CS765 Assignment 3 : Report

Atharva Bendale
22B0901
atharvaab@cse.iitb.ac.in

Vardan Verma
22B0902
22B0902@iitb.ac.in

Vishal Bysani
22B1061
vishalbysani@cse.iitb.ac.in

April 16, 2025

Implementation Details

DEX.sol

- `swap`: Implements a decentralized exchange with constant product formula (x * y = k)

```
function swap(string memory token , uint256 amount)
    public returns (uint256) {
    uint256 newTransferAmount = (amount * 997)/1000;  // 0.3% fee
    if (keccak256(abi.encodePacked(token)) ==
    keccak256(abi.encodePacked("TokenA"))) {
        uint256 newAmountA = getTokenABalance() + newTransferAmount;
        uint256 newAmountB = (getTokenABalance() * getTokenBBalance())
        / newAmountA;
        uint256 transferAmountB = getTokenBBalance() - newAmountB;
        // ... rest of the function
    }
}
```

- `depositTokens`: Allows users to add liquidity and receive LP tokens

```
        function depositTokens(uint256 amt1, uint256 amt2) public {
            if(!isRatioClose(amt1, amt2)) {
                return;
            }
            uint256 LPTokensReward = (tokenABalance == 0) ? 10**18 :
                (amt1 * 10**18) / tokenABalance;
            LPTok(LPTokens).generateTokens(LPTokensReward, sender);
        }
```

- `withdrawTokens`: Enables LP token holders to withdraw their share of tokens

```
        function withdrawTokens(uint256 LPAmt) public returns(uint256, uint256) {
            (success, balance) = LPTok(LPTokens).burn(LPAmt, sender);
            if (success) {
                uint256 balanceA = getTokenABalance();
                uint256 balanceB = getTokenBBalance();
                IERC20(tokenA).transfer(sender,
                (balanceA*LPAmt)/totalTokens);
                IERC20(tokenB).transfer(sender,
                (balanceB*LPAmt)/totalTokens);
                return ((balanceA*LPAmt)/totalTokens,
                (balanceB*LPAmt)/totalTokens);
            }
        }
```

- Implements ratio checks to ensure liquidity providers maintain the correct token ratio

LPTokens.sol

- Implements ERC20 LP tokens representing liquidity pool shares

```
    contract LPToken is ERC20 {
        address internal DexAddr;
        uint256 internal totalTokens = 0;
        mapping (address => uint256) internal _balance;

        constructor(address DEXAddr) ERC20("LPToken", "LPT") {
            DexAddr = DEXAddr;
        }
    }
```

- Security measures:

– Access control for minting/burning operations

```
        function generateTokens(uint256 _amount, address receiver)
        public returns (bool success, uint256) {
            if (msg.sender != DexAddr) return (false, _balance[receiver]);
            return (true, _mint(_amount, receiver));
        }

        function burn(uint256 _amount, address owner)
        public returns (bool success, uint256) {
            if(msg.sender != DexAddr) return (false, _balance[owner]);
            if (_balance[owner] < _amount) return (false, _balance[owner]);
            _balance[owner] -= _amount;
            totalTokens -= _amount;
            return (true, _balance[owner]);
        }
```

Arbitrage.sol

- Implements arbitrage between two DEXes

```
    function getArbitrargeProfit(address _dex1, address _dex2, bool ABA)
    internal returns (uint256) {
        if (ABA) {
            uint256 spotPriceA1 = (DEX_Interface(_dex1).spotPrice("TokenA") *
                (1000-fee))/1000;
            uint256 spotPriceB2 = (DEX_Interface(_dex2).spotPrice("TokenB") *
                (1000-fee))/1000;
            uint256 initialTokenA = 1*10**18;
            uint256 finalTokenA = (spotPriceA1*spotPriceB2)/(10**18);
            if(finalTokenA < initialTokenA) return 0;
            return finalTokenA - initialTokenA;
        }
    }
```

