# Code Appendix for "Quantifying the High-Frequency Trading 'Arms Race' by Matteo Aquilina, Eric Budish and Peter O'Neill, *Quarterly Journal of Economics*[*]

Code credits: Jiahao Chen, Natalia Drozdoff, Matthew O'Keefe, Jaume Vives, Zizhe Xia, Matteo Aquilina, Eric Budish, Peter O'Neill
Documentation credits: Jiahao Chen, Zizhe Xia, Matteo Aquilina, Eric Budish, Peter O'Neill

Version 1.0

September 2021

## 1 Overview

This document serves as a guide for future researchers, regulators or practitioners who wish to use financial-exchange message data to quantify latency arbitrage and study other aspects of speed-sensitive trading, following Matteo Aquilina, Eric Budish and Peter O'Neill, "Quantifying the High-Frequency Trading 'Arms Race"', *Quarterly Journal of Economics,* 2021 (hereafter, "ABO"). This guide should be used in conjunction with the Python and R code that we have made publicly available at `github.com/ericbudish/HFT-Races`.[1] The Python code processes the user's message data, detects trading races, and outputs a race-level statistical dataset along with complementary trading data. The R code produces race summary statistics, tables and figures analogous to all reported results in ABO. We also provide a small artificial data set that can be used to understand the data structure and to test one's configuration.

This code package is an improved version of the code we used in our study, edited (i) to be more universally usable (i.e., for any financial exchange message data set, abstracting from any conventions specific to our message data from the London Stock Exchange) and (ii) to improve run time.

The remainder of this document is structured as follows:

- Section 2. Basic Data Requirements. This section describes the requirements for a message data set to be able to conduct a study along the lines of ABO. No programming expertise is necessary to read this section.

- Section 3. Data Pre-Processing: Translating Your Message Data into the Package's Expected Data Format. This section goes through the necessary pre-processing steps to translate the

[*]We would be grateful for feedback or comments on the code, and are especially eager to hear from early users. Please address comments, questions, and any other feedback to eric.budish@chicagobooth.edu and hft.races.code.package@gmail.com.

[1]Version 1.0 of the code is also posted publicly on the *QJE's* website.

user's message data into the form expected by the Python package. It contains four sub-parts: Section 3.1: Required structure of the data. Section 3.2: New variables to construct. Section 3.3: Variable names and value conventions. Section 3.4: Required reference files (tick size information, symbol-date list, etc.).

- Section 4. Set Race Detection Parameters. This section details how to set parameters for the race detection code, such as time horizon and the minimum required number of participants. We walk the user through how to define a baseline scenario, sensitivity scenarios, and discuss the distinction between race-detection parameters (which are computationally time intensive) and post-race filters (which are computationally cheap).

- Section 5. Computational Setup. This section details the requirements for the computational environment and file structure setup. It also provides some rough magnitudes for the time it will take to run the rest of the code based on our own experience.

- Section 6. Run Message Data Processing Code. This section walks through how to run three key pieces of code for processing message data: Event Classification, Order Book Maintenance and Trading and Order Book Statistics. Event Classification translates combinations of inbound and outbound messages into market events (e.g., "New order adds to book", or "Failed immediate-or-cancel request"). Order Book Maintenance builds and maintains the order book throughout the trading day. Trading and Order Book Statistics produces a trade-level statistical dataset and a symbol-date-level statistical dataset. The trade-level dataset reports statistics related to each trade. The symbol-date-level dataset reports the aggregate statistics on volume, midpoint, and spread for each symbol-date. This step is relatively simple for the user but will be compute-time intensive. Technical details for what happens "under the hood" are in Section 10.

- Section 7. Run Race Detection Code. This section walks through how to run the Python race-detection code. This step is relatively simple for the user but will be compute-time intensive. Technical details for what happens "under the hood" are in Section 10.

- Section 8. Description of Python Outputs from Race Detection. This section details the output files produced by the Python code.

- Section 9. Produce Race Statistics, Figures and Tables using the R Code. This section walks the reader through how to use the Python output files and the R package to generate race statistics, figures and tables, to replicate the analysis in the Main Results section of ABO. We also cover how to perform sensitivity analyses using either new runs of race detection with alternative parameters or filters off of an existing run.

- Section 10. Technical Details for the Python Package. This section describes in detail the inner workings of the Python package for message data processing (corresponding to Section 6) and Race Detection (corresponding to Section 7)

For reference, the structure of the code on `github.com/ericbudish/HFT-Races` is provided in Figure 1.

Figure 1: **Code Package File Structure**

```
├─ PythonCode
│  ├─ data_validation.py Script to test whether the pre-processed data meet the requirements of the package
│  ├─ process_msg_data_main.py Main script to run message data processing
│  ├─ race_detection_main.py Main script to run race detection
│  ├─ log_monitor.py Script to monitor the progress of program runs
│  ├─ LatencyArbitrageAnalysis
│  │  ├─ Classify_Messages.py Functions to group messages into events
│  │  ├─ Prep_Order_Book.py Functions to construct order book based on message data
│  │  ├─ Trading_and_Order_Book_Stats.py Functions to calculate trading and order book statistics
│  │  ├─ Race_Detection_and_Statistics.py Functions to detect races and calculate race statistics
│  │  ├─ OrderBook
│  │  │  ├─ Orders.py Classes used in event classification
│  │  │  └─ OrderBook.py Classes used in order book construction
│  │  ├─ RaceDetection
│  │  │  ├─ Prep_Race_Data.py Functions to generate variables for race detection
│  │  │  ├─ Race_Detection_Functions.py Functions to detect races
│  │  │  ├─ Race_Statistics_Functions.py Functions to calculate race statistics
│  │  │  └─ Race_Msg_Outcome.py Functions to determine whether race messages are successes or fails
│  │  └─ utils
│  │     ├─ Dtypes.py Module with Pandas datatype for the program
│  │     ├─ Validate_Data.py Functions to validate the pre-processed message data
│  │     ├─ Logger.py Functions to log the process
│  │     ├─ Monitor_Logs.py Functions to monitor and aggregate the log files
│  │     └─ Collect_Statistics.py Functions to combine output statistics
│  └─ MiscABOCodeSnippets
│     ├─ preprocessing_example.py Example script for data pre-processing
│     └─ code_snippet_{analysis}.py Python code snippets for results that rely on details specific to ABO's setting
├─ RCode
│  ├─ MainResults.R R script to reproduce tables and figures in the Main Results section of ABO
│  ├─ Sensitivity.R R script to reproduce the sensitivity tables in ABO
│  └─ Functions.R R script for functions used in MainResults.R and Sensitivity.R
└─ ArtificialTestData
   ├─ Sample_All_SymDates.csv.gz Sample input file for symbol-date list
   ├─ Sample_Ticktables.pkl Sample input file for tick size information
   ├─ 2000-01-01/Raw_Msg_Data_2000-01-01_ABCD.csv.gz Sample pre-processed message data for a hypothetical symbol ABCD
   └─ Sample_Input_Race_Parameters.csv Sample input file for race detection parameters
```

## 2 Basic Data Requirements

This section describes the basic data requirements for a message data study along the lines of Aquilina, Budish and O'Neill (2021) (hereafter "ABO"). This section is written at a level that should be accessible to readers without programming expertise. A more technical description of data requirements is provided in the following section, Section 3.

### 2.1 Message Data

Every financial market has a different set of message specifications for the electronic communication between market participants and the exchange. However, at a high level, all message specifications should have a set of inbound messages which allow market participants to initiate requests to the exchange (e.g., send a new order), and a set of outbound messages which communicate the exchange activities back to the participants (e.g., the order was added to the book).

Users of this package should obtain all inbound and outbound messages between market participants and the exchange for a meaningful set of symbols and dates. The messages should have timestamps that are sufficiently accurate for race detection (see Section 2.2 of ABO for a discussion of the timestamps in the context of that study). The inbound and outbound messages should at least cover the following set of core message types:

- Inbound: "Plain vanilla" limit orders, immediate-or-cancel (IOC) limit orders, order modifications, order cancelations.
- Outbound: new orders added to the book, executed trades, successful cancels, failed cancels (often called "too late to cancel"), and failed immediate-or-cancel requests (often called "expired")

Other message types can be incorporated as well but are not essential. Messages should contain at least the following kinds of information:

- For all messages: timestamps, identifiers for the symbol and date, and identifiers that enable the researcher to link outbound messages to inbound messages
- For limit orders and IOCs: the price, side, and quantity of the order
- For cancels and order modifications: order identifying information that allows the researcher to identify which order the participant is requesting to cancel or modify
- For executed trades: information about the trade, including price, quantity, and identifiers that allow the researcher to identify which orders executed
- For failed cancels and failed IOCs: identifying information that allows the researcher to identify which cancel the participant was too late to cancel for, or which IOC order failed to execute. (These are important special cases of the more general principle that the researcher must be able to link outbound messages to the relevant inbound messages).

We emphasize that the identifying information can be anonymous (it was in ABO). In the ABO study, the identifying information enabled the research to link inbound and outbound messages as described above, and also to track over time whether the same anonymized participants were participating in multiple races. In principle much of the ABO study could be conducted without this latter information (e.g., if for each new order, the participant ID was re-anonymized), but some elements of the study such as the material in Figure 4.2, Figure 4.3, and Table 4.4 would not be possible in this case.

Relative to traditional limit order book data sets such as TAQ or exchange direct-feeds, the key features to underscore are: (i) the call-and-response nature of the message data (i.e., the combination of inbounds and outbounds and the ability to link these); (ii) the failed IOCs and failed cancels; and (iii) timestamps that are sufficiently accurate for race detection (exchange direct feeds often share this feature but TAQ may not).

**Notes on Executable Quotes** The package can handle message data with both ordinary orders and LSE-like executable quotes. An executable quote is a double-sided order, usually placed by registered market makers. When a quote is posted to the order book, it is treated as two separate and independent limit orders valid for the trading day. A particular market participant may only have one active quote per symbol. Some exchanges may not support executable quotes. The package is applicable to message data with or without such executable quotes. If there is no executable quote in the exchange, the package will skip quote handling.

**Note on Packet Loss** Ideally, the data should contain all inbound and outbound messages. In practice, the data recording device may not capture all messages due to packet loss. The package has built-in protections against a modest level of packet loss. Severe packet loss can introduce noise in the final output. Users of the package should ask their data provider for information about the extent of packet loss in their context.

**Note on Other Message Types** The package is designed primarily to handle the core inbound message types listed above: limit orders, IOCs, order modifications, and order cancelations. The package also includes support for several other message types (e.g., quotes, market orders, fill-or-kill, pegged, good-till), with judgment calls made in the code as to whether these other message types can count as part of races or not (none of these are quantitively important in the LSE context but they may be in other contexts). Please see Section 10.4.1 for details

In practice there are many other message types that are not explicitly supported in the code. For any additional message type, users of the package will have to decide whether they can be mapped to one of the message types recognized by the package, or, if they do not belong, the user can label them as "other messages" in the pre-processing steps discussed in in Section 3.

## 2.2 Reference Data

In addition to the message data, the package also requires the following reference datasets.

- Symbol-date list: A list of all symbol-date pairs in the message data.
- Tick tables: Tick size information for each symbol-date.
- Regular trading hours: Timestamps of the regular hours of continuous trading. While one can use the publicly available exchange trading hours (e.g. LSE trades from 8:00 to 16:30), the package works better with timestamps at microsecond level. For example, the London Stock Exchange opens at 8:00 am, but the actual opening time may not be exactly 8 o'clock due to randomness of the auction uncross time. Users need to convert this information into a variable of the message data in the pre-processing steps following Section 3.2.

- Information on the opening and intraday auctions: At a minimum, this should allow users of the package to construct dummy variables that separate outbound trade confirmations for opening and intraday auctions from regular trades. This can be timestamps for opening and intraday auctions, or identifiers attached to outbound trades that specify the trade occurred during an auction. One can also use the scheduled opening and closing auction times instead of microsecond level timestamps. But this may introduce noise because exchanges often have dynamic open times (such as price monitoring extensions that delay the auction uncross if certain thresholds are met), especially for exchanges with substantial auction volume. Users need to convert this information into a variable of the message data in the pre-processing steps following Section 3.2.

## 3 Data Pre-Processing: Translating Your Message Data into the Package's Expected Data Format

Raw data provided by exchanges can come in various formats. The first step for users of this package is to translate their data from its original format into a format that can be recognized by the package. This section serves as a guide on this step, called data pre-processing. From this section forward this document assumes a modest level of programming experience.

Users of the package should pre-process their raw data in four steps:

- Step 1: Parse and reshape the message data into symbol-date level dataframes and sort the messages by their timestamps. Section 3.1 provides detailed instructions on this step.
- Step 2: Construct the variables described in Section 3.2 based on other fields in the message data and the reference data.
- Step 3: Rename variables and re-label values following Section 3.3.
- Step 4: Create two reference data files with tick size information and a symbol-date list. Section 3.4 describes the preparation of the reference data files.

