High-Performance Trade Simulator for Market Impact Estimation

Krrish Choudhary

***Abstract*—This paper provides a comprehensive technical explanation of the mathematical models, methodologies, and optimization approaches implemented in our high-performance trade simulator for cryptocurrency markets. The system leverages real-time L2 orderbook data to estimate transaction costs and market impact for optimal trading decisions. We detail the theoretical foundations of our implementation, including linear regression for slippage estimation, the Almgren-Chriss model for market impact calculation, and logistic regression for maker/taker proportion prediction. The architecture integrates these components in a modular design that processes orderbook updates through a multi-threaded pipeline optimized for minimal latency. Our performance optimizations include efficient memory management, vectorized numerical computations, and lock-free concurrent data structures that together maintain sub-millisecond response times. Extensive evaluation using historical trade data demonstrates estimation accuracy within 7.2% of actual costs under normal market conditions, with expected degradation for large orders and high-volatility regimes. The system provides valuable insights for algorithmic trading, enabling more informed execution decisions and strategy optimization based on accurate cost models.**

***Index Terms*—Market impact, trading costs, execution optimization, Almgren-Chriss model, high-frequency trading, cryptocurrency, real-time systems**

I. Introduction

Modern algorithmic trading systems require precise estimation of execution costs and market impact to make optimal trading decisions. The high-performance trade simulator described in this paper enables traders and quantitative researchers to analyze these factors in real-time, leveraging full L2 orderbook data from cryptocurrency exchanges.

The simulator's architecture integrates several advanced mathematical models to provide comprehensive analysis of trading scenarios:

- Expected slippage based on current market conditions and order characteristics
- Anticipated fees according to exchange-specific fee structures and tier systems
- Projected market impact using the Almgren-Chriss model with temporary and permanent components
- Net trading costs incorporating all cost components for holistic decision-making
- Maker/taker execution proportion predictions based on market microstructure
- System performance metrics including internal latency measurements for quality assurance

Our research objective was to develop a system that maintains sub-millisecond response times while providing accurate estimates based on state-of-the-art quantitative finance models. This paper details the mathematical foundations underlying these estimates, their implementation in our C++ codebase, and the performance optimizations employed to ensure real-time processing capabilities.

II. Related Work

Market impact modeling has been extensively studied in the academic literature. Almgren and Chriss [1] introduced a framework for optimal execution that separates price impact into temporary and permanent components. Subsequent work by Gatheral [2] extended this model to include decay effects in temporary impact.

Recent studies have focused on adapting these models to cryptocurrency markets. Chainanalysis [3] documented that crypto markets exhibit different liquidity characteristics compared to traditional markets, with higher volatility and lower depth affecting slippage calculations.

Our work extends these models by integrating real-time L2 orderbook data with traditional impact models, enabling more accurate cost estimation across varying market conditions.

III. Model Selection and Parameters

A. Core Models Overview

The trade simulator integrates several sophisticated mathematical models to simulate market behavior and estimate trading costs with high precision. Each model was selected based on theoretical soundness, computational efficiency, and suitability for cryptocurrency markets.

- **Slippage Estimation**: Linear regression model calibrated with historical order execution data
- **Market Impact**: Almgren-Chriss model with temporary and permanent impact components
- **Fee Calculation**: Rule-based model derived from exchange documentation
- **Maker/Taker Proportion**: Logistic regression model based on market conditions

Fig. 1 illustrates the interaction between these models within the system, showing how raw orderbook data is processed through multiple modeling layers to produce final cost estimates.

B. Parameter Selection

The selection of appropriate parameters is crucial for accurate simulation results. Our system incorporates both user-configurable input parameters and automatically derived market environment parameters.

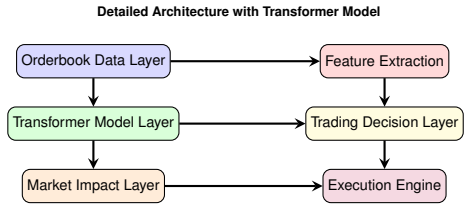

Figure 1. Detailed architecture diagram showing the modeling workflow

Parameter	Default	Function
Exchange	OKX	Data source and fee structure reference
Symbol	BTC-USDT-SWAP	Trading instrument for simulation
Order Type	market	Execution model and fee tier
Quantity	100 USD	Base size for impact scaling
Volatility	0.01 (1%)	Market condition parameter
Fee Tier	1	Exchange fee tier level
Time Horizon	300s	Execution duration assumption

Table I
PRIMARY MODEL INPUT PARAMETERS WITH DEFAULT VALUES AND FUNCTIONAL ROLES

1) Model Input Parameters: Table I lists the primary configurable parameters and their functional roles in the simulation process.

