left(1 + \frac{\gamma}{k}\right) \tag{1.19}$$

$$\delta_t^{(a)^*} = Q_t^{(a)} - S_t + \frac{1}{\gamma} \cdot \log\left(1 + \frac{\gamma}{k}\right) \tag{1.20}$$

which means $P_t^{(b)^*}$ and $P_t^{(a)^*}$ are equidistant from $Q_t^{(m)}$. Substituting these simplified $\delta_t^{(b)^*}, \delta_t^{(a)^*}$ in Equation (1.18) reduces the PDE to:

$$\begin{aligned} &\frac{\partial \theta}{\partial t} + \frac{\sigma^2}{2} \cdot \left(\frac{\partial^2 \theta}{\partial S_t^2} - \gamma \cdot \left(\frac{\partial \theta}{\partial S_t}\right)^2\right) + \frac{c}{k+\gamma} \cdot \left(e^{-k \cdot \delta_t^{(b)^*}} + e^{-k \cdot \delta_t^{(a)^*}}\right) = 0 \\ &\text{with boundary condition } \theta(T, S_T, I_T) = I_T \cdot S_T \end{aligned} \tag{1.21}$$

Note that this PDE (1.21) involves $\delta_t^{(b)^*}$ and $\delta_t^{(a)^*}$. However, Equations (1.19), (1.20), (1.14) and (1.15) enable expressing $\delta_t^{(b)^*}$ and $\delta_t^{(a)^*}$ in terms of $\theta(t, S_t, I_t - 1), \theta(t, S_t, I_t)$ and

$\theta(t, S_t, I_t + 1)$. This gives us a PDE just in terms of θ. Solving that PDE for θ would give us not only $V^*(t, S_t, W_t, I_t)$ but also $\delta_t^{(b)^*}$ and $\delta_t^{(a)^*}$ (using Equations (1.19), (1.20), (1.14) and (1.15)). To solve the PDE, we need to make a couple of approximations.

First we make a linear approximation for $e^{-k\cdot\delta_t^{(b)^*}}$ and $e^{-k\cdot\delta_t^{(a)^*}}$ in PDE (1.21) as follows:

$$\frac{\partial\theta}{\partial t} + \frac{\sigma^2}{2}\cdot(\frac{\partial^2\theta}{\partial S_t^2} - \gamma\cdot(\frac{\partial\theta}{\partial S_t})^2) + \frac{c}{k+\gamma}\cdot(1 - k\cdot\delta_t^{(b)^*} + 1 - k\cdot\delta_t^{(a)^*}) = 0 \tag{1.22}$$

Combining the Equations (1.19), (1.20), (1.14) and (1.15) gives us:

$$\delta_t^{(b)^*} + \delta_t^{(a)^*} = \frac{2}{\gamma}\cdot\log(1+\frac{\gamma}{k}) + 2\theta(t, S_t, I_t) - \theta(t, S_t, I_t + 1) - \theta(t, S_t, I_t - 1)$$

With this expression for $\delta_t^{(b)^*} + \delta_t^{(a)^*}$, PDE (1.22) takes the form:

$$\begin{aligned}\frac{\partial\theta}{\partial t} + \frac{\sigma^2}{2}\cdot(\frac{\partial^2\theta}{\partial S_t^2} - \gamma\cdot(\frac{\partial\theta}{\partial S_t})^2) + \frac{c}{k+\gamma}\cdot(2 - \frac{2k}{\gamma}\cdot\log(1+\frac{\gamma}{k}) \\ - k\cdot(2\theta(t, S_t, I_t) - \theta(t, S_t, I_t + 1) - \theta(t, S_t, I_t - 1))) = 0\end{aligned} \tag{1.23}$$

To solve PDE (1.23), we consider the following asymptotic expansion of θ in I_t:

$$\theta(t, S_t, I_t) = \sum_{n=0}^{\infty}\frac{I_t^n}{n!}\cdot\theta^{(n)}(t, S_t)$$

So we need to determine the functions $\theta^{(n)}(t, S_t)$ for all $n = 0, 1, 2, \ldots$

For tractability, we approximate this expansion to the first 3 terms:

$$\theta(t, S_t, I_t) \approx \theta^{(0)}(t, S_t) + I_t\cdot\theta^{(1)}(t, S_t) + \frac{I_t^2}{2}\cdot\theta^{(2)}(t, S_t)$$