- Main arbitrage execution logic:

```
    function arbitrage() external {
        // Compare spot prices and swap DEX addresses if needed
        if (spotPriceA1 < spotPriceA2) {
            address tmp = DEX1;
            DEX1 = DEX2;
            DEX2 = tmp;
        }

        // Calculate profits in both directions
        uint256 profitABA = getArbitrargeProfit(DEX1, DEX2, true);
        uint256 profitBAB = getArbitrargeProfit(DEX2, DEX1, false);

        // Execute most profitable path
        if(profitABA >= profitBABinTermsA) {
            // Execute A->B->A path
            IERC20(tokenA).approve(DEX1, 1*10**18);
            uint256 receivedB = DEX_Interface(DEX1).swap("TokenA", 1*10**18);
            IERC20(tokenB).approve(DEX2, receivedB);
            uint256 receivedA = DEX_Interface(DEX2).swap("TokenB", receivedB);
            // ... profit calculation and event emission
        }
    }
```

Handling of Floating-Point Values in Solidity and Sanity Checks

- Floating-point values are represented by multiplying them by 10^{18} and storing the result as a `uint256`. This approach provides a precision of up to 18 decimal places.

- When calculating formulas with floating point values, we first multiply and then divide to preserve precision.

- We also included sanity checks for subtractions in many places to ensure safe arithmetic.

```
    if (receivedA < 1*10**18) {
        emit ArbitrageResult(1*10**18, 0, "TokenA");
        return;
    }
    uint256 profit = receivedA - 1*10**18;
```

```
    if(finalTokenA < initialTokenA) return 0;
    uint256 arbitrageProfit = finalTokenA - initialTokenA;
```

- We included many other logical sanity checks to avoid inconsistencies.

```
    require(_tokenA != address(0) && _tokenB != address(0),
    "Invalid token address");
    require(_tokenA != _tokenB, "Tokens must be different");
```

To ensure that the addresses given correspond to different tokens in DEX

Theoretical Questions

Which address(es) should be allowed to mint/burn the LP tokens?

Only the DEX contract should be permitted to mint and burn LP tokens. This ensures that LP tokens, which represent a proportional share of the liquidity pool, are issued or destroyed strictly during liquidity addition or removal. This constraint maintains the integrity of ownership and prevents unauthorized inflation or reduction of LP tokens.

```
function generateTokens(uint256 _amount, address receiver)
    public returns (bool success, uint256) {
    if (msg.sender != DexAddr) return (false, _balance[receiver]);
    return (true, _mint(_amount, receiver));
}
```

Tokens can only be minted by DEX

```
function burn(uint256 _amount, address owner)
        public returns (bool success, uint256) {
        if(msg.sender != DexAddr) return (false, _balance[owner]);
        if (_balance[owner] < _amount) return (false, _balance[owner]);
        _balance[owner] -= _amount;
        totalTokens -= _amount;
        return (true, _balance[owner]);
    }
```

Tokens can only be burned by DEX

In what way do DEXs level the playing ground between a powerful and resourceful trader (HFT/institutional investor) and a lower resource trader (retail investors)?

Decentralized exchanges (DEXs) democratize trading by offering transparent pricing mechanisms via automated market makers (AMMs) and equal access to liquidity pools without intermediaries. Unlike centralized exchanges, where high-frequency traders (HFTs) can leverage order book insights or enjoy preferential access, DEXs apply the same rules (such as the constant product formula) to all users. This enforces fairness and reduces barriers for retail investors, enabling a more equitable trading environment.

Suppose there are many transaction requests to the DEX sitting in a miner's mempool. How can the miner take undue advantage of this information? Is it possible to make the DEX robust against it?

