Real Time Embedded Systems – Final Project (2025)

Name: Chouklis Athanasios // AEM: 10396 // GitHub Repository

The goal of this project is to write a program that collects and processes cryptocurrency trade data from OKX in real time, using the platform's public WebSocket channels.

As the program is meant to run on low-power devices (in this case, a Raspberry Pi Zero 2 W) for long periods of time, it must make efficient use of system resources. Consequently, the language chosen was C and care was taken to ensure multi-threading is safely utilized, via the **pthread.h** library. The **libwebsockets.h** library was used to interface with OKX's WebSocket API. The program is comprised of 4 threads:

1) The main thread takes all the steps necessary to initiate the program's working data (file pointers, FIFO queue for incoming trade data, etc.) before starting the WebSocket connection process. It sets libwebsockets' protocols struct, creates the context and sets all necessary parameters (address, port, path, SSL usage, retry/idle policy, etc.) before initiating the connection using **lws_sul_schedule()**. It then creates the other 3 threads; structs are used to pass bulk data to them, as **pthread_create()** can only pass one parameter to the thread it creates. It then sits idle until it receives either of the SIGINT or SIGTERM signals, at which point it begins the cleanup and shutdown process by setting certain flags, broadcasting pthread conditions and using **pthread_join()** on each thread, as well as destroying the libwebsockets context, closing file pointers, etc.

2) The "producer" thread receives the libwebsockets context (sent at creation by the main thread) and uses it to continuously call **lws_service()** until a termination flag is set. No other thread ever interacts with the LWS API (except for *main*, after joining *producer*), as it is single-threaded. When a WebSocket event occurs, **lws_service()** calls **callback_okx()** as defined in the protocols struct. This callback function responds to the event accordingly:

- On CLIENT_ESTABLISHED, the function subscribes to the 8 required symbols using **lws_callback_on_writable()**

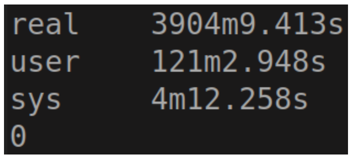
- On CLIENT_RECEIVE, the function checks what was received. "subscribe" messages are printed to console, "unsubscribe" messages prompt another **lws_callback_on_writable()** call to subscribe again and non-NULL "data" messages result in the addition of new trade data to the FIFO queue. Data comes in JSON format and is parsed via the **jansson.h** library

- On "CLIENT_CLOSED" or "CLIENT_CONNECTION_ERROR", reconnection is attempted indefinitely until it succeeds, using **lws_retry_sul_schedule_retry_wsi()**

3) The “consumer” thread removes the trade data from the FIFO queue and saves it to its respective file (each symbol has its own file). It also increments 3 variables: *numTrades*, *totalSize* and *sumPrices*, to keep track of the symbol’s number of trades conducted, total size traded and sum of all trade prices paid in the last minute. This trio of *numTrades*, *totalSize* and *sumPrices*, along with all the file pointers, comprise a struct dubbed “database” which is given its own pthread mutex (contained within it) to prevent race conditions, similar to the FIFO queue’s mutex.

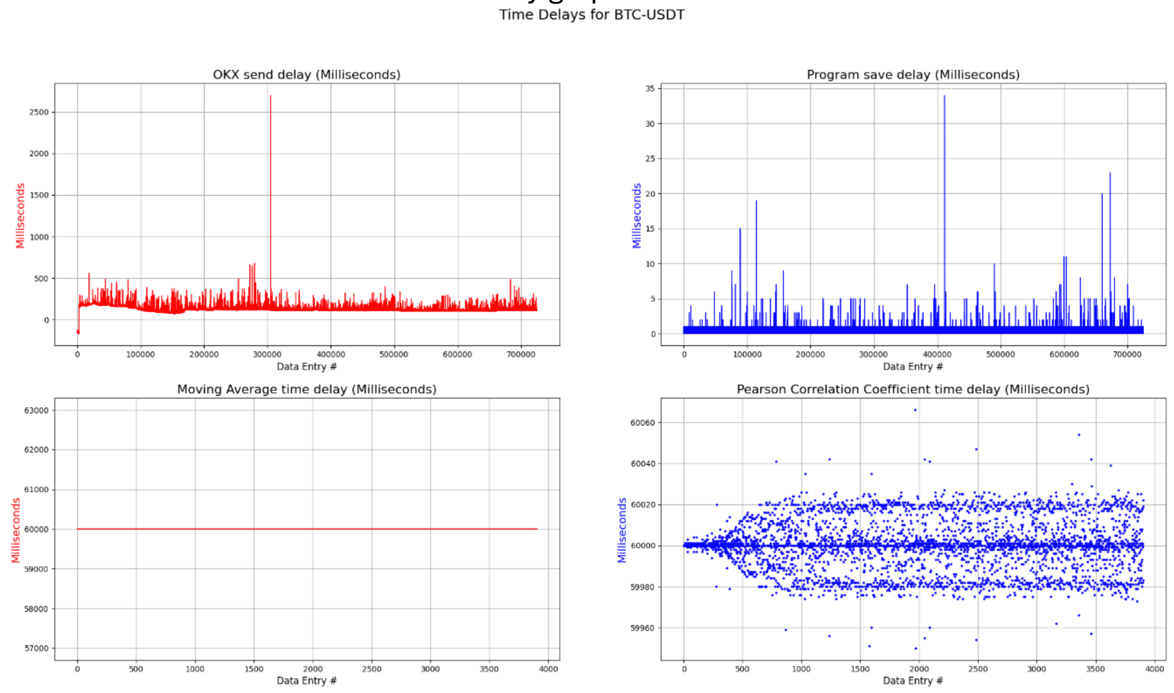
4) The ”everyInterval” thread wakes once every minute to save the data in the “database” struct to its own arrays; it then resets the struct’s data to 0 in order to prepare the recording of the next minute’s trade data. For each symbol, it eventually (after 15 minutes) accumulates and keeps the “results” of the most recent 15 minutes in arrays whose oldest value gets replaced every minute. This data is then on-the-spot (every minute) used to calculate and save (to files) the required moving average and Pearson correlation coefficient values of the minute. The coefficient calculation takes into account the program’s entire runtime, parsing the moving average files value by value, one by one, to find the absolute best correlation and the timestamp it corresponds to.

