Using crytpocurrencies and LispTick to evaluate trade signing accuracy

Cédric Joulain
Strigi-Form
`cedric.joulain@strigi-form.com`
`http://www.strigi-form.fr http://www.lisptick.org`

December 24, 2018

Abstract

This paper evaluates Tick Test, Mid Test and Lee-Ready methods to infer trade direction using cryptocurrencies intraday data and LispTick. Those procedures of trade signing, inferring if trade is market buy or market sell using intraday data, are validated with Bitstamp and implemented with LispTick. Bitstamp is a market place where you can trade some main cryptocurrencies and where real trade direction is part of the intraday data. LispTick is a service using extended Lisp language with integrated time series as an interface. Comparing the real direction with the inferring one shows each method accuracy. We also show how easy it is to implement algorithms and timestamp management with LispTick.

1 Introduction

Bitstamp is a European based cryptocurrency marketplace. Its web-socket API allows to retrieve lots of information for each trade and quote rarely available on standard market place. For example, for each trade we have timestamp, in second, price, amount... But we also have id, buy-order-id, sell-order-id and more important in our study: type, which is buy or sell. So having the real direction of the trade will allow us to compute an accuracy score for each method.

LispTick is an extension of Lisp language with new embedded types like time series, time, delays. With a powerful internal synchronization mechanism you can add, subtract, multiply or do what you want with several input time series to create a new output time series. Lisp is there just as an interface to describe the algorithm, this message is then parsed by a service written in Go, Google open source programming language.

Then and only then the algorithm is evaluated as a stream by pure Go. This streaming approach allows LispTick to work on very tiny machines like Raspberry Pi 3, all source code is also tested on this small platform.

All the following algorithms implemented with LispTick are validated against Bitstamp data. But they can be applied on any source included classical exchanges like Eurex, Nasdaq, EuroNext, LSE...

2 Used data and definitions

Each method is implemented with 4 input arguments and result is a time series of signs., -1 for sell, 1 for buy and 0 for undefined.

Input arguments are:

source the data source, here Bitstamp.

code the instrument code, here "BTC" "ETH" "XRP" "LTC" "BCH" "EUR".

start the beginning date, here 2018-01-19 .

stop the ending date included, here 2018-12-10 .

Output sign is:

-1 for sell.

0 for unsigned, for example trade price exactly math mid in mid test.

1 for buy.

The full number of trades for all currencies and the given period is **19,302,001**.

For each method we are counting how many trades are signed, name “cover”, how many trades have a 0 sign, name “no sign” and how many trade signs are correct, named “right sign”. Then we are able to compute percentage score using:

$$score = \frac{number\ of\ right\ sign}{number\ of\ trades}$$

For each method we also try several different delays to move quotes in time. This is not done for Tick Test as it is only based on trade price and not on bid/ask.

There are two different ways of moving quotes int time, adding the same delay to all timestamp or using previous or next update time.

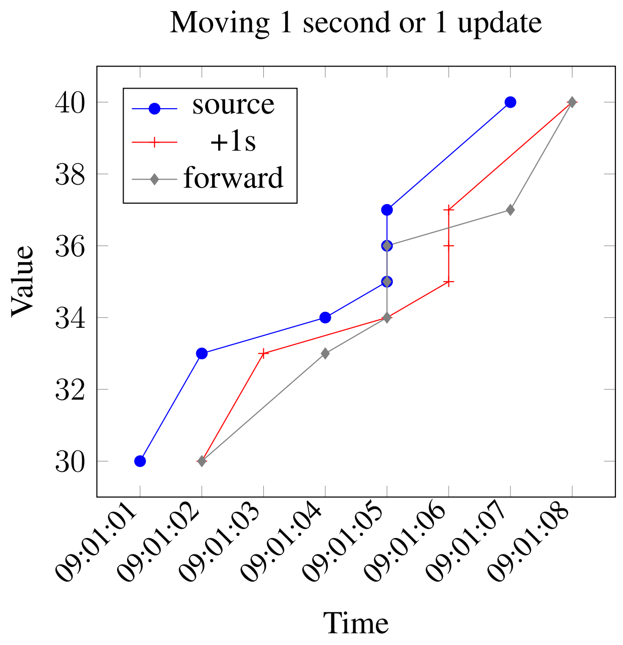

Figure 1: Different ways of moving in time

As you can see in Figure 1 adding 1 second to each point will move uniformly all the time series.