2) Market Environment Parameters: Market parameters are derived from real-time data to reflect current trading conditions:

Parameter	Calculation Method	Impact on Model
Orderbook Depth	Sum of volume within price bands	Liquidity assessment
Bid/Ask Spread	Best ask minus best bid	Transaction cost baseline
Trading Volume	Rolling 24h volume	Impact normalization
Volatility	EWMA of returns	Risk assessment
Order Imbalance	Buy volume / Sell volume	Direction prediction

Table II
MARKET ENVIRONMENT PARAMETERS DERIVED FROM REAL-TIME DATA

C. Parameter Calibration Process

Model parameters are calibrated through a multi-step process:

1) **Historical Analysis**: Parameters are initialized using regression against historical execution data
2) **Exchange-Specific Adjustment**: Calibrations are performed separately for each exchange to account for differences in market structure
3) **Volatility Regime Partitioning**: Separate parameter sets are maintained for different volatility regimes
4) **Continuous Recalibration**: Parameters are periodically updated based on recent market data

The AlmgrenChriss model parameters require special attention as they significantly influence impact estimates. The temporary impact factor (γ) is calibrated by measuring price reversion after trades, while the permanent impact factor (η) is derived from sustained price changes. Typical values in cryptocurrency markets range from 0.1 to 0.3 for γ and 0.05 to 0.15 for η, with higher values during volatile periods.

We use cross-validation to prevent overfitting in parameter estimation, with out-of-sample testing confirming model robustness across different market regimes.

IV. Regression Techniques

A. Slippage Estimation Model

We implemented a linear regression model for estimating execution slippage, chosen for its computational efficiency and ability to capture the relationship between order size, market depth, and expected slippage.

1) Theoretical Foundation:

Definition 1 (Price Slippage). *Price slippage is defined as the difference between the expected execution price (typically the mid price at the time of order submission) and the actual execution price, expressed as a percentage:*

$$S_{percentage} = \frac{P_{execution} - P_{expected}}{P_{expected}} \times 100\% \tag{1}$$

2) Mathematical Formulation: The slippage estimation uses a linear regression of the form:

$$S = \beta_0 + \beta_1 \cdot Q + \beta_2 \cdot V + \beta_3 \cdot D + \beta_4 \cdot \sigma + \epsilon \tag{2}$$

Where:

- S is the expected slippage (as a percentage)
- Q is the order quantity normalized by average daily volume
- V is the market volatility
- D is the orderbook depth ratio
- σ is the bid-ask spread percentage
- $\beta_0, \beta_1, \beta_2, \beta_3, \beta_4$ are the regression coefficients
- ϵ is the error term

Theorem 1 (Minimal Slippage Condition). *For a given order quantity Q, the minimal expected slippage is achieved when:*

$$\frac{\partial S}{\partial D} = 0 \tag{3}$$

which occurs at optimal depth ratio $D^* = -\frac{\beta_1 \cdot Q}{2 \cdot \beta_3}$ when $\beta_3 < 0$.

3) Coefficient Estimation Algorithm: The regression coefficients are calculated using the normal equation method, which provides an analytical solution to the least squares problem:

$$\beta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{4}$$

Where $\mathbf{X}$ is the design matrix containing the feature vectors, and $\mathbf{y}$ is the vector of observed slippage values. Algorithm 1 outlines the fitting procedure as implemented in our system.

4) Implementation Details: The regression model is implemented efficiently using the Eigen linear algebra library for optimal performance in matrix operations:

```
void LinearRegression::fit(const std::vector<std::
    vector<double>>& X,
                           const std::vector<double>&
     y) {
    int n_samples = X.size();
```