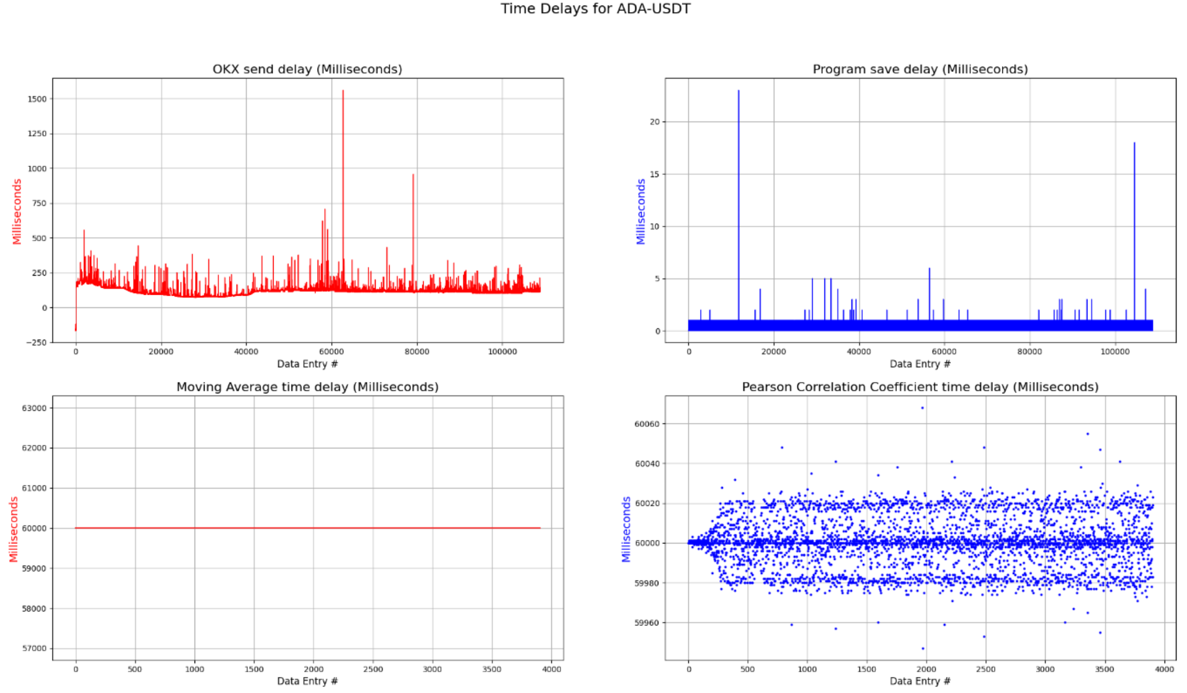
The implementation of the FIFO queue was taken from Andrae Muys’ 1997 Producer/Consumer Demo, with slight modification. The program was run using *time ./rtes-okx-raspi; echo $?* to get the information shown below (full runtime log on GitHub):

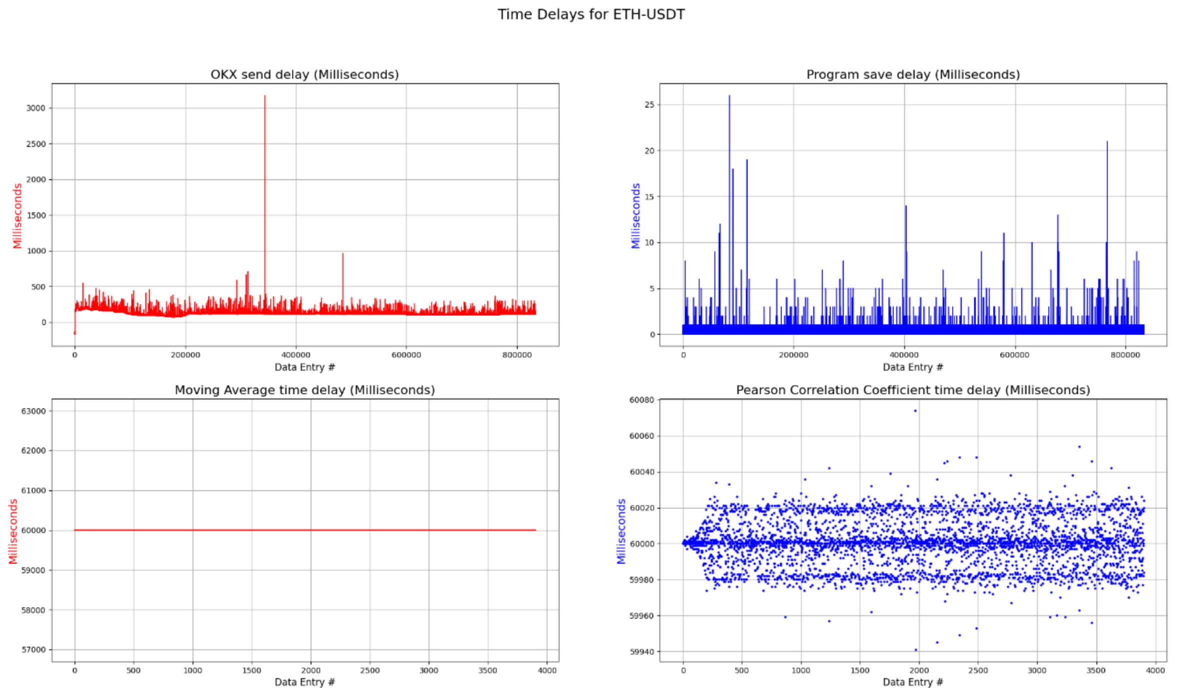


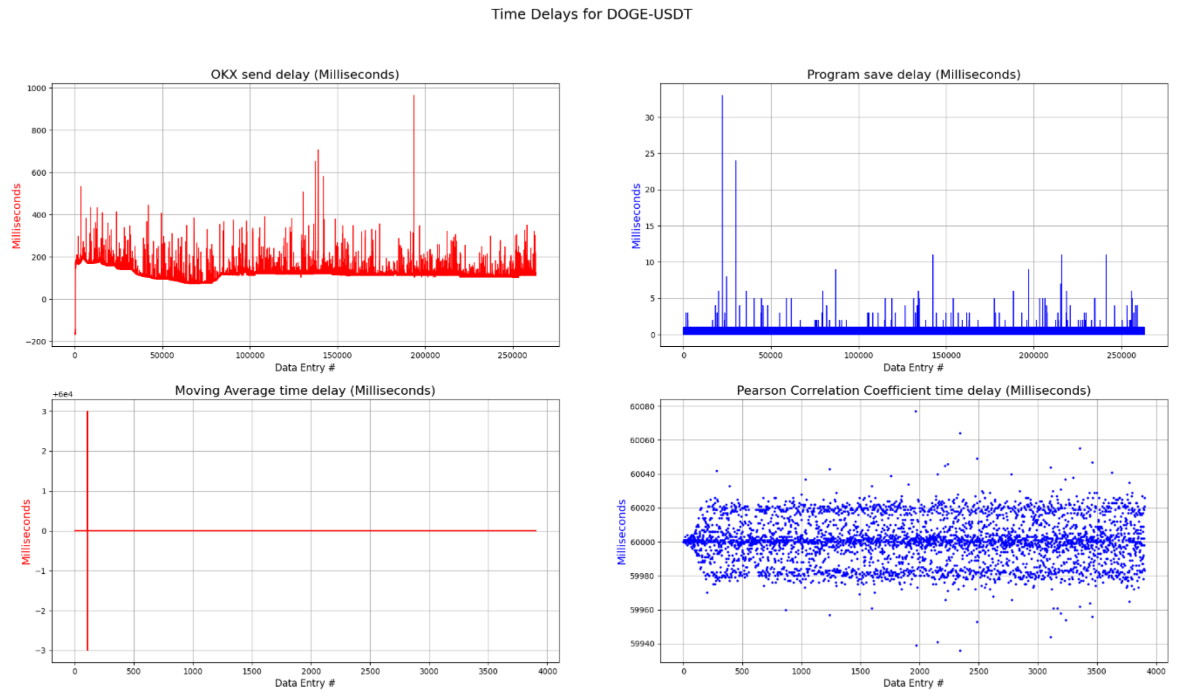
```
real    3904m9.413s
user    121m2.948s
sys     4m12.258s
0
```