We provide a code snippet for data preprocessing specific to the ABO setting at `/PythonCode /MiscABOCodeSnippets/preprocessing_example.py` as an example. Users may have to build their own code if the institutional details are sufficiently different. Depending on the nature of the original data, it may be better to perform these four steps in an alternative order. We also provide a sample pre-processed message data set at `/ArtificialTestData/2000-01-01/ Raw_Msg_Data_2000-01-01_ABCD.csv.gz` to help users better understand the package's expected data format. After data pre-processing, users of the package can check their data against Table 1 to make sure their pre-processed data comply with the expected format. We provide a Python script to check whether the pre-processed message data comply with the expected format. Users should follow the instructions in Section 5 to validate their data.

### 3.1 Restructure the Message Data

Users of the package should perform the following tasks and the output from this step should be symbol-date level dataframes of messages sorted by their timestamps. Each dataframe should be saved to a separate `.csv.gz` data file.

**Parse the Message Data** The Python package can only handle dataframes of messages. However, exchanges may provide message data in other forms including binaries, strings and JSON files. In this case, users have to parse their data into dataframes. A dataframe required by this package is a table of message data where each row represents an inbound or outbound message and each column represents a message data variable. For example, a dataset with one million messages and one hundred variables should be reshaped as a dataframe with one million rows and one hundred columns. Users can make use of open source message data parsers or construct their own parsers based on the exchange's message protocol specification documents.[2] Some exchanges may support multiple message formats. One may need a different parser for each message format. For instance, the London Stock Exchange accepts inbound messages in either the FIX or the Native interface, and each requires a different message data parser.

**Partition Messages by Symbol-Date** We ask users to partition the message data by the symbol and the date of the message. This is because the package analyzes the message data for each symbol-date independently. Specifically, after parsing the message data into dataframes, users should split these dataframes by symbol-date and save the data separately for each symbol-date. There should be a single `.csv.gz` data file for each symbol-date, and the file should contain all inbound and outbound messages in the given symbol-date.

**Sort Messages by Timestamp** The package also requires all messages within each symbol-date to be organized in the ascending order according to their timestamps. Users should sort the message data files if they are not in the correct order.

## 3.2 Construct New Variables

In our analysis of the LSE message data, we created several new variables based on existing fields of the message data and the reference data. These variables are necessary for the package but they are not directly available in our data and have to be constructed. Users of this package may encounter similar issues. This section provides detailed instructions on how to construct these necessary variables based on existing message data fields and the reference data. The output from this step should be symbol-date level message data files with the following variables created. This step can be skipped if the raw data has all the variables.

### 3.2.1 UniqueOrderID

**Description** `UniqueOrderID` is an identifier grouping inbound and outbound messages by orders and, if any, quotes. `UniqueOrderID` should link all inbounds and outbounds associated with the same order (quotes) throughout its lifetime. It is necessary to build this identifier because exchanges usually use multiple identifier fields at the order level, and the order identifiers may change during its life span. For example, in LSE, a market participant can supply a client-defined identifier when submitting a new order, and the matching engine will also assign a machine-defined identifier to that order. When the participant cancels and replaces the order, the machine-defined identifier may change. It is therefore convenient to define a unified identifier that stays the same from the initial submission of the order to the end of the order (e.g., full execution or expiration).

[2]Example of open source message data parser: FIX messages, pyfixmsg by Morgan Stanley `https://github.com/morganstanley/pyfixmsg`. Examples of exchange message specifications: FIX messages, `https://www.fixtrading.org/online-specification/`; Nasdaq OUCH, `https://www.nasdaqtrader.com/content/technicalsupport/specifications/TradingProducts/OUCH4.2.pdf`; NYSE, `https://www.nyse.com/publicdocs/NYSE_Pillar_Gateway_Binary_Protocol_Specification.pdf`.

**Construction** `UniqueOrderID` can be constructed upon a client-supplied identifier (this field is usually called the `ClientOrderID`) and a matching-engine-supplied identifier (this field is usually called the `MEOrderID`). To construct `UniqueOrderID`, loop through all messages within a symbol-date and use the client-supplied and the matching-engine-supplied identifiers to link together all inbounds and outbounds related to the same order or, if any, quote.

- All messages related to the same order should be assigned the same `UniqueOrderID`. This typically includes an inbound new order message which initializes the order, a few (possibly zero) inbound cancel or cancel/replace requests that attempt to cancel or update the order, and all outbounds associated to the order.
- All messages related to a quote, if any, should also be assigned the same `UniqueOrderID`. This typically includes one or more inbound new quote messages, a few (possibly zero) inbound cancel requests, and all associated outbounds. All quote-related messages from the same market participant within a symbol-date should be assigned the same `UniqueOrderID` because one market participant can only have one active quote at a time for each symbol.

In our pre-processing code example, we provide the code to generate `UniqueOrderID` for the LSE data. Users can read and learn from the example before constructing their own `UniqueOrderID` variable. Users may be able to apply the code with modest edits if their message data is similar to the LSE data. However, one may have to build their own code to construct this variable if the institutional details of the exchange are sufficiently different.

### 3.2.2 RegularHour

**Description** `RegularHour` is a boolean variable indicating whether the message is within the regular hours of continuous trading.

**Construction** Obtain the timestamps of the regular trading sessions from the reference data. Set `RegularHour` to `True` if the message is within the regular trading hours, `False` otherwise.

### 3.2.3 OpenAuctionTrade/AuctionTrade

**Description** `OpenAuctionTrade` and `AuctionTrade` are boolean variables indicating whether a trade confirmation is from an auction trade or an open auction trade.

**Construction** Obtain this information from reference data and construct the fields for outbound trade confirmations. Set `OpenAuctionTrade` or `AuctionTrade` to `True` if the trade confirmation is from an opening auction or auction trade, `False` otherwise.

### 3.2.4 QuoteRelated

**Description** `QuoteRelated` is a boolean variable indicating whether the message is related to a quote. This includes inbound messages to submit a new quote or to cancel an existing quote and all outbounds associated to the quote.

**Construction** If the exchange does not support executable quotes, set `QuoteRelated` to `False` for all messages and the package will skip quote handling. If there are executable quotes in the exchange, set `QuoteRelated` to `True` for all inbounds and outbounds related to quotes, and `False` for other messages.

## 3.3 Rename Variables and Re-Label Categorical Values

The same message attribute might receive different variable names across exchanges. For example, the client-supplied order identifier is called `ClientOrderID` in LSE and `ClOrdID` in NYSE. The values of categorical variables in the message data, usually encoded in alphanumeric strings, can also have exchange specific meanings. For example, in LSE FIX messages, "3" in the time qualifier field stands for immediate-or-cancel while Nasdaq OUCH messages use "0" for IOCs. To resolve the potential inconsistency, we ask users to rename the variables and re-label the categorical values following the convention of this package. This section describes the set of variable names and value labels recognized by the package. Users should reset the variable names and re-label values in their data accordingly.

While it is straightforward to reset variable names, one may have to make judgment calls to convert the values labels to the required format. If the user's data contains messages that are not explicitly supported by the code (e.g., messages with certain order types or time qualifiers not listed in Section 3.3.2), the user will have to decide whether they can be mapped to one of the message types recognized by the package, or set `MessageType=Other_Inbound/Other_Outbound`, in which case they will be ignored in the analysis. For example, orders in LSE can have multiple time qualifiers, including good-for-auction, immediate-or-cancel, fill-or-kill, good-till-date, good-till-time, etc. This package only distinguishes among good-till-X, IOC, FOK, and GFA. To map the LSE-specific value labels to our preferred convention, we combined good-till-date, good-till-time, and a few other time qualifier types into the "good-till-X" category. Users may need to make similar decisions for categorical variables following the exchange's message specification documents.

Section 3.3.1 lists out the recognized variables names and value labels for variables required for all messages. Sections 3.3.2-3.3.8 list out the recognized variables names and value labels for additional variables required for certain types of inbound and outbound messages. Table 1 summarizes the required variable names and value labels for all types of messages. Users should use this section and Table 1 as a checklist for data pre-processing. The output from this step should be symbol-date level data files that are compliant with Table 1.

### 3.3.1 Variables Required for All Messages

All messages should have the following variables. Missing values are not expected unless otherwise stated.[3]

#### Identifiers

- `Symbol`: str. The symbol of the message, representing the underlying security.
- `Date`: YYYY-MM-DD. The date of the message.
- `SessionID`: str. An identifier associated with the trading session of the message. A trading day in some exchanges can include multiple trading sessions separated by lunch breaks or intraday auctions. If there is only one trading session per trading day, one can use the date as the trading session identifier. This field can be N/A for messages not in the regular trading hours.

[3]Missing values are not allowed in the data unless otherwise stated. However, users can drop messages with missing values in the required data fields and treat them as packet loss. If the message data of a symbol-date contains too many missing values in required fields, we recommend dropping the symbol-date from the analysis.

Table 1: Required Message Data Variable Names and Format Conventions

Panel A: Data Fields Required for All Messages

| **Message Type** | **Variable Name** | **Value Format** |
|---|---|---|
| | | *Identifiers* |
| All Messages | `Symbol` | str. |
| | `Date` | YYYY-MM-DD. |
| | `SessionID` | str. This field can be N/A for non-regular-hour messages. |
| | `UserID` | str. |
| | `FirmID` | str; set `FirmID = UserID` if not available. |
| | `ClientOrderID` | str |
| | `MEOrderID` | str; N/A for `New_Order` inbounds. |
| | `UniqueOrderID` | str. |
| | | *Timestamps* |
| | `MessageTimestamp` | YYYY-MM-DD HH:MM:SS.ssssss(sss). |
| | | *Basic Information* |
| | `MessageType` | For inbounds: `New_Order`, `New_Quote`, `Cancel_Request`, `Cancel_Replace_Request`, `Other_Inbound`. For outbounds: `Execution_Report`, `Cancel_Reject`, `Other_Reject`, `Other_Outbound`. |
| | `Side` | `Bid`, `Ask`; N/A for quote-related inbounds. |
| | `QuoteRelated` | bool; set to `False` for all messages if quotes not supported. |
| | `RegularHour` | bool. |

Panel B: Additional Data Fields Required for Inbound Messages

| **Message Type** | **Variable Name** | **Value Format** |
|---|---|---|
| `New_Order` | `OrderType` | `Limit`, `Market`, `Stop`, `Stop_Limit`, `Pegged`, `Passive_Only`. |
| | `TIF` | `GoodTill`, `IOC`, `FOK`, `GFA` |
| | `OrderQty` | int. |
| | `DisplayQty` | int, set `DisplayQty = OrderQty` if not available. |
| | `LimitPrice` | float. |
| | `StopPrice` | float, N/A if not available. |
| `New_Quote` | `BidSize` | int. |
| | `AskSize` | int. |
| | `BidPrice` | float. |
| | `AskPrice` | float. |
| `Cancel_Replace_Request` | `OrigClientOrderID` | str. This and `MEOrderID` cannot both be missing. |
| | `MEOrderID` | str. This and `OrigClientOrderID` cannot both be missing. |
| | `OrderQty` | int. |
| | `DisplayQty` | int, set `DisplayQty = OrderQty` if not available. |
| | `LimitPrice` | float. |
| | `StopPrice` | float, N/A if not available. |
| `Cancel_Request` | `OrigClientOrderID` | str. This and `MEOrderID` cannot both be missing. |
| | `MEOrderID` | str. This and `OrigClientOrderID` cannot both be missing. |

Panel C: Additional Data Fields Required for Outbound Messages

| **Message Type** | **Variable Name** | **Value Format** |
|---|---|---|
| `Execution_Report` | `ExecType` | `Order_Accepted`, `Order_Cancelled`, `Order_Executed`, `Order_Expired` (including IOCs failed to trade), `Order_Rejected`, `Order_Replaced`, `Order_Suspended`, `Order_Restated`. |
| | `LeavesQty` | int. |
| | `DisplayQty` | int, set `DisplayQty = LeavesQty` if not available. |
| `Execution_Report` with `ExecType = Order_Executed` | `TradeMatchID` | str. |
| | `TradeInitiator` | `Aggressive`, `Passive`, `Other`. |
| | `OrderStatus` | `Partial_Fill`, `Full_Fill`. |
| | `ExecutedPrice` | float. |
| | `ExecutedQty` | int. |
| | `LeavesQty` | int. |
| | `DisplayQty` | int, set `DisplayQty = LeavesQty` if not available. |
| | `AuctionTrade` | bool. |
| | `OpenAuctionTrade` | bool. |
| `Cancel_Reject` | `CancelRejectReason` | `TLTC` (This is failed cancel attempt in race detection), `Other`. |

Notes: This table describes the required message data fields and the recognized value format. Users of the package should refer to this table and Section 3.3 as a checklist for data pre-processing. Panel A lists out the required data fields for all messages. Missing values in these fields are not allowed unless otherwise stated. Panel B and Panel C describe the additional variables required for certain types of inbound and outbound messages. Fields not related to a particular type of inbound or outbound message can be left blank. The table specifically denotes "N/A" as an option if the field is related to a particular inbound or outbound message but it is allowed to be blank. Messages types `Other_Inbound`, `Other_Reject`, and `Other_Outbound` do not require additional variables and are omitted from this table.

