CS765 - Introduction to Blockchains, Cryptocurrencies, and Smart Contracts
Homework 3: Building your own Decentralized Exchange

Sharvanee Sonawane (22B0943)
Ananya Rao (22B0980)
Deeksha Dhiwakar (22B0988)

April 2025

Task 1

Implementation Details

- Token.sol defines two ERC-20 compatible tokens: `TokenA` and `TokenB`.
- Both contracts inherit from OpenZeppelin's:
  - `ERC20` – the standard ERC-20 token implementation.
  - `ERC20Permit` – adds support for permit-based approvals (EIP-2612), allowing gasless approvals via signatures.
- Each contract has a constructor that takes an `initialSupply` and mints that amount of tokens to the deployer's address (`msg.sender`).
- Token details:
  - `TokenA`: Name = `"TokenA"`, Symbol = `"TKA"`
  - `TokenB`: Name = `"TokenB"`, Symbol = `"TKB"`

Task 2

Implementation Details

Here we implement a decentralized exchange (DEX) using a constant product Automated Market Maker (AMM) model, with smart contracts and a JavaScript simulation for testing.

Smart Contracts

- **DEX.sol**
  - Core DEX contract enabling liquidity addition, removal, and token swaps.
  - Uses the constant product formula $x \cdot y = k$ to determine swap outputs.
  - Accepts two ERC-20 tokens (`TokenA` and `TokenB`) as trading pairs.
  - Applies a swap fee of 0.3%, implemented using constants `FEE_NUM = 997` and `FEE_DEN = 1000`.
  - Once swap fees are collected, swap fee per LP token is calculated for each LP, and when an LP removes liquidity their swap fees corresponding to all the swaps so far is also returned in the form of tokenAs and tokenBs. (Share of swap fees is calculated immediately after the swap, and returned whenever an LP removes liquidity).
  - We also keep track of the fee already collected by an LP such that the fee of a particular swap doesn't reflect in removing the liquidity added after the swap was performed.
  - Mints and burns LP tokens via an instance of the `LPToken` contract.
  - Tracks internal reserves `reserveA` and `reserveB` updated after every transaction.
  - Provides utility functions `spotPrice()` and `getReserves()` for querying state.
- **LPToken.sol**
  - ERC-20 token representing Liquidity Provider (LP) shares.
  - Only the DEX contract (set via `msg.sender` at deployment) is allowed to mint or burn tokens.
  - Ensures proportional ownership of the liquidity pool.

JavaScript Simulation (simulate_dex.js)

- Runs a randomized simulation of 75 transactions involving:
  - 5 LPs performing deposits and withdrawals.
  - 8 traders executing token swaps.
- Randomizes transaction amounts within defined constraints (e.g., based on balances or reserves).
- Includes helper functions to validate whether a user can perform a given action (e.g., `canAddLiquidity`, `hasA`).

Plots:

Initially 1000 tokenAs and 2000 tokenBs were deployed. A total of 75 transactions (chosen uniformly randomly among swaps, deposits and withdrawals) are simulated. 5 of the users are LPs and the other 8 are traders. Every trader is handed 62.5 tokenA and 125 tokenB, whereas every LP is handed 100 tokenA and 200 tokenB.

