CS 765: Assignment 3

Building your own Decentralized Exchange

Soumitra Darshan Nayak (22B0984)
Kukudala Sai Aditya (22B0952)
Avanaganti Akshath Reddy (22B0992)

April 15, 2025

Contents

1 Introduction

This report documents our implementation and analysis of a custom Decentralized Exchange (DEX) system. We've built a DEX using Solidity smart contracts, created plots to analyze its behavior, and implemented arbitrage mechanisms between multiple DEX instances. The project involved three main tasks implemented using Solidity on the Remix IDE:

1. Creating and deploying two standard ERC20 tokens.
2. Implementing the core DEX contract with liquidity provision and swapping functionality, along with a Javascript simulation to study its dynamics.
3. Implementing an arbitrage bot contract to detect and execute profitable trades between two instances of the DEX.

This report details the implementation of each task, discusses the simulation results, and addresses theoretical questions related to DEXs and AMMs.

2 Task 1: ERC20 Token Implementation

2.1 Token Contract (`Token.sol`)

The first step was to create a standard ERC20 token contract. We used the OpenZeppelin ERC20 implementation for robustness and standard compliance. The contract `Token` inherits from 'ERC20' and 'Ownable'. The constructor initializes the token's name, symbol, and mints an initial supply entirely to the deployer account (`msg.sender`), who becomes the initial owner.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract MyToken is ERC20, Ownable {
    constructor(
        string memory name_,
        string memory symbol_,
        uint256 initialSupply_
    ) ERC20(name_, symbol_) Ownable(msg.sender) {
        _mint(msg.sender, initialSupply_);
    }
}
```

Listing 1: Basic ERC20 Token Contract ('Token.sol')

2.2 Deployment and Testing

This 'Token.sol' code was deployed twice on the Remix VM using 'accounts[0]' to create two independent token instances: Token A (e.g., "TKNA") and Token B (e.g.,"TKNB"), each with its own initial supply. Basic functionality was tested directly in Remix by using the 'transfer' function to send tokens between different Remix VM accounts and verifying the updated balances using the 'balanceOf' function, ensuring the core ERC20 operations worked correctly.

3 Task 2: AMM DEX Implementation and Simulation

3.1 LP Token Contract (`LPToken.sol`)

A separate ERC20 token contract, 'LPToken.sol', was created to represent liquidity providers' shares in the DEX pool. Similar to 'Token.sol', it inherits from OpenZeppelin's 'ERC20' and 'Ownable'. However, its constructor does not mint any initial supply. Instead, it includes 'mint' and 'burn' functions protected by the 'onlyOwner' modifier. The ownership of the deployed LPToken instance must be transferred to the corresponding DEX contract instance after deployment, ensuring only the DEX can mint or burn LP tokens when liquidity is added or removed.

```
contract LPToken is ERC20, Ownable {
    constructor(string memory name_, string memory symbol_)
        ERC20(name_, symbol_)
        Ownable(msg.sender)
    {}  // Deployer is initial owner

    function mint(address account, uint256 amount) public onlyOwner {
        _mint(account, amount);
    }  // Only owner (DEX contract) can mint

    function burn(address account, uint256 amount) public onlyOwner {
        _burn(account, amount);
    } // Only owner (DEX contract) can burn
}
```

Listing 2: LP Token Contract ('LPToken.sol')

3.2 DEX Contract (`DEX.sol`)

The 'DEX.so' contract implements a constant product Automated Market Maker (AMM) with the following key features:

- **Architecture:** Uses immutable token addresses (Token A, Token B, and LPToken) set during deployment, with state variables 'reserveA' and 'reserveB' to track liquidity. Incorporates 'SafeMath', 'SafeERC20', and 'nonReentrant' modifiers for security.
- **Liquidity Management:** Provides `addLiquidity` for depositing token pairs (with the first depositor setting the initial price ratio and subsequent ones maintaining the existing ratio) and `removeLiquidity` for withdrawing proportional shares. LP tokens represent ownership stakes, which are minted/burned based on liquidity changes. Both operations require prior user approval.
- **Trading Mechanism:** The `swap` function enables trading between tokens, applying a 0.3% fee that remains in the pool, calculating output amounts using the constant product formula $x \times y = k$, and ensuring that $(\texttt{reserveIn} + \texttt{amountInWithFee}) \times (\texttt{reserveOut} - \texttt{amountOut}) = \texttt{reserveIn} \times \texttt{reserveOut}$. Follows the Checks-Effects-Interactions pattern to prevent reentrancy attacks.
