meatpy

Release 0.0.3

Sep 20, 2023

Contents:

The Market Empirical Analysis Toolbox for Python (MeatPy) is a Python module aimed at researchers studying high-frequency market data feeds, focusing on full limit order book data. MeatPy aims to provide a set of standard, user-friendly open-source tools to lower the bar to entry into advanced empirical market microstructure research.

MeatPy's latest documentation is available at https://meatpy.readthedocs.io/en/latest/ and the source code is available on GitHub.

MeatPy is a work in progress, and a lot remains to be done before we reach version 1.0. As of the current version, MeatPy only supports Nasdaq ITCH 5.0 files.

CHAPTER 1

Installation

You can install MeatPy using `pip install meatpy`.

1.1 Overview of MeatPy

The Market Exchange Analysis Toolbox for Python (MeatPy) is a Python module aimed at researchers studying high-frequency market data feeds, focusing on full limit order book data. MeatPy aims to provide a set of standard, user-friendly open-source tools to lower the bar to entry into advanced empirical market microstructure research. The documentation is available on Read the Docs and the source code is available on GitHub.

The three building blocks of the MeatPy workflow are the parser, the market processor, and the recorders.

1.1.1 Parser

The parser is in charge of reading the data files to extract messages. It can be used to convert message files in a different format, to split full market data files into symbol-specific files and to feed messages to the market processor.

MeatPy implements a parser for Nasdaq ITCH 5.0:

1. `ITCH50MessageParser`

Reads and writes Nasdaq ITCH 5.0 binary files. It can split full market data files into symbol-specific files and read messages to feed to the market processor. For more details on messages, see the Nasdaq TotalView-ITCH 5.0 Specification.

1.1.2 Market Processor

The market processor is the engine that allows processing for one symbol/day. It receives messages one at a time and replays the day's events, keeping track of the limit order book's state.

MeatPy implements a market processor for Nasdaq ITCH 5.0:

1. `ITCH50MarketProcessor`

Handles messages according to the Nasdaq ITCH 5.0 specification.

1.1.3 Recorders

The market processor does not generate any output. Instead, attached recorders are used to record the desired output. This allows for efficient processing and flexibility in what data is generated.

Once a recorder is attached to a market processor, it can react to events (e.g., trade messages, trading status changes, limit order book updates, etc.) and record the desired data. Some recorders can be set to record only during specific market states (e.g., regular trading) or at specific timestamps (e.g. every one minute).

MeatPy implements six types of recorders:

1. `SpotMeasuresRecorder`

Records certain metrics, such as best quotes and Kyle's lambda.

2. `LOBRecorder`

Records snapshots of the limit order book. It supports parameters for limiting the recorder depth and level of detail.

3. `ITCH50TopOfBookMessageRecorder`

Records all messages that affect the top of the order book.

4. `ITCH50OrderEventRecorder`

Records order-related events, such as order additions, order executions, order cancelations, and order replacements.

5. `ITCH50ExecTradeRecorder`

Records executions and trades, including information about the executed limit order.

6. `ITCH50OFIRecorder`

Records the order flow imbalance.

> See Equations (4) and (10) of Cont, R., et al. (2013). "The Price Impact of Order Book Events." Journal of Financial Econometrics 12(1): 47-88.
>
> The recorder follows equation (10) but accounts for trades against hidden orders as well.

1.2 Getting Started

This section presents sample code for common use cases. The suggested workflow is the following:

- `Step0_ExtractSymbols.py` Extracting symbols from a Nasdaq ITCH file.
- `Step1_Parsing.py` Splitting Nasdaq ITCH files into per symbol individual ITCH files.
- `Step2_Processing.py` Process individual symbols.

1.2.1 Data

Sample Nasdaq ITCH files are available at ftp://emi.nasdaq.com/ITCH/. The following examples are based on the file `20190530.BX_ITCH_50.gz`, which contains Nasdaq BX messages from May 30, 2019. The message format for Nasdaq BX is the same as for the main Nasdaq exchange, but the files are smaller and thus more suited for examples.

