Homework 3

Building your own Decentralized Exchange

Chaitanya Garg (210050039), Omm Agrawal (210050110), Pulkit Goyal (210050126)

16th April 2025

1 Introduction

Decentralized Exchange (DEX) is a platform that facilitates direct cryptocurrency trades between users via smart contracts. This report presents the implementation of a simplified DEX based on the Automated Market Maker (AMM) model, inspired by popular protocols such as Uniswap.

The objective of this assignment is to understand, design, and deploy smart contracts that simulate the core operations of a DEX on the Ethereum blockchain. These operations include deploying ERC-20 tokens, building a constant product AMM (i.e., $x * y = k$) for swapping, tracking liquidity provider (LP) shares through LP tokens, and enabling arbitrage opportunities across multiple DEX instances.

The project was developed using Solidity, tested on Remix IDE, and simulated through JavaScript scripts to mimic realistic trading and liquidity provision scenarios. The implementation focuses not only on functional correctness but also on ensuring safe arithmetic, proper validations, and a practical grasp of economic concepts like price discovery, slippage, and arbitrage. Through this assignment, we explore how DeFi applications empower both institutional and retail users by offering transparent and permission-less financial instruments.

2 Implementation Description

2.1 Token.sol

The `Token.sol` contract implements a standard ERC-20 token using OpenZeppelin's `ERC20` library. This token contract serves as the base for creating both TokenA and TokenB, which are used in the liquidity pools of the DEX.

- The contract inherits from OpenZeppelin's `ERC20` implementation, ensuring compliance with the ERC-20 standard and the availability of tested, secure token functionality.
- The constructor takes an `initialSupply` parameter and mints that many tokens to the deployer's address.
- The token is initialized with the name **``Token''** and the symbol **``TK''**, which can be reused with different initial supplies to create distinct ERC-20 tokens (i.e., TokenA and TokenB).
- **Floating Point Arithematics:** Although not enforced, we treat uint256 value 10^6 as 1 token. In our simulations. This ensures that any rounded error made in any calculation is insignificant compared to 1 token, and hence floating point arithematics is safely emulated using integer arithematics.

2.2 LPToken.sol

The `LPToken.sol` contract implements a custom ERC-20 token representing liquidity provider (LP) shares in the DEX. It is built using OpenZeppelin's `ERC20` and `Ownable` contracts to ensure standardized and secure token behavior and access control.

- The contract inherits from both `ERC20` and `Ownable`. The `Ownable` pattern ensures that only the DEX contract (set as the owner) can mint or burn LP tokens.
- The token is initialized with the name ``LP Token'' and symbol ``LPT''.
- The `mint()` and `burn()` functions can only be called by the contract owner (i.e., the DEX contract), enforcing permissioned control over LP token issuance and destruction.
- The LP tokens represent a user's proportional share of the liquidity pool and are key to tracking ownership and withdrawals.

This design ensures that liquidity provision and removal are tightly controlled and transparent, maintaining the integrity of the pool share distribution.

2.3 DEX.sol

The `DEX.sol` contract implements an Automated Market Maker (AMM)-based Decentralized Exchange supporting two tokens. It enables liquidity provision, token swapping, and LP token management, while enforcing safeguards to ensure robustness, correctness, and security of operations.

2.3.1 Core Components

- **Tokens:** The DEX operates on two ERC-20 tokens, `tokenA` and `tokenB`, which are specified at deployment. LP tokens are issued using the `LPToken` contract.
- **Liquidity Pools:** The contract maintains reserves (`reserveA` and `reserveB`) corresponding to the two tokens. These reserves are automatically updated after every liquidity and swap operation.
- **LP Token Logic:** Liquidity providers receive LP tokens proportional to their contribution. They can burn these tokens to retrieve their share of the pool along with accumulated swap fees.
- **Fee System:** A swap fee of 0.3% is applied on every trade/swap. These fees are accumulated and proportionally distributed among all active LPs.
- **Data Structures:** The contract uses `EnumerableSet` to efficiently track and iterate over current liquidity providers, and maintains per-address mappings to manage accumulated fees.

2.3.2 Swap Fee Distribution Design Choice

In our simulation, gas costs are deliberately ignored. However, in real-world deployments on Ethereum or similar blockchains, every transaction incurs a gas fee proportional to its computational complexity. Directly transferring each liquidity provider's share of the swap fee at the time of every trade would result in substantial gas overhead, especially since swap fees are typically very small. This would make such a strategy highly inefficient, as the majority of the fee could be consumed by the gas cost of distribution itself.
To address this, we adopt a deferred fee distribution model. The collected swap fees are retained within the DEX contract, and a running record is maintained of the fees owed to each liquidity provider. These

recorded balances are kept separate from the pool reserves used in price calculations, ensuring that fee holdings do not distort the exchange rates or reserve ratios.
Whenever a liquidity provider initiates a withdrawal of any amount of LP tokens, the DEX transfers not only their proportional share of the token reserves but also the accumulated fees owed to them up to that point. This design strikes a balance between accurate reward allocation and gas efficiency.

2.3.3 Key Functionalities

- **addLiquidity():** Accepts equal-ratio deposits of `tokenA` and `tokenB`, updating reserves and minting LP tokens. The first liquidity provider sets the initial pool ratio.
- **removeLiquidity():** Burns the specified amount of LP tokens and transfers the provider's share of tokens and fees back. This function is protected by the `noReentranct` modifier to prevent reentrancy attacks.
- **swap():** Enables token swaps based on the constant product formula. It calculates the output amount factoring in the 0.3% fee and updates internal reserves post-swap. The collected fee is distributed across LPs.
- **distributeFee():** Proportionally distributes the fee collected from a swap to all active liquidity providers, updating their personal balances and the global fee state.
- **Utility Functions:** Functions such as `getReserve()`, `spotPrice()`, `getSpotPriceAtoB()`, `getSpotPriceBtoA()`, and `get_swaps_vol()`, `get_total_fees()` provide insights into the pool state and trading statistics.