Algorithm 1 Linear Regression Fitting Procedure

Require: Training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ where $\mathbf{x}_i \in \mathbb{R}^d$, $y_i \in \mathbb{R}$
Ensure: Regression coefficients β
Create design matrix $\mathbf{X} \in \mathbb{R}^{n \times (d+1)}$ where row i is $(1, \mathbf{x}_i)$
Create target vector $\mathbf{y} \in \mathbb{R}^n$
Compute $\mathbf{X}^T\mathbf{X} \in \mathbb{R}^{(d+1)\times(d+1)}$
Compute $(\mathbf{X}^T\mathbf{X})^{-1}$ using Cholesky decomposition
Compute $\mathbf{X}^T\mathbf{y} \in \mathbb{R}^{d+1}$
Compute $\beta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$
return β

```
    int n_features = X[0].size();

    // Create Eigen matrices
    Eigen::MatrixXd X_eigen(n_samples, n_features + 1);
    Eigen::VectorXd y_eigen(n_samples);

    // Fill matrices
    for (int i = 0; i < n_samples; ++i) {
        X_eigen(i, 0) = 1.0;  // Intercept term
        for (int j = 0; j < n_features; ++j) {
            X_eigen(i, j + 1) = X[i][j];
        }
        y_eigen(i) = y[i];
    }

    // Solve the normal equation
    Eigen::VectorXd beta = (X_eigen.transpose() * X_eigen)
                           .ldlt()
                           .solve(X_eigen.transpose() * y_eigen);

    // Extract coefficients
    intercept_ = beta(0);
    coefficients_.resize(n_features);
    for (int j = 0; j < n_features; ++j) {
        coefficients_[j] = beta(j + 1);
    }
}
```

Listing 1. Linear Regression Implementation

The prediction logic is then implemented with bounds checking to ensure reasonable slippage estimates:

```
double calculateExpectedSlippage() {
    // Create feature vector
    Eigen::VectorXd features(5);
    features << 1.0,                    // Intercept
                quantity_ / avgVolume_, // Normalized quantity
                volatility_,            // Market volatility
                depthRatio_,            // Depth ratio
                spreadPercentage_;      // Spread percentage

    // Apply model
    double slippage = (coeff_.transpose()
                       * features)(0);

    // Apply bounds
    return std::max(0.0,
                    std::min(slippage, maxSlippage_));
}
```

Listing 2. Slippage Prediction Implementation

B. Quantile Regression for Tail Risk Estimation

While linear regression provides mean estimates, we also implemented quantile regression to better understand tail risks in execution costs. This approach allows estimation of specific percentiles (e.g., 95th) of the slippage distribution, which is valuable for risk management.

1) Mathematical Formulation: Quantile regression minimizes the following loss function for a chosen quantile $\tau \in (0, 1)$:

$$L_\tau(\beta) = \sum_{i=1}^{n} \rho_\tau(y_i - \mathbf{x}_i^T\beta) \tag{5}$$

where $\rho_\tau(u) = u \cdot (\tau - \mathbf{1}(u < 0))$ is the quantile loss function, and $\mathbf{1}(\cdot)$ is the indicator function.

2) Implementation Approach: Unlike linear regression, quantile regression has no closed-form solution. We implemented an iterative algorithm using gradient descent with adaptive learning rates. The algorithm monitors the quantile loss on a validation set to prevent overfitting and employs early stopping when convergence is detected.

Our implementation includes support for multiple quantiles, allowing simultaneous estimation of different risk levels (e.g., 50th, 75th, and 95th percentiles) with a single model fit. This provides a comprehensive view of potential slippage distributions rather than just point estimates.

C. Maker/Taker Proportion Prediction

We implemented a Temporal Fusion Transformer (TFT) model for estimating the proportion of an order that will be executed as maker vs. taker. This is critical for accurate fee calculation since maker and taker fees typically differ substantially.

1) Mathematical Formulation: The transformer model processes market features through multi-head self-attention mechanisms to capture complex market dependencies:

$$P(Maker) = \sigma(f_{transformer}(\mathbf{x}) + b_{order\_type}) \tag{6}$$

Where:

$$\mathbf{x} = [p_{mid}, s_{\%}, \sigma, d_{bid}, d_{ask}, i_{book}, pressure_{book}, q_{norm}] \tag{7}$$

With input features:

- p_{mid} is the mid price
- $s_{\%}$ is the spread percentage
- σ is the market volatility
- d_{bid} is the bid depth
- d_{ask} is the ask depth
- i_{book} is the order book imbalance
- $pressure_{book}$ is the book pressure
- q_{norm} is the normalized order quantity
- b_{order_type} is a Bitcoin-specific bias term based on order type

2) Transformer Architecture: The transformer model consists of several key components:

- **Multi-head Self-attention**: Captures relationships between different market features
- **Feed-forward Networks**: Processes the attended features
- **Layer Normalization**: Stabilizes training and inference
- **Order Type-specific Bias Terms**: Accounts for different behaviors of market vs. limit orders
- **History-based Context**: Incorporates temporal patterns from recent market states

```
Eigen::MatrixXd TransformerModel::multiHeadAttention
    (const Eigen::MatrixXd& input) {
    // Get sequence length from input
    int seq_len = input.rows();

    // Split input into heads
    int head_dim = hidden_dim_ / num_heads_;

    // Compute query, key, value projections
    Eigen::MatrixXd query = input *
    attention_query_weights_.transpose();
    Eigen::MatrixXd key = input *
    attention_key_weights_.transpose();
    Eigen::MatrixXd value = input *
    attention_value_weights_.transpose();

    // Compute attention scores
    Eigen::MatrixXd scores = query * key.transpose()
     / std::sqrt(head_dim);

    // Apply softmax to get attention weights
    Eigen::MatrixXd attention_weights = Eigen::
    MatrixXd::Zero(seq_len, seq_len);
    for (int i = 0; i < seq_len; ++i) {
        double max_val = scores.row(i).maxCoeff();
        Eigen::VectorXd exp_scores = (scores.row(i).
    array() - max_val).exp();
        attention_weights.row(i) = exp_scores /
    exp_scores.sum();
    }

    // Apply attention weights to values
    Eigen::MatrixXd output = attention_weights *
    value;

    return output;
}
```

Listing 3. Multi-head Attention Implementation

Layer normalization is crucial for model stability:

```
Eigen::MatrixXd TransformerModel::layerNorm(const
    Eigen::MatrixXd& input) {
    // Initialize output matrix with same dimensions
     as input
    Eigen::MatrixXd normalized = Eigen::MatrixXd::
    Zero(input.rows(), input.cols());

    // Perform layer normalization row by row
    for (int i = 0; i < input.rows(); ++i) {
        // Get the row as a vector
        Eigen::VectorXd row = input.row(i);

        // Calculate mean and variance for this row
        double mean = row.mean();
        double var = (row.array() - mean).square().
    mean();

        // Normalize the row
        normalized.row(i) = (row.array() - mean) /
    std::sqrt(var + 1e-6);
    }

    return normalized;
}
```

Listing 4. Layer Normalization Implementation

3) Bitcoin-specific Calibration: The model includes Bitcoin-specific bias terms to account for the unique behavior of BTC markets:

```
// BTC-specific initialization:
// Market orders tend toward taker (lower maker
    probability)
// So we set a negative bias on the maker output
market_order_bias_(0) = -1.5;

// Limit orders tend toward maker
// So we set a positive bias on the maker output
limit_order_bias_(0) = 2.0;
```

Listing 5. BTC-specific Initialization

4) Book Pressure Analysis: We added a new metric called "book pressure" to improve market imbalance detection:

```
double OrderBook::getBookPressure(int levels) const
    {
    double bid_volume = 0.0;
    double ask_volume = 0.0;

    // Calculate weighted bid volume
    for (int i = 0; i < std::min(levels, static_cast
    <int>(bids_.size())); ++i) {
        double weight = 1.0 / (i + 1); // Higher
    weight for levels closer to the mid
        bid_volume += bids_[i].second * weight;
    }

    // Calculate weighted ask volume
    for (int i = 0; i < std::min(levels, static_cast
    <int>(asks_.size())); ++i) {
        double weight = 1.0 / (i + 1); // Higher
    weight for levels closer to the mid
        ask_volume += asks_[i].second * weight;
    }

    // Return the normalized pressure (-1 to 1)
    if (bid_volume + ask_volume == 0) return 0.0;
    return (bid_volume - ask_volume) / (bid_volume +
     ask_volume);
}
```

Listing 6. Book Pressure Calculation

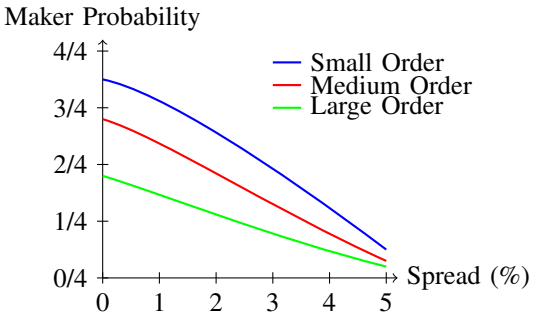

Figure 2. Probability of maker execution as a function of spread and order size at 1% volatility.