- `UserID`: str. An identifier associated with the participant sending the message. Some exchanges may allow a trading firm to have multiple accounts. They may also have further identifiers by trader and trading-desk. The package only requires one unique identifier for senders of messages.
- `FirmID`: str. Optional. An identifier associated with the firm of the participant who sends the message. The package can handle this optional `FirmID` on top of `UserID`. If `FirmID` is available in the data, the package can perform firm level analysis. If `FirmID` is not available, set `FirmID = UserID` for all messages and the package is still applicable. In this case, it will not be able to perform any firm level analysis.
- `ClientOrderID`: str. A participant-supplied identifier for the order associated with the message.
- `MEOrderID`: str. An exchange machine-engine-supplied identifier for the order associated with the message. This field can be N/A for new order inbounds.
- `UniqueOrderID`: str. An identifier that links together all inbounds and outbounds associated with the same order. Users of the package have to construct this variable following Section 3.2.

## Timestamp

- `MessageTimestamp`: YYYY-MM-DD HH:MM:SS.ssssss(sss). Timestamps with at least microsecond precision. Please refer to Section 2.2 of ABO for more detail. The package can handle timestamps with at most nanosecond precision. Due to the limitation of Pandas 1.0.0, the package is not able to go beyond nanoseconds. Timestamps with higher precision will be rounded to nanoseconds.

**Basic Information**

- `MessageType`: Indicates the type of the message.
  - This variable can only take the following values for inbound messages:
    - `New_Order`: Submits a new order.
    - `New_Quote`: Submits a new quote or updates an existing quote.
    - `Cancel_Request`: Requests to cancel an order or a quote.
    - `Cancel_Replace_Request`: Requests to cancel and replace an order.
    - `Other_Inbound`: Inbounds that cannot be categorized as any one of the above. Messages in this category will not be considered in the subsequent analysis.
  - This variable can only take the following values for outbound messages:
    - `Execution_Report`: Indicates an action of the matching engine. "Execution" here refers to an action performed by the matching engine, including, but not limited to, the execution of an order. Other examples include order acceptance, expiration, cancellation, etc. The type of the action must be described by the field `ExecType`.
    - `Cancel_Reject`: Indicates the cancel request of an orders or a quote is unsuccessful. The reason for the rejection must be given in the field `CancelRejectReason`.
    - `Other_Reject`: Rejects the request to submit or cancel an order or a quote due to technical reasons (e.g. unsupported message type, missing certain required field, wrong input).
    - `Other_Outbound`: Outbounds that cannot be categorized as any one of the above. Messages in this category will not be considered in the subsequent analysis.
- `Side`: `Bid` or `Ask`, N/A for quote-related inbound messages, if any.
- `RegularHour`: bool. `True` for messages sent during the regular trading hours, `False` otherwise. Users of the package have to construct this variable following Section 3.2.
- `QuoteRelated`: bool. `True` for messages associated with a quote, `False` otherwise. If the message data does not contain quotes, `QuoteRelated` should be `False` for all messages. Users of the package have to construct this variable following Section 3.2.

### 3.3.2 Additional Variables Required for Inbound `New_Order` Messages

Market participants use inbound `New_Order` messages to submit a new order to the exchange. In addition to the required fields in Section 3.3.1, inbound `New_Order` messages must contain the following additional variables. Missing values are not expected in these variables for `New_Order` messages unless otherwise stated. For other types of messages, these fields can be N/A.

**Order Type**

- `OrderType`: Indicates the type of the order. At a minimum, this field should distinguish limit orders from market orders if market orders exist on an exchange. Only the following values are allowed.

  - `Limit`: Limit order.
  - `Market`: Market order.
  - `Stop`: Stop order.
  - `Stop_Limit`: Stop limit order (inactive limit order until the market price reaches the stop price).
  - `Pegged`: Pegged orders with order price being pegged to the best bid/best ask.
  - `Passive_Only`: Limit orders that can only be accepted to the book if it does not match visible contra orders.

**Time Qualifier**

- `TIF`: Indicates the time qualifier of an order. At a minimum, this should distinguish immediate-or-cancel orders from good-till-X orders. Only the following values are allowed.
  - `GoodTill`: Good-till-X. Examples include good-for-day, good-till-time, good-till-date, good-till-cancel, etc.
  - `IOC`: Immediate-or-cancel. Either the order trades aggressively against the resting orders and the unmatched part, if any, expires immediately, or the order expires immediately.
  - `FOK`: Fill-or-kill. The order expires immediately if it is not fully filled.
  - `GFA`: Good for auction only. The order can only participate in auction sessions.

**Price and Quantity**

- `OrderQty`: int. Total quantity of the order. All orders must have an `OrderQty`.
- `DisplayQty`: int. Optional. Displayed quantity of the order. If this information is not available or the exchange does not support iceberg orders, set `DisplayQty = OrderQty`.
- `LimitPrice`: float. Limit price of a limit or stop limit order. All limit orders must have a `LimitPrice`.
- `StopPrice`: float. Optional. Stop price of a stop or stop limit order. Set `StopPrice` to N/A if it is not available.

### 3.3.3 Additional Variables Required for Inbound `New_Quote` Messages

Market participants use inbound `New_Quote` messages to submit a new quote or update an existing quote. In addition to the required fields in Section 3.3.1, inbound `New_Quote` messages must contain the following additional variables.[4] Missing values are not expected in these variables for `New_Quote` messages. For other types of messages, set these fields to N/A.

**Price and Quantity**

- `BidSize/AskSize`: int. Bid and ask quantities. Set both to N/A if the exchange does not support executable quotes.
- `BidPrice/AskPrice`: float. Bid and ask prices. Set both to N/A if the exchange does not support executable quotes.

[4]Note that if the exchange supports executable quote, an inbound `New_Quote` message is only required to have the price and quantity information. This is because the `OrderType` and `TIF` of an executable quote is fixed. It is treated as two separate limit orders valid for the trading day.

### 3.3.4 Additional Variables Required for Inbound `Cancel_Replace_Request` Messages

Market participants use inbound `Cancel_Replace_Request` messages to modify an existing order. In addition to the required fields in Section 3.3.1, inbound `Cancel_Replace_Request` messages must contain the following additional variables. Missing values are not expected in these variables for `Cancel_Replace_Request` messages unless otherwise stated. For other types of messages, these fields can be N/A.

**Order Referencing Information** At least one of the following identifiers must be non-empty.

- `OrigClientOrderID`: str. The participant-supplied order ID of the order to modify.
- `MEOrderID`: str. The matching-engine-supplied order ID of the order to modify.

**Price and Quantity** The same price and quantity variables as in inbound `New_Order` messages.

### 3.3.5 Additional Variables Required for Inbound `Cancel_Request` Messages

Market participants use inbound `Cancel_Request` messages to cancel an existing order. In addition to the required fields in Section 3.3.1, inbound `Cancel_Request` messages must contain the following additional variables. Missing values are not expected in these variables for `Cancel_Request` messages unless otherwise stated. For other types of messages, these fields can be N/A.

**Order Referencing Information** At least one of the following identifiers must be non-empty.

- `OrigClientOrderID`: str. The participant-supplied order ID of the order to cancel.
- `MEOrderID`: str. The matching-engine-supplied order ID of the order to cancel.

### 3.3.6 Additional Variables Required for Outbound `Execution_Report` Messages

Exchanges send execution reports to market participants to communicate an action related to an order. "Execution" here refers to an action executed by the matching engine. This includes, but is not limited to, the execution of an order. In addition to the required fields in Section 3.3.1, outbound `Execution_Report` messages must contain the following additional variables. Missing values are not expected in these variables for `Execution_Report` messages unless otherwise stated. For other types of messages, set these fields to N/A.

**Execution Report Type**

- `ExecType`: Indicates the reason the execution report is being sent. Only the following values are allowed.
  - `Order_Accepted`: Indicates a new order or a new quote has been accepted. This message is also sent when a parked order (for instance, parked pegged order, parked GFA order, other parked order) is injected and added to the order book without receiving an execution.
  - `Order_Cancelled`: Indicates that an order cancel or quote cancel request has been accepted and successfully processed.

  - `Order_Executed`: Indicates that the message is a trade confirmation. Acknowledges the market participants an order or a quote has been partially or fully filled. The execution details must be specified.
  - `Order_Expired`: Indicates that an order or a quote has expired in terms of its time qualifier or due to an execution limit, including the case where an IOC order does not execute aggressively and expires.
  - `Order_Rejected`: Indicates a new order or a new quote has been rejected.
  - `Order_Replaced`: Indicates that an order cancel/replace request, or a new quote update has been accepted and successfully processed.
  - `Order_Suspended`: Indicates that an order or a quote has been parked by the system without adding it into the order book. This message is sent when an incoming stop or stop limit order, an incoming pegged order, or an incoming good-for-auction (GFA) order is put into the parked state.
  - `Order_Restated`: Indicates an order or a quote has been amended by the service desk.

**Quantity Information**

- `LeavesQty`: int. Remaining quantity of the order after the execution report has been sent.
- `DisplayQty`: int. Optional. Displayed quantity after the execution report has been sent. If this information is not available or the exchange does not support iceberg orders, set `DisplayQty = LeavesQty`.

**Additional Variables when `ExecType = Order_Executed`** If the execution report indicates the execution of an order, the message should also include the following variables and missing values are not expected. For non-`Order_Executed Execution_Report` messages, these fields can be N/A.

- `TradeMatchID`: str. An identifier associated to a trade match. This field should link together the pair of outbound messages sent to the two parties of the trade match. When an aggressive new order trades against multiple resting orders at one or more price levels, it is considered as multiple trade matches. Each trade match should receive a different `TradeMatchID` and the matching engine should send trade confirmations to the two parties of each trade match separately.
- `TradeInitiator`: Indicates whether a trade confirmation is for the aggressive or the passive part of the trade.[5] Only the following values are allowed.
  - `Aggressive`: The order traded aggressively against a resting order in the order book.
  - `Passive`: The resting order traded passively against an aggressive order.
  - `Other`: Auction trades and other trades where the aggressive or passive part is unclear.
- `OrderStatus`: Indicates whether the order execution is in part or in full. Only the following values are allowed.
  - `Partial_Fill`: Executes part of `OrderQty`.

[5]In the LSE data, this information is conveyed in a data field called `Container` (whether the order is contained in the order book or in the new order sequence).

  - `Full_Fill`: Executes all of `OrderQty`.

- `OpenAuctionTrade/AuctionTrade`: bool. `True` if the trade confirmation is from an opening auction or auction trade, and `False` otherwise. Users of the package have to construct this variable following Section 3.2.
- `ExecutedPrice`: float. Price at which the trade is executed.
- `ExecutedQty`: int. Quantity executed in the trade.
- `LeavesQty`: int. Remaining quantity after the trade. This is also a required variable for non-`Order_Executed` `Execution_Report` messages.
- `DisplayQty`: int. Optional. Displayed quantity after the trade. If this information is not available or the exchange does not support iceberg orders, set `DisplayQty = LeavesQty`.

### 3.3.7 Additional Variables Required for Outbound `Cancel_Reject` Messages

Exchanges send `Cancel_Reject` messages to indicate a cancel request is unsuccessful. In addition to the required fields in Section 3.3.1, outbound `Cancel_Reject` messages must contain the following additional variable. Missing values are not expected in this variable for `Cancel_Reject` messages. For other types of messages, set the field to N/A.

#### Cancel Reject Reason

- `CancelRejectReason`: Indicates the reason for the cancel reject. This should distinguish too-late-to-cancel from other cancel reject reasons. Only the following values are allowed.
  - `TLTC`: Too-late-to-cancel. Cancel requests rejected due to `TLTC` are counted as failed cancel attempts in race detection.
  - `Other`: Any reason other than `TLTC`.

### 3.3.8 Other Messages

For `Other_Inbound`, `Other_Outbound`, and `Other_Reject` messages, no additional variable is required beyond the variables in Section 3.3.1.

## 3.4 Prepare Reference Data Files

Two reference data files should be prepared as follows.