The total runtime was 3904 minutes and 9.413 seconds, or approximately **65 hours** without issue. The sum of “user” and “sys” comprises the total amount of time the CPU was busy for; dividing it by “real” gives the approximate result of **0.032**, which is the fraction of the total runtime for which the CPU was busy. This means that the CPU was busy approximately **3.2%** of the time, and therefore it was idle **96.8%** of the time. “0” is the exit status. Below are the time delay graphs that resulted from this run:

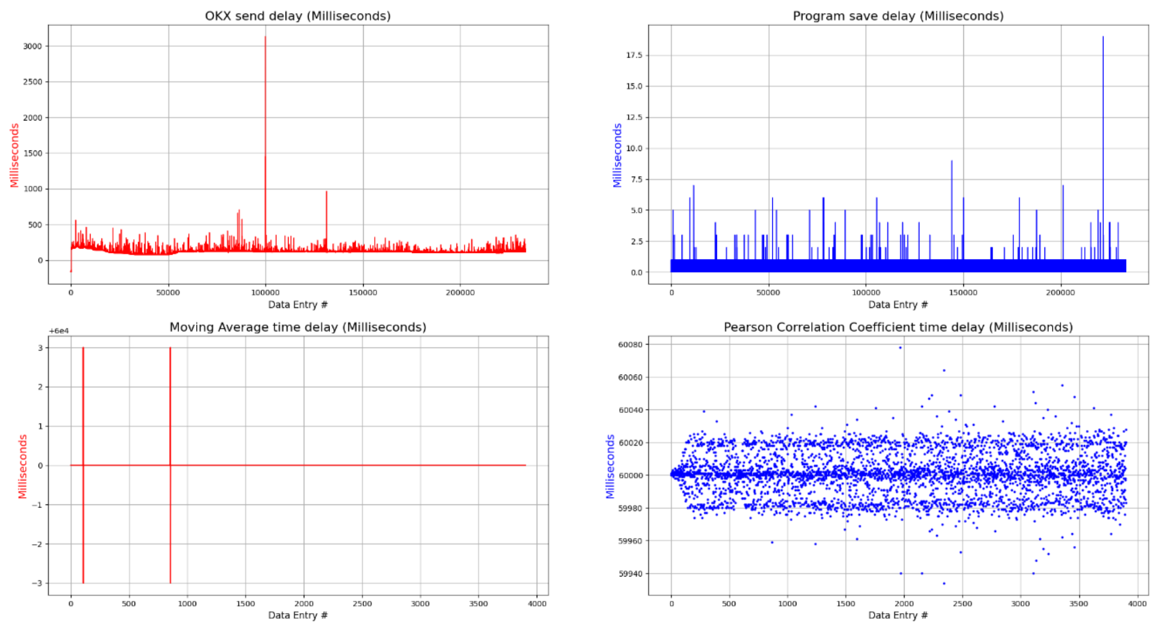



Time Delays for ADA-USDT
OKX send delay (Milliseconds)
Program save delay (Milliseconds)
Moving Average time delay (Milliseconds)
Pearson Correlation Coefficient time delay (Milliseconds)
Milliseconds
Data Entry #

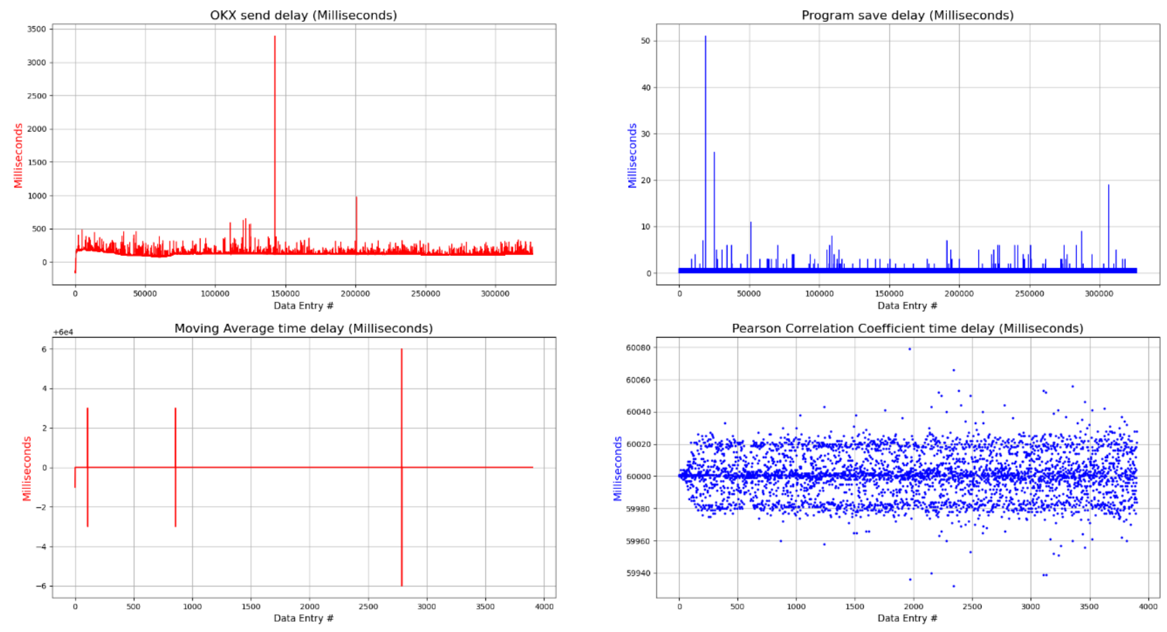

Time Delays for ETH-USDT
OKX send delay (Milliseconds)
Program save delay (Milliseconds)
Moving Average time delay (Milliseconds)
Pearson Correlation Coefficient time delay (Milliseconds)
Milliseconds
Data Entry #


