Fast Arbitrage Detection

Claudio Raimondi

June 13, 2025

Abstract

Triangular arbitrage offers opportunities to exploit price inefficiencies between three currency pairs. While several algorithms exist for triangular arbitrage detection, their computational efficiency and accuracy can vary. In this report, I present a straight forward method that combines logarithmic properties with numerical representation techniques and advanced parallelization strategies to accurately and efficiently detect arbitrage opportunities.

Contents

1 Introduction

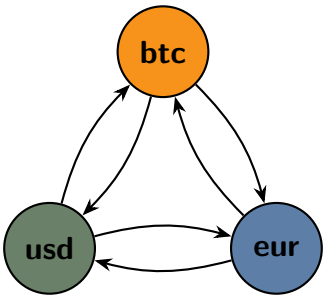
The process to exploit triangular arbitrage opportunities involves 3 generic steps, each crucial for ensuring the success of the strategy:

1. **Data Collection:** Gather real-time exchange rates for the currency pairs involved.
2. **Data Manipulation:** Use the collected data to detect profitable trades.
3. **Execution:** Execute trades to lock in profits.

For all three steps, the **speed** and **accuracy** of the algorithm are paramount. All the efforts become vain if somebody else collets, manipulates, and executes the arbitrage opportunity before you do. On this report, I will focus on the **second step**—*data manipulation*—, which is the most short-lived. The other two steps rely on external factors, such as networks and exchange APIs.

2 Visualization

To visualize triangular arbitrage, we can represent it as a **directed graph** (more precisely a clique), where each node represents a currency and each edge represents an exchange rate.



3 Arbitrage Formula

3.1 Base Formula

The formula to detect a profitable triangular arbitrage path, given three exchange rates R, is the following inequality:

$$R_1 \cdot R_2 \cdot R_3 > 1 \tag{1}$$

3.2 Price Format

If the *data collection* layer is serious about efficiency and precision, the prices are provided in **fixed point** representation as a combination of **mantissa** and **exponent** such that $R = m \cdot 10^e$. So the formula becomes:

$$(m_1 \cdot 10^{e_1}) \cdot (m_2 \cdot 10^{e_2}) \cdot (m_3 \cdot 10^{e_3}) > 1 \tag{2}$$

$$m_1 \cdot m_2 \cdot m_3 \cdot 10^{e_1+e_2+e_3} > 1 \tag{3}$$

$$m_1 \cdot m_2 \cdot m_3 > \frac{1}{10^{e_1+e_2+e_3}} \tag{4}$$

$$m_1 \cdot m_2 \cdot m_3 > 10^{-(e_1+e_2+e_3)} \tag{5}$$

3.3 Logarithmic Form

In most computer systems, where registers are limited to 64 bits, the product of three mantissas can easily overflow. To address this, we can use logarithmic properties to transform the **product** into an **addition** of much smaller numbers and the **power** into a **product**:

$$\log_b(m_1 \cdot m_2 \cdot m_3) > \log_b(10^{-(e_1+e_2+e_3)}) \tag{6}$$

$$\log_b(m_1) + \log_b(m_2) + \log_b(m_3) > -(e_1 + e_2 + e_3) \cdot \log_b(10) \tag{7}$$

3.4 Inverse Direction

As of now, we have only considered the **forward direction** of the triangular arbitrage, i.e. the forward cycle path. But since the rates are bidirectional, we can also exploit price inefficiencies in the **reverse direction**. Assuming we don't have the rates for the reverse direction, we can compute them as the inverse of the forward rates, and the formula becomes:

$$\frac{1}{R_1} \cdot \frac{1}{R_2} \cdot \frac{1}{R_3} > 1 \tag{1}$$

$$\frac{1}{R_1 \cdot R_2 \cdot R_3} > 1 \tag{2}$$

$$R_1 \cdot R_2 \cdot R_3 < 1 \tag{3}$$

$$\log_b(m_1) + \log_b(m_2) + \log_b(m_3) < -(e_1 + e_2 + e_3) \cdot \log_b(10) \tag{4}$$

As expected, the only difference is the direction of the inequality. This means that at any time there always exists a profitable arbitrage opportunity, either in the forward or reverse direction (unless the market is perfectly efficient). However in practice, not every opportunity is worth exploiting, wether due to **fees** or the **spread**[1].