2.3.4 Security and Safeguards

The contract employs multiple protective patterns and runtime checks to ensure safe operation:

- `require` **Checks:**
  - Ensures non-zero input amounts in liquidity and swap functions.
  - Validates that tokens are correctly paired and distinct in swap operations.
  - Confirms that sufficient user balances and contract reserves exist before transfers.
  - Verifies adherence to the expected token ratio when adding liquidity (with a small allowed deviation/floating point error).
  - Rejects invalid token queries in `getReserve()`.
- **Reentrancy Protection:** A custom modifier `noReentranct` prevents nested calls to `removeLiquidity()`, mitigating potential reentrancy attacks.
- **Ownership Restrictions:** LP token minting and burning can only be performed by the DEX contract, which owns the `LPToken` instance. This ensures only authorized mint/burn actions occur.
- **Fee Accounting Safety:** Fee balances are individually tracked per provider and reset after withdrawal. Total fee pool variables ensure accurate reserve calculations.

2.3.5 Mathematical Operations

- The `sqrt()` function implements the Babylonian method to compute square roots and is used during the initial issuance of LP tokens. The first liquidity provider determines the initial ratio between the two tokens, and the number of LP tokens to be minted must reflect a balanced representation of both reserves. Choosing a fixed value for LP token issuance could lead to scaling issues when the initial token amounts are either too small or too large. To address this, we mint an amount of LP tokens equal to the geometric mean of the deposited TokenA and TokenB amounts, which ensures proportionality and avoids imbalance. The square root calculation is essential for this geometric mean.
- Output amounts in swaps are derived using the constant product formula:

$$output = \texttt{outputReserve} - \frac{\texttt{inputReserve} \times \texttt{outputReserve}}{\texttt{inputReserve} + \texttt{inputAmount} - \texttt{feeAmount}}$$

2.3.6 Conclusion

The `DEX.sol` contract forms the backbone of the decentralized exchange, integrating liquidity, trading, and accounting functionalities while ensuring safety through well-audited libraries, rigorous state checks, and strong encapsulation of privileges.

2.4 arbitrage.sol

The `arbitrage.sol` contract implements an arbitrageur that detects and executes profitable arbitrage opportunities between two decentralized exchanges (DEXes). It leverages interfaces to communicate with the DEX contracts and relies on rigorous checks and fee adjustments to guarantee safe and profitable trades.

2.4.1 Key Components and Interfaces

- **DEX Interface:** An `IDEX` interface is defined to interact with external DEX contracts. This interface includes:
  - `spotPrice()`, which returns the current reserves (reserveA and reserveB) of the DEX.
  - `swap()`, which executes token swaps.
  - `tokenA()` and `tokenB()`, which return the associated ERC-20 token instances.
- **Ownership:** The `Arbitrageur` contract inherits from OpenZeppelin's `Ownable`, restricting critical functions (such as executing arbitrage and updating thresholds) solely to the contract owner.
- **Parameters:**
  - A swap fee of 0.3% (represented as 30/10000) is consistently applied, matching the fee logic used in the DEXes.
  - A minimum profit threshold (defaulting to 0.05% or 5/10000) ensures that only transactions providing sufficient net gain are executed.

2.4.2 Arbitrage Detection Logic

The contract contains two internal functions to detect arbitrage opportunities in different trading directions:

- `detectArbitrageA_B_A`: This function considers a strategy where a trader swaps Token A for B on one DEX and then swaps back from B to A on the other DEX. It computes:

  - The effective output of Token B from the initial swap, accounting for the swap fee.
  - The final amount of Token A after swapping back, again deducting the respective fee.

  If the final amount exceeds the original investment by more than the minimum profit threshold, the function returns the net profit; otherwise, it returns zero.

- `detectArbitrageB_A_B`: This function mirrors the previous one but starts with Token B, swapping it for A and then back to B using the respective DEXes. It follows an analogous computation process to determine if the arbitrage would be profitable.

2.4.3 Arbitrage Execution Process

- `executeArbitrage`: Once an arbitrage opportunity is detected, this function carries out the arbitrage transaction:
  1. It first transfers the required input tokens (from the owner's account) to the contract.
  2. It approves and executes the first swap on the selected DEX.
  3. After receiving the swapped tokens, it approves and performs the reverse swap on the second DEX.
  4. Finally, it transfers the resultant tokens back to the owner.
- **Helper Functions:**
  - `helper1` and `helper2` are internal functions that simulate arbitrage trades in two different directional approaches. They inspect balances, reserve ratios, and calculate potential profits. Depending on which direction yields a higher computed profit (and exceeds the minimum profit threshold), the corresponding arbitrage is executed.
- `calculateAndExecuteArbitrage`: This external function, accessible only by the owner, triggers the arbitrage detection by calling the helper functions sequentially. If no profitable opportunities are detected, the transaction is reverted.

2.4.4 Security and Safeguards

The contract includes several important safety checks and design considerations:

- `require` **Statements:**
  - Ensure that input amounts are greater than zero.
  - Verify that the reserves in both DEXes are valid (non-zero) before performing calculations.
  - Confirm that token addresses obtained from the DEX contracts are valid.
  - Check that the owner has a sufficient token balance before initiating arbitrage.
- **Approval Mechanism:** Prior to executing any swap operation, tokens are approved for transfer by the respective DEX contract. This guard ensures that the contract cannot inadvertently overspend tokens.
- **Fee and Profit Checks:**
  - The swap fee is meticulously deducted during each swap simulation.
  - The profit is computed only if the net output exceeds the input by more than the prescribed minimum profit threshold, avoiding losses from negligible trades.
- **Owner Restriction:** Critical functions (`calculateAndExecuteArbitrage`) are restricted to the owner, preventing unauthorized use of arbitrage logic.