We note that the Optimal Value Function V^* can depend on S_t only through the current *Value of the Inventory* (i.e., through $I_t\cdot S_t$), i.e., it cannot depend on S_t in any other way. This means $V^*(t, S_t, W_t, 0) = -e^{-\gamma(W_t+\theta^{(0)}(t,S_t))}$ is independent of S_t. This means $\theta^{(0)}(t, S_t)$ is independent of S_t . So, we can write it as simply $\theta^{(0)}(t)$, meaning $\frac{\partial\theta^{(0)}}{\partial S_t}$ and $\frac{\partial^2\theta^{(0)}}{\partial S_t^2}$ are equal to 0. Therefore, we can write the approximate expansion for $\theta(t, S_t, I_t)$ as:

$$\theta(t, S_t, I_t) = \theta^{(0)}(t) + I_t\cdot\theta^{(1)}(t, S_t) + \frac{I_t^2}{2}\cdot\theta^{(2)}(t, S_t) \tag{1.24}$$

Substituting this approximation Equation (1.24) for $\theta(t, S_t, I_t)$ in PDE (1.23), we get:

$$\begin{aligned}&\frac{\partial\theta^{(0)}}{\partial t} + I_t\cdot\frac{\partial\theta^{(1)}}{\partial t} + \frac{I_t^2}{2}\cdot\frac{\partial\theta^{(2)}}{\partial t} + \frac{\sigma^2}{2}\cdot(I_t\cdot\frac{\partial^2\theta^{(1)}}{\partial S_t^2} + \frac{I_t^2}{2}\cdot\frac{\partial^2\theta^{(2)}}{\partial S_t^2}) \\ &- \frac{\gamma\sigma^2}{2}\cdot(I_t\cdot\frac{\partial\theta^{(1)}}{\partial S_t} + \frac{I_t^2}{2}\cdot\frac{\partial\theta^{(2)}}{\partial S_t})^2 + \frac{c}{k+\gamma}\cdot(2 - \frac{2k}{\gamma}\cdot\log(1+\frac{\gamma}{k}) + k\cdot\theta^{(2)}) = 0\end{aligned} \tag{1.25}$$

with boundary condition:

$$\theta^{(0)}(T) + I_T\cdot\theta^{(1)}(T, S_T) + \frac{I_T^2}{2}\cdot\theta^{(2)}(T, S_T) = I_T\cdot S_T$$

Now we separately collect terms involving specific powers of I_t, each yielding a separate PDE:

- Terms devoid of I_t (i.e., I_t^0)
- Terms involving I_t (i.e., I_t^1)
- Terms involving I_t^2

We start by collecting terms involving I_t:

$$\frac{\partial \theta^{(1)}}{\partial t} + \frac{\sigma^2}{2} \cdot \frac{\partial^2 \theta^{(1)}}{\partial S_t^2} = 0 \text{ with boundary condition } \theta^{(1)}(T, S_T) = S_T$$

The solution to this PDE is:

$$\theta^{(1)}(t, S_t) = S_t \tag{1.26}$$

Next, we collect terms involving I_t^2:

$$\frac{\partial \theta^{(2)}}{\partial t} + \frac{\sigma^2}{2} \cdot \frac{\partial^2 \theta^{(2)}}{\partial S_t^2} - \gamma \cdot \sigma^2 \cdot (\frac{\partial \theta^{(1)}}{\partial S_t})^2 = 0 \text{ with boundary condition } \theta^{(2)}(T, S_T) = 0$$

Noting that $\theta^{(1)}(t, S_t) = S_t$, we solve this PDE as:

$$\theta^{(2)}(t, S_t) = -\gamma \cdot \sigma^2 \cdot (T - t) \tag{1.27}$$

Finally, we collect the terms devoid of I_t

$$\frac{\partial \theta^{(0)}}{\partial t} + \frac{c}{k+\gamma} \cdot (2 - \frac{2k}{\gamma} \cdot \log(1 + \frac{\gamma}{k}) + k \cdot \theta^{(2)}) = 0 \text{ with boundary } \theta^{(0)}(T) = 0$$

Noting that $\theta^{(2)}(t, S_t) = -\gamma\sigma^2 \cdot (T - t)$, we solve as:

$$\theta^{(0)}(t) = \frac{c}{k+\gamma} \cdot ((2 - \frac{2k}{\gamma} \cdot \log(1 + \frac{\gamma}{k})) \cdot (T - t) - \frac{k\gamma\sigma^2}{2} \cdot (T - t)^2) \tag{1.28}$$

This completes the PDE solution for $\theta(t, S_t, I_t)$ and hence, for $V^*(t, S_t, W_t, I_t)$. Lastly, we derive formulas for $Q_t^{(b)}, Q_t^{(a)}, Q_t^{(m)}, \delta_t^{(b)*}, \delta_t^{(a)*}$.