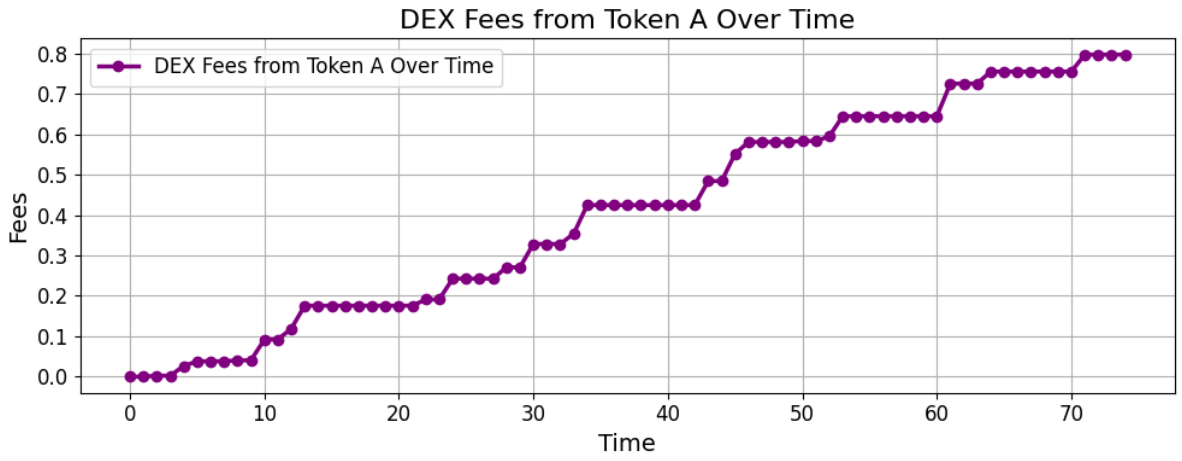

Figure 1: DEX Fees from Token A Over Time

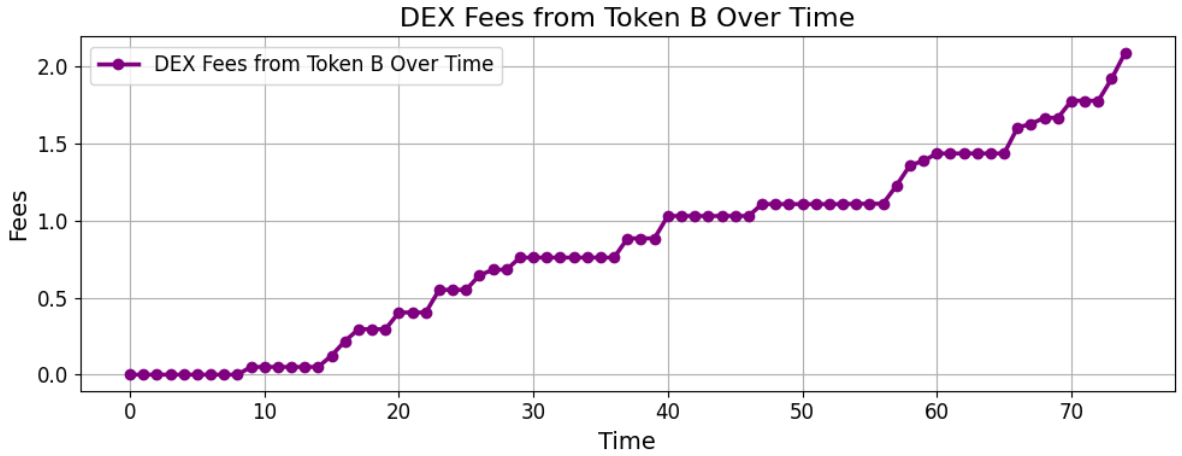

Figure 2: DEX Fees from Token B Over Time

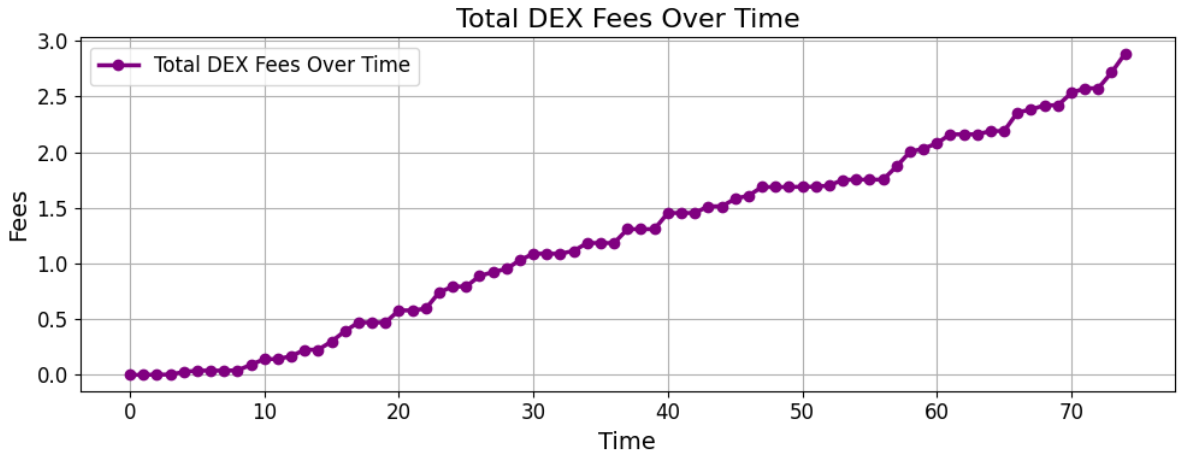