Another moving option is to move forward, meaning using next timestamp for current value, move seems less uniform but it has the advantage to adapt to the underlying instrument liquidity. Moving by an absolute amount of time will not have the same impact on time-series depending on instrument liquidity. For a very liquid instrument, 1 second can be huge, for an unliquid one it's small. Using previous or next update time is intrinsically linked to instrument liquidity.

We test for each model delays from -4 seconds to 9 seconds, second by second as Bitstamp timestamp is only precise to the second. We also test several update moves, backward (use previous update time), forward, 2x forward and 3x forward. Base and 0s being the same as original time series without any change.

All the Listings that follow are the exact LispTick code used to compute results. All words in bold are reserved LispTick keywords, others are user defined.

Listing 1: Tick Test LispTick implementation

```
(defn tick-test[source code start stop]
  (*
    (clockref
      (one
        (timeserie @trade-price source code start stop)
      )
      0 ;reference value for sign when no delta price available
    )
    (keep
      (sign
        (delta
          (timeserie @trade-price source code start stop)
        )
      )
      not= 0
    )
  )
)
```

model	cover	no sign	right sign	score
tick-test	19,302,001	3,331	14,392,078	74.56%

Table 1: Tick Test accuracy

3 Tick Test

The tick test infers the direction of a trade by comparing its price to the price of the preceding trade(s). The test classifies each trade into four categories: an uptick, a downtick, a zero-uptick, and a zero-downtick. A trade is an uptick (downtick) if the price is higher (lower) than the price of the previous trade. When the price is the same as the previous trade (a zero tick), if the last price change was a downtick, then the trade is a zero-downtick. A trade is classified as buy, 1 for our time series, if it occurs on an uptick or zero-uptick; otherwise it is classified as a sell, -1 for our time series.

3.1 Implementation

LispTick implementation is quite compact, see Listing 1. We create a time series of not null sign from the delta of each price. We then multiply by a time series of 1 at each trade timestamp. Doing so each missing sign will be replaced by previous sign multiplied by 1…

3.2 Accuracy

Score obtained in Table 1 is in line with other studies like [2], where accuracy is estimated around 72%.

Listing 2: Mid Test LispTick implementation

```
;Best result for mid-test is to use quotes 1 step in past
(defn move-in-time[ts]
  (forward
    ts
  )
)

(defn base-mid[source code start stop]
  (/
    (+
      (timeserie @bid-price source code start stop)
      (timeserie @ask-price source code start stop)
    )
    2
  )
)

(defn mid-test[source code start stop]
  (sign
    (-
      (clockref ;only one point per trade
        (timeserie @trade-price source code start stop)
        0 ;reference value for sign when no quote available
      )
      ;ask previous day quotes to match all trades
      (move-in-time
        (base-mid source code start stop)
      )
    )
  )
)
```

4 Mid Test

Mid test is based on middle price, $mid = \frac{best\,ask + best\,bid}{2}$, if trade price is higher (lower) this is a buy (sell) trade. Trade with price exactly matching mid can't be signed.

4.1 Implementation

LispTick implementation is also straightforward is this case, see Listing 2.

The only trick is to define trade price timeseries as a **`clockref`** for the subtraction so there will be only one update per trade and not one update for every new trade and mid.

Second refinement, we move quotes in time in Listing 2 with the `move-in-time` user defined function. This allows to match each trade with quote 1 update in past.

model	cover	no sign	right sign	score
bwd	19,302,001	395,918	13,684,793	70.90%
base	19,302,001	263,351	15,016,541	77.80%
fwd	19,302,001	102,391	15,624,687	80.95%
2fwd	19,302,001	58,377	15,584,930	80.74%
3fwd	19,302,001	54,532	15,158,212	78.53%

Table 2: Mid Test accuracy per update delay

lag	cover	no sign	right sign	score
-4	19,302,001	129,797	12,557,386	65.06%
-3	19,302,001	172,271	13,020,824	67.46%
-2	19,302,001	249,422	13,513,357	70.01%
-1	19,302,001	379,156	14,017,530	72.62%
0	19,302,001	263,351	15,016,541	77.80%
1	19,302,001	112,140	15,581,440	80.72%
2	19,302,001	66,431	15,550,195	80.56%
3	19,302,001	55,986	15,365,819	79.61%
4	19,302,001	51,488	15,190,967	78.70%
5	19,302,001	48,898	15,033,891	77.89%
6	19,302,001	47,071	14,895,261	77.17%
7	19,302,001	45,509	14,791,234	76.63%
8	19,302,001	44,571	14,699,256	76.15%
9	19,302,001	43,584	14,612,926	75.71%