- `Ticktables.pkl`: dict. {`(date, sym): pandas.DataFrame`}, where each key `(date, sym)` is a tuple of two strings representing the symbol-date, e.g. `('2000-01-01', 'AAPL')`, and each value is a Pandas dataframe containing the corresponding ticktable for that symbol-date. A ticktable maps the price to its tick size. It should have two columns, `tick` for the tick size and `p` for the smallest price of that tick size. Please see the example below. The reason for specifying a ticktable for each symbol-date is that ticktables may vary across symbol-dates in some exchanges (e.g., LSE). If all symbol-dates in the user's data have the same ticktable, the user can simply specify the same ticktable for all symbol-dates in the Python dictionary. We provide an example in `/ArtificialTestData/Sample_Ticktables.pkl`.

| Ticktable | | Meaning |
|---|---|---|
| p | tick | |
| 0 | 0.0001 | - If $0 \leqslant$ price $< 1$, ticksize = 0.0001; |
| 1 | 0.0005 | - If $1 \leqslant$ price $< 5$, ticksize = 0.0005; |
| 5 | 0.0010 | - If price $\geqslant 5$, ticksize = 0.0010. |

- `All_SymDates.csv.gz`: A `.csv.gz` file that lists out all symbol-dates covered in the message data. It should have two columns, `Date` and `Symbol`. Each row should represent a symbol-date with `Symbol` being the underlying security and `Date` being the trading day in the form of YYYY-MM-DD. We provide an example in `/ArtificialTestData/Sample_All_SymDates.csv.gz`.

# 4 Set Race Detection Parameters

This section walks through the user how to set parameters for race detection. The user should read this section alongside Section 3 of ABO, "Defining and Measuring Latency Arbitrage Races".

We first cover how to set a single set of race parameters, e.g., for a baseline scenario (Section 4.1). We then cover how to get the code to execute multiple race scenarios for sensitivity analyses, which we strongly recommend especially given the ambiguity of what the "same time" means in different contexts (Section 4.2). In this section we also cover the idea of a "filter" as distinct from a race parameter, to save computational time. For example, running race detection requiring 2+ participants, but then filtering down to the set of races with 3+ participants. Filtering can be performed in R and is computationally trivial relative to re-running race detection.

## 4.1 How to Set a Single Set of Race Detection Parameters

As a reminder, ABO define a race according to four conceptual criteria:

1. Multiple market participants acting on the same symbol, price, and side.
2. Either a mix of take attempts and cancel attempts (equilibrium emphasized in Budish, Cramton and Shim, 2015), or all take attempts (if the liquidity provider is slow, see Appendix F.1 of ABO).
3. Some succeed, some fail.
4. All at the "same time".

The Python Race Detection code requires users to input parameters for these criteria in the following format. Please see Table 2 for a summary.

**Minimum Number of Participants** The race parameter is called `min_num_participants` and takes an integer value.

ABO baseline set this equal to 2. ABO sensitivities explored values of 3 and 5, primarily using filters as opposed to new runs.

**Number of Takes and Cancels** The race parameters are called `min_num_takes` and `min_num_cancels` and each takes an integer value.

ABO baseline set these equal to 1 and 0, respectively. Together with requiring 2+ participants, this implies that a baseline race could consist of 1+ take and 1+ cancel, or 2+ takes and 0 cancels.

Table 2: Race Detection Parameters

| | | | Values in ABO | |
|---|---|---|---|---|
| **Variable Name** | **Description** | **Allowed Values** | **Baseline** | **Sensitivities** |
| `min_num_participants` | Minimum number of participants. | int. | 2 | 3 and 5 as filter. |
| `min_num_takes` | Minimum number of takes. | int. | 1 | 2 as filter. |
| `min_num_cancels` | Minimum number of cancels. | int. | 0 | 1 as filter. |
| `strict_fail` | See text for details. | bool. | `False` | `True` (as race parameter and as filter). |
| `strict_success` | See text for details. | bool. | `False` | `True` (as filter). |
| `method` | Time horizon method for race detection. | `Info_Horizon` or `Fixed_Horizon`. | `Info_Horizon` | Both methods. |
| `min_reaction_time` | Minimum time to respond to a matching engine update in microseconds. | int, N/A for `Fixed_Horizon`. | 29 | 29 |
| `info_hor_upper_bound` | Upper bound of the information horizon in microseconds. | int, N/A for `Fixed_Horizon`. | 500 | 100, 1000. |
| `len_fixed_hor` | Length of the fixed horizon in microseconds. | int, N/A for `Info_Horizon`. | N/A | 50, 100, 200, 500, 1000, 2000, 3000. |

Notes: This table summarizes the Race Detection parameters the Python Race Detection code requires the users to set. See text of Section 4.1 for detail.

ABO sensitivities explored requiring 1+ cancel and requiring 2+ takes, primarily as filters as opposed to new runs.

**Success and Fail** The user sets two race parameters: `strict_fail` and `strict_success`, each of which takes a Boolean value. For each, `False` uses the baseline definition of Fail and Success in ABO, and `True` uses a more restrictive definition.

`strict_fail`. If set to `False`, then failed IOC orders, failed cancels, and plain-vanilla limit orders that are marketable at the race price but fail to take in the race count as fails. If set to `True`, then only failed IOC orders and failed cancels count as fails. ABO set `strict_fail = False` in the baseline, but set `strict_fail = True` in sensitivities exploring longer time horizons, where limit orders intended to provide post-race liquidity provision are a concern. ABO also performed analysis of strict fails using the filter approach.

`strict_success`. If set to `False`, then a success is achieved by any of the following: a limit order or IOC order that executes a positive quantity at the race price; a cancel request that cancels at least in part at the race price. If set to `True`, then we additionally require that a limit order or IOC order fails to execute at the race price – this implies that the full price level was cleared in the race by either successful takes or successful cancels. Please note that in ABO we did this as a filter which also allowed for a slightly more precise definition of strict success: apart from observing a limit order or IOC order that fails to execute at the race price, we also allow proof that the full price level was cleared to come from observing quantity traded plus quantity canceled equal to (or strictly larger than, if there is hidden quantity) the displayed depth at the start of the race. This criterion is difficult to implement in race detection per se and instead was implemented as a post-race filter.

**Time Horizon** The reader is advised to consult Section 3.4 of ABO. As a reminder, there are two possible methods for defining the race horizon: the "Information Horizon" method and the "Fixed Horizon" method.

The user first specifies which method: this is a binary variable `method` with the choices {`Info_Horizon`, `Fixed_Horizon`}.

If the user has selected the Information Horizon method, the user must specify (i) the minimum reaction time (`min_reaction_time`), and (ii) the upper bound of information horizon (`info_hor_upper_bound`), both in integer number of microseconds. ABO set `min_reaction_time` = 29 microseconds and set `info_hor_upper_bound` = 500 microseconds.Note that the race detection code will compute the other component of the Information Horizon calculation, the Actual Observed Latency of the matching engine at a particular point in time (see Section 3.4 of ABO), automatically.

If the user has selected the Fixed Horizon method, the user must specify the length of the fixed horizon (`len_fixed_hor`) in integer number of microseconds. ABO considered fixed horizons ranging from 50 microseconds to 3000 microseconds (3 milliseconds).

## 4.2 Running Sensitivities – Race Runs and Race Filters

We strongly encourage the researcher to run sensitivity analysis.

From a code implementation perspective, we provide two ways to run sensitivity analyses: multiple runs and filters. A new run of race detection under different parameters will find the complete set of races satisfying those parameters. A filter of race detection finds a subset of races – that is, it takes a baseline set and filters down to a more restrictive subset. Filters are computationally trivial to implement, whereas in our environment each new run took about 9 hours.

In ABO, we used both approaches. Primarily, we used new runs for different time horizons, and filters for most other sensitivities.

**Multiple Race Runs** To perform multiple race runs, users should set up a `.csv` file with their desired race parameters for each race run. Each column of the `.csv` file should represent a race parameter in Table 2 with the variable names as the column headers. Each row should represent a race run to be explored with the parameters for the run specified in each column. We have provided a sample such `.csv` file, where the user wants to run race detection for either 2+ or 3+ participants, and time horizons of Info Horizon, 50, 100, 500, 1000 microseconds. The Info Horizon rows use the time parameters in ABO for illustration.

**Race Filters** It is computationally much easier to analyze alternative race parameters as filters as opposed to by using a fresh run of race detection. We provide a large number of race filters in the R package. Please see Section 9 and Table 6 for detail.

**Remark: Building your own Race Parameters or Race Filters** Building your own additional race filters is straightforward. Often, each filter takes just 1 or a few lines of R code. Please see Section 9 for a detailed example.

Building your own new race parameters will require more programming expertise but should be accessible for moderately experienced programmers. To build your own race parameters, users need to edit the race detection functions in `/PythonCode/LatencyArbitrageAnalysis/RaceDetection/Race_Detection_Functions.py` to account for the new parameters, and edit `/PythonCode/race_detection_main.py` and the `.csv` input file to include the new parameters.

Please read the code comments in the two scripts carefully and make sure that you fully understand the code before making any edits.

## 5 Computational Setup

Before applying the package, users should set up the computational environment and validate the pre-processed message data as described in this section. Readers of this section are assumed to be familiar with Python, R, MacOS/Linux command line interface and cluster or cloud computing.

### 5.1 Computing Environment

Users should set up the computing environment as follows.

**Operating System** MacOS or Linux.[6]

**Software Requirements** Python ($\geq$ 3.6), Pandas ($\geq$ 1.1.0), Numpy ($\geq$ 1.18.0), R ($\geq$ 4.0.0).[7]

**Hardware Requirements** We recommend users to run the package on a multi-core computing cluster with sufficient memory and storage. The package processes symbol-date-level message data in parallel and is computationally intensive. In our own experience, the key computational details were:

- Data size: The London Stock Exchange data used in ABO contain about 15,000 symbol-dates with about 150,000 messages per symbol-date, or a total of 2.2 billion messages. The raw message data has a total size of 70 GB in `.csv.gz` files.
- Hardware: We used a 128-core AWS instance with 1 TB of memory and 1.5 TB of storage to run our baseline analysis and all the sensitivities.
- Running time of the program: In our final code re-run, it took 15 hours to run the Message Data Processing code. This translates into 0.9 hours for one core to process a million messages. For the Race Detection code, it on average took 9 hours per specification, which means we spent 0.5 hours to process a million messages on one core. In general, more inclusive specifications take longer to finish because there are more races to be detected and processed.

Users can estimate the execution time based on the size of the message data and our running time information. The running time is roughly linear in the total number of messages and the speedup is also roughly linear in the number of cores utilized. We also encourage users to test the program on a small sample of symbol-dates to obtain the running time information specific to their environment before running the program at scale. This can help users decide on the computing resources to allocate and make a better trade-off between financial costs and computing efficiency.

### 5.2 File Structure Setup

Users should organize the file structure as follows.

[6] Our code does not support the Windows operating system. We recommend Windows 10 users set up Windows Subsystem for Linux and run the code in the virtual Linux system.

[7] We recommend users first try to run the code using the latest versions of software, and revert to older versions if they encounter errors.

**Download and Unzip the Replication Package** Download the replication package. Create a directory for the replication package and unzip all files in that directory.

**Organize Input Message Data Files** Create a directory for the pre-processed message data and organize the data files as

`path/to/data/dir/{date}/Raw_Msg_Data_{date}_{sym}.csv.gz`

For example, `path/to/data/dir/2000-01-01/Raw_Msg_Data_2000-01-01_ABCD.csv.gz` should be the path to the message data of date 2000-01-01 and symbol ABCD. Please refer to Section 3 for a guide on data pre-processing.

**Organize Input Reference Data Files** Create a directory for the reference data and place the reference data files `Ticktables.pkl` and `All_SymDates.csv` in the directory. Please refer to Section 3 for instructions on reference data preparation.

**Organize Input Race Detection Parameter File** Create a `.csv` file with the race detection parameter for each race run following Section 4.

**Create a Directory for Intermediate Files** Create a directory for intermediate files to be produced by program runs. The directory must be created on a disk with enough free space. In our analysis of the LSE data, the intermediate files for baseline and all sensitivities filled up over 800 GB of space. Intermediate files can be removed only after the analysis has been finished.

**Create a Directory for Log Files** Create a directory for logs to be produced by program runs.

**Create a Directory for Output Files** Create a directory for intermediate files to be produced by program runs. The directory must be created on a disk with enough free space. In our analysis of the LSE data, the output files for baseline and all sensitivities filled up 20 GB of space.