Figure 3: Total DEX Fees Over Time

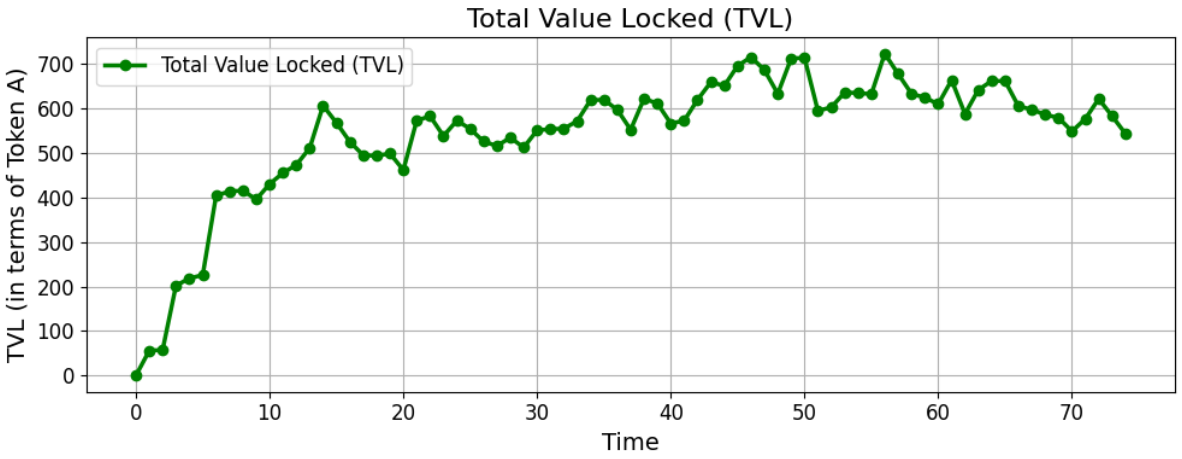

Figure 4: Total Value Locked (TVL)

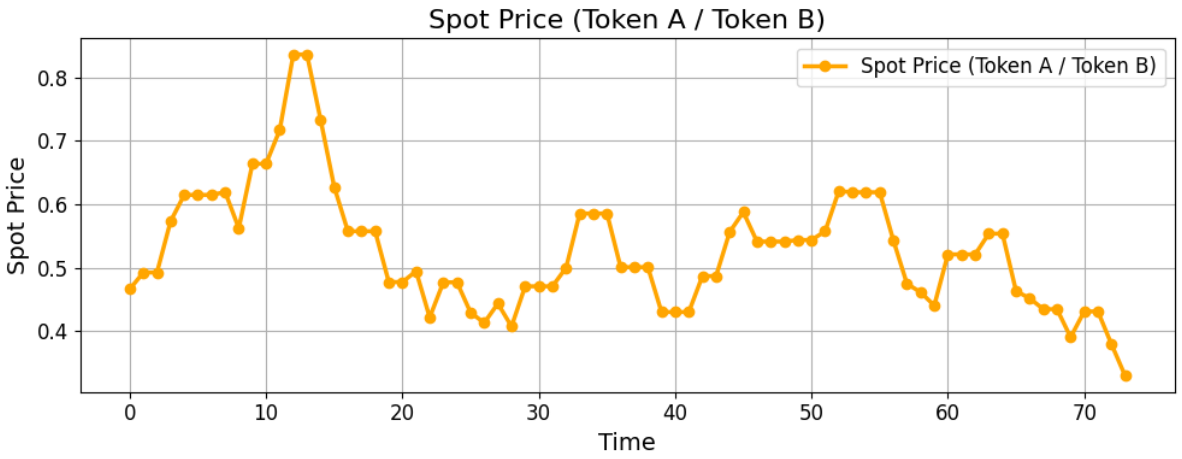

Figure 5: Spot Price (Token A / Token B)