Table 3: Mid Test accuracy per s lag

4.2 Accuracy

LispTick computed results can be seen in Table 2 for delay in number of updates and in Figure 3 for lag in seconds. A more graphical view can be seen in Figure 2. And in Figure 3 we can see scores split by currency. EUR have a different behavior perhaps because it's not a crypto and, on this place, it is less liquid, have far lower volatility with slighly higher spread.

Concerning quantity of signed trades, as starting quotes are 1 day in past coverage is always 100% of trades. We always have enough information to put a mid in-front of each trade price.

Around 1% of trades have a zero sign, labeled "no sign", meaning trade price was exactly equal to mid price. This percentage decrease as we move forward, i.e. as we use quotes from the past.

Best scores are obtained by using quotes one update in the past, or with a 1 second lag when we use an absolute time as delay.

5 Lee-Ready

Lee-Ready procedure [1], is based on both trade prices and quotes. First it uses what we call a "Side Test", if trade price is higher than (lower than) best ask (best bid) it is a buy (a sell). For trade with price between bid and ask it uses Tick Test result.

5.1 Side Test Implementation

Side Test LispTick implementation is done in 3 parts see Listing 3:

1^{st} `ask-side` a time series of 1 when trade price is higher or equal to ask price and 0 other ways.

2^{nd} `bid-side` a time series of -1 when trade price is lower or equal to bid price and 0 other ways.

3^{rd} Then we sum both, so trade between bid and ask will keep a 0 sign.

Ask (bid) side is done by computing sign of $trade\,price - ask(bid)$ then we add 1 (-1) and once again we only keep sign.

Listing 3: Side Test LispTick implementation

```
;Best result for side-test is to use quotes 1 step in past
(defn move-in-time[ts]
  (forward
    ts
  )
)

(defn ask-side[source code start stop]
  (sign
    (+
      1 ; only prices < ask will have 0 sign, others 1
      (sign
        (-
          (clockref
            (timeserie @trade-price source code start stop)
            0 ;reference value for sign when no quote available
          )
          (move-in-time
            (timeserie @ask-price source code start stop)
          )
        )
      )
    )
  )
)

(defn bid-side[source code start stop]
  (sign
    (+
      -1 ; only prices > bid will have 0 sign, others -1
      (sign
        (-
          (clockref
            (timeserie @trade-price source code start stop)
            0 ;reference value for sign when no quote available
          )
          (move-in-time
            (timeserie @bid-price source code start stop)
          )
        )
      )
    )
  )
)

(defn side-test[source code start stop]
  (+
    (ask-side source code start stop)
    (bid-side source code start stop)
  )
)
```

model	cover	no sign	right sign	score
bwd	19,302,001	5,438,131	10,837,939	56.15%
base	19,302,001	4,594,681	12,605,479	65.31%
fwd	19,302,001	4,773,232	13,031,930	67.52%
2fwd	19,302,001	4,428,995	12,934,968	67.01%
3fwd	19,302,001	5,004,558	12,090,309	62.64%

Table 4: Side Test accuracy per delay

So `ask-side` gives:

$trade > ask \ \rightarrow sign(sign(trade - ask) + 1) = sign(1 + 1) = 1$

$trade = ask \ \rightarrow sign(sign(trade - ask) + 1) = sign(0 + 1) = 1$

$trade < ask \ \rightarrow sign(sign(trade - ask) + 1) = sign(-1 + 1) = 0$

And `bid-side` gives:

$trade > bid \ \rightarrow sign(sign(trade - bid) + 1) = sign(1 - 1) = 0$

$trade = bid \ \rightarrow sign(sign(trade - bid) + 1) = sign(0 - 1) = -1$

$trade < bid \ \rightarrow sign(sign(trade - bid) + 1) = sign(-1 - 1) = -1$

Once again trade price time series are used as **`clockref`** to get one and only one point per trade update.