## 5.3 Data Validation

We provide a data validation Python script to check whether the pre-processed message data satisfy the requirements of the package. We strongly recommend users of the package to validate the data before applying the package. To validate the data, users should set up the computing environment and the file structure following Sections 5.1 and 5.2, and work with `/PythonCode/data_validation.py` through the following steps:

- Step 1: Specify the path to pre-processed message data in `data_validation.py`.
- Step 2: Select a small sample of symbol-dates to check. This can be a random sample of several symbol-dates. Organize the symbol-date pairs as a list in `data_validation.py`.
- Step 3: Run `data_validation.py` interactively or in the console. The program will go through each symbol-date in the sample and check if the data meet the requirements. Users will be notified whether the data pass the tests and how to fix it if the data fail the tests.

## 6 Run Message Data Processing Code

After pre-processing the data following Section 3 and setting up the computational environment as in Section 5, the next step is to run the Message Data Processing code. The program will group messages into economically meaningful events, construct the order book based on these events and produce trade-level and symbol-date-level statistical datasets. This section guides the users through how to execute the Message Data Processing code. We also discuss how to monitor the progress of the program and provide useful tools for troubleshooting. Please refer to Section 10 for a description of how message data processing is done "under the hood". To execute the Message Data Processing code, users of the package only need to work with `/PythonCode/process_msg_data_main.py` through the following two steps:

- Step 1: Specify technical parameters in `process_msg_data_main.py`.
- Step 2: Execute `process_msg_data_main.py`.

### 6.1 Specify Technical Parameters

Specify the following technical parameters in `process_msg_data_main.py`. Please refer to the code comments in the script for more detailed instructions.

#### Specify File Paths

- Specify directories for the pre-processed message data and paths to reference data files.
- Specify directories for log files, intermediate files and output files to be generated by the program run.

#### Specify the Multi-Processing Parameter

- Specify number of cores for multi-processing.[8]

#### Specify the Floating Point Protection Parameter

- Specify the maximum decimal scale of the price-related variables. This should be the scale of the smallest tick size in the data. For example, if the prices are quoted as 123.45000 with at most 5 digits after the decimal point, the maximum decimal scale is 5. The program uses this parameter to convert all price-related variables into large integers in the intermediate steps to avoid floating point errors. All variables are converted back to the original price unit in the output.

### 6.2 Execute the Code

Users of the package can execute `process_msg_data_main.py` in the console once the technical parameters are specified. It is recommended to conduct a test run on a few symbol-dates before running the program at scale.

Depending on the computational environment, there can be multiple ways to execute the code. We suggest using the following command:

[8]During data pre-processing, exchange message data is partitioned into discrete chunks of symbol-date level files which allows for parallel processing. The processing time scales almost linearly by the number of cores or CPUs utilized.

```
$ nohup python3 path/to/code/main.py > path/to/main/log/run.log 2>&1&
```

Please change `path/to/code` to the actual path to the `PythonCode` folder accordingly and change `path/to/main/log/run.log` to a path to store the the standard output and the standard error of the program run.

The `nohup ...&` command makes the program immune to hangups. Therefore, the process can continue even after the user disconnects from the computing cluster. The first half of the command, `python3 path/to/code/main.py`, executes a Python script. The second part, `> path/to/main/log/run.log 2>&1`, redirects the standard output and the standard error of the program run to file `path/to/main/log/run.log`.

After the program starts to run, it will automatically save the technical parameters of the run, including the floating point protection parameter and file paths, to `{runtime}.txt` under the `/path/to/logs/TechnicalSpecification`s folder, where `/path/to/logs/` is the directory for log files specified by the users.

### 6.3 Monitor the Progress

Users can monitor the progress of the program run while waiting for the results. The program uses symbol-date level log files to keep track of the progress. The log files are created under `/path/to/logs/MessageDataProcessing_{runtime}/`.

To monitor the progress, users should go through the following steps:

- Step 1: Specify the `runtime` of the program run to be monitored in `/PythonCode/log_monitor.py`. We use the program `runtime` to identify each run and monitor the progress.
- Step 2: Specify the path to log files in `/PythonCode/log_monitor.py`.
- Step 3: Execute the following command interactively or in a console:
  ```
  $ python3 path/to/PythonCode/log_monitor.py
  ```
  A log summary will be printed to the console. This will also save a log summary file to `/path/ to/logs/LogLogSummary/LogSummary_{runtime}.csv`. Users should make sure all symbol-dates have been processed before moving on to the next step.

## 7 Run Race Detection Code

After running the Message Data Processing code, users can execute the Race Detection code to detect races and produce statistics for each race run. Please refer to Section 4 for instructions on how to choose the Race Detection parameters. The output from the Race Detection code contains a race-level statistical dataset for each race detection scenario. This section describes how to run the Race Detection code for one or multiple race runs. We also discuss how to monitor the progress of the program and provide guidance on troubleshooting. Please refer to Section 10 for a description of our race detection algorithm. To execute the Race Detection code, users of the package only need to work with `/PythonCode/race_detection_main.py` through the following three steps:

- Step 1: Specify technical parameters in `race_detection_main.py`.
- Step 2: Specify race detection parameters.
- Step 3: Execute `process_msg_data_main.py`.

## 7.1 Specify Technical Parameters

Specify the following technical parameters in `race_detection_main.py`. Users must use the same file paths and floating point protection parameters as in `process_msg_data_main.py`. Please refer to the code comments in the script for more detailed instructions.

**Specify File Paths**

- Specify paths to reference data files. The reference data files *must* be the same as in `process_msg_data_main.py`.
- Specify directories for log files, intermediate files and output files to be generated by the program run. The directories *must* be the same as in `process_msg_data_main.py`.

**Specify the Multi-Processing Parameter**

- Specify number of cores for multi-processing.

**Specify the Floating Point Protection Parameter**

- Specify the maximum decimal scale of the price-related variables. This variable *must* be the same as in `process_msg_data_main.py`.

## 7.2 Specify Race Parameters

In addition to the technical parameters, users of the package also need to specify the race detection parameters for each race run to explore.

- Create a `.csv` input file for the set of race run to explore. Each row of the table represents the race parameters for a race run and each column is a race detection parameter. Please refer to Section 4 for details. We provide an example in `/ArtificialTestData/Sample_Input_Race_Parameters.csv`.
- Specify the path to the `.csv` input file.

## 7.3 Execute the Code

After specifying the parameters in `race_detection_main.py`, the program is ready to run. It is recommended to conduct a test run on a few symbol-dates before running the package at scale. If the user only runs race detection for one scenario, the program will process the symbol-dates for the scenario in parallel using the number of cores specified by the user. If the user runs multiple scenarios of race detection in a single program run, the program will go through them one at a time, and within each scenario, the symbol-dates will be processed in parallel. The steps to execute the program are exactly the same regardless of the number of race runs. Please refer to Section 6.2 for detail.

### 7.4 Monitor the Progress

Users of the package should follow the same steps as in Section 6.3 to monitor the progress of a program run. If users only run a single race detection scenario, the steps are exactly the same as in Section 6.3. If users run multiple race detection scenarios in a single program run, each scenario will receive a different `runtime`. One can use the `runtime`s to monitor the progress for each scenario separately following Section 6.3. The program will also print out notifications when a scenario has been finished.

## 8 Description of Python Outputs from Race Detection

The outputs of the Python code include a trade-level and a symbol-date-level statistical datasets from the Message Data Processing code, and, for each race detection scenario, a race-level statistical dataset from the Race Detection code. This section describes the output variables in each statistics dataset.

**Trade-Level Statistics** The trade-level dataset provides a set of statistics for each regular-hour non-auction trade. The event where an order trades aggressively against multiple resting orders is considered to have multiple trades. The variables are described in Table 3. Each row in the trade-level data represents a trade and each column is a trade-related statistic.

**Symbol-Date-Level Statistics** The symbol-date-level statistics provide aggregate information on volume, midpoint, and spread for each symbol-date. Each row represents a symbol-date. Each column contains a symbol-date-level aggregate statistic. The variables are described in Table 4.

**Race-Level Statistics** The race-level dataset contains a set of statistics for each race. The program loops through every race in the race record dataset and computes the race statistics as described in Table 5. Each row in the race statistics dataset represents a race. Each column is a race-related statistic.

Table 3: Trade Statistics

| **Variable** | **Description** |
|---|---|
| | *Identifiers* |
| Date, Symbol | str. Date and Symbol of the trade. |
| TradeMatchID | str. Trade identifier unique within each symbol-date. |
| | *Timestamp* |
| MessageTimestamp | timestamp. Timestamp of the trade confirmation to the aggressive part. |
| MessageTimestamp_CP | timestamp. Timestamp of the trade confirmation to the passive part. |
| | *Trade Execution* |
| Side | str. Side of the aggressor. |
| ExecutedQty | float. Executed quantity of the trade. |
| ExecutedPrice | float. Executed price of the trade. |
| ExecutedValue | float. Executed value of the trade, ExecutedPrice multiplied by ExecutedQty. |
| | *Midpoint Prices (T = 1ms, 10ms, 100ms, 1s, 10s, 30s, 60s, 100s)* |
| MidPt | float. Midpoint price at the aggressor's trade confirmation outbound. |
| MidPt_TickSize | Tick size of the midpoint price at the aggressor's trade confirmation outbound. |
| MidPt_Inb | float. The midpoint price at the inbound of the aggressor. |
| MidPt_f_T | float. Midpoint price $T$ after the inbound of the aggressor. |
| | *Effective Spread and Price Impact (T = 1ms, 10ms, 100ms, 1s, 10s, 30s, 60s, 100s)* |
| Eff_Spread_PerShare | float. Per-share effective spread, defined as the difference between the executed price and the midpoint at the inbound of the aggressor in monetary unit. |
| Flag_Spread | bool. Whether the trade is flagged due to missing midpoint, missing inbound message of the aggressor, or negative effective spread. This can happen due to packet loss or zero-to-one-sided market. |
| PriceImpact_PerShare_T | float. Per-share price impact, defined as the difference between the midpoint price at the inbound of the aggressor and the midpoint price $T$ after that inbound. |
| Flag_Spread_T | bool. Whether the trade is flagged because of missing midpoint, missing inbound message of the aggressor, missing midpoint $T$ after the inbound, or negative effective spread. This can happen due to packet loss or zero-to-one-sided market. |
| | *Information on the Aggressive Part in the Trade* |
| UniqueOrderID | str. UniqueOrderID of the aggressor. |
| UserID, FirmID | str. UserID / FirmID of the aggressor. |
| idx | int. The message index of the trade confirmation to the aggressor. |
| UnifiedMessageType | str. UnifiedMessageType of the trade confirmation to the aggressor. |
| Inbound_Missing | bool. Whether the inbound message sent by the aggressor to participate in this trade is missing. |
| UnifiedMessageType_Inb | str. UnifiedMessageType of the aggressor's inbound. N/A if inbound missing. |
| | *Information on the Passive Part in the Trade* |
| UniqueOrderID_CP | str. UniqueOrderID of the passive part. |
| UserID_CP, FirmID_CP | str. UserID / FirmID of the passive part. |
| idx_CP | int. The message index of the execution outbound to the passive part. |
| UnifiedMessageType_CP | str. The UnifiedMessageType of the execution outbound to the passive part. |

Table 4: Symbol-Date Statistics

| **Variable** | **Description** |
|---|---|
| | *Basic information* |
| Date, Symbol | str. Date and symbol. |
| | *Message and Trade Counts* |
| N_Msgs | int. Number of regular trading hour messages of the symbol-date. |
| N_Msgs_Inbound | int. Number of inbound messages of the symbol-date. |
| N_Msgs_Inbound_NBBO | int. Number of inbound messages that are either take attempts at the current BBO or better, or cancel attempts at the current BBO. |
| N_Tr | int. Number of regular hour trades. |
| | *Volume and Depth* |
| Vol_Sh | float. Number of shares traded in all regular-hour trades |
| Vol | float. Total volume traded in all regular-hour trades in monetary unit. |
| Avg_Depth_Sh | float. Time-weighted average depth at the best ask and the best bid prices in shares. |
| Avg_Depth | float. Time-weighted average depth at the best ask and the best bid prices in monetary unit. |
| | *Tick Size* |
| Avg_Tick | float. Time-weighted average tick size in monetary unit. |
| Avg_Tick_bps | float. Time-weighted average tick size in basis points. |
| Min_Tick | float. Minimum tick size for best bid and ask prices in the symbol-date. |
| Max_Tick | float. Maximum tick size for best bid and ask prices in the symbol-date. |
| | *Midpoint and Spread* |
| MidPt_Distance | float. Midpoint distance travelled in monetary unit during the regular trading hours of the symbol-date. |
| MidPt_Distance_Tx | float. Midpoint distance travelled in ticks during the regular trading hours of the symbol-date. |
| Avg_Half_Spr_Time_Weighted | float. Time-weighted average half spread in monetary unit. |
| Avg_Half_Spr_Time_Weighted_Tx | float. Time-weighted average half spread in ticks. |
| Avg_Half_Spr_Time_Weighted_bps | float. Time-weighted average half spread in basis points (half spread over midpoint price). |
| Avg_Half_Spr_Qty_Weighted_Tx | float. Quantity-weighted average half spread in ticks. |
| Avg_Half_Spr_Qty_Weighted_bps | float. Quantity-weighted average half spread in basis points (half spread over midpoint price). |
| Avg_MidPt | float. Time-weighted average midpoint price. |
| Pct_Spr_1Tx | float. Percentage of day in which spread is 1 tick wide. |
| Pct_Spr_2Tx | float. Percentage of day in which spread is 2 tick wide. |
| Time_MidPt_Missing | float. Total time in seconds when midpoint price is missing during regular trading hours. This can happen due to packet loss or zero-to-one-sided market. |
| | *Effective Spread and Price Impact (T = 1ms, 10ms, 100ms, 1s, 10s, 30s, 60s, 100s)* |
| Eff_Spread_bps | float. Value-weighted average effective spread in basis points. |
| Eff_Spread_Paid | float. Total effective spread paid of the symbol-date in monetary unit. |
| PriceImpact_bps_T | float. Value-weighted average price impact in basis points. |
| PriceImpact_Paid_T | float. Total price impact paid of the symbol-date in monetary unit. |