2.4.5 Conclusion

The `arbitrage.sol` contract extends the functionality of the DEX system by introducing a systematic approach for detecting and exploiting price discrepancies between two DEXes. Its modular design, extensive use of safeguards, and careful fee calculations ensure that arbitrage transactions are executed only when profitable, while minimizing the risks and potential losses due to transaction fees and market fluctuations.

3 Testing & Simulation

3.1 simulate_DEX.js

The `simulate_DEX.js` script is responsible for testing the deployed DEX contracts by simulating user interactions in a controlled environment. It connects to the already deployed TokenA, TokenB, LPToken, and DEX contracts using Web3.js and Remix artifacts.
The script first initializes the test accounts, assigning the first 5 as liquidity providers (LPs) and the next 8 as traders. Token balances are redistributed to ensure fair participation. It then runs a loop over a fixed number of iterations (`N = 100`), where in each iteration one of three actions is randomly chosen:

- **Deposit**: A randomly selected LP provides liquidity to the DEX by depositing TokenA and TokenB, either in any ratio (if the pool is empty) or maintaining the reserve ratio (if the pool has liquidity).
- **Withdraw**: A randomly selected LP removes a random portion of their LP tokens, triggering a proportional withdrawal of TokenA and TokenB from the pool.
- **Swap**: A randomly selected trader swaps TokenA for TokenB or vice versa. The swap amount is randomized but capped to limit slippage and maintain stability.

For each action, the script logs key metrics:

- `Total Value Locked` (TVL) in TokenA units
- `Reserve ratio` (TokenA/TokenB)
- `Spot price`
- `Swap volumes` for both tokenA and TokenB
- `Fees collected` in both TokenA and TokenB
- Estimated `Slippage` based on the difference between expected and actual exchange rates

Due to lack of file system and file write operations in remix, The metrics are printed on console in CSV format at end of simulation, and needs to be manually copied and pasted to a CSV file for offline analysis and visualization.
The script enables stress testing the DEX and studying emergent behaviors such as impermanent loss, price volatility, and fee accumulation in a dynamic environment.

3.2 simulate_arbitrage.js

The `simulate_arbitrage.js` script is designed to test arbitrage opportunities between two decentralized exchanges (DEX1 and DEX2) by using a `Arbitrageur` smart contract. It connects to the already deployed TokenA, TokenB, LPToken1, LPToken2, DEX1, DEX2 and Arbitrageur contracts using Web3.js and Remix artifacts.
The script first initializes the test accounts, assigns the first account as liquidity provider (LP) and the second accound as trader. The script requires and ensures that the owner/deployer of the Arbitrageur contract is the trader. The Token balances are equally distributed among the 2 to allow them to participate in transactions.

1. **Profitable Arbitrage:**
    - LP adds liquidity to DEX1 and DEX2 such that a price difference exists between them.
    - Trader approves the arbitrage contract to spend their tokens.
    - The arbitrage contract's `calculateAndExecuteArbitrage()` method is invoked.
    - Trader's TokenA and TokenB balances are logged before and after to verify profit.
2. **Unprofitable Arbitrage:**
    - LP adds liquidity in a way that introduces only a negligible price difference.
    - The same arbitrage execution steps are followed.
    - This tests whether the arbitrage logic avoids or executes a loss-inducing trade.
    - Trader's TokenA and TokenB balances are logged before and after to verify no loss.

Please note that the script does print an error in case of Unprofitable Arbitrage and is expected behaviour as The Arbitrageur contract reverts the transaction when no opportunity exists.
The goal is to simulate both profitable and unprofitable arbitrage scenarios and observe how token balances change as a result.
This scripts shows how the Arbitrageur behaves when arbitrage opportunity may or may not exist And observe how a trader can profit based of exchange rate difference between exchanges.

4 Results & Analysis

4.1 DEX

Liquidity

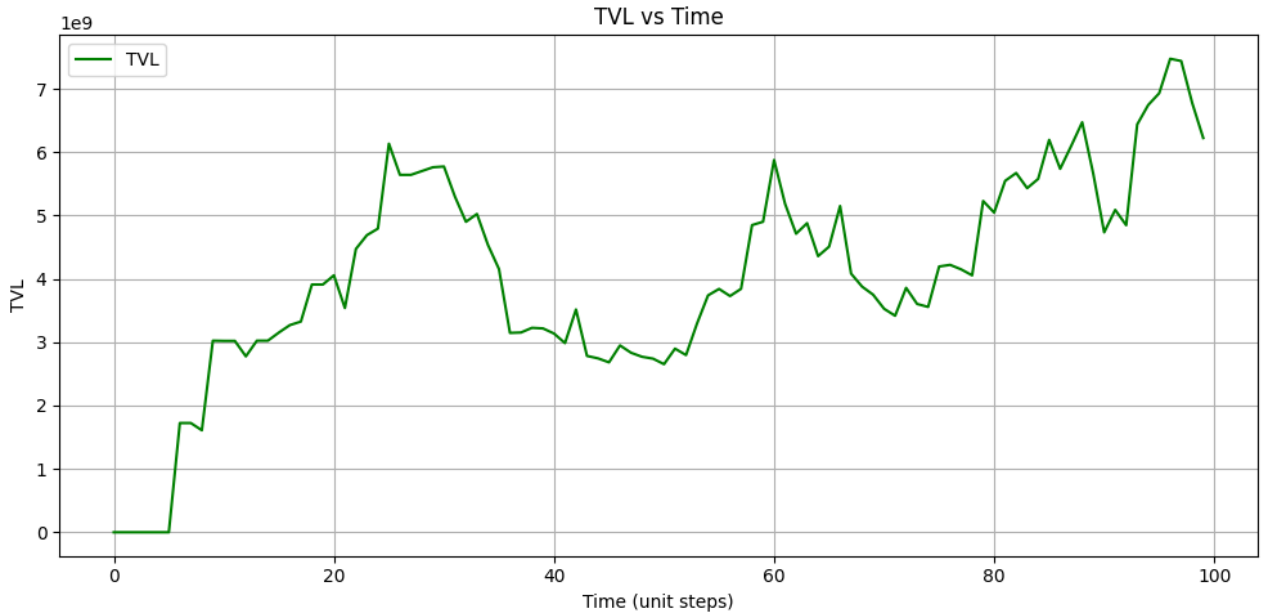

Figure 1: Total Value Locked in DEX in terms of Token A.