Time Delays for DOGE-USDT
OKX send delay (Milliseconds)
Program save delay (Milliseconds)
Moving Average time delay (Milliseconds)
+6e4
Pearson Correlation Coefficient time delay (Milliseconds)
Milliseconds
Data Entry #

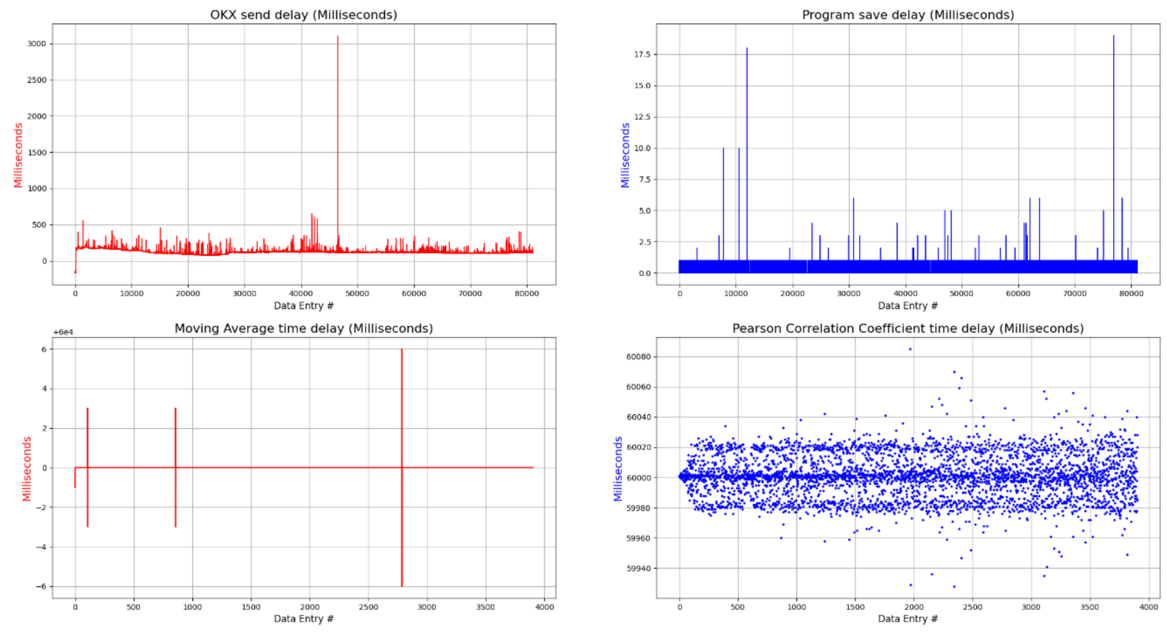
Time Delays for XRP-USDT



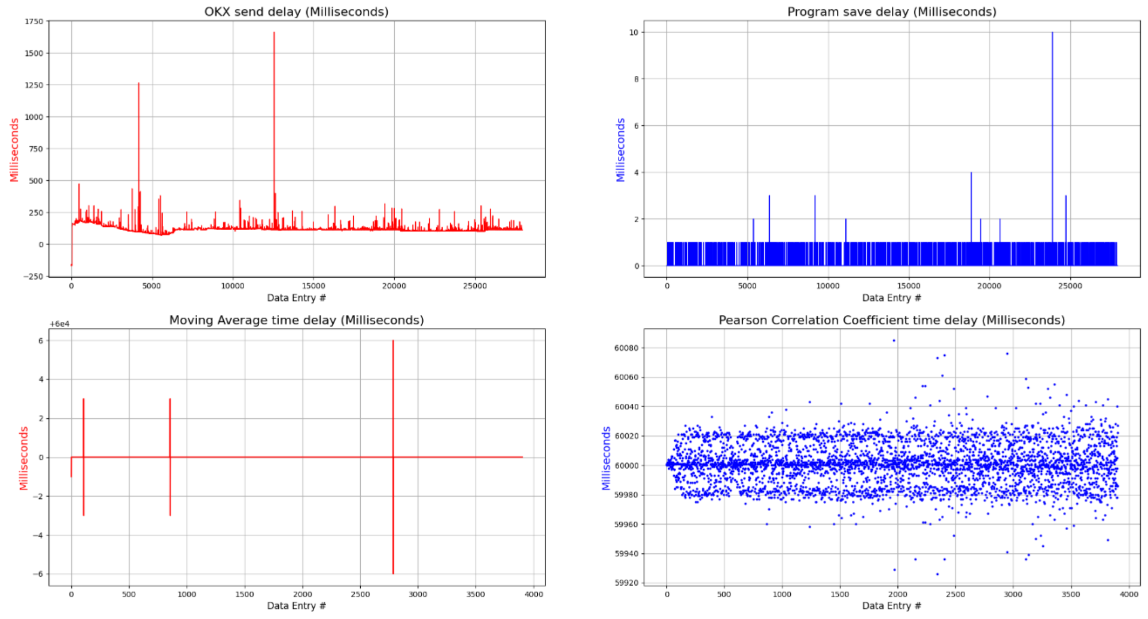
Time Delays for SOL-USDT



Time Delays for LTC-USDT



Time Delays for BNB-USDT



When it comes to the trade data itself, most of the time delay is on the part of OKX sending the data, ranging in the hundreds of milliseconds, whereas the program itself takes only a few tens of milliseconds to save it (mostly below 10 ms). This makes sense as sending data over the internet should take longer than waiting for mutexes to unlock and writing to a file.