5.2 Side Test Accuracy

As expected, see Table 4, Side Test score is smaller than Mid Test as all trades between bid and ask are unsigned. The best score is also obtained using quotes one update in past (fwd).

5.3 Lee-Ready Implementation

Lee-Ready LispTick full implementation, see Listing 4, uses previously defined functions `tick-test` and `side-test`. We split `side-test` in two parts, unsigned $= 0$ and signed $not = 0$. We keep signed part as it is and we **`merge`** it with unsigned part replaced by `tick-test`. To do so we sum unsigned trades with `tick-test` using **`clockref`** to get one and only one update per unsigned `side-test` and not an update for every `tick-test`.

Listing 4: Lee-Ready LispTick implementation

```
(defn lee-ready[source code start stop]
  (merge
    (keep
      (side-test source code start stop)
      not= 0
    )
    (+
      (clockref
        (keep
          (side-test source code start stop)
          = 0
        )
      )
      (tick-test source code start stop)
    )
  )
)
```

model	cover	no sign	right sign	score
bwd	19,302,001	963	14,774,263	76.54%
base	19,302,001	834	15,773,449	81.72%
fwd	19,302,001	1,291	16,148,241	83.66%
2fwd	19,302,001	1,430	15,690,022	81.29%
3fwd	19,302,001	1,574	15,209,800	78.80%

Table 5: Lee-Ready accuracy per delay

5.4 Performance as a function of lag

Lee and Ready [1] observe that the current quote is often not synchronized to the trades, so they suggest to compare the trade price to the quote 5 seconds before. Doing this with LispTick is as easy as adding 5s to the time series, meaning adding 5s to the timestamp of the time series.

This 5 seconds lag from Lee-Ready was purely empirical, so we tried several lags to find the best for our data. In our case the best absolute lag is 1s for Lee-Ready and Mid-Test.

Our results for Lee-Ready score in Figure 2 are in line with Ioane Muni Toke study [3] Section 5, our graphs have an opposite direction as we add time instead of subtracting it. We have the same shape, performance drops quite fast on one side and more progressively on the other. Absolute maximum is more in line with recent studies than older ones.

As LispTick can move time series by a certain number of updates we also try several update lags, see Figure 2 and Table 5.