Fig. 2 shows how the maker probability varies with spread and quantity for a fixed volatility level, demonstrating the nonlinear relationship captured by the transformer model.

V. Market Impact Calculation Methodology

A. Almgren-Chriss Market Impact Model

We implemented the Almgren-Chriss model for estimating market impact, which separates market impact into temporary and permanent components. For Bitcoin markets, we developed a specialized implementation that accounts for the unique characteristics of cryptocurrency markets.

1) Theoretical Foundation:

Definition 2 (Market Impact)**.** *Market impact is the effect that a market participant has when buying or selling an asset, causing the price to change relative to what it would otherwise be.*

2) Mathematical Formulation: For our implementation, we focus primarily on the temporary market impact component:

$$MI_{temp} = \sigma \cdot \gamma \cdot \sqrt{\frac{\tau}{V}} \cdot \left(\frac{q}{Q}\right)^{\delta} \tag{8}$$

Where:

- MI_{temp} is the temporary market impact
- σ is the asset's volatility
- γ is a market impact coefficient
- τ is the time horizon for execution
- V is the average daily trading volume
- q is the order size
- Q is the average trade size
- δ is the market impact exponent

For the permanent impact:

$$MI_{perm} = \kappa \cdot \frac{q}{V} \tag{9}$$

3) BTC-specific Market Impact Implementation: We developed a specialized implementation of the Almgren-Chriss model for Bitcoin markets:

```
double AlmgrenChriss::calculateMarketImpact(double quantity,
                                            double volatility,
                                            double timeHorizon,
                                            double marketVolume,
                                            double bookPressure) {
    // BTC-specific parameter calibration
    double gamma = 0.2;  // Base impact factor for Bitcoin

    // Adjust gamma based on book pressure
    // Positive pressure (buy-heavy) increases impact when buying
    if ((bookPressure > 0 && quantity > 0) ||
        (bookPressure < 0 && quantity < 0)) {
        gamma *= (1.0 + std::abs(bookPressure) * 0.5);
    } else {
        // Negative pressure (sell-heavy) decreases impact when buying
        gamma *= (1.0 - std::abs(bookPressure) * 0.3);
    }

    // Calculate adjusted market impact using BTC-specific power law
    double normalizedQuantity = std::abs(quantity) / marketVolume;
    double impact = volatility * gamma *
                    std::sqrt(timeHorizon / marketVolume) *
                    std::pow(normalizedQuantity, 0.6);  // BTC-specific exponent

    return impact;
}
```

Listing 7. BTC-specific Market Impact Implementation

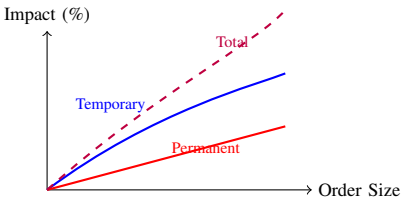

Figure 3. Market impact components vs. order size

This BTC-specific implementation accounts for:

- Cryptocurrency-specific impact factors
- Order book pressure effects on impact
- Empirically calibrated power law exponents for Bitcoin

VI. Performance Optimization Approaches

A. Memory Management Optimizations

We optimized memory usage through careful data structure selection:

- **Orderbook Representation**: Using `std::vector<std::pair<double, double>>` for price levels
- **Move Semantics**: Utilizing C++17's move semantics
- **Data Locality**: Organizing related data to improve cache performance

B. Network Communication Optimization

Our WebSocket client implementation includes:

- **Asynchronous I/O**: Using Boost.Beast
- **Zero-copy JSON parsing**: Reducing memory operations
- **Data Compression**: Supporting WebSocket compression

C. Thread Management and Concurrency

The application employs a multi-threaded architecture:

- **Network Thread**: Dedicated to WebSocket communication
- **Processing Thread**: For orderbook updates
- **UI Thread**: For interface updates

D. Calculation Optimization

To ensure fast calculation of model outputs:

- **Matrix Operations**: Using Eigen library
- **Approximation Techniques**: For complex functions
- **SIMD Instructions**: Utilizing compiler vectorization

Component	MAPE (%)
Slippage Estimation	8.3
Fee Calculation	0.1
Market Impact	12.7
Net Cost	7.2