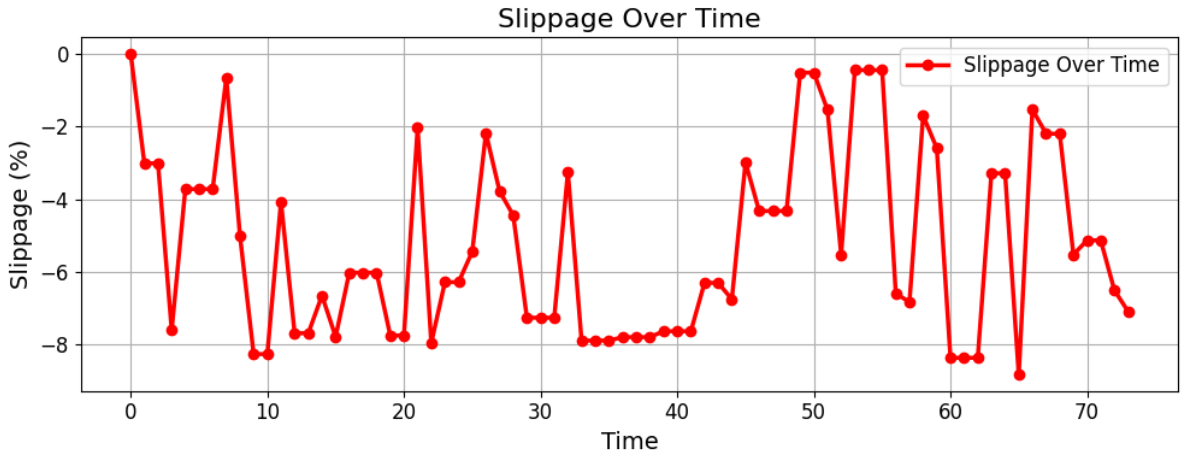

Figure 6: Slippage Over Time

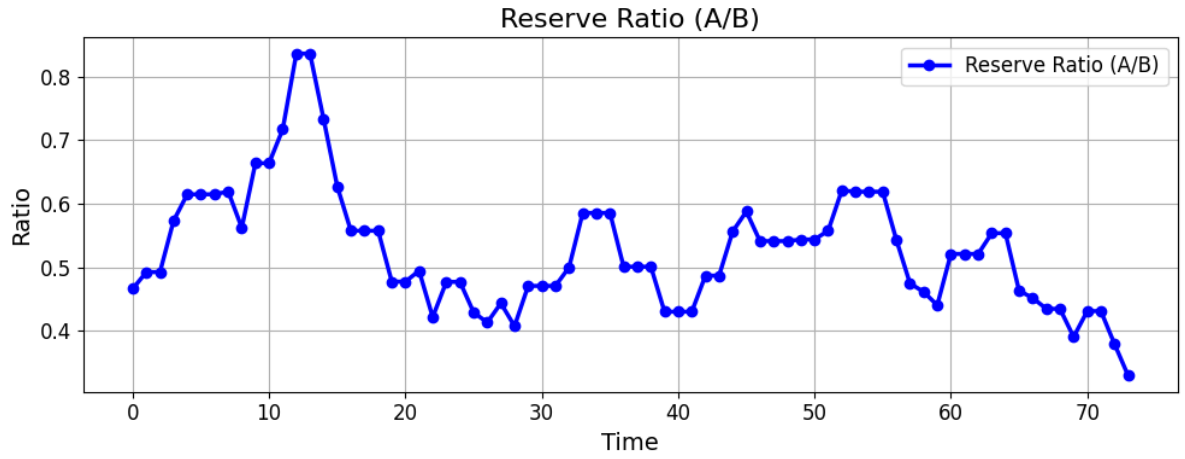

Figure 7: Reserve Ratio (Token A / Token B)