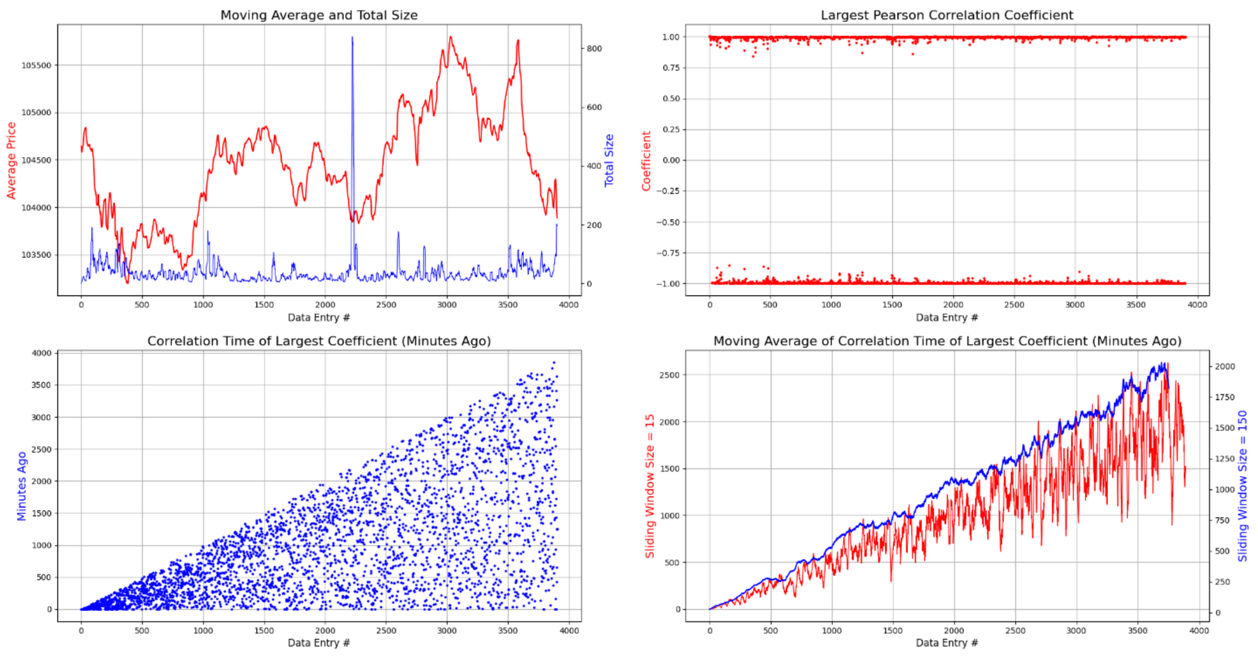
When it comes to the per-minute processing and saving delays (calculated as “next” minus “previous”, which should be 1 minute apart), the moving averages are nearly always exactly 60000 ms apart (as expected), whereas the Pearson correlation coefficients take a few tens of milliseconds to be saved. This is also expected, because the maximum coefficient is searched for among all previous moving average values, which involves file reading – a “heavy” investment in terms of time, compared to working directly with memory.

Nevertheless “drift” is avoided by incrementing the “everyInterval” thread’s **pthread_cond_timedwait()** timer by 60 seconds *immediately* upon each non-spurious activation/wakeup, *before* any of the processing takes place, ensuring the “clock” will “ring” exactly 1 minute later, every time, without being impacted by the processing’s runtime. As such, no entries are “missed”: 3904 minutes results in 3904 entries. The only way “drift” could occur here is if the files got so large that parsing them took more than a whole minute – in which case this program will have indeed reached its limit. The *timespec* struct (incremented at the beginning of each wakeup) used for **pthread_cond_timedwait()** (which implements the timer) as well as the attached pthread condition are implemented using CLOCK_MONOTONIC, which is independent of network time changes, unlike CLOCK_REALTIME. This way the internal timing cannot be disrupted by outside changes.

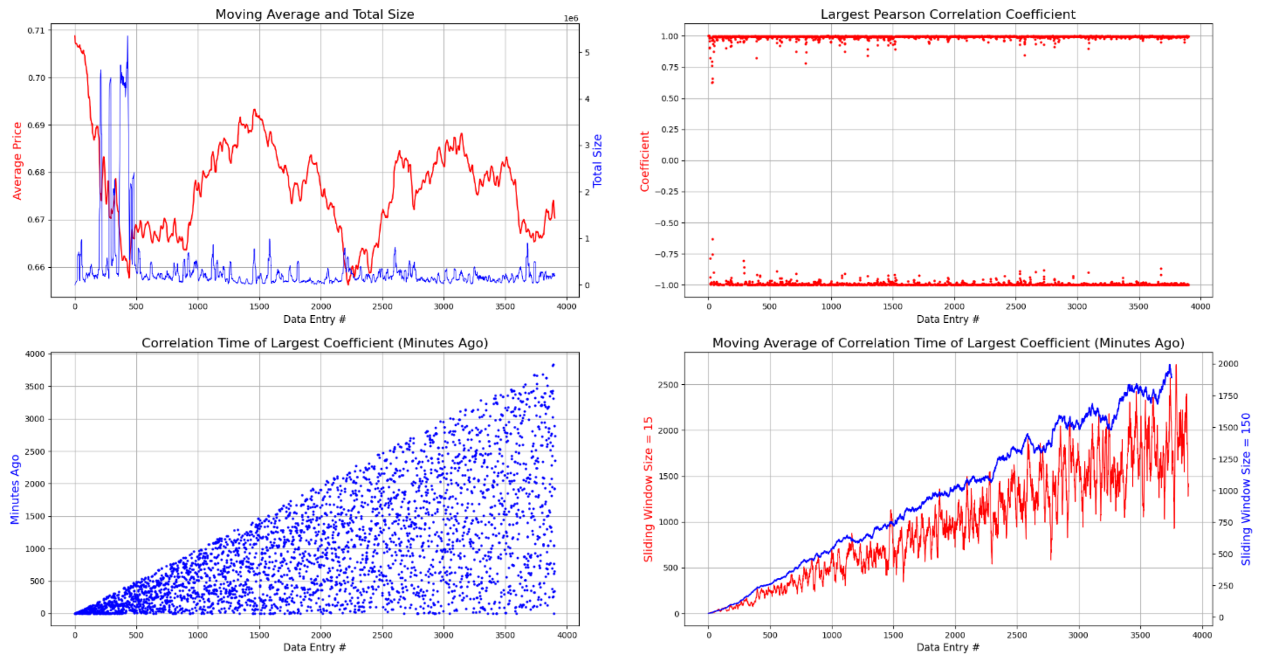
Signal handlers are NOT used as very few functions can be safely called in them (must be *async-signal-safe*). Mutex related functions do not belong in this category. Instead, the main thread sleeps until SIGINT/SIGTERM appears using **sigwait()**. The other threads are barred from responding to any signal by means of signal masking.