Sample code files are located in the `samples` directory. The sample data file should be placed in the `sample_data` directory.

1.2.2 Extracting symbols from a Nasdaq ITCH file

This program uses a `ITCH50MessageParser` to parse an individual Nasdaq ITCH 5.0 file and extract all the traded symbols from stock directory messages. This can be useful to list all the symbols that are present in the file.

```python
"""Sample code for extracting the symbols from a ITCH 5.0"""

import gzip
from meatpy.itch50 import ITCH50MessageParser

sample_dir = '../sample_data/'

fn = '20190530.BX_ITCH_50.gz'
outfn = 'Symbols_20190530_BX_ITCH.txt'

# Initialize the parser
parser = ITCH50MessageParser()

# Keep only the Stock Directory Messages
parser.keep_messages_types = b'R'

# Stock Directory Messages are also copied in a separate list by the parser,
# so we can avoid keeping track of stock-specific messages, which saves
# memory.
parser.skip_stock_messages = True

# Parse the raw compressed ITCH 5.0 file.
# Note: This can take a while. If we were to run this on many files,
# it might make sense to modify the message parser to stop after a given
# number of messages since the stock directory messages are at the
# start of the day.
with gzip.open(sample_dir + fn, 'rb') as itch_file:
    parser.parse_file(itch_file)

# We only care about symbols, so let's extract those.
symbols = [x.stock for x in parser.stock_directory]

# Output the list of symbols, one per row.
lines = [x.decode() + '\n' for x in symbols]
with open(sample_dir + outfn, 'w') as out_file:
    out_file.writelines(lines)
```

The first few lines of the output file look like this:

Table 1: Symbols_20190530_BX_ITCH.txt

A
AA
AAAU
AABA
AAC
AADR
AAL
AAMC
AAME
AAN
AAOI
AAON
AAP
AAPL
AAT

1.2.3 Splitting Nasdaq ITCH files

This program uses a `ITCH50MessageParser` to parse an individual Nasdaq ITCH 5.0 file and split the aggregate daily Nasdaq file into symbol-specific valid Nasdaq ICTH 5.0 files for the desired symbols. The resulting files are smaller, so it is more efficient for archival if only some symbols are needed. This makes parallel processing much easier because symbol-specific files can be processed in parallel on one computer using multiple cores or on computing clusters. Reading and writing ITCH files in binary format is also much faster than using human-readable formats such as CSV.

```
"""Sample code for parsing a ITCH 5.0 file"""

import gzip
from datetime import datetime
from meatpy.itch50 import ITCH50MessageParser

sample_dir = '../sample_data/'

date = datetime(2019, 5, 30)
dt_str = date.strftime('%Y%m%d')

fn = dt_str + '.BX_ITCH_50.gz'

# List of stocks to extract, in byte arrays.
# Note that all Nasdaq ITCH symbols are 8 bytes long (ticker + whitespace)
stocks = [b'AAPL    ', b'ALGN    ']

# Initialize the parser
parser = ITCH50MessageParser()

# Setup parser to minimize memory use. A smaller buffer uses less memory
# by writes more often to disk, which slows down the process.
parser.message_buffer = 500  # Per stock buffer size (in # of messages)
parser.global_write_trigger = 10000  # Check if buffers exceeded

# We only want our stocks. This is optional, by default MeatPy
# extracts all stocks.
```

(continues on next page)

(continued from previous page)

```
parser.stocks = stocks

# Set the output dir for stock files
# Using a file prefix is good practice for dating the files.
# It also avoids clashes with reserved filenames on Windows, such
# as 'PRN'.
parser.output_prefix = sample_dir + 'BX_ITCH_' + dt_str + '_'

# Parse the raw compressed ITCH 5.0 file.
with gzip.open(sample_dir + fn, 'rb') as itch_file:
    parser.parse_file(itch_file, write=True)
```

1.2.4 Processing Nasdaq ITCH files

This program processes a symbol-specific ICTH 5.0 file to extract limit order book snapshots and data related to order book events and executions.