Table 5: Race Statistics

| Variable | Description |
|---|---|
| | *Basic Information* |
| Date, Symbol | str. Date and symbol of the symbol-date |
| SingleLvlRaceID | int. Unique race identifier within each symbol-date. |
| Race_Start_Idx | int. Index of the race starting message in the message dataset of the symbol-date. |
| Side | str. "Bid" or "Ask". Side of the order book on which the race happens. |
| RacePrice | float. Price level of the race. |
| TickSize | float. Tick size of the race price. |
| NearTickBoundary | bool. Whether the race price is within 10 ticks of the boundary between tick levels. |
| Race_Msgs_Idx | list. Indices of the race messages. |
| Race_Horizon | float. Length of the race horizon in $\mu s$. |
| Processing_Time | float. Time from the race starting message to its first outbound in $\mu s$. |
| | *Race Volume and Trade Activity* |
| Num_Trades | int. Number of trades executed at the race price in the race. Note that when a successful take attempt executes against multiple resting orders and receives multiple aggressive execution outbounds, it is counted as multiple trades. |
| Qty_Traded, Qty_Cancelled | int. Number of shares traded / canceled in the race at the race price. |
| Value_Traded, Value_Cancelled | float. Value traded / canceled in the race at the race price. |
| Qty_Active | int. Active quantity in the race at the race price, defined as the sum of Qty_Traded and Qty_Cancelled. |
| Qty_Remaining | int. Number of shares remaining in the book at the race price after the race. |
| Qty_Remaining_Disp | int. Displayed depth in shares remained at the race price after the race. |
| Qty_Remaining_Disp_Pct | int. Percent of displayed depth remained at the race price after the race. |
| | *Race Profits (T = 1ms, 10ms, 100ms, 1s, 10s, 30s, 60s, 100s)* |
| Race_Profits_PerShare_T | float. Per-share profits mark-to-market at $T$, defined as the difference between the race price and the midpoint price $T$ after the race starting point. |
| Race_Profits_PerShare_Tx_T | float. Per-share profits mark-to-market at $T$ in ticks. |
| Race_Profits_PerShare_bps_T | float. Per-share profits mark-to-market at $T$ in basis points, defined as per-share profits over the midpoint price. |
| Race_Profits_DispDepth_T | float. Per-share profits multiplied by displayed depth at the beginning of the race. |
| Race_Profits_TotalDepth_T | float. Per-share profits multiplied by total depth at the beginning of the race. |
| Race_Profits_ActiveQty_T | float. Per-share profits multiplied by active quantity in the race. |
| LossAvoidance_T | float. Total loss avoided in the race, defined by the per-share race profits multiplied by the quantity cancelled in the race. |
| | *Effective Spread and Price Impact (T = 1ms, 10ms, 100ms, 1s, 10s, 30s, 60s, 100s)* |
| Eff_Spread_PerShare_Race | float. Per-share effective spread in the race, defined as the difference between the race price level and the midpoint price at the beginning of the race. |
| Eff_Spread_Paid_Race | float. Total effective spread paid by the race participants, defined by per-share effective spread multiplied by quantity traded in the race at the race price. |
| PriceImpact_PerShare_Race_T | float. Per-share price impact, defined by the difference between the midpoint at the beginning of the race and the midpoint $T$ after the race starting point. |
| PriceImpact_Paid_Race_T | float. Total price impact paid in the race, defined by per-share price impact multiplied by quantity traded in the race at the race price. |

Race Statistics (Continued)

| **Variable** | **Description** |
|---|---|
| | *Order Book Information (T = 1ms, 10ms, 100ms, 1s, 10s, 30s, 60s, 100s)* |
| BestBid, BestAsk | float. Best bid / best ask price at the race starting point. |
| MidPt | float. Midpoint price at the race starting point. |
| BestBidQty, BestAskQty | int. Quantity displayed at best bid / best ask at the race starting point. |
| Depth_Disp, Depth_Total | int. Displayed / total depth at the race price and side at the race starting point. |
| BestBid_f_T, BestAsk_f_T | float. Best bid / best ask price $T$ after the race starting point. |
| MidPt_f_T | float. Midpoint price $T$ time after the race starting point. |
| | *Race Timing* |
| Time_M1 | timestamp. The timestamp of the first message in the race. |
| Time_MLast | timestamp. The timestamp of the last message in the race. |
| Time_M1_F1 | float. Time in $\mu$s from the first race message to the first failed race message. |
| Time_M1_S1 | float. Time in $\mu$s from the first race message to the first successful race message. |
| Time_S1_F1 | float. Time in $\mu$s from the first successful to the first failed race message. |
| Time_S1_F1_Max_0 | float. Max{Time_S1_F1, 0}. |
| | *Race Messages* |
| S1_Type, F1_Type | str. Whether the first successful / failed message is an attempt to take or cancel. |
| S1_UserID, S1_FirmID | str. The UserID / FirmID of the first successful race message. |
| F1_UserID, F1_FirmID | str. The UserID / FirmID of the first failed race message. |
| N_All | int. Number of distinct UserIDs in the race. |
| F_All | int. Number of distinct FirmIDs in the race. |
| M_All | int. Number of race messages in the race. |
| M_Take, M_Canc | int. Number of take / cancel attempts in the race. |
| M_Take_IOC, M_Take_Lim | int. Number of new IOC / new regular limit order take attempts in the race. |
| M_Success_All | int. Number of successful race messages in the race. |
| M_Success_Take | int. Number of successful take attempts in the race. |
| M_Success_Canc | int. Number of successful cancel attempts in the race. |
| M_Fail_All | int. Number of failed race messages in the race. |
| M_Fail_Take | int. Number of failed take attempts in the race. |
| M_Fail_Canc | int. Number of failed cancel attempts in the race. |
| M_Fail_Take_IOC | int. Number of failed IOC take attempts in the race. |
| M_Fail_Take_at_P | int. Number of failed take attempts at the race price in the race. |
| M_QR | int. Number of quote related messages in the race. |
| M_IOC_Expired | int. Number of expired IOC in the race. |
| M_RaceRlvtNoResponse | int. Number of race messages with no outbound. |
| | *Race Participation (T = 50μs, 100μs, 200μs, 500μs, 1ms, 2ms, 3ms)* |
| N_Within_T | int. Number of distinct UserIDs that attempt to take or cancel at the race price within $T$ of the race starting point. |
| F_Within_T | int. Number of distinct FirmIDs that attempt to take or cancel at the race price within $T$ of the race starting point. |
| M_Within_T | int. Number of take and cancel attempts within $T$ of the race starting point. |
| M_Take_Within_T | int. Number of take attempts within $T$ of the race starting point. |
| M_Take_IOC_Within_T | int. Number of new IOC take attempts at the race price within $T$ of the race starting point. |
| M_Take_Lim_Within_T | int. Number of new limit order take attempts at the race price within $T$ of the race starting point. |
| M_Canc_Within_T | int. Number of cancel attempts at the race price within $T$ of the race starting point. |
| M_Success_Within_T | int. Number of successful cancel and take attempts at the race price within $T$ of the race starting point. |
| M_Fail_Within_T | int. Number of failed cancel and take attempts at the race price within $T$ of the race starting point. |

Race Statistics (Continued)

| **Variable** | **Description** |
|---|---|
| | *Pre-Race Order Book Activity (T = 10μs, 50μs, 100μs, 500μs, 1ms)* |
| Stable_Prc_RaceBBO_since_T_PreRace | bool. Whether the race-side BB/BO price is stable continuously since T before the race. |
| RaceBBO_Improved_since_T_PreRace | 1, -1, or 0. 1 indicates the race-side BB/BO price improves. -1 indicates the price worsens. 0 indicates the race-side BB/BO price at T before the race is the same as the beginning of the race (this also includes the case where the price changes first and then changes back). |

## 9 Produce Race Statistics, Figures and Tables using the R Code

The last step to replicate the analysis in Aquilina, Budish and O'Neill is to reproduce the figures and tables in the paper.

- We provide an R package in `/RCode` that allows users to replicate most analysis in the paper. The R code takes the Python package outputs as inputs and reproduces most figures and tables presented in the paper.
- A few figures and tables are difficult to generalize because they make use of data that is very specific to the ABO dataset (e.g., the set of balanced versus aggressive firms in the Top 6, or LSE's trading hours for the Poisson exercise). We provide separate code snippets that we used to produce such figures and tables in `/PythonCode/MiscABOCodeSnippets`. In order to reproduce these results, users will have to adapt the code snippets to their context depending on the specific details of their setting.

This section walks through how to replicate the analysis in ABO using the scripts in `/RCode`. For implementation details, please refer to the notes for figures and tables in the paper and the R code scripts. For tables and figures that cannot be reproduced by the R code, please refer to `/PythonCode/MiscABOCodeSnippets` for the code snippets.

**Replication of the Main Results Section for a Baseline** To replicate the analysis presented in the Main Results section of the paper for a baseline scenario, users of the package should do the following.

- Step 1: Pick a baseline race run and obtain the Python package outputs by running the Race Detection code. Please refer to Section 4 and Section 7 for detail.
- Step 2: Specify the input and output paths in `MainResults.R`.
- Step 3: Execute `MainResults.R`.

**Replication of the Main Results Section for Sensitivity Race Run(s)** To replicate the Main Results section of the paper for sensitivity race run(s), users of the package should do the following.

- Step 1: Pick a set of sensitivity race runs and obtain the Python package outputs by running the Race Detection code. Please refer to Section 4 and Section 7 for detail.

Table 6: Race Filters

| **Description** | **Filter Variable** | **Sample Race Filter Code** |
|---|---|---|
| Requiring 3+ participants | `N_All` | `> race.stats = race.stats[N_All >= 3]` |
| Requiring 5+ participants | `N_All` | `> race.stats = race.stats[N_All >= 5]` |
| Requiring multiple takes | `M_Take` | `> race.stats = race.stats[M_Take >= 2]` |
| Requiring cancels | `M_Canc` | `> race.stats = race.stats[M_Canc >= 1]` |
| Requiring multiple takes and cancels | `M_Take`, `M_Canc` | `> race.stats = race.stats[M_Take >= 2 & M_Canc >= 1]` |
| Requiring 2+ firms | `F_All` | `> race.stats = race.stats[F_All >= 2]` |
| Strict success | `M_Fail_Take_at_P` | `> race.stats = race.stats[M_Fail_Take_at_P >= 2]` |
| Strict fail | `M_Fail_Take_IOC`, `M_Fail_Canc` | `> race.stats = race.stats[M_Fail_Take_IOC + M_Fail_Canc >= 1]` |
| Race-side BB/BO is stable for 100 $\mu s$ pre-race | `Stable_Prc_RaceBBO _since_100us_PreRace` | `> race.stats = race.stats[ Stable_Prc_RaceBBO_since_100us_PreRace == TRUE]` |

Notes: This table summarizes the race filters used in ABO. `race.stats` in the example code is a data.table for some baseline race statistics dataset. A race filter simply selects a subset of a baseline race statistics dataset usually by a line of R code. Readers can follow the examples to build their own filters.

- Step 2: Specify the input and output paths in `MainResults.R` for one race run.
- Step 3: Execute `MainResults.R`. This produces the tables and figures corresponding to the run.
- Step 4: If there are multiple race runs, repeat Steps 2 and 3 for each race run.