- **View Functions:** `getReserves()` to read current liquidity levels, `spotPriceA()` & `spotPriceB()` to obtain price ratios (scaled by 10^{18} for converting into Wei), and inbuilt token address getters to support the arbitrage contract interface.

3.2.1 Security Measures

Our implementation incorporates multiple security measures to protect against common vulnerabilities:

- **Arithmetic Protection:** Used Solidity 0.8+ built-in checks and OpenZeppelin's SafeMath to prevent overflow/underflow issues in calculations involving token amounts, fees, and liquidity ratios.
- **Safe Token Transfers:** Implemented SafeERC20 for all token operations to handle non-standard ERC20 implementations correctly and ensure failed transfers revert the transaction rather than silently failing.
- **Reentrancy Prevention:** Applied nonReentrant modifiers to critical functions (addLiquidity, removeLiquidity, swap, executeArbitrage) to prevent malicious contracts from exploiting recursive call patterns.
- **Checks-Effects-Interactions Pattern:** Followed the secure execution pattern by performing validations first, updating state next, and external calls last to minimize attack surfaces during token transfers.
- **Input Validation:** Used require statements throughout the codebase to verify addresses, amounts, balances, and other parameters before executing critical operations.
- **Access Control:** Restricted sensitive functions with Ownable modifiers, particularly for LP token minting/burning and administrative operations to prevent unauthorized access.
- **Immutable Architecture:** Declared critical addresses and configuration parameters as immutable to prevent post-deployment tampering.

These protections collectively ensure contract integrity against common attacks while maintaining efficient operation.

3.3 DEX Simulation (`simulate_DEX.js`)

A Javascript script (`simulate_DEX.js`) was implemented to test the DEX functionality and study its dynamics, following the structure of `ballot_example.js`.

- **Setup:** The script initializes by fetching contract ABIs, getting deployed contract addresses (Token A/B, LPToken, DEX), obtaining 13 user accounts (5 LPs, 8 Traders), and instantiating web3 contract objects. It automates the initial funding of user accounts and sets necessary ERC20 approvals for the DEX contract. It also adds initial liquidity using the first LP account.
- **Simulation Loop:** It runs for N (i.e. 75) transactions. In each iteration:
  - A user is selected uniformly randomly.
  - An action (add liquidity, remove liquidity, swap) is chosen randomly.
  - Transaction amounts are determined randomly based on assignment rules: liquidity amounts based on user holdings, swap amounts based on a uniform random distribution between 0 and minimum of user balance and 10% of the relevant reserve.
  - The corresponding DEX function is called from the selected user's account.

  - Necessary data (reserves, prices, volumes, slippage, fees, LP balances) is recorded after each transaction.

- **Data Logging:** After the loop, the script logs the final state and prints the collected time-series data arrays in JSON format for external plotting.

3.4 Simulation Results and Plots

The simulation (`simulate_DEX.js`) was run successfully after compilation and deployment of the contracts and setting the addresses of ABIs and deployed contracts. The script executed various swap, add liquidity, and remove liquidity transactions initiated by different users. The following key metrics were tracked and data was logged for plotting against transaction index/time:

1. **TVL (Total Value Locked):** The total amounts of Token A and Token B held by the DEX contract over time. $TVL = Reserves_A + Reserves_B \cdot \frac{Reserves_A}{Reserves_B} = 2 \cdot Reserves_A$ Here we are assuming $Reserves_A$ as USD.
2. **Reserve Ratio / Spot Price:** The ratio reserveA / reserveB, representing the spot price of Token B in terms of Token A.
3. **LP Token Distribution:** A snapshot of each simulated user's LP token balance after every transaction, visualized as a stacked area chart to show distribution over time.
4. **Swap Volume:** The cumulative amount of Token A and Token B swapped into the DEX.
5. **Fee Accumulation:** The cumulative fees collected in Token A and Token B, calculated based on the 0.3% fee per swap.
6. **Slippage:** The percentage difference between the actual execution price for each swap transaction and the expected price (reserve ratio before swap), calculated using the formula provided.