What follows are graphs showcasing some patterns (or lack thereof) on the moving average, size and Pearson correlation coefficient data collected over these 65 hours:

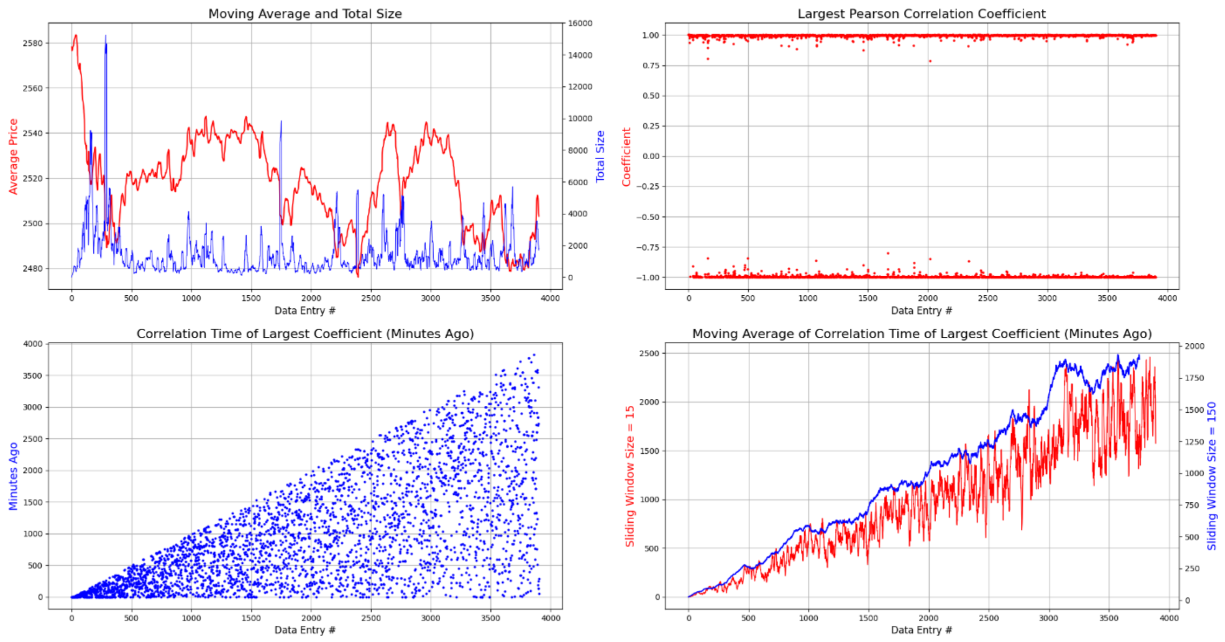
Moving Average, Size and Pearson Correlation Coefficient data for BTC-USDT



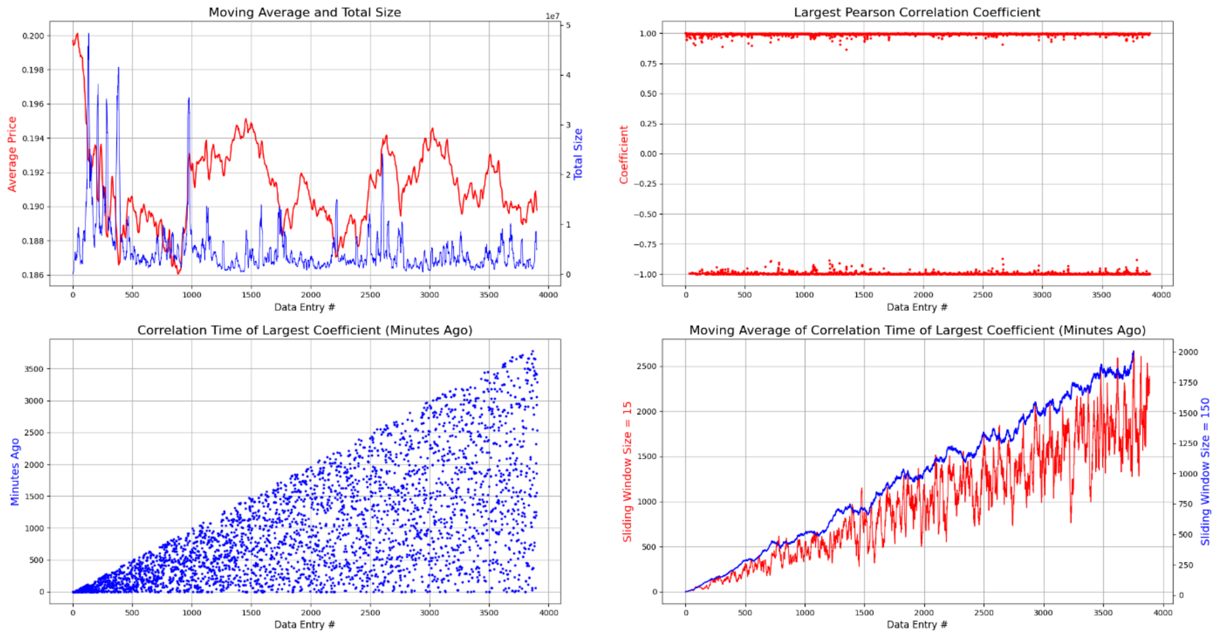
Moving Average, Size and Pearson Correlation Coefficient data for ADA-USDT



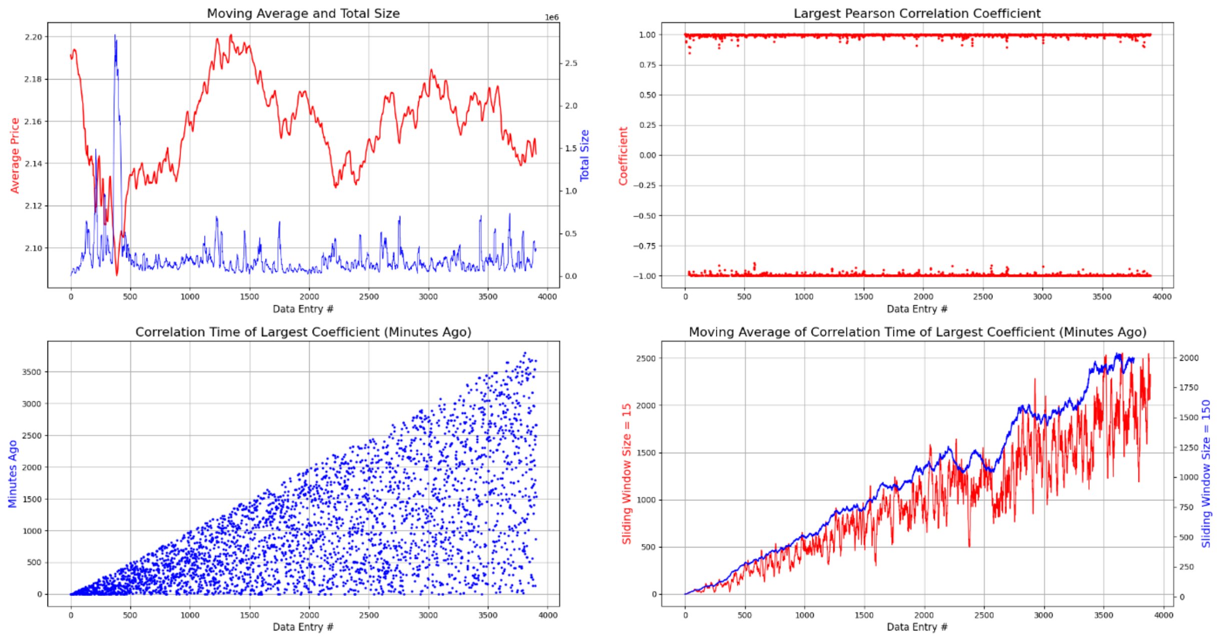
Moving Average, Size and Pearson Correlation Coefficient data for ETH-USDT