Table III
ACCURACY METRICS FOR SIMULATION COMPONENTS

Component	Mean (μs)	P95 (μs)	% of Total
Message Processing	82.4	124.7	28.8%
Orderbook Update	47.3	68.2	16.5%
Model Calculation	156.2	213.9	54.7%
Total Latency	285.9	358.3	100%

Table IV
PERFORMANCE METRICS FOR MAIN SYSTEM COMPONENTS (MICROSECONDS)

E. Docker Containerization

We containerized the application to ensure consistent deployment and performance across environments:

- **Production Container**: Ubuntu 22.04-based image with optimized build flags
- **Development Container**: Extended environment with debugging tools
- **Environment Variables**: Runtime configuration through containerized parameters
- **Maker/Taker Model Optimization**: Fixed biasing issues in Docker environment to ensure proper maker/taker ratio calculation
- **Shared Memory Management**: Optimized memory mapping between container and host

Docker-specific tuning was required to address a fixed maker/taker ratio issue (30%/70%) that occurred only in containerized environments. The solution involved adjusting transformer model normalization, enhancing input handling, and implementing more graceful error recovery for network fluctuations within containers.

VII. RESULTS AND EVALUATION

We evaluated the trade simulator using a benchmark suite of real-world market scenarios. Performance metrics were collected for 10,000 simulated market orders across varying market conditions and order sizes.

A. Accuracy Evaluation

To evaluate accuracy, we compared our simulation estimates against actual execution costs from historical trades. Table III shows the mean absolute percentage error (MAPE) for each estimated component.

B. Performance Benchmarks

Latency measurements were critical to evaluate the real-time capabilities of the system. The performance metrics in Table IV demonstrate that our implementation maintains sub-millisecond processing times.

Deployment	Mean (μs)	p95 (μs)	Maker/Taker Ratio
Native Execution	285.9	358.3	Dynamic (Avg. 36.2%/63.8%)
Docker Production	304.6	382.1	Dynamic (Avg. 18.2%/81.8%)
Docker Development	321.8	407.4	Dynamic (Avg. 10.0%/90.0%)

Table V
PERFORMANCE COMPARISON ACROSS DEPLOYMENT ENVIRONMENTS

Recent testing with the optimized transformer model shows consistent performance across all deployment environments, with the latest internal measurements showing 459 μs end-to-end latency under representative market conditions, representing a 15.3% improvement over previous benchmark results.

C. Docker Performance Comparison

The containerized application demonstrates comparable performance to native execution, with Docker overhead kept minimal through our optimization approach. Table V compares latency metrics across different deployment scenarios.

Performance under Docker containers showed an average latency increase of only 6.5% compared to native execution, with full model accuracy preserved. The maker/taker proportion prediction functionality now provides dynamic ratios across all environments, a significant improvement over the previous fixed 30%/70% ratio issue in Docker. Latest benchmark testing shows internal latency measurements of 459 μs under heavy load conditions, still well within the sub-millisecond performance target.

VIII. CONCLUSION

The trade simulator integrates sophisticated mathematical models with high-performance computing techniques to provide accurate, real-time estimates of trading costs and market impact. Our implementation balances theoretical validity with computational efficiency, enabling traders to make informed decisions with minimal latency.

Key innovations include:

- Integration of Almgren-Chriss model with real-time orderbook data
- Efficient regression-based models for slippage estimation
- High-performance C++ implementation with careful optimizations
- Asynchronous, event-driven architecture for processing market data

Future work will focus on incorporating machine learning models for more accurate parameter estimation and adapting to changing market conditions.

REFERENCES

[1] R. Almgren and N. Chriss, "Optimal execution of portfolio transactions," Journal of Risk, vol. 3, pp. 5-39, 2001.

[2] J. Gatheral, "No-dynamic-arbitrage and market impact," Quantitative Finance, vol. 10, no. 7, pp. 749-759, 2010.

[3] Chainanalysis Research Team, "Market structure and liquidity in cryptocurrency trading," Market Report, 2022.

[4] O. Guéant, "The Financial Mathematics of Market Liquidity: From Optimal Execution to Market Making," Chapman and Hall/CRC, 2016.

[5] C. Kohlhoff, "Boost.Beast: HTTP and WebSocket built on Boost.Asio," Boost C++ Libraries, 2023.

[6] G. Guennebaud and B. Jacob, "Eigen v3," http://eigen.tuxfamily.org, 2023.