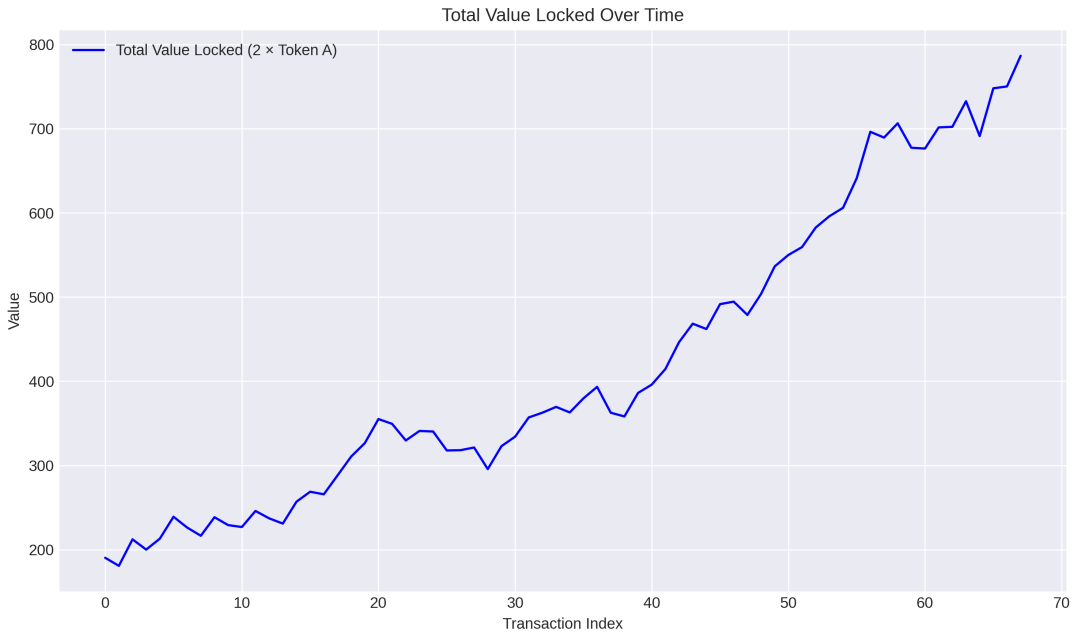

Figure 1: Plot of Total Value Locked (TVL) vs. Transaction Index

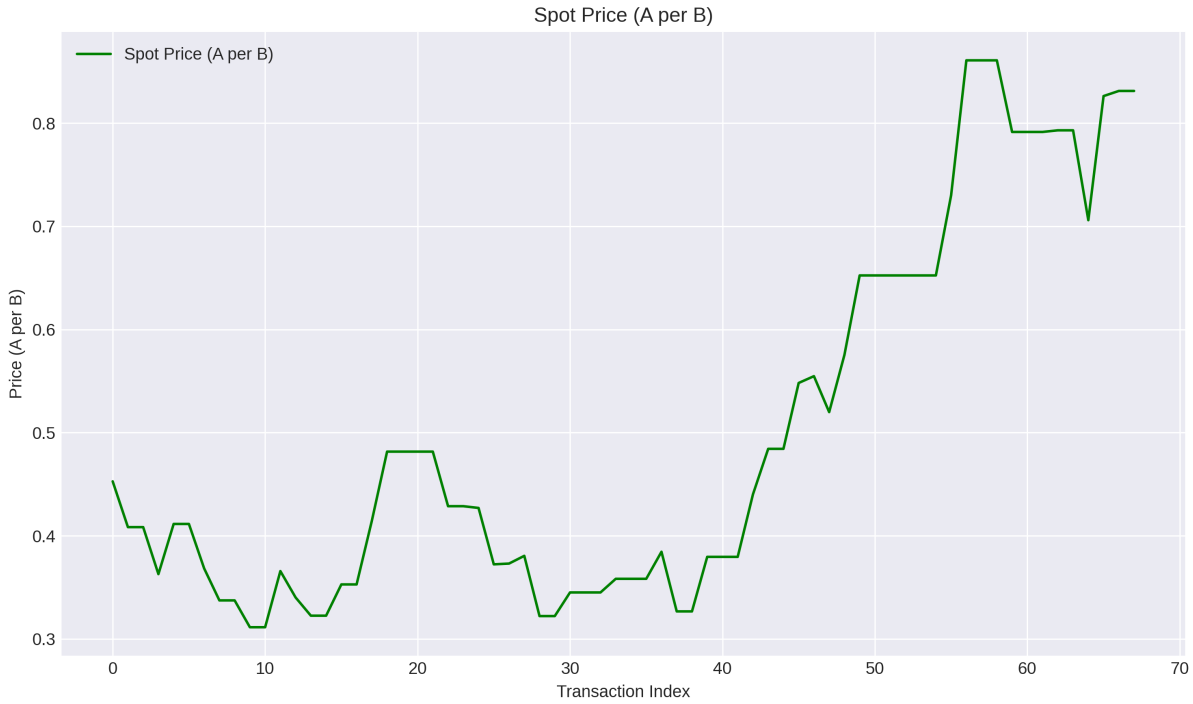

Figure 2: Plot of Reserve Ratio / Spot Price vs. Transaction Index

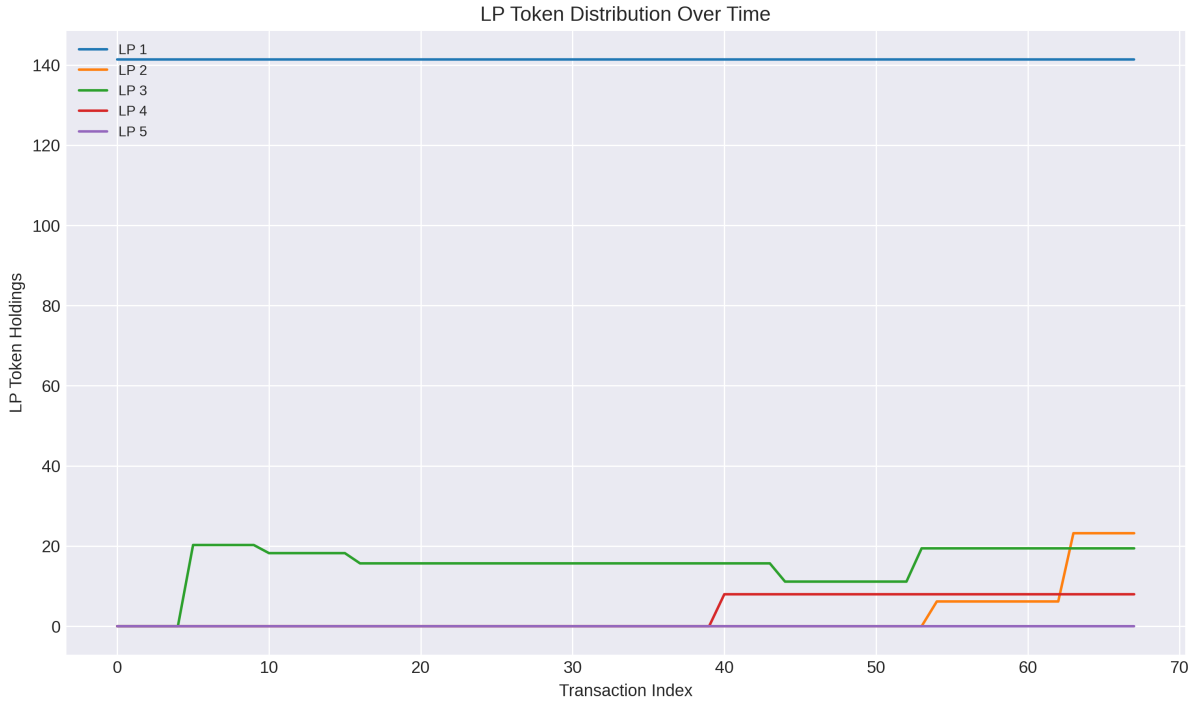

Figure 3: Plot of LP Token Distribution vs. Transaction Index

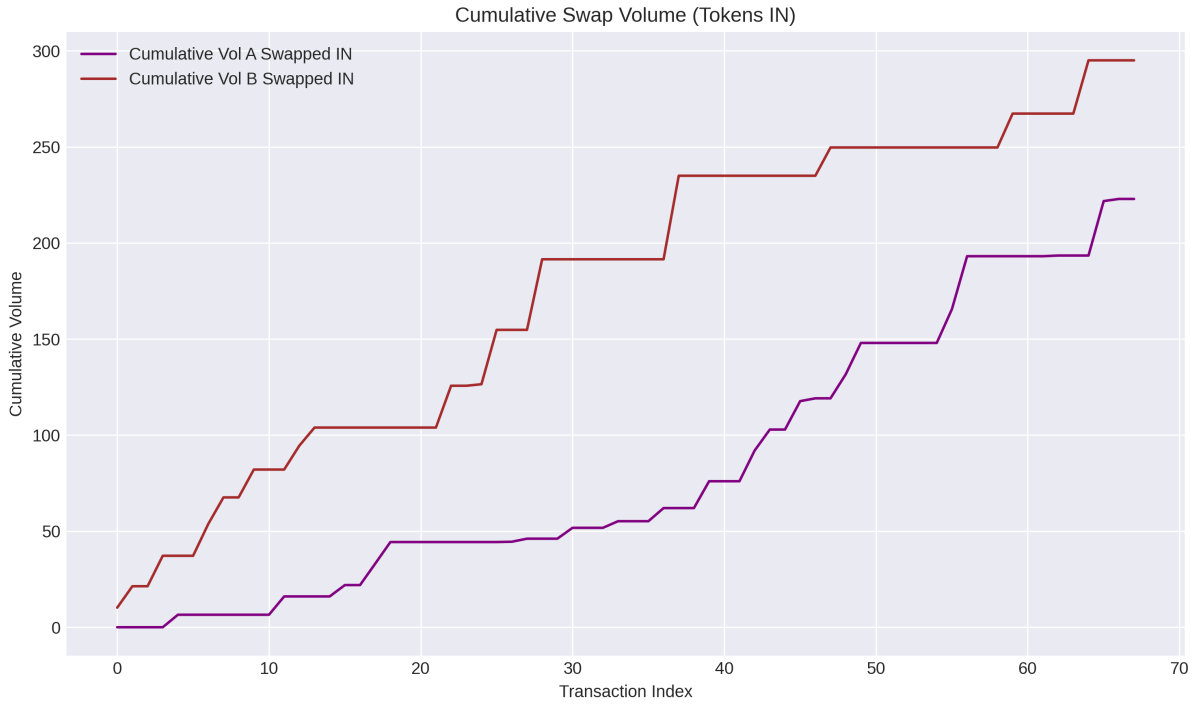

Figure 4: Plot of Cumulative Swap Volume vs. Transaction Index