3.5 The Spread Problem

At first, the two inequalities seem to differ only in the direction of the inequality. However, in practice, the two inequalities are not equivalent. In fact, due to the **spread**, we need to use different exchange rate values based on the direction of the trade. Specifically, we need to use the **ask prices** when buying and the **bid prices** when selling. So the inequalities become:

$$\log_b(m_{\text{bid}_1}) + \log_b(m_{\text{bid}_2}) + \log_b(m_{\text{bid}_3}) > -(e_1 + e_2 + e_3) \cdot \log_b(10) \tag{1}$$

$$\log_b(m_{\text{ask}_1}) + \log_b(m_{\text{ask}_2}) + \log_b(m_{\text{ask}_3}) < -(e_1 + e_2 + e_3) \cdot \log_b(10) \tag{2}$$

3.6 Threshold

Often times, to avoid false positives and very short-lived arbitrage opportunities, we need to set a profitability **threshold**. This threshold is a percentage value that represents the minimum profit we want to achieve from the arbitrage opportunity and allows us to filter out the noise. Let τ be the threshold as a decimal (e.g., 0.001 for 0.1%). Then the adjusted arbitrage inequalities become:

$$\log_b(m_{\text{bid}_1}) + \log_b(m_{\text{bid}_2}) + \log_b(m_{\text{bid}_3}) > -(e_1 + e_2 + e_3) \cdot \log_b(10) + \log_b(1 + \tau) \tag{1}$$

$$\log_b(m_{\text{ask}_1}) + \log_b(m_{\text{ask}_2}) + \log_b(m_{\text{ask}_3}) < -(e_1 + e_2 + e_3) \cdot \log_b(10) - \log_b(1 + \tau) \tag{2}$$

4 Algorithm

4.1 The choice of the base

Up to this point, we have considered the base b of the logarithm to be arbitrary. But in practice, we have to choose the one that best fits our needs.

- $\log_{10}$ allows for a simplification of the formula: $\log_{10}(10) = 1$.
- $\log_2$ can be computed with fast bit-shifts, and it is the natural choice for binary systems[3].

In this implementation, I chose the latter. This decision is motivated by both **efficiency** and **precision** gains. In fact the logarithm base 2 has some unique properties:

- the integer part corresponds to 63 minus the number of leading zeros in the binary representation of the value.
- since floating point numbers are naturally represented in base 2, there is virtually no loss of precision.

Furthermore, the small advantage of using base 10 becomes irrelevant since we can precompute the value of $\log 2(10)$ anyways.

4.2 Precomputation

Assuming exponents change very rarely, we can use an optimistic approach and **precompute** the entire right side of the inequality, storing it in a constant variable, raising an exception or even updating the precomputed constant only when the exponents change.

4.3 Fixed Point Representation

Since the logarithm produces a real number, I will use a **fixed point** representation to store the result. This allows for much faster additions and comparisons compared to floating point numbers. For reference, a float add takes about 4-5 cycles on modern CPUs[2], while a fixed point addition takes only 1 cycle. Specifically, since the maximum possible value of the logarithm is $log_2(2^{63}) = 63$ and our formula sums three of them together, we need $\lceil \log_2(64 * 3) \rceil = 8$ bits to store integers up to 192. The fractional part can be chosen based on the desired precision. At this point we have three options for the fixed point format:

- Q8.8 as 16 bits $\rightarrow 2^{-8} = 0.00390625$ step size.
- Q8.24 as 32 bits $\rightarrow 2^{-24} = 0.00000006$ step size.
- Q8.56 as 64 bits $\rightarrow 2^{-56} = 0.00000000000000014$ step size.