Since addition and removal of Liquidity is random, Figure 1 is mostly random.

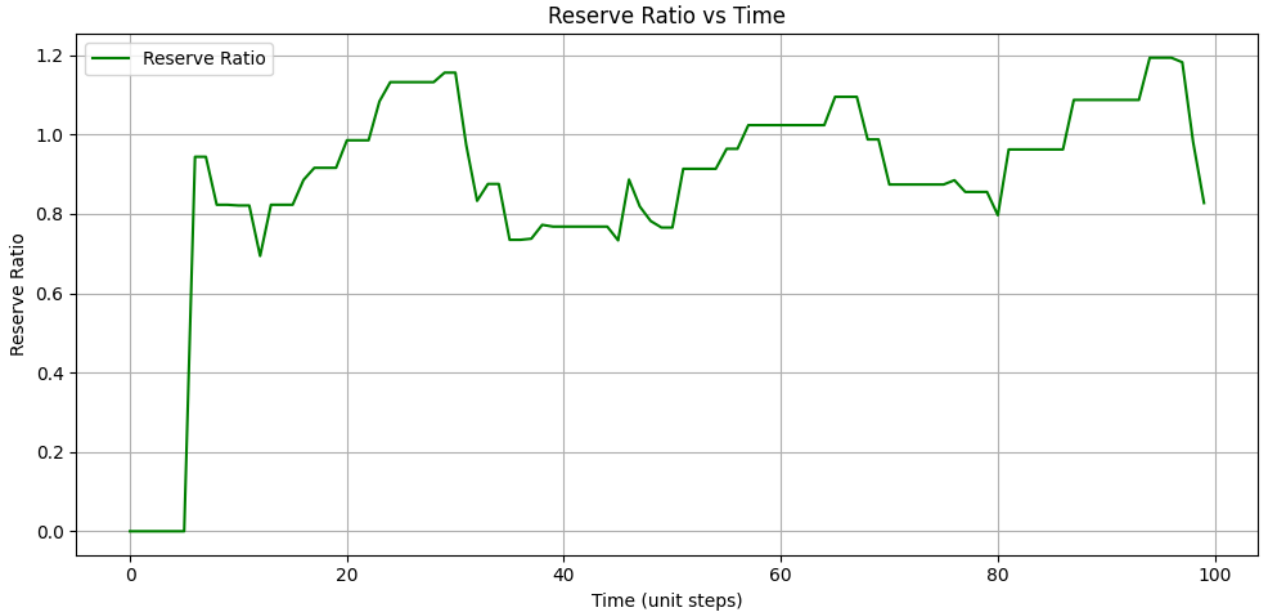

Figure 2: Ratio of reserves (TokenA/TokenB) in DEX.

Figure 2 indicates that the Reserve Ratio fluctuates around 1 in our simulation, which is expected as Operations on Token A and B are symmetric and uniformly random.

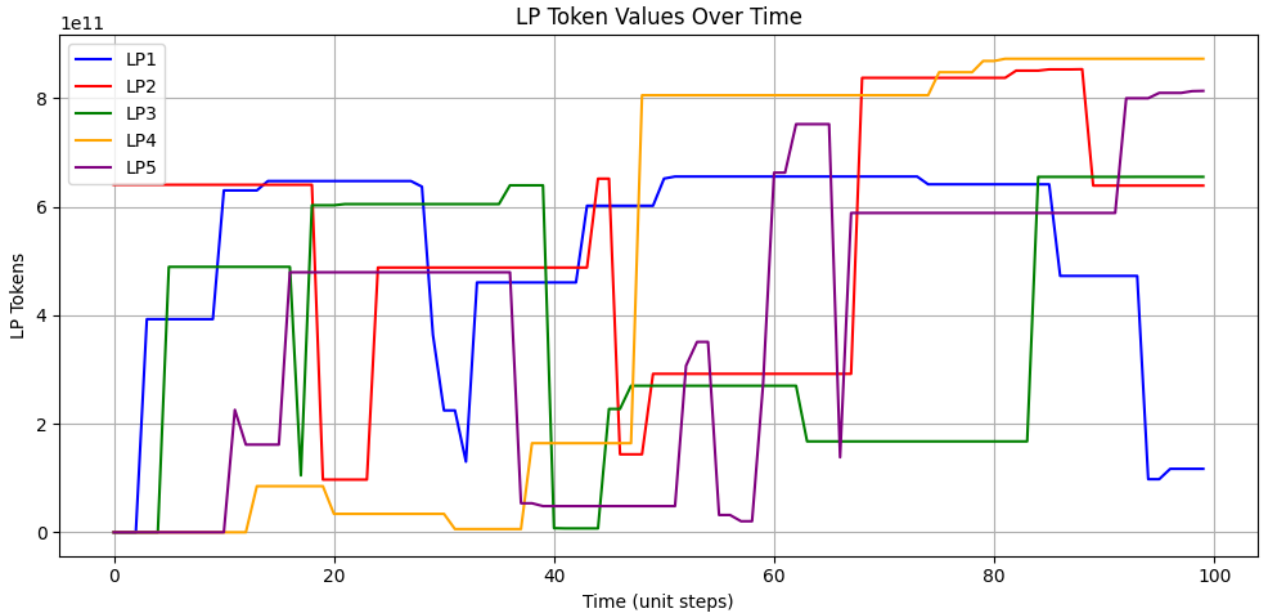

Figure 3: LPToken Holdings of each LP.

