

Fast Arbitrage Detection

Claudio Raimondi

June 11, 2025

Abstract

Triangular arbitrage offers opportunities to exploit price inefficiencies between three currency pairs. While several algorithms exist for arbitrage detection, their computational efficiency and accuracy can vary. In this report, I present a method that combines established graph-based algorithms with **fixed-point** representation techniques to detect triangular arbitrage efficiently and accurately. The approach leverages the **Bellman-Ford algorithm** for cycle detection, enhanced by numerical optimizations to minimize floating-point errors. Experimental results demonstrate that this method improves detection speed and reliability, making it suitable for real-time financial applications.

Contents

1	Introduction	1
2	Representation	2
3	Arbitrage Formula	2
3.1	Base Formula	2
3.2	Floating-Point Representation	2
3.3	Logarithmic Form	2
3.4	Inverse Direction	3
4	Algorithm	3
4.1	Precomputation	3
4.2	Fixed-Point Numbers	3
5	Bellman-Ford Algorithm	4
6	Relative Error	4
7	Optimization	4

1 Introduction

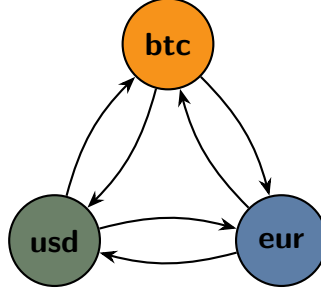
The process to exploit triangular arbitrage opportunities involves 3 generic steps, each crucial for ensuring the success of the strategy:

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 Representation

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 Floating-Point Representation

In practice, each price is a floating-point number, and often times, if the *data collection* layer is serious about efficiency, they are provided as **mantissa** (m) and **exponent** (e) separately. So our inequality 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:

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

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

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

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

$$m_1 \cdot m_2 \cdot m_3 < \frac{1}{10^{e_1+e_2+e_3}} \quad (5)$$

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

$$\log_b(m_1 \cdot m_2 \cdot m_3) < \log_b(10^{-(e_1+e_2+e_3)}) \quad (7)$$

$$\log_b(m_1) + \log_b(m_2) + \log_b(m_3) < -(e_1 + e_2 + e_3) \cdot \log_b(10) \quad (8)$$

4 Algorithm

Once the formula is derived, the algorithm is just a matter of applying the two inequalities. From a first analysis, it seems that the only difference between the forward and reverse inequalities is their direction. However, we have to account for the **spread**, which is the difference between the bid and ask prices. This means that we have to consider the **ask** price for the forward direction and the **bid** price for the reverse direction. Therefore, we still have to compute both inequalities separately.

4.1 Precomputation

Assuming exponents change very rarely, we can use an optimistic and precompute the entire right side of the inequality, storing it in a constant variable, and revert (or possibly shutdown) if the exponents change.

4.2 Fixed-Point Numbers

On the left side of the inequality, we have another problem: the logarithms produce floating-point numbers.

Up to this point, and for the sole purpose of simplifying the formula, we have considered the base b of the logarithm to be arbitrary. But in reality things are more complicated:

- A larger base b will result in a smaller logarithm, which can lead to a loss of precision.
- \log_2 is often faster to compute on most hardware, thanks to bit-shifting operations.
- \log_{10} allows for a simplification of the formula: $\log_{10}(10) = 1$.

- 5 Bellman-Ford Algorithm
- 6 Relative Error
- 7 Optimization