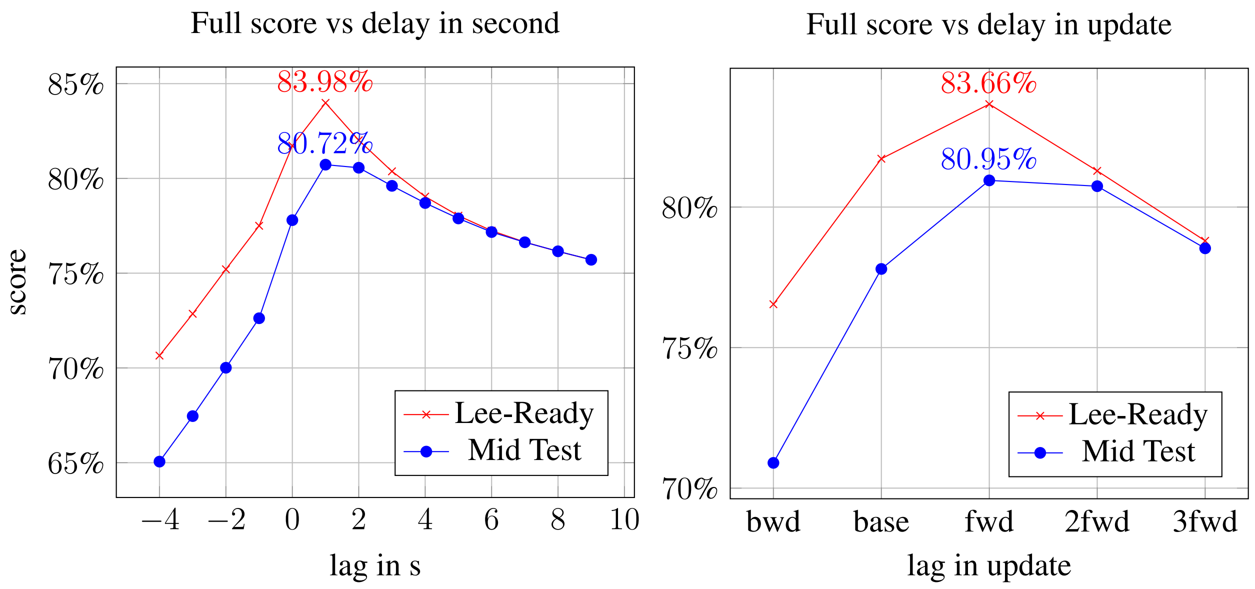

Figure 2: Lee-Ready vs Mid Test

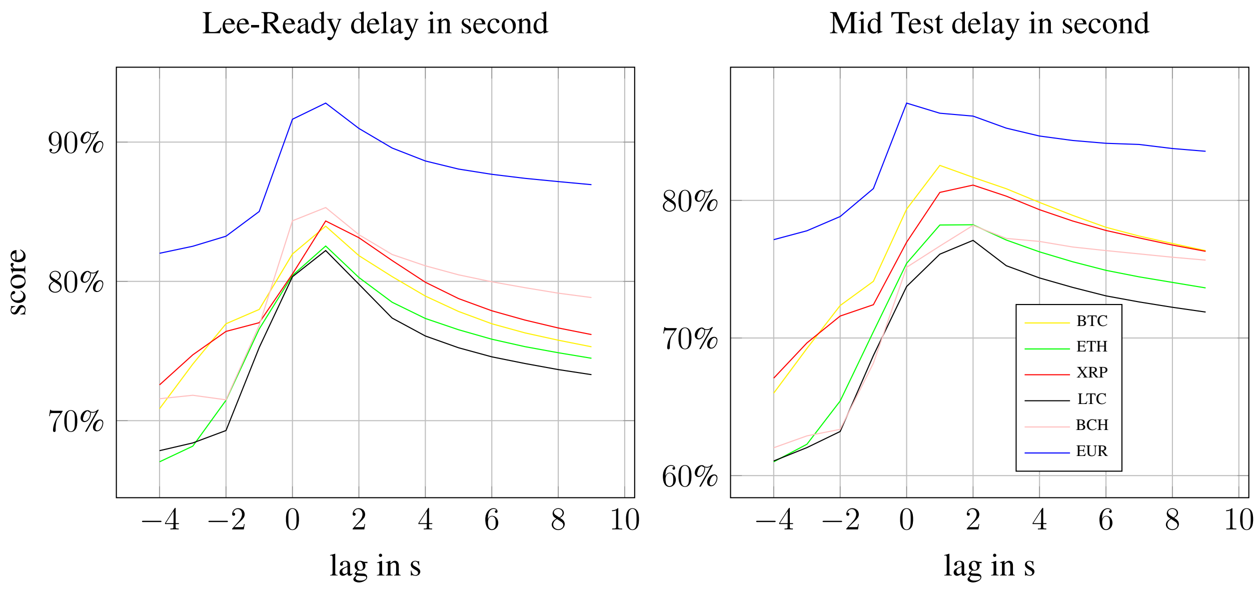

Figure 3: Lee-Read & Mid Test score for each currency

6 LispTick more in-depth

We present here more technical information about LispTick. Thanks to streaming LispTick is very frugal with memory resources: it works on a Raspberry Pi 3, a small single-board computer costing $35. In fact, all the preceding examples didn't use more than 150MB of resident memory.

6.1 Computing Score

In Listing 5, you can see how score computation is done calling `right-sign`. Inferred sign, `side` function that can be `tick-test`, `mid-test` or `lee-ready`, is multiplied by the real sign coming from source time series **`@sign`**. Then we keep result of multiplication only when value > 0, so only when signs are identical and not equal to 0. Final result is the length (**`len`**) of this time series, the score of this method will by this number divided by full number of trades.

Similarly number of unsigned trades is computed by counting number of signs $= 0$ by the function `no-sign`.

6.2 Parallel-Sample

To benefit from the increasing number of cores in our today processors LispTick has implicit and explicit parallelization mechanisms.

In our case, we present how to compute score day by day and sum each day using all cores, for that we use **`parallel-sample`** and a function to compute one day score, see Listing 6&5.

How it works: first argument of **`parallel-sample`** is the function that will be called several times, here `nbr-right` counting number of rightly sign for one day one code see Listing 6. Then we have how data is sampled, here **`sum`** all values. And to finish we have the arguments to iterate on, all the dates and all the codes.