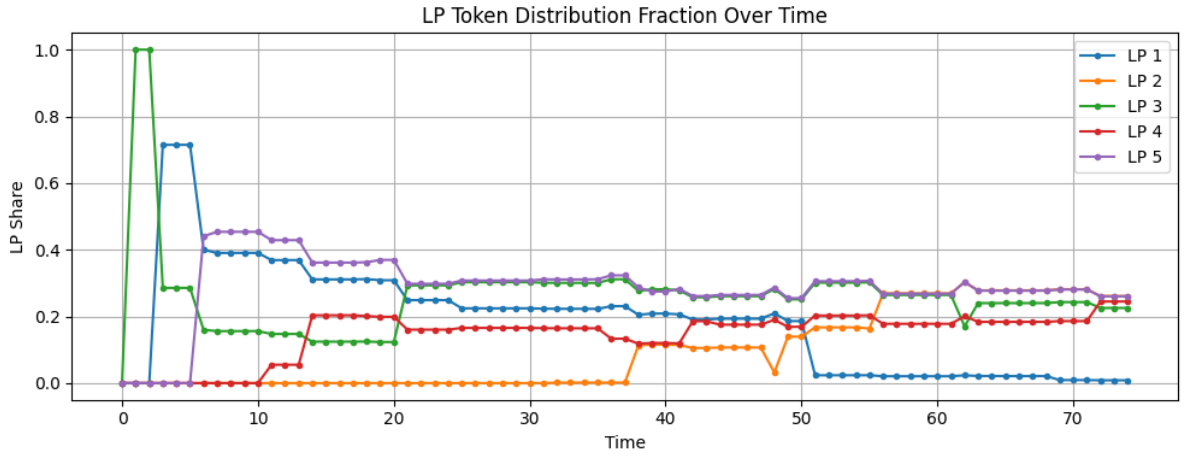

Figure 8: LP Token Distribution Fraction Over Time

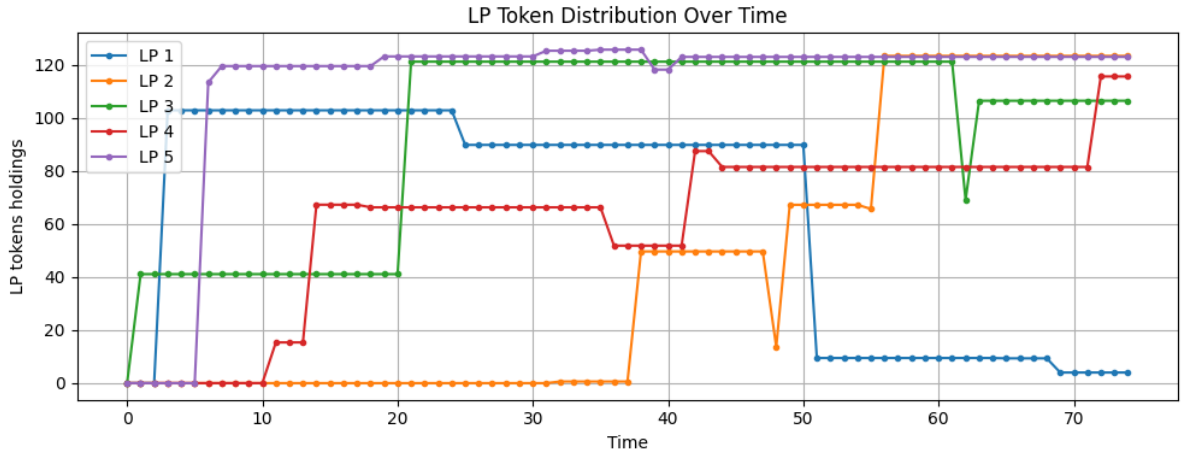

Figure 9: LP Token Distribution Over Time

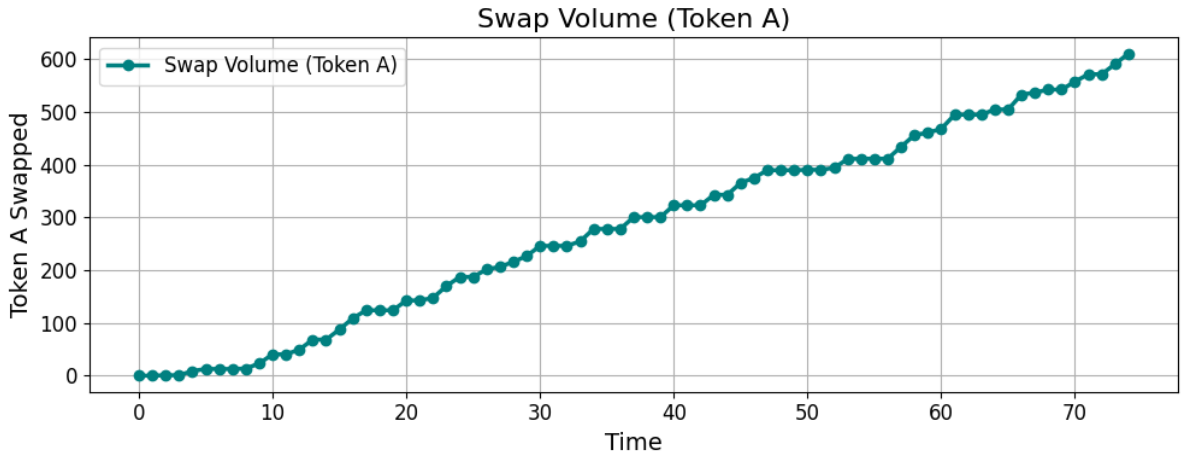

Figure 10: Swap Volume (Token A)