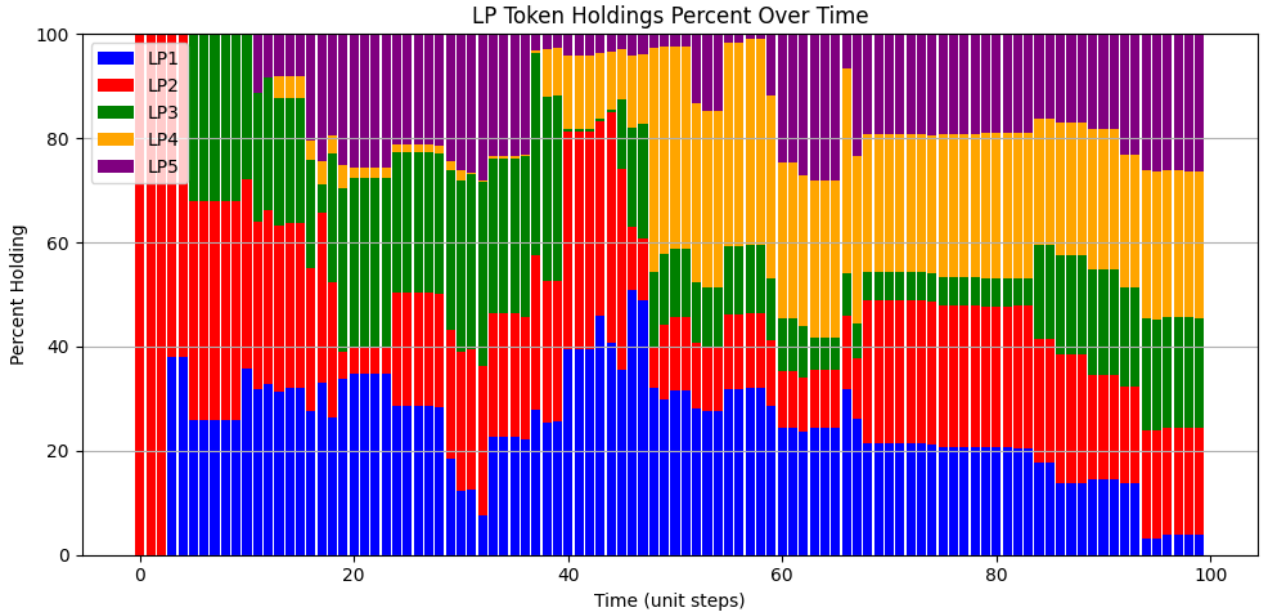

Figure 4: LPToken Distribution.

Trading Activity

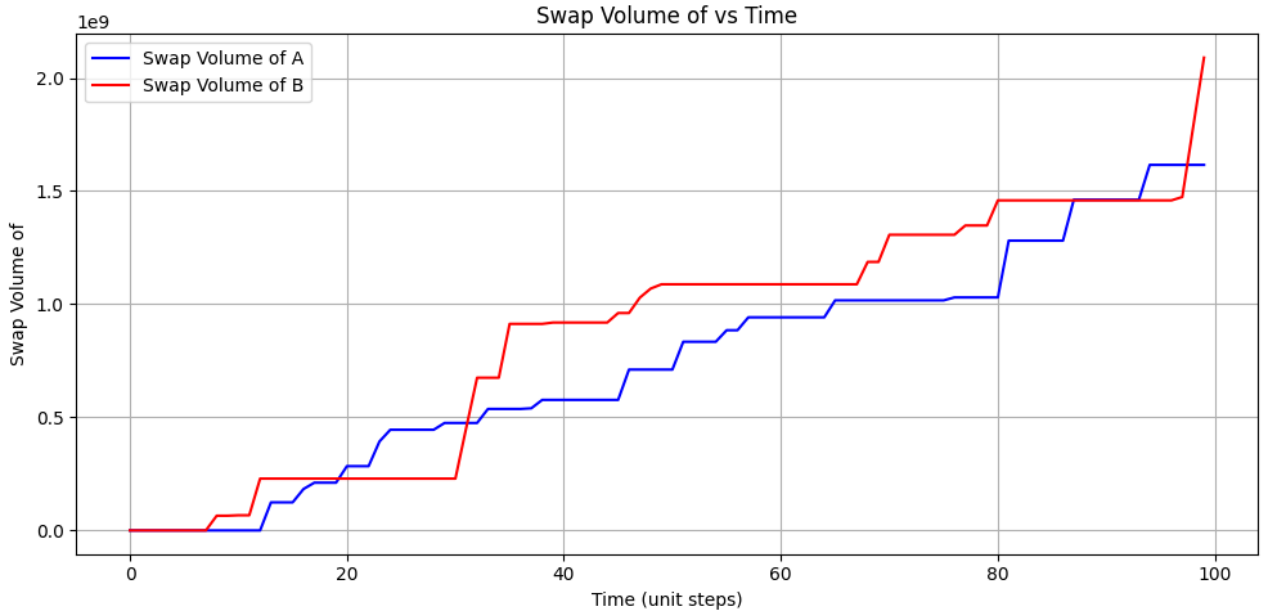

Figure 5: Swap Volume of Token A and B.

Swap volume of both A and B gradually and similarly increases as more Swap actions are performed.