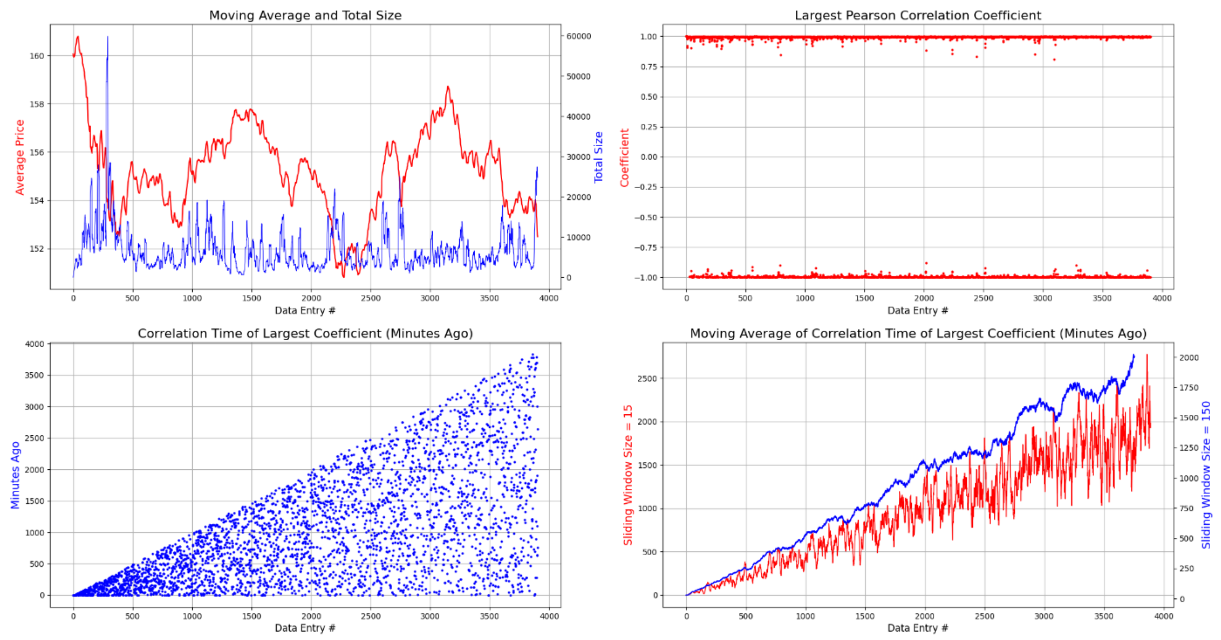
Moving Average, Size and Pearson Correlation Coefficient data for DOGE-USDT



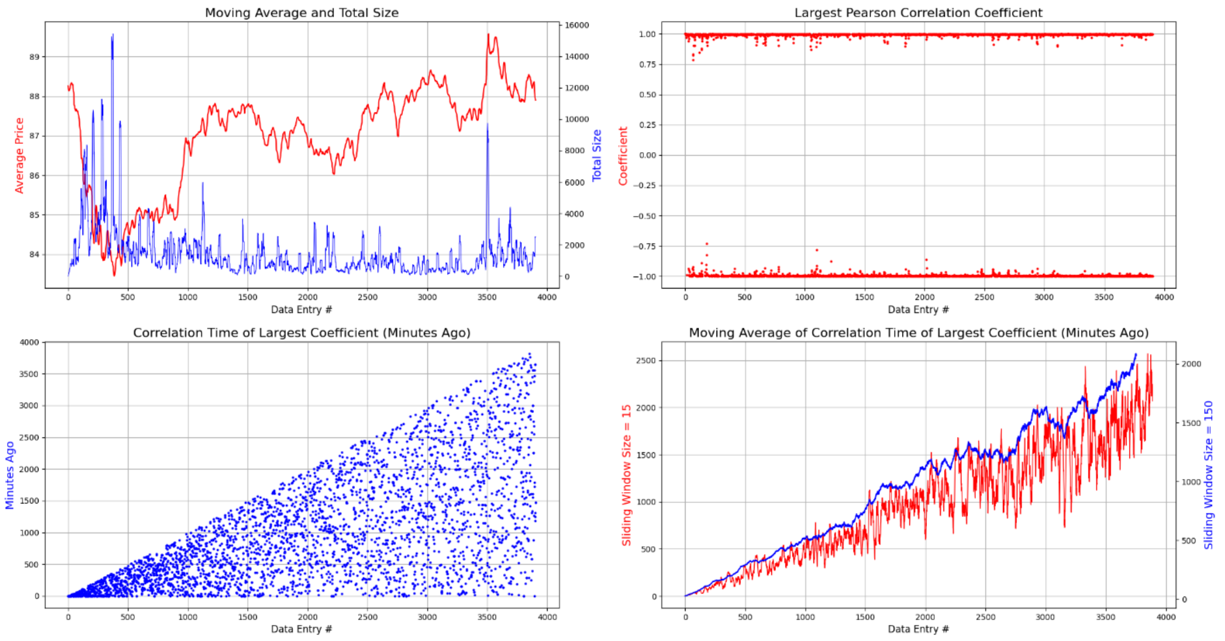
Moving Average, Size and Pearson Correlation Coefficient data for XRP-USDT