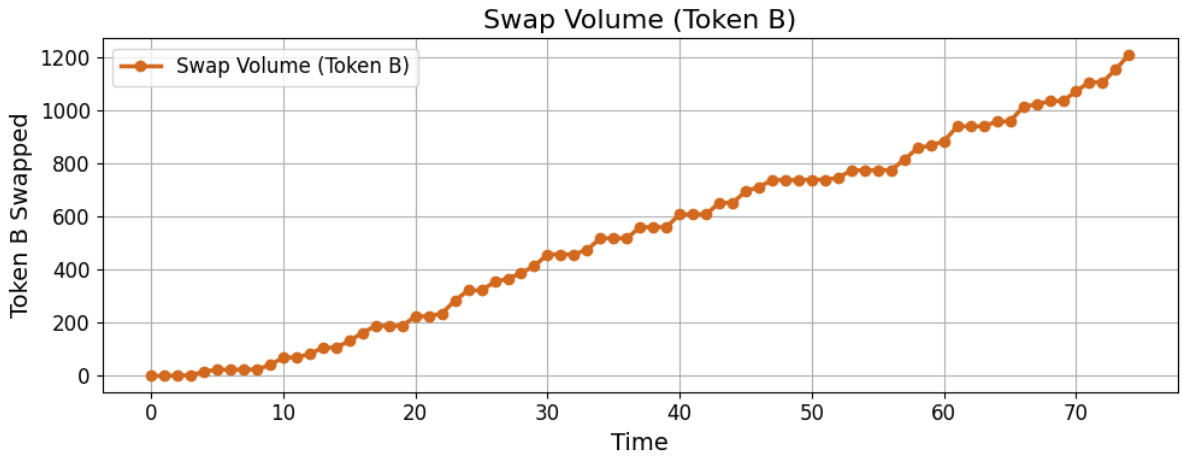

Figure 11: Swap Volume (Token B)

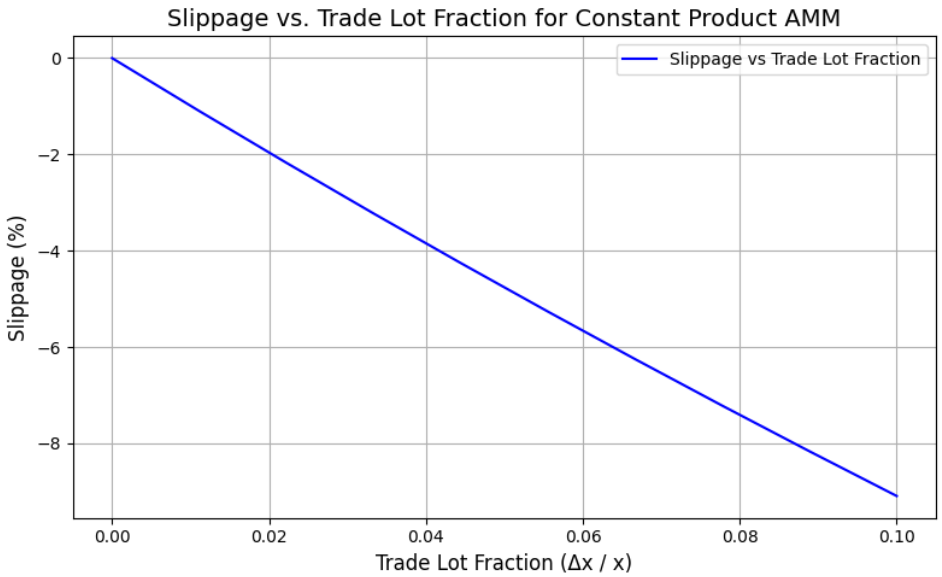

Figure 12: Slippage vs Trade Lot Fraction

Security and Validation Measures

The following security and validation mechanisms are implemented to ensure the robustness and correctness of the DEX:

- **Access Control:**
  - Only the DEX contract is authorized to mint or burn LP tokens via checks in `LPToken.sol`.
- **Input Validation:**
  - Swap function checks for valid token addresses and non-zero input amounts.
  - Liquidity functions verify sufficient balances and enforce proportional deposits and withdrawals.
- **Safe Arithmetic:**
  - Arithmetic operations are performed using Solidity 0.8.x, which includes built-in overflow/underflow checks.
  - The ‘Math.mulDiv()‘ function is used for precise proportional LP token minting.
- **State Consistency:**
  - Internal reserves `reserveA` and `reserveB` are updated after every liquidity or swap operation using the `_updateReserves()` function.
- **Fail-Safe Transfers:**
  - All ERC20 token transfers use `require()` to ensure they succeed, preventing silent failures.