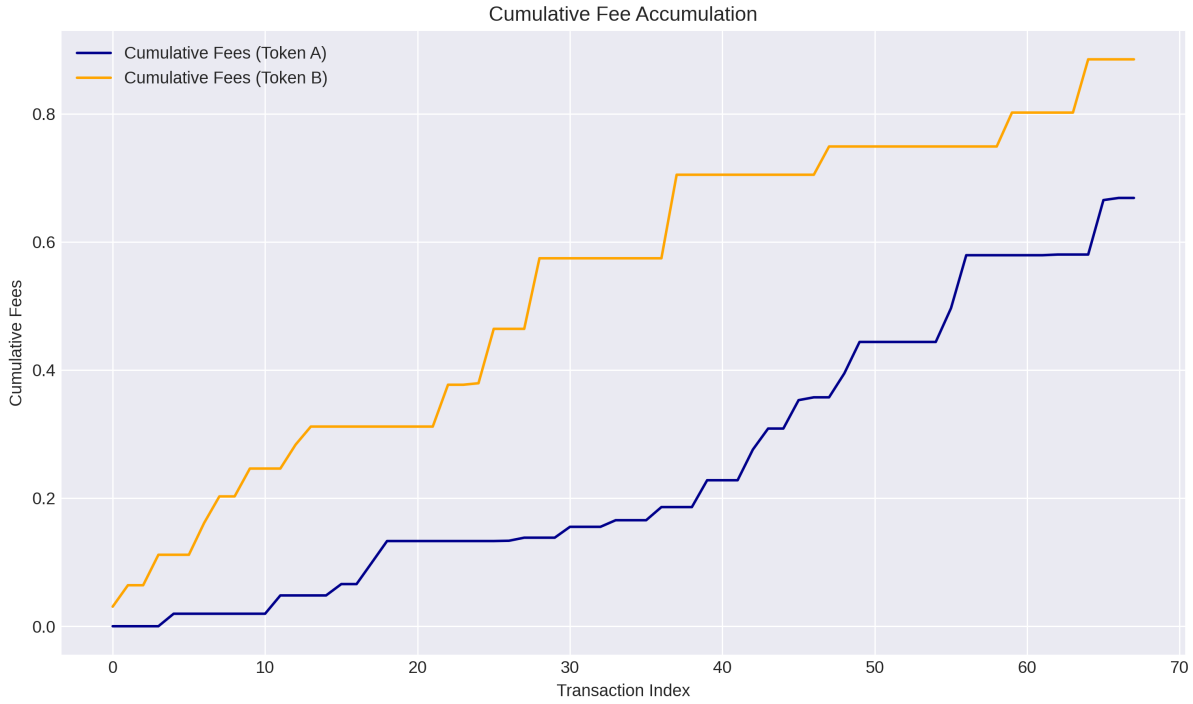

Figure 5: Plot of Cumulative Fee Accumulation vs. Transaction Index

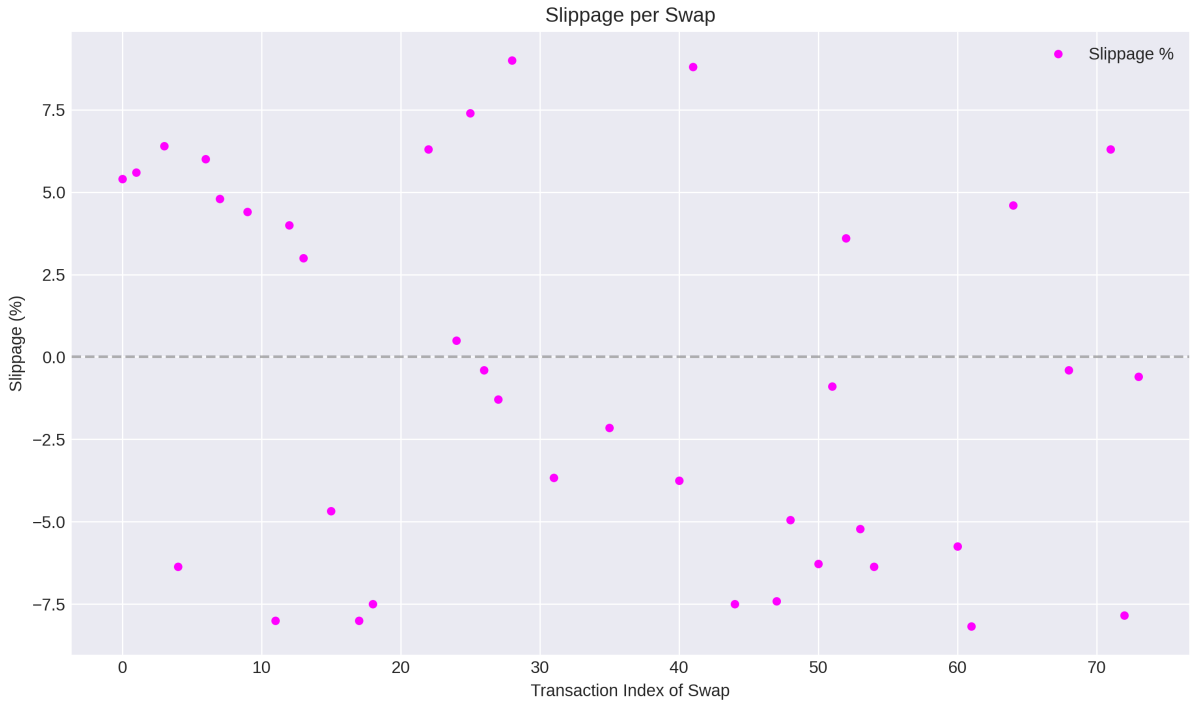

Figure 6: Plot of Slippage per Swap vs. Transaction Index

4 Task 3: Arbitrage Bot Implementation and Simulation

4.1 Arbitrage Bot Contract (`arbitrage.sol`)

An arbitrage bot contract was implemented to exploit price differences between two instances of the previously built DEX.

- **Setup:** The contract stores the addresses of Token A, Token B, DEX1, and DEX2, which are set during deployment. It requires initial funding with both Token A and B to execute trades. It includes 'Ownable' for control and withdrawal functions.
- **Price Fetching:** It uses an internal helper function `getAmountOut` to query the reserves of a specified DEX and predict the output of a potential swap, accounting for the 0.3% fee and the constant product formula (implicitly estimating price impact).
- **Opportunity Calculation:** The main function `executeArbitrage` calculates the potential profit for all four arbitrage paths ($A \to 1 \to B \to 2 \to A$, $A \to 2 \to B \to 1 \to A$, $B \to 1 \to A \to 2 \to B$, $B \to 2 \to A \to 1 \to B$) by calling `getAmountOut` twice for each path.

4.2 Arbitrage Simulation (`simulate_arbitrage.js`)

A Javascript script (`simulate_arbitrage.js`) was created to test the arbitrage bot.