**Replication of the Main Results Section for Sensitivity Race Filter(s)** Table 6 summarizes the race filters we used in our analysis. Users can also build their own filters. To replicate the Main Results section of the paper for sensitivity race filter(s), users of the package should do the following:

- Step 1: Pick a baseline race run and obtain the Python package outputs by running the Race Detection code. Please refer to Section 4 and Section 7 for detail.
- Step 2: Specify the input and output paths in `MainResults.R`.
- Step 3: Implement a filter in the "Race Filter" section of `MainResults.R`. This is usually a 1-line R code that subsets the race statistics `data.table` by one or more filter variables in the race statistics dataset. For example, to run a race filter for 3+ participants, one can add the following line in the "Race Filter" section of `MainResults.R`.

  ```
  > race.stats = race.stats[N_All >= 3]
  ```

  where `race.stats` is the `data.table` object of the baseline race statistics dataset, and `N_All` is the race statistic variable for the total number of participants in the race (see Section 8 for a full list of variable names). See Table 6 for more examples.
- Step 4: Execute `MainResults.R`. This produces the tables and figures corresponding to the filter.
- Step 5: If there are multiple race filters, repeat Steps 3 and 4 for each race filter.

**Replication of the Sensitivity Summary Table** To replicate the sensitivity summary table in the Sensitivity Section of the paper, users of the package should do the following.

- Step 1: Pick a set of race runs and race filters. Obtain the Python package outputs for each race run by running the Race Detection code. Please refer to Section 4 and Section 7 for detail.
- Step 2: Obtain the data related to each race run or race filter in the sensitivity table.
  - For each race run, specify the input and output paths in `Sensitivity.R` and execute the code. This produces the column corresponding to the race run in the sensitivity table.
  - For each race filter, specify the input and output paths in `Sensitivity.R` for its baseline, implement the race filter in the "Race Filter" section of `Sensitivity.R` and execute the code. This produces the column corresponding to the race filter in the sensitivity table.

  When investigating a large number of runs/filters, users may want to create a loop for this step, looping over the file paths of each run/filter instead of manually copying and pasting each of them one-at-a-time.
- Step 3: Combine the columns for the race runs and filters of interest into a single sensitivity table.

## 10 Technical Details for the Python Package

This section explains what is going on "under the hood" in the message data processing code and race detection code. This section aims to provide an overview of each of these processes. Please refer to the code files under the `/PythonCode/LatencyArbitrageAnalysis/` folder for more detailed information.

### 10.1 Event Classification

The program first groups inbound and outbound messages into market events inside the exchange. Please refer to `/PythonCode/LatencyArbitrageAnalysis/Classify_Messages.py` for implementation details.

**Input** Pre-processed message data and reference data.

**Output** Processed inbound and outbound messages with messages assigned economic events and price information.

#### Steps

1. Assign each message a unified message type, which combines multiple message attributes to improve readability and simplify the code. The assignment logic is presented in Table 7.

2. Group messages into economic events according to their associated inbound and outbound messages. The program identifies and labels economic events for each order (identified by `UniqueOrderID`) based on the `UnifiedMessageTypes` of the inbounds and the associated outbounds. It generates the field `Event` for the first message in each event to represent the type of this event, and the field `EventNum` for all messages in each event as an event identifier within each order. Quote-related messages are processed separately in a similar manner. Please refer to Table 8 for detail on the assignment logic

## 10.2 Order Book Reconstruction

In this section, the package constructs and maintains the limit order book, as at every message, based on the market events. This is necessary because the inbound and outbound messages do not explicitly contain information about the state of the order book. An important consideration in this section is dealing with missing messages. If the data is perfectly complete, every order book activity can be observed in the data as a market event, and the program can reconstruct the limit order book by sequentially processing all market events. In practice, some order book activities might be missing due to packet loss.[9] To address this, the program searches for inconsistencies in the limit order book and corrects them. Please refer to `/PythonCode/LatencyArbitrageAnalysis/Prep_Order_Book.py` for implementation details of order book reconstruction.

**Input** Processed message data with events assigned to each message, outputs from event classification.

**Output**

- Top-of-book dataset, containing best bid and offer (BBO) prices at each discrete update during regular trading hours, organized by symbol-date.
- Depth-of-book dataset, containing all changes in depth at each price and side at each discrete update during regular trading hours, organized by symbol-date.

**Steps**

1. Construct the order book based on events in the matching engine. Detail in Section 10.2.1
2. Correct the order book based on its internal logic. Detail in Section 10.2.2

### 10.2.1 Order Book Construction

**Identify Book-Updating Events** Only certain events in the matching engine can change the order book. Table 9 provides a list of all order book updating events. The program loops through the events and flags all order book updating events.

[9]Packet loss is the term for when a computer network recording device records strictly less than 100.0% of all activity. Packet loss was extremely low in the LSE dataset but when there is packet loss it is important to correct for it. Otherwise, a misleading state of the order book can persist for the rest of a trading day, which among other concerns can affect any calculation that relies on the midpoint between the best bid and best offer.

Table 7: Unified Message Types

| MessageType | OrderType | TIF | ExecType | OrderStatus | TradeStatus | CancelRejectReason | UnifiedMessageType |
|---|---|---|---|---|---|---|---|
| | | | | *Inbounds* | | | |
| New_Order | Market | - | - | - | - | - | In: New Order (Market) |
| | Limit | Other | - | - | - | - | In: New Order (Limit) |
| | | IOC | - | - | - | - | In: New Order (IOC) |
| | Stop | - | - | - | - | - | In: New Order (Stop) |
| | Stop_Limit | - | - | - | - | - | In: New Order (Stop Limit) |
| | Pegged | - | - | - | - | - | In: New Order (Pegged) |
| | Passive_Only | - | - | - | - | - | In: New Order (Passive Only) |
| New_Quote | - | - | - | - | - | - | In: New Quote |
| Cancel_Request | - | - | - | - | - | - | In: Cancel |
| Cancel_Replace_Request | - | - | - | - | - | - | In: Cancel/Replace |
| Other_Inbound | - | - | - | - | - | - | Other Inbound |
| | | | | *Outbounds* | | | |
| Execution_Report | - | - | Order_Expired | - | - | - | Out: Order Expire |
| | - | - | Order_Cancelled | - | - | - | Out: Cancel Accept |
| | - | - | Order_Replaced | - | - | - | Out: Cancel/Replace Accept |
| | - | - | Order_Suspended | - | - | - | Out: Order Suspended |
| | - | - | Order_Restated | - | - | - | Out: Order Restated |
| | - | - | Order_Accepted | - | - | - | Out: New Order Accept |
| | - | - | Order_Rejected | - | - | - | Out: Order Reject |
| | - | - | Order_Executed | Partial_Fill | Other | - | Out: Partial Fill (Other) |
| | - | - | | | Passive | - | Out: Partial Fill (P) |
| | - | - | | | Aggressive | - | Out: Partial Fill (A) |
| | - | - | | Full_Fill | Other | - | Out: Full Fill (Other) |
| | - | - | | | Passive | - | Out: Full Fill (P) |
| | - | - | | | Aggressive | - | Out: Full Fill (A) |
| Cancel_Reject | - | - | - | - | - | TLTC | Out: Cancel Reject (TLTC) |
| | - | - | - | - | - | Other | Out: Cancel Reject (Other) |
| Other_Reject | - | - | - | - | - | - | Out: Other Reject |
| Other_Outbound | - | - | - | - | - | - | Other Outbound |

**Notes:** Some fields are not populated for certain types of messages. They are marked as "-" in the table. For instance, a `New_Order` message does not have a `CancelRejectReason`.

Table 8: Event Classification

| **Initial Message** | **Followed by** | **Followed by** | **Followed by** | **Event** |
|---|---|---|---|---|
| | | *New Order Events* | | |
| In: New Order | Out: New Order Accept | - | - | New order accepted |
| | Out: Order Expire | - | - | New order expired |
| | Out: Order Suspended | - | - | New order suspended |
| | Out: Order Reject | - | - | New order failed |
| | Out: Other Reject | - | - | New order failed |
| | Out: Full Fill (A) | - | - | New order aggressively executed in full |
| | $\geqslant$ 1 Out: Partial Fill (A) | Out: Full Fill (A) | - | New order aggressively executed in full |
| | | Other or No Messages | - | New order aggressively executed in part |
| | Other or No Messages | - | - | New order no response |
| In: New Quote | Out: New Order Accept | - | - | New quote accepted |
| | Out: Cancel/Replace Accept | - | - | New quote updated |
| | Out: Order Expire | - | - | New quote expired |
| | Out: Order Suspended | - | - | New quote suspended |
| | Out: Order Reject | - | - | New quote failed |
| | Out: Other Reject | - | - | New quote failed |
| | Out: Full Fill (A) | - | - | New quote aggressively executed in full |
| | $\geqslant$ 1 Out: Partial Fill (A) | Out: Full Fill (A) | - | New quote aggressively executed in full |
| | | Other or No Messages | - | New quote aggressively executed in part |
| | Other or No Messages | - | - | New quote no response |
| | | *Existing Order Events* | | |
| In: Cancel | Out: Cancel Accept | - | - | Order or Quote cancel accepted |
| | Out: Cancel Reject (TLTC) | - | - | Order or Quote cancel rejected |
| | Out: Cancel Reject (Other) | - | - | Order or Quote cancel failed |
| | Out: Other Reject | - | - | Order or Quote cancel failed |
| | Other or No Messages | - | - | Order or Quote cancel no response |
| In: Cancel/Replace | Out: Cancel/Replace Accept | Out: Full Fill (A) | - | Cancel/replace aggressively executed in full |
| | | $\geqslant$ 1 Out: Partial Fill (A) | Out: Full Fill (A) | Cancel/replace aggressively executed in full |
| | | | Other or No Messages | Cancel/replace aggressively executed in part |
| | | Other or No Messages | - | Cancel/replace accepted |
| | Out: Cancel Reject (TLTC) | - | - | Cancel/replace rejected |
| | Out: Cancel Reject (Other) | - | - | Cancel/replace failed |
| | Out: Other Reject | - | - | Cancel/replace failed |
| | Other or No Messages | - | - | Cancel/replace no response |
| | | *Outbound-Only Events* | | |
| Out: Partial Fill (P) | - | - | - | Order or Quote passively executed in part |
| Out: Full Fill (P) | - | - | - | Order or Quote passively executed in full |
| Out: Partial Fill (Other) | - | - | - | Order or Quote executed in part (other) |
| Out: Full Fill (Other) | - | - | - | Order or Quote executed in full (other) |
| Other or No Messages | - | - | - | Other outbound activity |

**Notes:** "In: New Order" includes all types of new order messages (Market, Limit, IOC, Stop, Stop Limit, Pegged and Passive Only). Whether the event is an order-related or a quote-related event (e.g. Cancel request accepted or Quote cancel accepted) depends on the field `QuoteRelated`. `Other_Inbound` and `Other_Outbound` are omitted.

**Construct the Order Book** The program goes through book-updating events and updates the order book accordingly as at the outbound of these events. Updating the book on outbounds allows us to handle both the core message types (IOC, Limit Orders, etc.) and other message types (Stop Limit orders, Pegged orders, etc.) correctly. [10]

- For order executed events, quantity is removed from the order book at the earliest message of the two outbound messages of the trade (one for each side of the execution). In cases where the aggressive party in the trade posts the unexecuted quantity of their order to the book after partial fill(s), the quantity is added as at the time of the last outbound to the aggressive party, or the passive counterpart of this message, which ever shows up earlier in the data.
- For other book-updating events, the book is updated when the outbound message of the event is observed. Examples include cancel accepted, cancel/replace accepted, etc.

**Notes on Order Priority** In many exchanges, orders resting in the book are executed based on certain priority rules. For example, limit orders at the same price level will be executed first if it arrives earlier. Our order book construction code focuses on the quantity at each price level and does not track the priority of the orders. This is because the exchange messages allow us to figure out which orders are executed if there is a trade.

**Notes on Orders Valid for Multiple Trading Days** In many exchanges, orders that remain in the order book at the end of the trading day are purged by the exchange. Some exchanges allow orders to stay in the order book for more than one trading day. Our package is designed to process discrete symbol-dates in parallel. As a result, the package does not retain orders still active at the end of a trading day to subsequent days. In subsequent trading days, these orders will be treated as missing inbound new order messages due to packet loss. Not carrying these orders over over from previous trading days is unlikely to cause an issue because:

- We can observe the effect of these orders through trades and cancel requests.
- Our order book correction steps described in Section 10.2.2 resolves the impact of these missing orders.