6.3 Benchmarks

In Table 6, you can see number of trades signed per second depending on method and computer. In most cases, data is in OS cache as we forget values from 1st run. Those are best results when cpus are efficiently cooled. Some overclocked cpus may suffer thermal throttling as LispTick maximize all cores usage.

The AMD Ryzen Threadripper 1950X is a 64-bit microprocessor with 16 cores and 32 threads (SMT enabled) and a frequency of 3.4Ghz. Benchmarks are 20 to 30% slower when disabling SMT, Simultaneous multithreading, Listing 5: Score LispTick implementation

```
;number of sign OK
(defn right-sign[source code start stop]
  (len
    (keep
      (*
        (clockref
          (side source code start stop)
        )
        (timeserie @sign source code start stop)
      )
      > 0
    )
  )
)

;signed with 0
(defn no-sign[source code start stop]
  (len
    (keep
      (side source code start stop)
      = 0
    )
  )
)

;simple functions for parallel sample
(defn nbr-trade[d code]
  (subsample 1D count
    (timeserie @trade-price "bitstamp" code d)
  )
)

(defn nbr-side[d code]
  (subsample 1D count
    (side "bitstamp" code d d)
  )
)

(defn nbr-no-sign[d code]
  (no-sign "bitstamp" code d d)
)

(defn nbr-right[d code]
  (right-sign "bitstamp" code d d)
)
```

Listing 6: Parallel score LispTick implementation

```
(parallel-sample nbr-right sum dates codes)
```

model	Tick-Test	Mid-Test	Lee Ready
amd1950X	7,200,000	919,000	535,000
i9-7900X	6,100,000	780,000	420,000
Ryzen 7 2700X	5,200,000	637,000	345,000
E3-1270v6	2,860,000	326,000	175,000
2xE5530	2,730,000	326,000	173,000
Celeron J3455	840,000	99,000	53,000
i3-7100U	670,000	90,000	47,000
RPi-3	130,000	15,000	7,600

Table 6: Signing in trades per second, the more the best

The Intel Core i9-7900X is a 64-bit microprocessor with 10 cores and 20 threads (Hyper-Threading enabled) and an overclocked frequency of 4.0Ghz. Benchmarks are 30% slower when disabling Hyper-Threading.

The AMD Ryzen 7 2700X is a 64-bit microprocessor with 8 cores and 16 threads (SMT enabled) and a frequency of 3.7Ghz.

The Intel Xeon E3-1270 v6 is a 64-bit server microprocessor with 4 cores and 8 threads with a frequency of 3.8Ghz.

The Intel Xeon 2xE5530 is a 64-bit bi-microprocessors server with 2x4 cores and 2x8 threads with a frequency of 2.4Ghz.

The Intel Celeron J3455 is a Synology DS918+ NAS 64-bit microprocessor with 4 cores and 4 threads and a frequency of 1.5Ghz. This shows how easy it is to install LispTick, even directly on a NAS.

The Intel Core i3-7100U is a laptop 64-bit microprocessor with 2 cores and 4 threads and a frequency of 2.4Ghz.

The RPi-3 ARM Cortex-A53 is a 64-bit microprocessor with 4 cores and a frequency of 1.2Ghz.

7 Conclusion

With a tool like LispTick, it was very simple to implement and test all the methods. The accuracy found empirically for Lee-Ready is in line with previous theoretical studies. But the comparison with a simple Mid-Test shows that the gain in accuracy is small versus the increase in complexity. Tick test is very fast but the amount of errors increases a lot. Using other sources of data than Bitstamp will allow to refine our results. We are currently looking at Bitfinex, were timestamp accuracy is ms, but also at conventional (not crypto) market places with embedded sign if exists. And it will be straightforward as all the algorithms have already been implemented in LispTick, we just have to change the data source!

References

[1] Charles Lee and Mark Ready. Inferring trade direction from intraday data. *Journal of Finance*, 46:733–747, 01 1991.

[2] Marcelo Perlin, Chris Brooks, and Alfonso Dufour. On the performance of the tick test. *The Quarterly Review of Economics and Finance*, 54(1):42 – 50, 2014.

[3] Ioane Muni Toke. Reconstruction of order flows using aggregated data. *Market Microstructure and Liquidity*, 02(02):1650007, 2016.