Using Equations (1.14) and (1.15), we get:

$$Q_t^{(b)} = \theta^{(1)}(t, S_t) + (2I_t + 1) \cdot \theta^{(2)}(t, S_t) = S_t - (2I_t + 1) \cdot \frac{\gamma \cdot \sigma^2 \cdot (T - t)}{2} \tag{1.29}$$

$$Q_t^{(a)} = \theta^{(1)}(t, S_t) + (2I_t - 1) \cdot \theta^{(2)}(t, S_t) = S_t - (2I_t - 1) \cdot \frac{\gamma \cdot \sigma^2 \cdot (T - t)}{2} \tag{1.30}$$

Using equations (1.19) and (1.20), we get:

$$\delta_t^{(b)*} = \frac{(2I_t + 1) \cdot \gamma \cdot \sigma^2 \cdot (T - t)}{2} + \frac{1}{\gamma} \cdot \log(1 + \frac{\gamma}{k}) \tag{1.31}$$

$$\delta_t^{(a)*} = \frac{(1 - 2I_t) \cdot \gamma \cdot \sigma^2 \cdot (T - t)}{2} + \frac{1}{\gamma} \cdot \log(1 + \frac{\gamma}{k}) \tag{1.32}$$

$$\text{Optimal Bid-Ask Spread } \delta_t^{(b)^*} + \delta_t^{(a)^*} = \gamma \cdot \sigma^2 \cdot (T-t) + \frac{2}{\gamma} \cdot \log(1 + \frac{\gamma}{k}) \tag{1.33}$$

$$\text{Optimal Pseudo-Mid } Q_t^{(m)} = \frac{Q_t^{(b)} + Q_t^{(a)}}{2} = \frac{P_t^{(b)^*} + P_t^{(a)^*}}{2} = S_t - I_t \cdot \gamma \cdot \sigma^2 \cdot (T-t) \tag{1.34}$$

Now let's get back to developing intuition. Think of $Q_t^{(m)}$ as *inventory-risk-adjusted* mid-price (adjustment to S_t). If the market-maker is long inventory ($I_t > 0$), $Q_t^{(m)} < S_t$ indicating inclination to sell rather than buy, and if market-maker is short inventory, $Q_t^{(m)} > S_t$ indicating inclination to buy rather than sell. Think of the interval $[P_t^{(b)^*}, P_t^{(a)^*}]$ as being around the pseudo mid-price $Q_t^{(m)}$ (rather than thinking of it as being around the OB mid-price S_t). The interval $[P_t^{(b)^*}, P_t^{(a)^*}]$ moves up/down in tandem with $Q_t^{(m)}$ moving up/down (as a function of inventory I_t). Note from Equation (1.33) that the Optimal Bid-Ask Spread $P_t^{(a)^*} - P_t^{(b)^*}$ is independent of inventory I_t.

A useful view is:

$$P_t^{(b)^*} < Q_t^{(b)} < Q_t^{(m)} < Q_t^{(a)} < P_t^{(a)^*}$$

with the spreads as follows:

$$\text{Outer Spreads } P_t^{(a)^*} - Q_t^{(a)} = Q_t^{(b)} - P_t^{(b)^*} = \frac{1}{\gamma} \cdot \log(1 + \frac{\gamma}{k})$$

$$\text{Inner Spreads } Q_t^{(a)} - Q_t^{(m)} = Q_t^{(m)} - Q_t^{(b)} = \frac{\gamma \cdot \sigma^2 \cdot (T-t)}{2}$$

This completes the analytical approximation to the solution of the Avellaneda-Stoikov continuous-time formulation of the Optimal Market-Making problem.