### 10.2.2 Order Book Corrections

As discussed above, packet loss, even if rare, can cause the computed state of the order book to differ from the actual state of the order book, which can in turn cause a confusing or misleading computed state of the order book to persist for an entire trading day. The code contains the following order book corrections to account for this issue:

**Correction at the opening auction** At the end of the opening auction, the order book should be fully uncrossed (wherein there are no buy orders that are priced to match with sell orders, or the best bid is below the best ask price). In practice, the book might remain crossed if some execution

[10]For example, when a user submits a new stop limit order, we first see an inbound message submitting the new order followed by an outbound message informing order suspension. Once the stop price is reached, we see another outbound message confirming that the order is posted to the book. When the order is executed, we observe outbound message(s) of trade confirmation. By updating the book on outbounds, we can correctly update the state of the book throughout the lifetime of this order. For more details, please refer to the Python script `/PythonCode/LatencyArbitrageAnalysis/Prep_Order_Book.py`.

Table 9: Book Updating Events

| **Event First Message** | **Event** | **Updating** | **Type of Update** |
|---|---|---|---|
| | *New Order Events* | | |
| In: New Order (Limit, or Stop Limit) | New order accepted | Y | Add depth |
| | New order aggressively executed in part | Y | Add depth |
| | New order aggressively executed in full | N | None |
| In: New Order (IOC) | New order expired | N | None |
| | New order aggressively executed in part | N | None |
| | New order aggressively executed in full | N | None |
| In: New Order (Passive Only) | New order accepted | Y | Add depth |
| | New order expired | N | None |
| In: New Order (Market) | New order aggressively executed in full | N | None |
| In: New Order (Stop) | New order suspended | N | None |
| In: New Order (Pegged) | New order suspended | N | None |
| In: New Quote | New quote accepted | Y | Add depth |
| | New quote updated | Y | Add and remove depth |
| | New quote aggressively executed in part | Y | Add and remove depth |
| | New quote aggressively executed in full | Y | Remove depth |
| Any New Order/Quote Inbound | New order, or quote failed | N | None |
| | *Existing Order Events* | | |
| In: Cancel | Order, or Quote cancel accepted | Y | Remove depth |
| | Order, or Quote cancel rejected, or failed | N | None |
| In: Cancel/Replace | Cancel/replace accepted | Y | Remove depth |
| | Cancel/replace rejected, or failed | N | None |
| | Cancel/replace aggressively executed in part | Y | Add and remove depth |
| | Cancel/replace aggressively executed in full | Y | Remove depth |
| Out: Partial Fill (P) | Order, or Quote passively executed in part | Y | Remove depth |
| Out: Full Fill (P) | Order, or Quote passively executed in full | Y | Remove depth |
| | *Other Outbound Activity Events* | | |
| Out: Partial Fill (Other) | Order executed in part (other) | N | None |
| Out: Full Fill (Other) | Order executed in full (other) | N | None |
| Out: Partial Fill (A) | Other outbound activity | Y | Remove depth |
| Out: Full Fill (A) | Other outbound activity | Y | Remove depth |
| Out: New Order Accept | Other outbound activity | Y | Add depth |
| Out: Order Suspend | Other outbound activity | Y | Remove depth |
| Out: Order Expire | Other outbound activity | Y | Remove depth |
| Out: Cancel Accept | Other outbound activity | Y | Remove depth |
| Out: Cancel/Replace Accept | Other outbound activity | Y | Add and remove depth |

**Notes:** All no response events are not book updating and are omitted in this table. Other outbound activity events include some nontypical events such as an event of one message "Out: Full Fill (A)". These events can result from packet loss and they are considered in order book construction.

outbounds are missing in the data due to packet loss. The program resolves this issue as follows at the last outbound of the opening auction.

- Remove all orders to buy at prices strictly above the auction price and all orders to sell at prices strictly below the auction price.
- For orders to buy and sell at the auction price, remove all the orders on the side with a smaller total quantity since this side is more likely to be the result of packet loss.

**Correction on trades**

- When an outbound trade confirmation is observed, remove all orders to buy at prices strictly greater than the execution price and all orders to sell at strictly less than the execution price.
- When an aggressive full fill is observed, remove all liquidity at the execution price on the same side of the book.
- When a partial fill is observed, remove all liquidity at the execution price on the opposite side of the book.

**Correction on new orders** When a new order accepted, cancel/replace accepted, or new IOC expired occurs without a corresponding trading message being observed,

- Remove all orders on the ask side at prices weakly lower than the order price if the message is on the bid side.
- Remove all orders on the bid side at prices weakly greater than the order price if the message is on the ask side.

**Remark: Audit Statistics Re Order Book Corrections in ABO** For the analysis in ABO, we produced audit statistics on both (i) the magnitude of the corrections, and (ii) the % of time that our order book state performs as expected. In a high-volume symbol (Vodafone) on a typical-volume day (2015-09-23), we are correct 99.95% of the time about whether a new limit order should trade against the book versus post to the book. On the highest-volume day of our sample (2015-08-24), which contained a mini-flash-crash and was noticeably an outlier on many measures relative to the other days, we are correct in this manner 99.82% of the time. Also reassuring, most of the time that we had to execute an order book correction, the correction concerned just a single level of the book, and involved a number of shares that was less than the mean depth at the top level of the book. These figures are consistent with the LSE's comments to us that the occurrence of packet loss is extremely low.

## 10.3 Trading and Order Book Statistics

After classifying messages into economic events and constructing the order book, the program will produce summary statistics for trading and order book. This includes a trade-level statistical dataset and a symbol-date-level statistical dataset. For implementation details, please see `/PythonCode/LatencyArbitrageAnalysis/Trading_and_Order_Book_Stats.py`. Please refer to Section 8 for a complete list of output variables.

**Input**

- Processed message data with events assigned to each message, an output from event classification.
- Top-of-book dataset, an output from order book reconstruction.
- Depth-of-book dataset, an output from order book reconstruction.

**Output**

- Trade-level statistical data. Each row represents a trade. Each column is a statistic related to the trade. The statistics are independent of the definition of a race.
- Symbol-date-level statistical data. Each row represents a symbol-date. Each column is an aggregate statistic on volume, midpoint, and spread for the symbol-date. The statistics are independent of the definition of a race.

**Steps**

1. Compute trade-level statistics for non-auction trades in regular hours. A trade is identified as a discrete aggressive execution outbound message, irrespective of whether a partial or full fill. Statistics on trade execution, timing, spread are computed for each trade. Please refer to Section 8 for a complete list of output variables.
2. Compute symbol-date-level statistics. For each symbol-date, the program generates a set of aggregate statistics on volume, midpoint, and spread. Please refer to Section 8 for a complete list of output variables.

## 10.4 Race Detection

Race detection is the core step of the package. To detect races, the program first identifies the complete set of potential race starting points (discrete inbound messages) and price levels. For each potential race starting point and price level, the program then checks whether the race criteria are satisfied. Please refer to `Race_Detection_and_Statistics.py` and the `RaceDetection` folder under `/PythonCode/LatencyArbitrageAnalysis/` for implementation details.

**Input**

- Processed message data with events assigned to each message, an output from event classification. Note that the messages have already been sorted by timestamps in the pre-processing steps.
- Top-of-book dataset, an output from order book reconstruction.
- Depth-of-book dataset, an output from order book reconstruction.
- Race detection parameters which specify the race criteria.

**Output**

- Record-of-race dataset, basic identifying information for each race detected.

**Steps**

1. Find potential starting point and price levels of races as defined in Section 10.4.1.
2. Determine whether there is a race at each potential starting point and price level. Detail in Section 10.4.2.

### 10.4.1 Find Potential Race Starting Points and Price Levels

**Potential Race Starting Points** A potential race starting point is an inbound message that submits a marketable order to take liquidity (that is potentially stale), or attempts to cancel an existing order (that is potentially stale). The program finds potential race starting points in the following steps.

- Loop through inbound messages and identify orders that are marketable at the current BBO: that is, orders to buy at the best ask or higher, or orders to sell at the best bid or lower. This includes new IOC orders and limit orders.[11]
- Loop through inbound messages and identify attempts to cancel an existing order. This also includes requests to cancel cancel and replace an order with a worse price.[12]

**Potential Race Prices** For each potential race starting point identified, potential race prices are defined as follows:

- For attempts to take liquidity at the current BBO or better, any price between the current BBO and the order price of the message is considered a potential race price.
- For attempts to cancel an existing order, the price of the quote the message is attempting to cancel is considered the potential race price.

### 10.4.2 Check for Races

For each potential race starting point $m$ and potential race price $P$, the program checks against the race criteria to determine whether there is a race. The program loops over all potential race starting points and determines whether a race is triggered in the following way:

- Given a potential race starting point $m$, calculate its potential race horizon as follows.
  - If `method` = "`Info_Horizon`", the program uses the information horizon as its race horizon. The information horizon is defined as the sum of the actual observed latency and the minimum reaction time, capped at `info_hor_upper_bound`. The actual observed latency is measured as the time difference between $m$ and its first outbound. The minimum reaction time is specified by `min_reaction_time`.

[11]In this initial step of finding potential race starting points, other order types are included as well if they can be immediately executable at the current BBO: market orders, fill-or-kill orders, new quotes (which could cross the BBO and execute), and cancel-and-replaces with more aggressive prices (which could cross the BBO and execute). That said, in the ABO data, over 90% of marketable orders in races are immediate-or-cancels, and nearly all of the remainder are plain-vanilla limit orders (see Section 4.2 of ABO, Number of Takes and Cancels). Stop, Stop Limit, Pegged and Passive Only orders are ignored because they are unlikely to be used in races.

[12]In this initial step of finding potential race starting points, the code does not check whether an attempt to cancel is at or near the current BBO. Practically, however, only attempts to cancel that are at or near the current BBO will be parts of races.

  - If `method = "Fixed_Horizon"`, the program uses a fixed race horizon specified by `len_fixed_hor` in microseconds.

- Loop over the potential race prices at *m*. For each potential race price *P*, go through the subsequent inbound messages of *m* and collect all attempts to take at *P* and all attempts to cancel at *P* within the potential race horizon of *m*. The first message *m* is also included in the collection.
  - An attempt to take at *P* is an inbound message that is a limit or IOC order that is priced aggressively with respect to *P* (i.e., an offer to buy weakly higher than *P* or an offer to sell weakly lower than *P*).[13]
  - An attempt to cancel at *P* is an inbound message that attempts to cancel an existing order at *P*.

- Determine whether the set of messages collected consist of a race. The set of messages is a race if *all* of the following conditions are met:[14]
  1. The messages are sent by at least `min_num_participants` distinct `UserID`s.
  2. At least `min_num_takes` messages are attempts to take at *P* or a better price, and at least `min_num_cancels` messages are attempts to cancel at *P*.
  3. At least one of the messages is successful.
     (a) An attempt to take at *P* is successful if it trades a positive quantity at *P* or a better price.
     (b) An attempt to cancel at *P* is successful if it cancels a positive quantity at *P*.
     (c) If `strict_success = True`, the program additionally requires that at least one of the messages is a limit order or IOC order that fails to execute at *P*. This implies that the full price level is cleared in the race by either successful takes or successful cancels.
  4. At least one of the messages fails.
     (a) An attempt to cancel at *P* is considered a fail if it does not cancel a positive quantity at *P*. That is, it receives a Too Late To Cancel message in reply.
     (b) An attempt to take at *P* is considered a fail if it does not execute a positive quantity at *P* (or a strictly better price, meaning a lower price if seeking to buy or a higher price if seeking to sell). If `strict_fail = True`, the program only allows immediate-or-cancel orders to satisfy this condition (i.e., plain vanilla limit orders cannot count as fails).

- If a race is detected, record its timestamp and message indices in the record-of-race dataset, and stop detecting races at *P* within the race horizon of *m*. Race detection at *P* will continue after the race horizon of *m*. This prevents any two races at the same price level from overlapping.

- After checking for races at *P*, go to the next potential race price *P'*. If all potential race prices at *m* have been exhausted, go to the next potential race starting point *m*'.

[13] As noted above, some other order types are included for the purpose of race detection if they can be immediately executable at the BBO. Stop, Stop Limit, Pegged and Passive Only orders are ignored because they are unlikely to appear in races.

[14] The reader should consult Section 3 of ABO alongside this piece of code for detailed explanations of each race criterion.

## 10.5 Calculating Race Statistics

As the last step, the program computes a set of race-level statistics for all races. The program loops through every race in the record-of-race dataset and computes a set of statistics for each race including race volume, race profits, spread, race participation, etc. For implementation details, please refer to `Race_Detection_and_Statistics.py` and the `RaceDetection` folder under `/PythonCode/LatencyArbitrageAnalysis/`. Please refer to Section 8 for a complete list of output variables.

### Input

- Processed message data with events assigned to each message, an output from event classification.
- Top-of-book dataset, an output from order book reconstruction.
- Depth-of-book dataset, an output from order book reconstruction.
- Record-of-race dataset, an output from race detection.

### Output

- Race-level statistical data. Each row represents a race, and each column is a statistic related to the race. Please refer to Section 8 for a complete list of output variables.