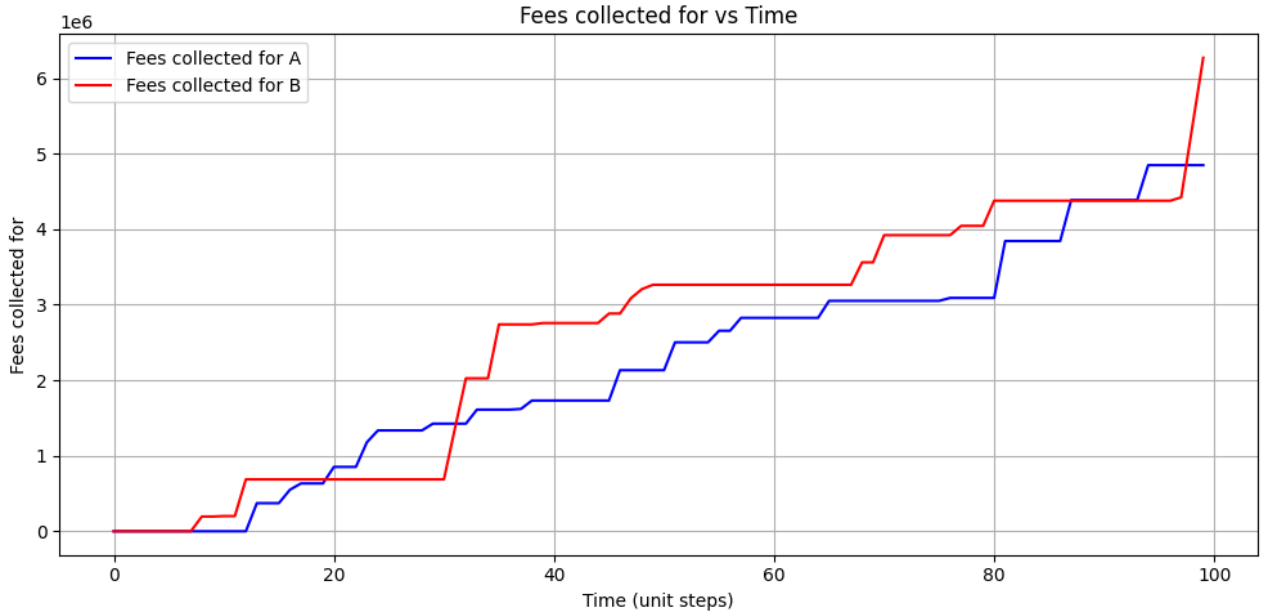

Figure 6: Fee Accumulation of Token A and B.

We can see that graph in Figure 6 is identical to graph in Figure 5 with a difference in scale. This is quite expected as Fees Collected is proportional to the swap amount, with factor of 0.3%.

Price Dynamics

Figure 7: Spot Price in DEX.

Spot Price is same as Reserve Ratio.

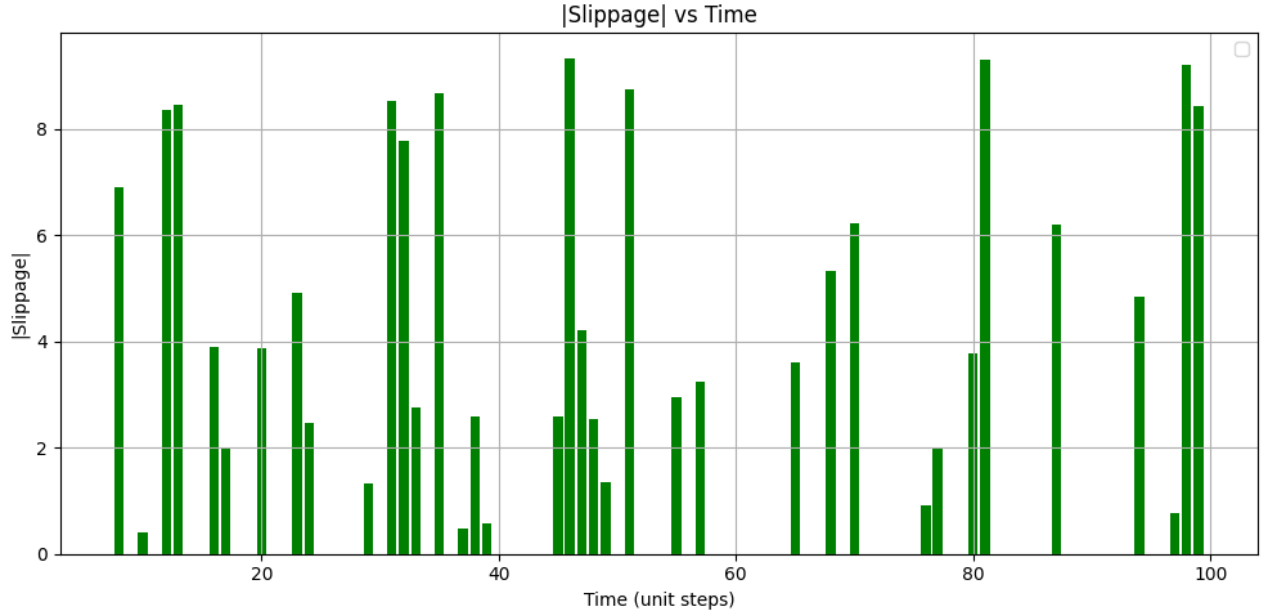

Figure 8: Slippage.

Slippage represents the difference between the actual exchange rate experienced and the expected exchange rate based on the spot price.
The formula for Slippage reduces to (This reduction is shown in this Section):

$$S \approx -\frac{\text{token } X \text{ deposited in Swap}}{\text{token } X \text{ reserves before Swap}} \cdot (1 - \text{Swap Fee})^2 - \text{ Swap Fee}$$

This indicates that slippage can occur when the tokens deposited are large compared to that token's reserves. This is one of the reasons why many constant product AMMs allow only a small fraction of their reserve value to be swapped, to maintain stability.
Slippage also occurs due to Swap Fee.

4.2 Arbitrage

We consider two decentralized exchanges, DEX θ and DEX θ'.

Profitable Arbitrage Scenario

We have chosen and set the reserve ratios of the 2 DEXes as follows to show a profitable Arbitrage:

DEX θ	**DEX** θ'
Token A Initial: X	Token A Initial: X
Token B Initial: $2X$	Token B Initial: X

Both DEX charge 0.3% swap fees.