Floating Point Simulation

Since Solidity does not support floating point numbers, we simulate decimal precision using fixed-point arithmetic by scaling integer values by a factor of 10^{18}. This approach allows us to perform operations that would otherwise require decimal precision (e.g., calculating proportional shares, prices, or slippage). For example, when computing ratios or percentages, we multiply values by 10^{18} before division, and scale them back down afterward if needed. This technique ensures accurate mathematical operations without introducing rounding errors that would otherwise occur with truncated integer division.

Task 3

Implementation Details

Smart Contract: `Arbitrage.sol`

- The `Arbitrage` contract is owner-restricted and allows only the contract deployer to execute arbitrage.
- It supports two arbitrage paths:
  1. `TokenA` → `TokenB` → `TokenA`
  2. `TokenB` → `TokenA` → `TokenB`
- The contract calculates the expected output using reserves from each DEX and simulates swaps to check profitability.
- A minimum profit threshold is enforced; arbitrage is only executed if the profit is at least 0.05% of the input.
- The `executeArbitrage` function handles:
  1. Approval of input token for the buy DEX.
  2. Swap on the first DEX.
  3. Approval of output token for the sell DEX.
  4. Swap on the second DEX.
  5. Verification of profit and transfer of capital plus profit back to the owner.

JavaScript Simulation: `simulate_arbitrage.js`

- Loads the ABIs and deployed addresses for `TokenA`, `TokenB`, both DEXes, and the `Arbitrage` contract.
- Two arbitrage scenarios are demonstrated:
  1. **Failed Arbitrage:** Both DEXes are initialized with the same reserves (100 TokenA, 100 TokenB). Arbitrage fails due to no price discrepancy.
  2. **Successful Arbitrage:** In this case the reserves for DEX1 are 100 TokenA, 100 TokenB and for DEX2 are 100 TokenA, 400 TokenB. Arbitrage is successfully executed, and profit is returned to the arbitrageur.

Theory Questions

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

Only the DEX smart contract (i.e., the Automated Market Maker contract) should be allowed to mint and burn LP tokens.

This is because LP tokens, unlike TokenA and TokenB which represent real world assets, simply represents a proportional share of the liquidity pool (TokenA and TokenB reserves). The DEX contract should automatically mint tokens when an LP adds liquidity, and automatically burn tokens when an LP removes liquidity. Letting users mint and burn LP tokens themselves could break the proportional ownership rule and security of the DEX.

2. 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!)?

One major advantage of DEXs over traditional centralized systems involving banks or middlemen is that it allows small and large players to participate on equal footing. It does so in the following ways:

- **Transparency:** All transactions happen on a blockchain and are publicly visible to all players. This prevents unfair informational advantages (a problem in centralized exchanges that can be exploited by market makers or insiders).
- **Permissionless access:** There is no concept of approval or minimum balance; anyone with a crypto wallet can trade and participate in the market. This removes the entry barriers often imposed by centralized exchanges, allowing small traders to enter the market.
- **Decentralization:** There is no centralized order book, meaning HFTs do not gain advantage by exploiting microsecond-level events, since trades are executed only against the liquidity pool using deterministic pricing. This makes the market more accessible to small players.
- **Equal opportunity for liquidity provision:** Anyone can become an LP and earn trading fees; it is no longer a privileged role reserved for market makers.
- **No censorship:** Since DEXs operate on blockchains, no exchange can freeze or limit access arbitrarily. This protects small traders from policy based exclusions.

3. 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?

When users submit transactions to the DEX, they first enter the public mempool before being included in a block. Miners can inspect these pending transactions and identify profitable opportunities. One such technique is front-running. When the miner sees that a trader has submitted a large swap to the mempool (that could significantly impact token price), the miner submits his own trade right before it, offering a marginally higher gas fee to ensure priority. This allows him to buy tokens cheaply before the price impact, and by immediately selling them after the trader's transaction he obtains a risk free profit.

Techniques like this undermines fairness and can reduce user confidence in the DEX.

Some countermeasures to make the DEX robust against this are:

- Traders first submit a hash of their intended transaction, and later reveal it, so that miners cannot gain any insider information.
- Instead of processing trades sequentially, execute all trades in a batch, neutralizing the advantage of ordering.
- Allow users to set maximum acceptable slippage to prevent execution under unfavorable conditions.

4. 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?

