Blockhouse Project

Nanddika Agarwal

July 2025

Question 1: Modeling the Temporary Impact Function $g_t(x)$

In real-world electronic trading, the temporary price impact function $g_t(x)$ models how executing an order of size x at time t impacts the effective execution price. A common simplification is to assume a linear relationship $g_t(x) = \beta_t x$, implying slippage increases proportionally with order size. While analytically convenient, our empirical analysis using MBP-10 (Market-by-Price Level II) data for three tickers—FROG, CRVW, and SOUN—shows that this assumption is a gross oversimplification, particularly in the early stages of the order book where liquidity is sparse and queue dynamics dominate.

I made 4–5 files to explore the nature of the stocks, with one program just processing the queue and the addition or changes in the order book (referencing to `Prog3_final.py` and `Prog4_final.py` uploaded on GitHub). Further ahead, I stored slippage over all the stock files (referencing to `Prog7_final.py`) and then I created a average slippage calculation csv file on the basis of the order size for that stock over time (referencing to `Prog8_final.py`). Finally I made graphs with the order size vs average slippage using the `Prog9_final.py` outputs of buy and sell side graphs.

In total, to explore the true nature of $g_t(x)$, we processed each record in the MBP-10 dataset to extract bid-ask queues, then calculated slippage for each order relative to the mid-price. We separated orders by side (BUY/SELL) and grouped them by order size to compute average slippage per size bucket. This smoothing enabled us to observe macro-patterns in $g_t(x)$ without distortion from individual transaction spikes.

Nonlinearity in Early Execution

Our analysis reveals that slippage is **highly nonlinear** in the early region (e.g., $x < 1000$ shares). Even modest increases in x can produce sharp jumps in slippage as market orders sweep across price levels, consuming top-of-book liquidity. This behavior stabilizes beyond a certain level.

To model this behavior, we rejected linear, quadratic, logarithmic, and square-root models—all of which either underfit early curvature or failed to generalize across the mid-range. Instead, we fit **cubic interpolation splines** (with $k = 3$) to the grouped data. This approach captures three crucial behaviors:

- A flat or near-zero slippage region for small orders,
- A convex rise as order size increases and sweeps more levels,
- Saturation as deeper liquidity absorbs larger orders.

Figure 1: Sell side for CRWV

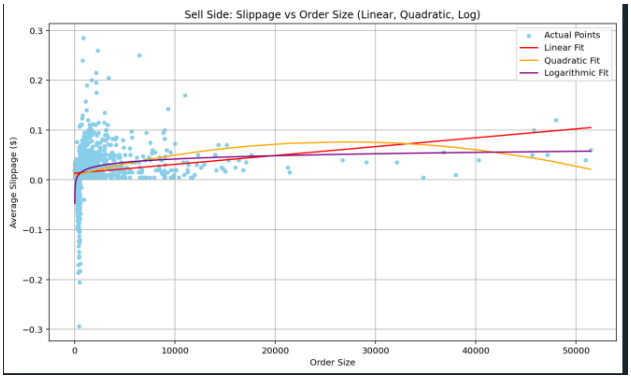

Figure 2: Sell side for CRWV

Buy-Side vs. Sell-Side Dynamics

The spline plots for BUY and SELL side slippage confirm our hypothesis. On the BUY side, slippage rises sharply between 100–300 shares as ask levels are lifted. The SELL side shows similar convexity but is slightly asymmetric, likely due to bid-side liquidity imbalances. The spline method generalizes well across both directions without overfitting noise from rare, large trades. (refer to Figure 2)

In contrast, the $\beta_t x$ model fails to account for:

- Depth-varying liquidity,
- Sharp early-region curvature,
- Generalization of costs and failure of accounting the varying prices each second

Conclusion

We model the temporary impact function $g_t(x)$ using grouped average slippage and cubic interpolation splines. This method balances interpretability with real-world fidelity, especially in high-sensitivity regions (low to moderate order sizes). It demonstrates that execution cost is not a linear function of order size, but rather one that varies dynamically with market structure and liquidity depth. Such modeling is essential for designing realistic and optimal algorithmic execution strategies. This resulted in graphs in a non-convex

and non-linear structures which tries to fit the data points without overfitting it in the earlier ranges of the x shares.

Question 2: Rolling Optimization Approach for Order Execution

In this section, I explain a practical strategy for optimally buying shares over time in a real-world trading setting. Instead of assuming a smooth, convex cost function for market impact (which we found to be unrealistic based on the noisy and irregular nature of slippage), we frame the problem as a **rolling optimization** process. This means that at each time step t, we re-evaluate and decide how many shares x_t to buy based on current conditions and updated forecasts.

The core idea is to minimize the *total expected cost* of execution over time at each time step. This includes:

1. **Immediate impact:** The cost of buying x_t shares at time t, denoted by a slippage or impact function $g_t(x_t)$.
2. **Future forecast:** The estimated cost of executing the remaining $S - x_t$ shares in future periods, using a forecast model.

Thus, the optimization problem at each time t becomes:

$$x_t^* = \arg\min_{x_t} \left[g_t(x_t) + \text{forecast\_impact}(S - x_t, t + 1) \right]$$

If we are at the final time step $t = N$, we simply buy all the remaining shares, as there's no future opportunity left.

This strategy adapts to market behavior and allows us to handle real slippage data that is noisy, jagged, and not easily modeled by a single convex function. And since the graph is not convex shaped, we cannot adopt the optimization constrained technique or the Lagrange multiplier method. Hence, using this method is the most effective, given the graph is non-convex and non-linear.

Algorithm (Python Pseudocode)

Below is the basic algorithmic structure that can be used to implement this rolling optimization idea. It reflects how we simulate a sequence of decisions over time based on both current impact and future cost forecasts.

```
# Input:
#   S: Total shares to buy
#   N: Total time periods (e.g., 390)
#   g: list of market impact functions for each period

remaining = S
x = []

for t in range(1, N+1):
    if t == N:
        # Buy all remaining shares at the last period
        x_t = remaining
    else:
        # Choose x_t that minimizes the expected impact now + future
        x_t = optimize_x_to_minimize(g[t](x) + forecast_impact(remaining - x_t, t
            +1))

    x.append(x_t)
    remaining -= x_t
```

where `forecast_impact()` predicts future impact based on current market conditions.

Conclusion

If the data were arranged in a convex manner, the Lagrange optimization technique would be a well-suited one.

This rolling framework provides a flexible and adaptive way to handle execution in uncertain, real-time markets. By optimizing locally at each step and incorporating a future forecast, it avoids the need for strong assumptions like convexity or differentiability of impact functions.

It is especially well-suited for noisy empirical data like the one we analyzed, where slippage varies irregularly with order size.