- **Setup:** The script automates the deployment setup. This includes deploying Token A/B, two separate LP Tokens, two DEX instances (DEX1 and DEX2), transferring ownership of LP Tokens to their respective DEXes, funding an initial LP account ('accounts[1]'), adding initial liquidity to DEX1 and DEX2 with different ratios (e.g., 1A:2B vs 1A:2.1B)

to create a price difference, deploying the 'Arbitrage' contract linked to the tokens and DEXes, and funding the 'Arbitrage' contract with starting capital (e.g., 10 A, 10 B).

- **Scenario 1 (Profitable Arbitrage):** The script calls `executeArbitrage` on the deployed bot. The console output confirmed that the `ArbitrageExecuted` event was emitted, and the arbitrage contract's token balance increased, demonstrating a successful profitable execution as required.
- **Scenario 2 (Insufficient Profit):** The script then calls `setMinProfitThreshold` on the arbitrage contract to set an extremely high threshold. It calls `executeArbitrage` again. The console output confirmed that no `ArbitrageExecuted` event was emitted and the contract balances did not change, demonstrating correct non-execution when profit is below the threshold, as required.

The simulation successfully demonstrated both required scenarios, which validates the arbitrage bot's functionality.

5 Task 4: Theory Questions

5.1 Q1: LP Token Mint/Burn Permissions

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

Answer: Only the DEX contract address itself should have the permission to mint and burn its corresponding LP tokens. This is crucial for maintaining the integrity of the pool. Allowing any other address (including the original deployer or individual users) to freely mint or burn LP tokens would break the fundamental link between the LP tokens and the actual reserves held in the pool, enabling theft or manipulation. In our implementation, this is achieved by making the `LPToken` contract `Ownable` and transferring ownership to the deployed `DEX` contract. The `mint` and `burn` functions in `LPToken` have the `onlyOwner` modifier.

5.2 Q2: DEXs fair playing ground?

Question: 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!)?

Answer: DEXs offer several features that can partially level the playing ground compared to centralized exchanges (CEXs), although advantages for resourceful traders still exist:

- **Transparency:** All trading rules (the AMM smart contract code) and transaction history are publicly visible on the blockchain. There are no hidden order books or special access privileges (like co-location for HFTs on CEXs). Everyone interacts with the same contract under the same rules.
- **Permissionless Access:** Anyone with a wallet can interact with the DEX; there are typically no complex onboarding processes, API access tiers, or preferential treatment based on trading volume that large players might exploit on CEXs.
- **Deterministic Execution (Mostly):** AMM logic is deterministic based on pool reserves. While miners can influence transaction order, the outcome of a specific swap against a specific pool state (before the swap) is predictable by the formula.

- **Reduced Counterparty Risk:** Users typically retain custody of their assets until the point of trade (interacting via their own wallet), reducing the risk associated with exchange hacks or insolvency prevalent in CEXs.

However, resourceful traders still have advantages in DeFi through concepts like Miner Extractable Value (MEV), superior infrastructure for monitoring mempools and executing faster, and the capital to perform large trades or provide significant liquidity.

5.3 Q3: Miner Advantage (MEV)

Question: 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?

Answer: Miners have visibility into the pool of pending transactions (mempool) before they are included in a block. They also have the power to arbitrarily order transactions within the block they produce. This gives the miners opportunities for Miner Extractable Value (MEV) that exploit DEX users:

- **Front-running:** A miner sees a large user swap incoming (e.g., buying Token B with Token A) that will increase the price of Token B. The miner inserts their own transaction just before the user's swap to buy Token B at the lower price. After the user's large swap pushes the price up, the miner can insert another transaction to sell their Token B at the new, higher price, capturing a risk-free profit.
- **Back-running:** A miner sees a transaction that creates an arbitrage opportunity (e.g., adding liquidity that significantly changes the price relative to another DEX). They execute the user's transaction and immediately insert their own transaction after it to capture the arbitrage profit before anyone else can.
- **Sandwich Attack:** A combination where the miner places a buy order just before the user's buy order (front-running) and a sell order just after it (back-running), profiting from the price impact caused by the user's trade.

To make a DEX completely robust against MEV is extremely difficult, as miners fundamentally control block production. However, strategies exist to mitigate it:

- **Slippage Tolerance:** Users setting tight slippage tolerances on their swaps can limit the potential profit from sandwich attacks, causing the second leg of the attack to fail.
- **Batch Auctions:** Some DEX designs move away from instant AMM execution and use batch auctions (collecting orders over a short period and settling them at a uniform clearing price) which makes front-running harder.