While MeatPy does not have built-in multiprocessing support, multiple instances of this code can be executed in parallel using Python's `multiprocessing` package.

```
"""Sample code for processing ITCH 5.0 file and extracting measures"""
import gzip
import sys
from datetime import datetime
from meatpy.itch50 import ITCH50MessageParser, ITCH50MarketProcessor, \
ITCH50ExecTradeRecorder, ITCH50OrderEventRecorder
from meatpy.event_handlers import LOBRecorder
from meatpy import ExecutionPriorityException, \
VolumeInconsistencyException, ExecutionPriorityExceptionList

sample_dir = '../sample_data/'

parser = ITCH50MessageParser()

with open(sample_dir + 'BX_ITCH_20190530_ALGN.txt', 'rb') as itch_file:
    parser.parse_file(itch_file)

# There should only be one stock in the file.
stocks = [s for s in parser.stock_messages]
stock = stocks[0]

processor = ITCH50MarketProcessor(stock, datetime(2019, 5, 30))
# Create a LOB recorder. By default, it records all LOB events.
# That means we will have an event everytime an order enters or exits the book.
# Create one to record the top of book (level 1), all events
tob_recorder = LOBRecorder()
# We only care about the top of book
tob_recorder.max_depth = 1

# We create another one to record 1-minute snapshots on the book
lob_recorder = LOBRecorder()
# We only want every minute. Nasdaq timestamps are in nanoseconds since 12am.
seconds_range = [x * 1000000000 for x in range(34130, 57730+1, 60)]
seconds_range.sort(reverse=True)
lob_recorder.record_timestamps = seconds_range
```

(continues on next page)

(continued from previous page)

```python
# Create the trade recorder
trade_recorder = ITCH50ExecTradeRecorder()
# Create the order event recorder
order_recorder = ITCH50OrderEventRecorder()

# Attach the recorders to the processor
processor.handlers.append(tob_recorder)
processor.handlers.append(lob_recorder)
processor.handlers.append(trade_recorder)
processor.handlers.append(order_recorder)

# Process the messages
for m in parser.stock_messages[stock]:
    try:
        processor.process_message(m)
    except ExecutionPriorityException as e:
        sys.stderr.write('Warning,' + stock.decode() +
                         ',' + e.args[0] + ',"' + e.args[1] + ' (' +
                         str(e[2]) + ')"\n')
    except VolumeInconsistencyException as e:
        sys.stderr.write('Warning,' + stock.decode() +
                         ',' + e[0] + ',"' + e[1] + '\n')
    except ExecutionPriorityExceptionList as eList:
        for e in eList.args[1]:
            sys.stderr.write('Warning,' + stock.decode() +
                             ',' + e.args[0] + ',"' + e.args[1] + ' (' +
                             str(e.args[2]) + ')"\n')

# Output files
with gzip.open(sample_dir + 'tob.csv.gz', 'w') as outfile:
    tob_recorder.write_csv(outfile, collapse_orders=True)
with gzip.open(sample_dir + 'lob.csv.gz', 'w') as outfile:
    lob_recorder.write_csv(outfile, collapse_orders=False)
with gzip.open(sample_dir + 'tr.csv.gz', 'w') as outfile:
    trade_recorder.write_csv(outfile)
with gzip.open(sample_dir + 'or.csv.gz', 'w') as outfile:
    order_recorder.write_csv(outfile)
```

The first few lines of each output file look like this:

Table 2: lob.csv (lob recorder, full book)