Out of the three, the second option offers the best balance between precision and size. For instance, a maximum error of step_size/2 = 0.0.00000006/2 = 0.00000003 in the logarithmic value corresponds to a relative error of $2^{0.00000003} - 1 \approx 2.1 \cdot 10^{-8} = 0.0000021\%$ in the original value, which is more than enough precision for our needs.

4.4 Log2

The actual implementation of the `log2` function relies on the **clz** (count leading zeros) intrinsic for the integer part, and a **lookup table** for the fractional part. The table is generated by sampling the logarithm function at regular intervals in the range $[0, 1]$. The **table size** directly influences the precision of the fractional part. Specifically, it determines the number of samples taken in the $[0, 1]$ range along the horizontal axis of the logarithm function.

TABLE_SIZE	Memory Usage	Horizontal Step Size	Vertical Step Size (approx.)
256	1 KB	0.00391	~ 0.00026
512	2 KB	0.00195	~ 0.00013
1024	4 KB	0.00098	~ 0.000068

Table 1: Trade-offs between memory usage and precision for various table sizes

A lookup table with **256 entries** is the best compromise between precision and memory usage. In fact, this brings down the step size from 0.00000006 to 0.00026, or a maximum error of 0.00026/2 = 0.00013, which corresponds to a relative error of $2^{0.00013} - 1 \approx 9 \cdot 10^{-5} = 0.009\%$ in the original value. This is a perfectly reasonable trade-off, considering that most arbitrage opportunities are above the 0.1% threshold. Furthermore, the error does not accumulate, since the additions that would follow in our formula are all performed on the fixed point numbers discussed earlier, which have a much greater margin of error. Considering that modern CPUs have **L1 caches** of at least 32 KB, a 1 KB lookup table will most likely always be present in the cache, making the lookup operation extremely fast

```cpp
template <uint8_t IntBits, uint8_t FracBits = 32 - IntBits>
constexpr FixedPoint<IntBits, FracBits> FixedPoint<IntBits, FracBits>::log2(const uint64_t value)
{
  static constexpr uint32_t TABLE_BITS = std::min(8, FracBits); //2^8 = 256 entries
  static constexpr uint32_t TABLE_SIZE = 1 << TABLE_BITS;
  static constexpr uint32_t SHIFT_AMOUNT = 63 - TABLE_BITS;
  static constexpr uint32_t MASK = TABLE_SIZE - 1;

  static constexpr std::array<uint32_t, TABLE_SIZE> log2_table = precomputeLog2Table<TABLE_SIZE>();

  const uint64_t leading_zeroes = __builtin_clzll(value);
  const uint32_t integer_part = 63 - leading_zeroes;
  const uint64_t shifted = value << leading_zeroes;
  const uint32_t table_index = (shifted >> SHIFT_AMOUNT) & MASK;
  const uint32_t frac_part = log2_table[table_index];

  const int32_t raw_value = (static_cast<int32_t>(integer_part) << FracBits) + frac_part;
  return FixedPoint<IntBits, FracBits>::fromRaw(raw_value);
}
```

In this implementation the **zero check** is avoided to prevent unnecessary **branching**. Since the logarithm of zero is undefined, it is safe and coherent to leave it undefined behavior. After all these considerations, this function is able to compile in just ~ 10 instructions, which is 3x faster than optimized standard log2 implementations[4].

5 Parallelization

References

[1] Wikipedia, *Bid–ask spread*, Available at: https://en.wikipedia.org/wiki/Bid%E2%80%93ask_spread.

[2] Agner Fog, *Instruction Tables: Lists of instruction latencies, throughputs and micro-operation breakdowns*, 2024. Available at: https://www.agner.org/optimize/instruction_tables.pdf.

[3] Sean Eron Anderson, *Bit Twiddling Hacks*, 2005. Available at: https://graphics.stanford.edu/~seander/bithacks.html.

[4] GNU C Library, *log2*, 2023. Available at: https://github.com/bminor/glibc/blob/master/sysdeps/ieee754/dbl-64/e_log2.c.