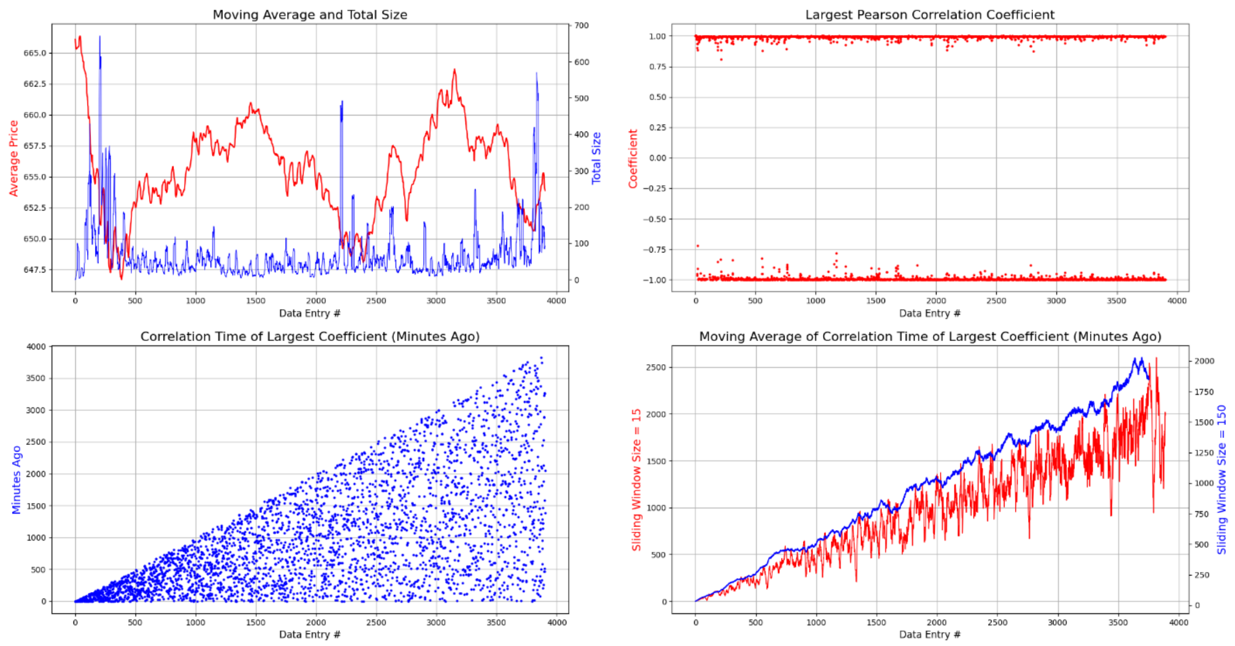
Moving Average, Size and Pearson Correlation Coefficient data for SOL-USDT



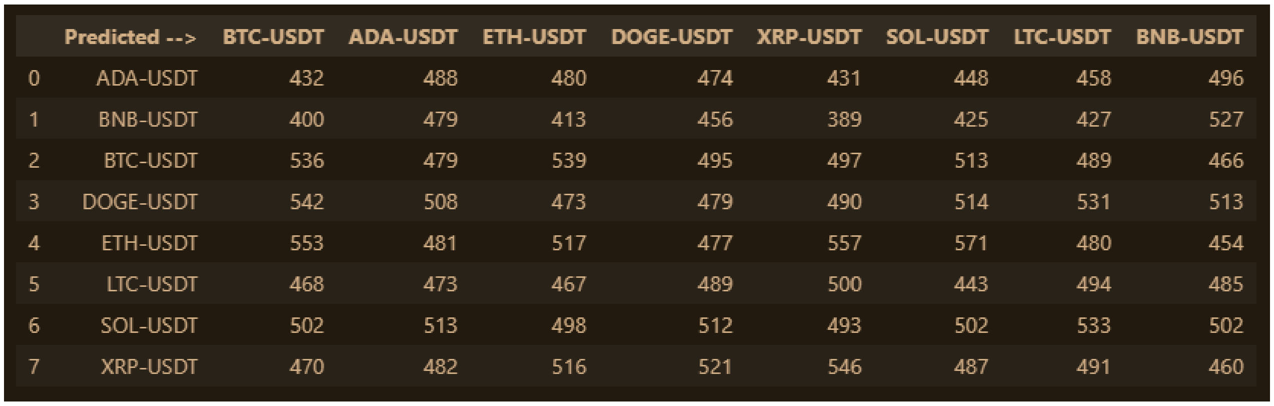
Moving Average, Size and Pearson Correlation Coefficient data for LTC-USDT



Moving Average, Size and Pearson Correlation Coefficient data for BNB-USDT



The “Moving Average and Total Size” graphs mostly serve as confirmation that the data was collected accurately (by comparing them to online sources tracking the same crypto-coins over the same period), but also uncanny similarities can be seen between them. This is corroborated by the Pearson correlation data, which although interesting, is difficult to make use of for predictive purposes. The “Largest Pearson Correlation Coefficient” graphs show that a coefficient very near 1 (or -1) with past data can almost always be found, but finding a stable leading indicator appears exceedingly difficult, as can be shown below:



	Predicted -->	BTC-USDT	ADA-USDT	ETH-USDT	DOGE-USDT	XRP-USDT	SOL-USDT	LTC-USDT	BNB-USDT
0	ADA-USDT	432	488	480	474	431	448	458	496
1	BNB-USDT	400	479	413	456	389	425	427	527
2	BTC-USDT	536	479	539	495	497	513	489	466
3	DOGE-USDT	542	508	473	479	490	514	531	513
4	ETH-USDT	553	481	517	477	557	571	480	454
5	LTC-USDT	468	473	467	489	500	443	494	485
6	SOL-USDT	502	513	498	512	493	502	533	502
7	XRP-USDT	470	482	516	521	546	487	491	460

The way to read this is: ADA-USDT was the leading indicator (i.e. provided the best Pearson correlation coefficient) for BTC-USDT **432** times over the program’s runtime; BNB-USDT did the same **400** times, and so on and so forth. As such, each column adds up to **3903**, since the program was run for **3904** minutes and each minute has its respective best indicator for each coin (the calculation is impossible for the first minute, hence 3903 instead of 3904).

It is immediately obvious that though there is an “overall winner” for every coin (i.e. a coin that was the leading indicator for it most often), this “winner” is always on a narrow margin over the rest. Every coin ends up “leading” every other coin hundreds of times; examining the raw data (text) files, there appears to be no easily discernible pattern governing which one takes over as the best predictor at which time.

Looking at the times themselves, the “Correlation Time of Largest Coefficient” graphs show how far back in time (in minutes) the best correlation coefficient is found to be, for each minute. These results are also not very promising. There is a clear rising trend, meaning that the longer the program runs, the further back in time the best correlation is to be found; but this rising trend is only *on average*, as indicated by the “Moving Average of Correlation Time of Largest Coefficient” graphs. The actual “Correlation Time” graphs maintain a huge variance (standard deviation **>858** on the whole) all throughout, meaning the rise of the average value can simply be explained by the fact that as the program runs, the upper bound of “Correlation Time” increases while the lower bound remains the same, hence raising the average whilst still remaining unpredictable.