Gas fees determine the usability and economic viability of DEXs. On blockchains like Ethereum, every smart contract interaction (swaps, deposits, withdrawals, etc.) requires users to pay gas. If the gas fees are high,

- Small trades become unprofitable. If the gas cost to execute a trade is comparable to or exceeds the value being exchanged, users may lose money.
- Arbitrage opportunities are limited. Since arbitrage relies on small price differences between markets, if gas costs are high, only large-volume trades that yield high profits can cover the fees. This allows price imbalances to persist longer.

- LP participation is discouraged, since each operation costs a significant amount of gas. This could result in a less dynamic liquidity pool and slower reaction to market demand.
- It becomes difficult for new participants to enter the market, since fees may be too high relative to their trading size.

Overall, high gas fees reduce the volume and frequency of interactions on the DEX. This lowers liquidity, increases slippage, and makes the platform less advantageous over centralized exchanges.

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

Yes, gas fees could lead to undue advantages to users who are able to pay higher fees, leading to an unfair DEX system, in the following ways:

- **Transaction prioritization:** Miners and validators prioritize transactions with higher gas fees. This allows wealthier users to front-run others by simply offering a higher gas price, ensuring their trades are executed first.
- **Arbitrage opportunities:** Arbitrage depends on being first to act on price discrepancies. High gas fees filter out users with smaller capital, leaving such opportunities exclusively to large players who can afford to absorb the cost.
- **Strategic timing:** Large traders can adjust their gas fees to ensure their trades are mined at optimal times, such as just before large swaps or during high slippage periods, thus gaining an edge over smaller traders.

While DEXs were designed to democratize access to financial services, high gas costs reintroduce the same barriers faced in traditional finance - where only those with capital and tools can fully participate.

6. What are the various ways to minimize slippage (as defined in 4.2) in a swap?

Slippage is usually caused by insufficient liquidity or large trade sizes. In AMMs, slippage increases with trade size relative to the pool and can lead to unfavorable exchange rates for users. Some ways to minimize slippage in a DEX are:

- **Increase liquidity in the pool:** Slippage is inversely proportional to the size of the reserves in the pool. Larger liquidity pools absorb bigger trades with less impact on the spot price. Liquidity can be increased by incentivizing LPs.
- **Limit trade size relative to reserves:** Instead of executing a single large swap, users can split trades into multiple smaller ones or limit the trade to a smaller percentage of the pool. This reduces the price impact due to the constant product formula $x \cdot y = k$.
- **Set slippage tolerance:** Allow users to specify a maximum slippage tolerance. If the actual trade exceeds this threshold, the transaction reverts. This ensures users are not unknowingly subject to high slippage.

7. 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.

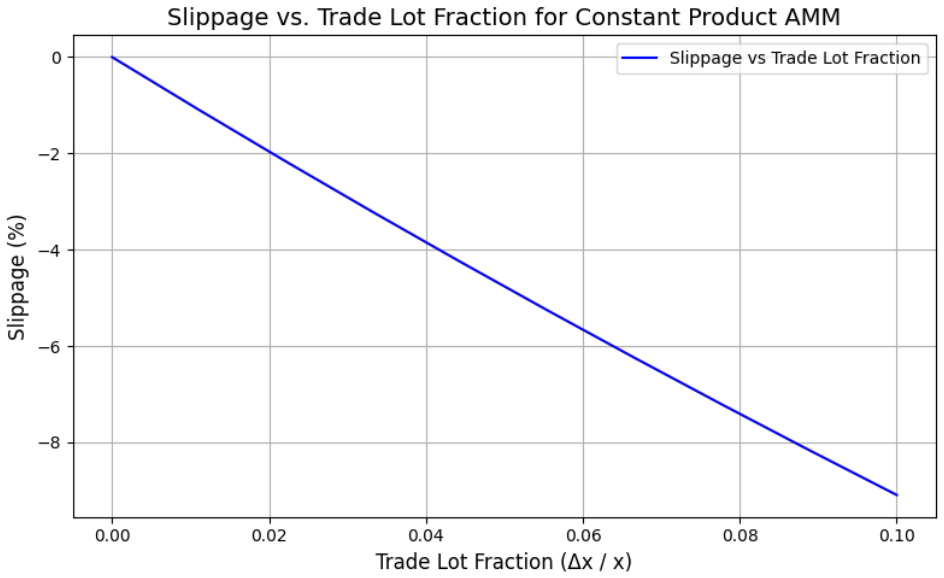

Figure 13: Slippage vs Trade Lot Fraction

The formula for slippage percentage reduces to -100 $*$ $\frac{f}{1+f}$ where f is the trade loss fraction. The graph obtained for the same from the 75 random transactions of task 2 is as shown above.