Miners can exploit the mempool through front-running and back-running. In front-running, a miner inserts their own transaction before a large swap to benefit from anticipated price movements. In back-running, they execute trades after a large swap to exploit price rebounds. While complete protection is difficult due to miners' control over transaction ordering, mechanisms such as commit-reveal schemes, batch auctions, and slippage protection can mitigate these risks. These strategies obscure transaction intent or restrict execution to acceptable price ranges, reducing vulnerability to such attacks.

How do gas fees influence the economic viability of the entire DEX and arbitrage?

Gas fees are critical to the economic viability of decentralized exchanges. They increase the cost of executing transactions, which can render small trades or narrow-margin arbitrage opportunities unprofitable. For arbitrageurs, any potential gain must exceed the associated gas costs to justify the trade. DEXs that are gas-efficient (e.g., by optimizing smart contract operations) tend to be more viable, as they lower transaction costs and attract more active users.

Could gas fees lead to undue advantages to some transactors over others? How?

Yes, gas fees can create inequalities among users. Wealthier participants can afford to pay higher gas fees, enabling them to prioritize their transactions during times of congestion. This allows them to front-run others or secure trades faster, which is especially advantageous in time-sensitive situations like arbitrage. On the other hand, retail users with limited budgets may experience delays or transaction failures, leading to a less fair trading environment.

What are the various ways to minimize slippage in a swap?

Slippage can be minimized through several strategies:

- **Increasing pool liquidity:** Larger reserves dampen price fluctuations caused by trades.
- **Reducing trade size:** Smaller trades result in less deviation from the expected price.
- **Slippage protection:** Trades can be programmed to revert if the price deviates beyond a specified threshold.
- **Trading during low volatility:** Avoiding trades during periods of high price swings helps maintain stable rates.

- **Routing through multiple pools:** Splitting trades across multiple liquidity pools can reduce the price impact.

Slippage vs Trade Lot Fraction

The theoretical formula obtained for this is:

$$\text{Slippage} = \left(\frac{1 - \text{fees}}{1 + (1 - \text{fees}) \times \text{k}} - 1 \right) \times 100\% \tag{1}$$

Here k is 'Trade Lot Fraction' and fees is 0.003 in this case. We can observe that slippage is always negative for all swaps.

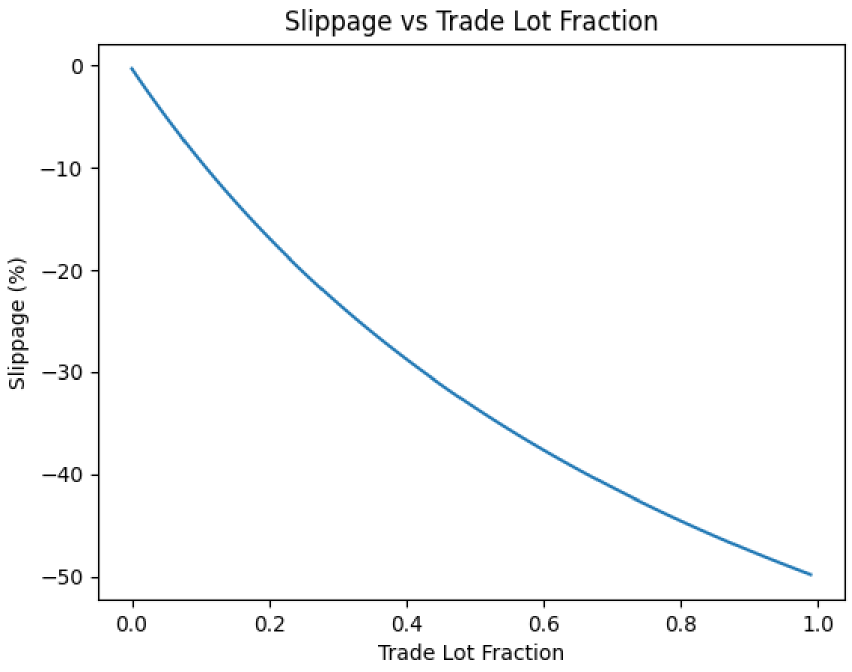

Figure 1: Slippage vs Trade Lot Fraction

Plots

Slippage

Our simulation allows a trade lot fraction of maximum 10%, we can observe from the above graph that this fraction restricts slippage to a minimum of -9% (approx), the below graph is consistent with these calculations.

