

Fast Arbitrage Detection

Claudio Raimondi

June 14, 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 to accurately and efficiently detect arbitrage opportunities.

Contents

1	Introduction	1
2	Visualization	2
3	Arbitrage Formula	2
3.1	Base Formula	2
3.2	Price Format	2
3.3	Logarithmic Form	2
3.4	Inverse Direction	2
3.5	The Spread Problem	3
3.6	Threshold	3
4	Algorithm	3
4.1	The choice of the base	3
4.2	Precomputation	4
4.3	Fixed Point Representation	4
4.4	Log2	4

1 Introduction

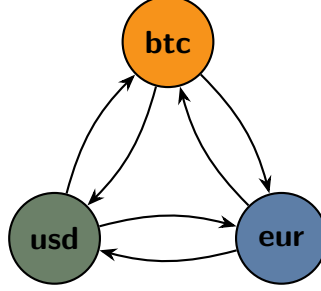
The process to exploit triangular arbitrage opportunities involves 3 generic steps, each crucial for ensuring the success of the strategy:

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 collects, 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:

$$R_1 \cdot R_2 \cdot R_3 > 1 \quad (1)$$

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:

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

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

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

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

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**:

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

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

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:

$$\frac{1}{R_1} \cdot \frac{1}{R_2} \cdot \frac{1}{R_3} > 1 \quad (1)$$

$$\frac{1}{R_1 \cdot R_2 \cdot R_3} > 1 \quad (2)$$

$$R_1 \cdot R_2 \cdot R_3 < 1 \quad (3)$$

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

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, whether 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:

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

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

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:

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

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

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 $\rightarrow 2^{-56} = 0.0000000000000000014$ step size.

Out of the three, the second option offers the best balance between precision and size. For instance, a maximum error of $\text{step_size}/2 = 0.00000006/2 = 0.00000003$ in the logarithmic value corresponds to a relative error of $0.00000003 \cdot \ln(2) \approx 0.0000021\%$ which is more than enough 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 $[1, 2]$. The **table size** directly influences the precision of the fractional part. Specifically, it determines the number of samples taken along the horizontal axis of the logarithm function. We can calculate the **relative error** on the original formula as follows:

$$\text{horizontal step size} = \frac{1}{\text{table size}} \quad (1)$$

$$\text{horizontal error} = \frac{\text{horizontal step size}}{2} \quad (2)$$

$$\text{vertical error} = \text{horizontal error} \cdot \max_{x \in [1, 2]} \left| \frac{d}{dx} \log_2(x) \right| \quad (3)$$

$$\text{accumulated error} = \text{vertical error} \cdot 3 \quad (4)$$

$$\text{relative error} = \ln(2) \cdot \text{accumulated error} \quad (5)$$

TABLE_SIZE	Memory Usage	Vertical Error	Relative Error
256	1 KB	~ 0.0028	$\sim 0.59\%$
512	2 KB	~ 0.0014	$\sim 0.29\%$
1024	4 KB	~ 0.0007	$\sim 0.15\%$

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

Considering that for maximum performance we want our table to always fit in **L1 cache**, these conclusions are not optimal yet. In fact, even with 4KB of memory, an error of 0.15% is

unacceptable. To address this, we can use **linear interpolation**^[4] between two adjacent table entries. Linear interpolation approximates the function between two points using a straight line. The error is determined by how much the actual function deviates from this linear approximation. For a smooth function $f(x)$, the interpolation error is bounded by:

$$\text{interpolation error} \leq \frac{h^2}{8} \max_{x \in [x_i, x_{i+1}]} |f''(x)| \quad (1)$$

where h is the step size between adjacent table entries. For $\log_2(x)$, we need the second derivative:

$$\frac{1}{x^2 \ln(2)} \quad (2)$$

The maximum absolute value of the second derivative in the range $[1, 2]$ occurs at $x = 1$:

$$\frac{1}{\ln(2)} \approx 1.44 \quad (3)$$

Therefore, the maximum interpolation error for the three \log_2 calculation becomes:

$$\text{accumulated error} = 3 \cdot \frac{h^2}{8} \cdot \frac{1}{\ln(2)} = \frac{3}{8 \ln(2) \cdot \text{table size}^2} \quad (4)$$

$$\text{relative error} = \ln(2) \cdot \text{accumulated error} = \frac{3}{8 \cdot \text{table size}^2} \quad (5)$$

TABLE_SIZE	Memory	Error (no interp)	Error (linear interp)	Improvement
256	1 KB	0.59%	$\sim 0.00057\%$	1035×
512	2 KB	0.29%	$\sim 0.00014\%$	2070×
1024	4 KB	0.15%	$\sim 0.000036\%$	4140×

Table 2: Comparison of lookup table approaches with and without linear interpolation

For a **1KB** table, we get over 1000× better precision, transforming an unacceptable 0.59% error into a negligible 0.00057% error.

```

template <uint8_t IntBits, uint8_t FracBits = 32 - IntBits>
constexpr FixedPoint<IntBits, FracBits> FixedPoint<IntBits, FracBits>::log2(const uint64_t value)
{
    static constexpr uint8_t TABLE_BITS = std::min(8u, FracBits); // 2^8 = 256 entries
    static constexpr uint16_t TABLE_SIZE = 1 << TABLE_BITS;
    static constexpr uint8_t SHIFT_AMOUNT = 63 - TABLE_BITS;
    static constexpr uint16_t MASK = TABLE_SIZE - 1;
    static constexpr uint8_t FRAC_BITS_FOR_INTERP = FracBits - TABLE_BITS;
    static constexpr uint8_t FRAC_SHIFT_OFFSET = SHIFT_AMOUNT - FRAC_BITS_FOR_INTERP;
    static constexpr uint16_t FRAC_MASK = (1u << FRAC_BITS_FOR_INTERP) - 1;

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

    const uint32_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_offset = (shifted >> FRAC_SHIFT_OFFSET) & FRAC_MASK;
    const uint32_t val0 = log2_table[table_index];
    const uint32_t val1 = log2_table[table_index + 1];
    const uint64_t diff = static_cast<uint64_t>(val1) - val0;
    const uint64_t prod = diff * frac_offset;
    const uint32_t interp = val0 + static_cast<uint32_t>(prod >> FRAC_BITS_FOR_INTERP);

    const int32_t raw_value = (static_cast<int32_t>(integer_part) << FracBits) + interp;
    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 the considerations above, this function is able to compile in just ~ 10 instructions, which is 3x faster than optimized standard log2 implementations^[5].

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] Wikipedia, *Linear interpolation*, Available at: https://en.wikipedia.org/wiki/Linear_interpolation.
- [5] GNU C Library, *log2*, 2023. Available at: https://github.com/bminor/glibc/blob/master/sysdeps/ieee754/dbl-64/e_log2.c.