Swap $b = 0.01X$ Token B in DEX θ' (Since swap cap is at 1%):

$$b\ B \xrightarrow[fee]{0.3\%} 0.997b\ B \xrightarrow{\theta'} ((X-a)(X+0.997b) = X^2 \implies a = 0.987b\ A)$$

$$0.987b\ A \xrightarrow[fee]{0.3\%} 0.984b\ A \xrightarrow{\theta} ((X+0.984b)(2X-b') = 2X^2 \implies b' = 1.949b\ B)$$

This shows that our b Token B get swapped into $1.949b$ Token B, yielding a sweet profit.

Unprofitable Arbitrage Scenario

We have chosen and set the reserve ratios of the 2 DEXes as follows to show a unprofitable Arbitrage:

DEX θ	**DEX** θ'
Token A Initial: X	Token A Initial: X
Token B Initial: $2X$	Token B Initial: $2.05X$

Both DEX charge 0.3% swap fees.

Swap $b = 0.02X$ Token B in DEX θ (Since swap cap is at 1%):

$$b\ B \xrightarrow[fee]{0.3\%} 0.997b\ B \xrightarrow{\theta} ((X-a)(2X+0.997b) = 2X^2 \implies a = 0.493b\ A)$$

$$0.493b\ A \xrightarrow[fee]{0.3\%} 0.492b\ A \xrightarrow{\theta'} ((X+0.492b)(2.05X-b') = 2.05X^2 \implies b' = 0.998b\ B)$$

This shows that our b Token B would get swapped to 0.998 Token B, which would yield a loss, hence no arbitrage opportunity exists.
Note that the swap $A \xrightarrow{\theta} B \xrightarrow{\theta'} A$ and $B \xrightarrow{\theta'} A \xrightarrow{\theta} B$ are clearly not profitable, even without swap fee. And the swap $A \xrightarrow{\theta'} B \xrightarrow{\theta} A$ can be shown to be unprofitable with similar calculation as above.

5 Theory Solutions

Q1: Which address(es) should be allowed to mint/burn the LP tokens?

Only the DEX smart contract itself should have the authority to mint and burn LP tokens.

The LP tokens should only be created when liquidity is added to DEX and destroyed when liquidity is removed from DEX, to maintaining the invariant of representing a share of the pool.

Allowing only DEX to mint and burn LP tokens is essential to prevent manipulation of pool ownership.

In our code, the LPToken contract is deployed by the DEX smart contract in it's constructor itself. And the LPToken contract sets it's deployer as it's owner. The `onlyOwner` modifier of `@openzeppelin`'s `Ownable` interface, which is applied to mint and burn functions of LPToken ensures that only the onwer of LPToken contract (i.e. DEX) can mint or burn these Tokens.

Q2: 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, like you and me!)?

Decentralized Exchanges (DEXs) aim to democratize trading access through protocol-level transparency and neutrality. Unlike centralized exchanges (CEXs), where high-frequency traders (HFTs) and institutional players often enjoy structural advantages, DEXs inherently promote fairness through design:

- **Equal Access:** On a DEX, everyone interacts directly with the protocol through smart contracts. There are no privileged APIs, private data feeds, or exclusive agreements. If a user can pay the required gas fee, they can interact with the protocol just like anyone else — no special treatment.

  In contrast, CEXs often grant HFT firms faster access to order books, colocated infrastructure, and lower latency trade execution — advantages completely absent in the DEX model.

  This also means anyone can contribute assets to the Liquidity Pool at will, with minimal entry barrier. While CEXs may gatekeep certain contribution schemes and prioratise resourceful individuals.

- **Algorithmic Pricing:** DEXs use Automated Market Makers instead of traditional order books. Trades are executed against a pool of assets using a deterministic pricing formula.

  This removes the advantage of placing and canceling orders faster than others — a core strategy in HFT. Since prices are set by an algorithm and not by who can act fastest, the system inherently favors no one.

- **No Need for a Counterparty:** Every trade interacts with a liquidity pool, not another trader. This makes price execution independent of market depth or matching speed, further minimizing any edge that institutions might have based on superior infrastructure.

In our code, no special treatment is given to any particular address. All functionality are deterministic and public. Swap Fee is distributed according to the LPToken share or each LP at the time of swap transaction. Use of AMM ensures no Centralized matchmaking. All these ensure a level playing ground for all individuals, irrespective of power and resources.

Q3: 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?

Yes — miners can exploit unconfirmed transactions to gain profit, using techniques like **front-running** or **sandwich attacks**.
Consider a scenario where a user submits a large trade — for example, swapping 500 Token X for Token Y. This will cause the price of Y to go up.
Before this transaction is confirmed and added to a block, a miner observes it sitting in the mempool. Using their power to control transaction ordering, the miner can:

- Insert their own trade ahead of the user's (e.g., buying Token Y),
- Let the user's large swap impact the price (moving Token Y's price up), and then
- Sell back Token Y after the user's trade, pocketing the difference.

Due to this, the user suffers worse pricing and the miner profits from the price movement caused by the user's transaction.
To make DEXs more resistant to such attacks, several strategies can be employed:

- **Slippage Tolerance Controls:** Users can specify a maximum acceptable slippage (price movement) when submitting a trade. If the execution price exceeds this tolerance — potentially due to a front-running attack — the trade automatically fails.
- **Commit-Reveal Schemes:** Instead of broadcasting trade details immediately, users can submit a commitment hash of the trade info. The actual trade details are revealed in a later transaction. This obscures the intent, making front-running much harder.
- **Batch Auctions:** Batching trades and executing them at a fixed interval prevents attackers from taking advantage of transaction ordering. Everyone gets the same clearing price, eliminating priority-based exploits.
- **Gas-Based Ordering:** Protocols can introduce gas-based prioritization rules or include fairness layers (like MEV-resistant sequencers or private mempools) to discourage miners from reordering transactions for personal gain.