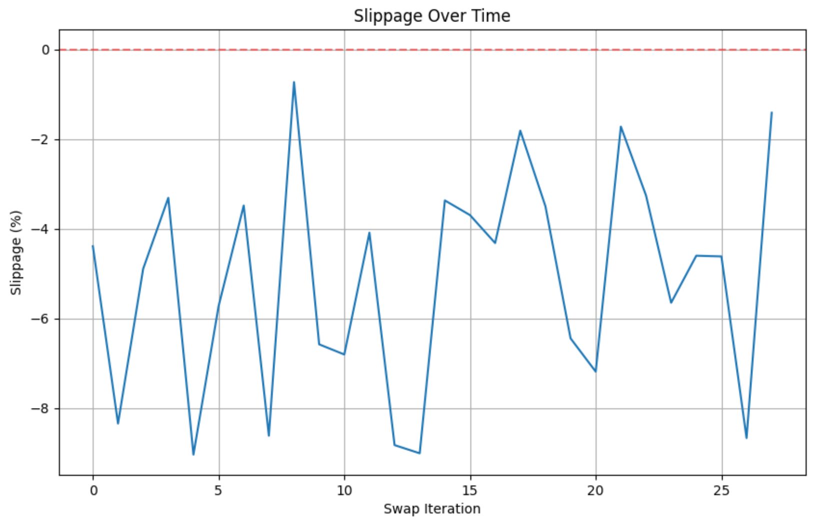

Figure 2: Slippage

LPToken Distribution

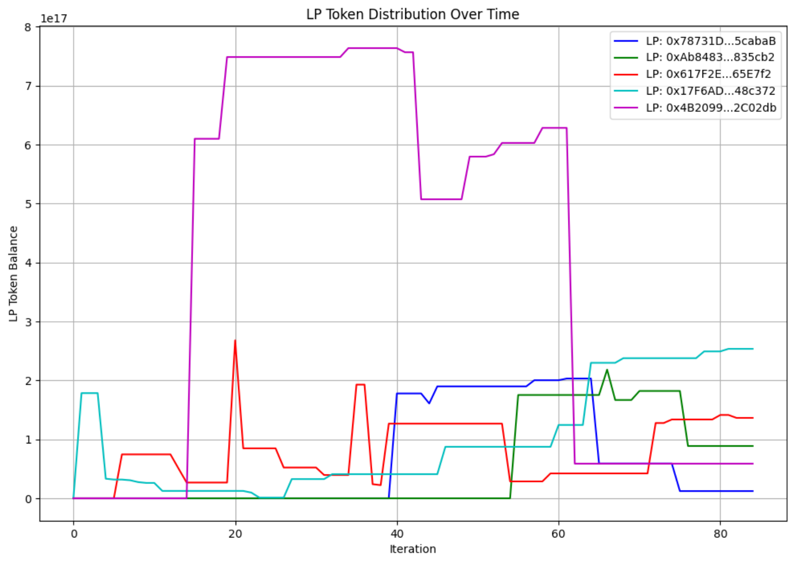

Figure 3: LPToken Distribution

Spot Price

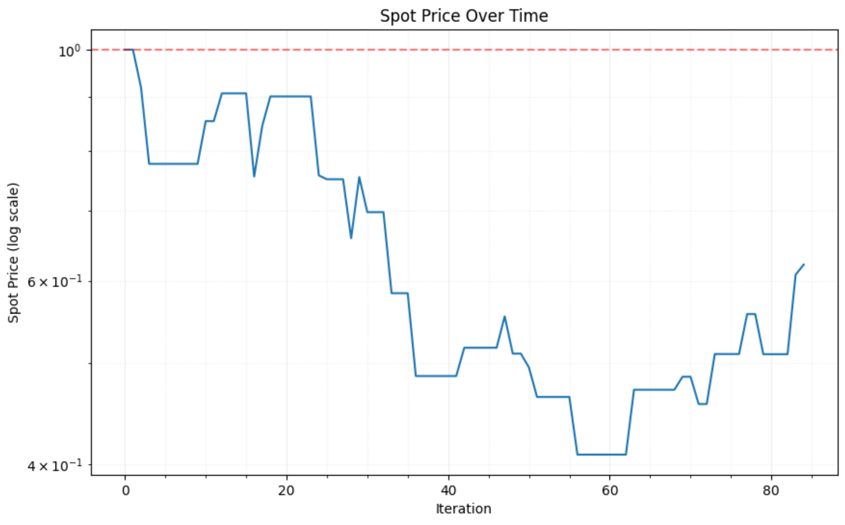

Figure 4: Spot Price

Reserve Ratio

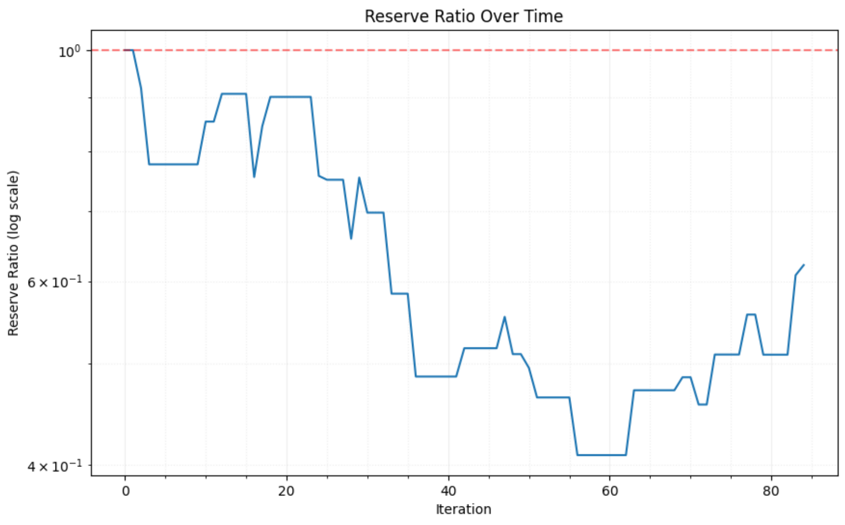

Figure 5: Reserve Ratio

Total Value Locked

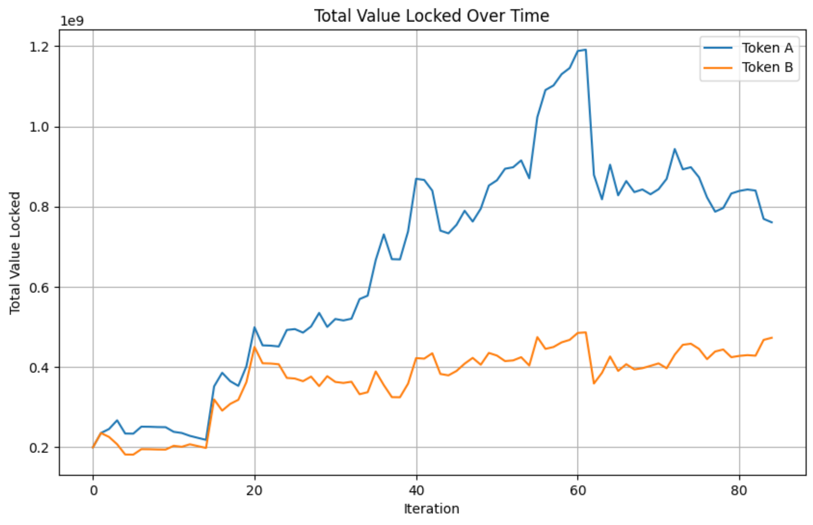

Figure 6: Total Value Locked

Trading Volume