1.3.4 Real-World Market-Making

Note that while the Avellaneda-Stoikov continuous-time formulation and solution is elegant and intuitive, it is far from a real-world model. Real-world OB dynamics are time-heterogeneous, non-linear and far more complex. Furthermore, there are all kinds of real-world frictions we need to capture, such as discrete time, discrete prices/number of shares in a bid/ask submitted by the market-maker, various constraints on prices and number of shares in the bid/ask, and fees to be paid by the market-maker. Moreover, we need to capture various market factors in the *State* and in the OB Dynamics. This invariably leads to the *Curse of Dimensionality* and *Curse of Modeling*. This takes us down the same path that we've now got all too familiar with - Reinforcement Learning algorithms. This means we need a simulator that captures all of the above factors, features and frictions. Such a simulator is basically a *Market-Data-learnt Sampling Model* of OB Dynamics. We won't be covering the details of how to build such a simulator as that is outside the scope of this book (a topic under the umbrella of supervised learning of market patterns and behaviors). Using this simulator and neural-networks-based function approximation of the Value Function (and/or of the Policy function), we can leverage the power of RL algorithms (to be covered in the following chapters) to solve the problem of optimal market-making in practice. There are a number of papers written on how to build practical and useful market simulators and using Reinforcement Learning for Optimal Market-Making. We refer you to two such papers here:

- A paper from University of Liverpool (Spooner et al. 2018)
- A paper from J.P.Morgan Research (Ganesh et al. 2019)

This topic of development of models for OB Dynamics and RL algorithms for practical market-making is an exciting area for future research as well as engineering design. We hope this section has provided sufficient foundations for you to dig into this topic further.

1.4 Key Takeaways from this Chapter

- Foundations of Order Book, Limit Orders, Market Orders, Price Impact of large Market Orders, and complexity of Order Book Dynamics.
- Casting Order Book trading problems such as Optimal Order Execution and Optimal Market-Making as Markov Decision Processes, developing intuition by deriving closed-form solutions for highly simplified assumptions (eg: Bertsimas-Lo, Avellaneda-Stoikov formulations), developing a deeper understanding by implementing a backward-induction ADP algorithm, and then moving on to develop RL algorithms (and associated market simulator) to solve this problem in a real-world setting to overcome the Curse of Dimensionality and Curse of Modeling.

Bibliography

Almgren, Robert, and Neil Chriss. 2000. "Optimal Execution of Portfolio Transactions." *Journal of Risk*, 5–39.

Avellaneda, Marco, and Sasha Stoikov. 2008. "High-Frequency Trading in a Limit Order Book." *Quantitative Finance* 8 (3): 217–24. http://www.informaworld.com/10.1080/14697680701381228.

Bertsimas, Dimitris, and Andrew W. Lo. 1998. "Optimal Control of Execution Costs." *Journal of Financial Markets* 1 (1): 1–50.

Ganesh, Sumitra, Nelson Vadori, Mengda Xu, Hua Zheng, Prashant P. Reddy, and Manuela Veloso. 2019. "Reinforcement Learning for Market Making in a Multi-Agent Dealer Market." *CoRR* abs/1911.05892. http://dblp.uni-trier.de/db/journals/corr/corr1911.html#abs-1911-05892.

Gueant, Olivier. 2016. *The Financial Mathematics of Market Liquidity: From Optimal Execution to Market Making*. Chapman; Hall/CRC Financial Mathematics Series.

Nevmyvaka, Yuriy, Yi Feng, and Michael J. Kearns. 2006. "Reinforcement Learning for Optimized Trade Execution." In *ICML*, edited by William W. Cohen and Andrew W. Moore, 148:673–80. ACM International Conference Proceeding Series. ACM. http://dblp.uni-trier.de/db/conf/icml/icml2006.html#NevmyvakaFK06.

Spooner, Thomas, John Fearnley, Rahul Savani, and Andreas Koukorinis. 2018. "Market Making via Reinforcement Learning." *CoRR* abs/1804.04216. http://dblp.uni-trier.de/db/journals/corr/corr1804.html#abs-1804-04216.

Vyetrenko, Svitlana, and Shaojie Xu. 2019. "Risk-Sensitive Compact Decision Trees for Autonomous Execution in Presence of Simulated Market Response." *CoRR* abs/1906.02312. http://dblp.uni-trier.de/db/journals/corr/corr1906.html#abs-1906-02312.