Q4: We have left out a very important dimension on the feasibility of this smart contract- the gas fees! Every function call needs gas. How does gas fees influence economic viability of the entire DEX and arbitrage?

Every action on a DEX — swap, add liquidity, remove liquidity — involves executing smart contract code, which requires gas. This affects DEX economics in following ways :

- **Discourages Small Trades:** For small trades gas fees can overweigh benefits. This erodes value and makes micro-transactions economically unviable.
- **Raises the Bar for Profitable Arbitrage:** For arbitrage to be viable:

  $$\text{Profit from price diff} > \text{Gas costs} + \text{Slippage}$$

  If gas fees are high, only large-value trades or highly optimized bots can profit. This pushes retail users out of the arbitrage game. Also it encourage moving logic for detection of arbitrage off-chain, and do that locally.

- **Impacts Liquidity Provision ROI:** High gas makes LP earning smaller and gas fees for providing and removing liquidity can eat away big chunk of profit, especially for small players.

DEXs may become unattractive in high-gas environments, pushing users toward L2s or centralized options.

In our code, Arbitrageur only considers a swap profitable only is the profit exceeds a minimum threshold. This threshold can also be increased to account for gas fees.

Q5: Could gas fees lead to undue advantages to some transactors over others? How?

Yes — gas fees can introduce systemic advantages for users with greater financial or technical resources, particularly in decentralized exchange (DEX) environments.

- **Priority Access Through Higher Fees:** Most blockchains allow users to bid for faster transaction inclusion by paying higher gas fees or tips. This enables:
  - **Front-running** of user transactions by bots or miners.
  - **Preemptive execution** of arbitrage trades before others can react. Users who can afford to consistently pay more gain first-mover advantage on every profitable opportunity.
- **Retail User Disadvantages:** For everyday users making small-value trades:
  - **High gas prices** during network congestion can push their transactions to the back of the queue or leave them stuck in the mempool.
  - They may experience **failed trades** if execution happens after a significant price shift.
  - Arbitrage opportunities may disappear by the time their transaction is confirmed.
- **Block Space Competition and Miner Collusion:** Sophisticated actors — such as high-frequency trading bots or MEV searchers — can exploit blockspace economics to dominate transaction ordering. They may even collude with miners or validators to ensure their transactions are prioritized, effectively outbidding retail users for on-chain opportunities.

Q6: What are the various ways to minimize slippage in a swap?

Slippage — the difference between expected and actual execution price — increases with trade size and lower liquidity. It can be mitigated through the following strategies:

- **Smaller Trades:** The AMM pricing curve ($x * y = k$) is non-linear. Large trades shift the price significantly (causing higher slippage), while smaller ones preserve rate stability.
- **Deep Liquidity Pools:** Greater reserve balances reduce price impact for a given trade. Slippage drops as pool depth increases, though liquidity availability depends on participation by many LPs.
- **Split Trades Over Time:** Breaking a large trade into smaller chunks can reduce instantaneous price impact — but this may introduce timing risk and isn't always feasible.
- **Smart Routing (Multi-Hop Swaps):** Indirect routes like $A \to B \to C$ may result in less slippage if those pools have better liquidity.
- **Use Stable AMMs or Dynamic Fee Models:** Platforms like Curve or Uniswap v3 minimize slippage using optimized pricing curves for correlated assets and concentrated liquidity.

- **Trade During Low Volatility:** Swapping during quieter periods helps reduce price swings due to fewer competing transactions.
- **Set Slippage Tolerance Parameters:** While not reducing slippage itself, this ensures the trade fails if the price impact exceeds acceptable bounds — protecting users from extreme losses.

Q7: Having defined what slippage, plot how slippage varies with the "trade lot fraction" for a constant product AMM?. Trade lot fraction is the ratio of the amount of token X deposited in a swap, to the amount of X in the reserves just before the swap.

Formula Reduction:

Let:

$$\begin{array}{cc} \text{S} = \text{ Slippage;} & \text{f} = \text{ Swap Fee (Fraction);} \\ \text{A} = \text{ token X reserves before swap;} & \text{a} = \text{ token X deposited in swap;} \\ \text{B} = \text{ token Y reserves before swap;} & \text{b} = \text{ token Y received in swap;} \end{array}$$

$$T = \text{ Trade Lot Fraction } = \frac{a}{A}$$

Then:

$$S = \left(\frac{\frac{b}{a} - \frac{B}{A}}{\frac{B}{A}} \right) * 100\%$$

$$\implies S = \left(\frac{b}{a} \cdot \frac{A}{B} - 1 \right) * 100\%$$

By the AMM's constant product formula:

$$(A + a * (1 - f)) * (B - b) = A * B$$

$$\implies \left(1 + \frac{a}{A} * (1 - f)\right) * \left(1 - \frac{b}{B}\right) = 1$$

$$\implies \frac{a}{A} \cdot (1 - f) = \frac{b}{B} \left(1 + \frac{a}{A} \cdot (1 - f)\right)$$

$$\implies (1 - f) = (S + 1) \left(1 + \frac{a}{A}(1 - f)\right)$$

$$\implies S = -\left(\frac{f + \frac{a}{A}(1 - f)}{1 + \frac{a}{A}(1 - f)}\right)$$

$$\implies S = -\left(\frac{f + T \cdot (1 - f)}{1 + T \cdot (1 - f)}\right)$$

If $T \ll 1$:

$$S \approx -\left(f + T \cdot (1 - f)^2\right)$$

Which suggests that Slippage is (almost) Linear with respect to Trade Lot Fraction.

Empirical Plot:

Figure 9: Slippage vs Trade Lot Fraction

Figure 9 from our experiment matches with the theoretical result as Slippage and Trade Lot Fraction come out to be almost Linear.