Timestamp	Type	Level	Price	Order ID	Volume	Order Timestamp
34130000000000	Ask	1	3010100	656801	400	34052727737823
34130000000000	Bid	1	2942000	669949	200	34085725901583
34190000000000	Ask	1	3010100	656801	400	34052727737823
34190000000000	Bid	1	2942000	669949	200	34085725901583
34250000000000	Ask	1	3010100	656801	400	34052727737823
34250000000000	Ask	2	3040000	845161	30	34202154392271
34250000000000	Ask	3	3142000	783433	100	34200414784684
34250000000000	Ask	4	3471000	774589	100	34200317659936
34250000000000	Bid	1	2958900	837589	200	34201826545548
34250000000000	Bid	2	2829900	783425	100	34200414765177
34250000000000	Bid	3	2502200	774585	100	34200317644668
34310000000000	Ask	1	3040000	845161	30	34202154392271
34310000000000	Ask	2	3142000	783433	100	34200414784684
34310000000000	Ask	3	3471000	774589	100	34200317659936

Table 3: or.csv (order event recorder)

Timestamp	MessageType	BuySellIndicator	Price	Volume	OrderID	NewOrderID	AskPrice	AskSize	BidPrice	BidSize
34052727727406	AddOrder	B	2954000	400	656797		None	None	None	None
34052727737823	AddOrder	S	3010100	400	656801		None	None	2954000	400
34084825837342	OrderDelete				656797		3010100	400	2954000	400
34085725901583	AddOrder	B	2942000	200	669949		3010100	400	None	None
34200317644668	AddOrderMPID	B	2502200	100	774585		3010100	400	2942000	200
34200317659936	AddOrderMPID	S	3471000	100	774589		3010100	400	2942000	200
34200414765177	AddOrderMPID	B	2829900	100	783425		3010100	400	2942000	200
34200414784684	AddOrderMPID	S	3142000	100	783433		3010100	400	2942000	200
34200777056480	OrderDelete				669949		3010100	400	2942000	200
34201826545548	AddOrder	B	2958900	200	837589		3010100	400	2829900	100
34202154392271	AddOrder	S	3040000	30	845161		3010100	400	2958900	200
34272871221455	OrderDelete				837589		3010100	400	2958900	200
34272871225602	OrderDelete				656801		3010100	400	2829900	100
34471992679916	AddOrder	B	2992600	3	2939241		3040000	30	2829900	100

Table 4: tob.csv (lob recorder, top of book only)

Timestamp	Type	Level	Price	Volume	N Orders
34052727727406	Bid	1	2954000	400	1
34052727737823	Ask	1	3010100	400	1
34052727737823	Bid	1	2954000	400	1
34084825837342	Ask	1	3010100	400	1
34085725901583	Ask	1	3010100	400	1
34085725901583	Bid	1	2942000	200	1
34200317644668	Ask	1	3010100	400	1
34200317644668	Bid	1	2942000	200	1
34200317659936	Ask	1	3010100	400	1
34200317659936	Bid	1	2942000	200	1
34200414765177	Ask	1	3010100	400	1
34200414765177	Bid	1	2942000	200	1
34200414784684	Ask	1	3010100	400	1
34200414784684	Bid	1	2942000	200	1

Table 5: tr.csv (trade recorder)

Timestamp	MessageType	Queue	Price	Volume	OrderID	OrderTimestamp
34703242608927	Exec	Ask	3008000	31	4426365	34692733984765
34703242648024	Exec	Ask	3008000	60	4426365	34692733984765
34729950074550	Exec	Bid	3017000	4	4635649	34729950038510
35149267156862	ExecHid	Bid	3025000	100		
35290544186992	ExecHid	Bid	3026200	100		
35290544190321	ExecHid	Bid	3026200	100		
35290544574482	ExecHid	Bid	3026200	100		
35401142766421	ExecHid	Bid	3027100	100		
35518105042925	ExecHid	Bid	3035200	75		
35518105042925	ExecHid	Bid	3035000	25		
35574799640110	ExecHid	Bid	3032500	75		
35574799640110	ExecHid	Bid	3032500	25		
35703478335449	Exec	Bid	3024500	17	7939453	35327271048191
35778872267499	ExecHid	Bid	3023500	100		

CHAPTER 2

Credits

MeatPy was created by Vincent Grégoire (HEC Montréal) and Charles Martineau (University of Toronto). Javad YaAli provided excellent research assistance.