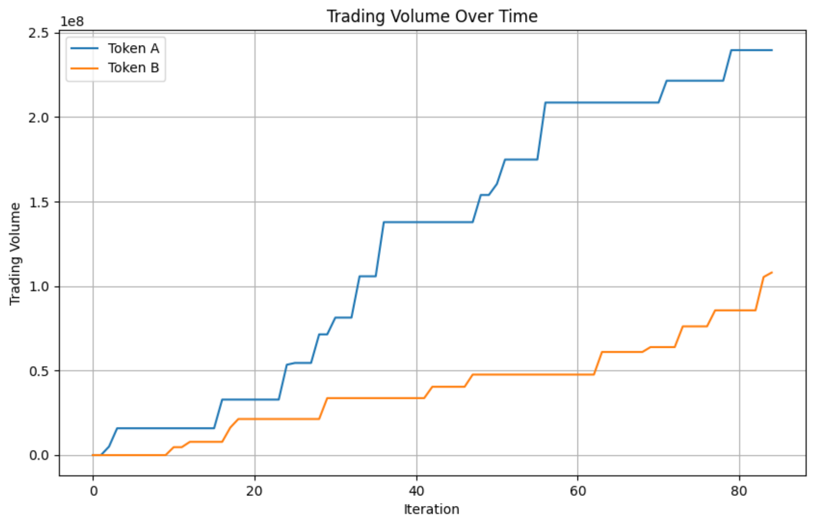

Figure 7: Trading Volume

Fees Collected

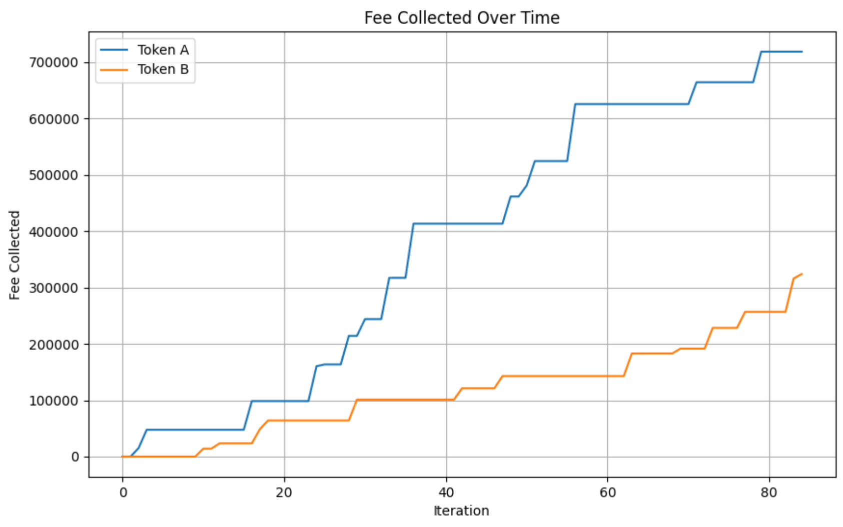

Figure 8: Fees Collected

Arbitrage Distribution

Profitable Arbitrage Execution

We simulate a scenario involving two different decentralized exchanges (DEXs). In the first DEX, liquidity providers (LPs) deposit tokenA and tokenB in a 1 : 1 ratio, while in the second DEX, the same tokens are deposited in a 1 : 2 ratio. This imbalance creates an arbitrage opportunity, and our simulation results confirm the profitability of executing arbitrage in this setup.

Failed Arbitrage

We simulate another scenario with two different DEXs, where LPs deposit tokenA and tokenB in a 1 : 1 ratio on both exchanges. Since the price ratios are identical, no arbitrage opportunity arises, and our simulation results validate this outcome.