- **MEV-aware Protocols:** Services like Flashbots allow users to submit transactions privately to miners, bypassing the public mempool. Miners participating in these systems can execute bundles atomically, potentially preventing front-running by others but still allowing the miner to capture MEV internally or through auctions.
- **Encrypted Mempools:** Future solutions involve encrypting transactions in the mempool so miners cannot see the content until after the block order is committed.

Our simple AMM in this assignment is vulnerable to these MEV strategies.

5.4 Q4: Gas Fees Influence

Question: 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?

Answer: Gas fees are a critical factor in the economic viability of DEXs and arbitrage:

- **DEX Viability (User Perspective):** High gas fees can make swapping small amounts uneconomical. If the gas cost for a swap transaction is a large percentage of the trade value, users will be hesitant. This particularly affects smaller retail traders. DEXs on Layer 1 blockchains with high gas fees (like Ethereum mainnet during congestion) become less attractive compared to CEXs or DEXs on cheaper Layer 2 solutions.
- **DEX Viability (LP Perspective):** Adding or removing liquidity also costs gas. LPs need to earn enough in trading fees (the 0.3% in this case) over time to cover the gas costs of their deposit and withdrawal transactions, plus compensate for impermanent loss risk, to be profitable. High gas fees reduce LP profitability, potentially leading to lower liquidity.
- **Arbitrage Viability:** Arbitrage opportunities often involve small price discrepancies. An arbitrage bot must calculate if the potential profit from exploiting a price difference across two DEXs is greater than the combined gas cost of executing the necessary transactions (at least two swaps, plus approvals if needed). During high gas periods, many small arbitrage opportunities become unprofitable, allowing price discrepancies between venues to persist for longer. Arbitrage bots have to compete by using sophisticated gas price strategies to ensure execution while minimizing cost.

5.5 Q5: Gas Fee Advantages

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

Answer: Yes, gas fees can definitely create advantages, primarily favoring those with more capital or sophisticated infrastructure:

- **Priority Gas Auctions (PGAs):** When network congestion is high, users compete for block space by bidding higher gas prices. Wealthy traders or bots (like front-runners or arbitrageurs) can afford to pay significantly higher gas prices to ensure their transactions are processed faster and included before others. This allows them to consistently capture time-sensitive opportunities (like arbitrage or front-running) before retail users whose transactions might get stuck due to lower gas prices.
- **Transaction Ordering:** Miners/validators are incentivized to include transactions with higher gas prices first. This gives an advantage to those who can pay more for priority, effectively allowing them to cut in line in the mempool.
- **Economic Exclusion:** As mentioned in Q4, high base gas fees can simply make certain actions (small swaps, frequent LP adjustments) economically infeasible for users with limited capital, while remaining viable for larger players. This restricts participation for smaller users.
- **MEV Exploitation:** Actors specifically searching for MEV opportunities are often willing to pay very high gas fees (sometimes even exceeding the direct arbitrage profit, if they can chain MEV opportunities) to guarantee execution and front-run others.

5.6 Q6: Minimizing Slippage

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

Answer: Slippage in an AMM swap is the difference between the expected price (based on reserves before the swap) and the actual execution price (based on the reserves during/after the swap). It's higher for larger trades relative to pool liquidity. Ways to minimize it include:

- **Trading Smaller Sizes:** Executing smaller individual trades has less price impact on the pool reserves, resulting in lower slippage compared to one large trade. Users can manually break down large orders (though this incurs more gas fees).
- **Using High-Liquidity Pools:** Pools with larger reserves (x and y) experience less price change for a given trade size (Δx), thus resulting in lower slippage. Trading on established DEXs with high liquidity for the desired pair of tokens is the key.
- **DEX Aggregators:** Services like 1inch or Matcha route trades across multiple DEXs and liquidity pools, splitting the order to find the path with the lowest overall slippage and best net execution price, often executing partial orders on different pools simultaneously.
- **Limit Orders (on specific DEXs):** Some DEXs (often using limit order book mechanisms alongside or instead of AMMs, like Serum or dYdX, or protocols like Uniswap v3 with concentrated liquidity ranges) allow users to place limit orders that only execute at a specified price or better, effectively providing zero slippage if the order fills at the desired price.
- **Setting Slippage Tolerance:** While this doesn't reduce the slippage inherent in the trade itself, setting a tight slippage tolerance in the user interface prevents the transaction from executing if the actual price moves beyond an acceptable percentage from the expected price (e.g., due to front-running or other simultaneous trades), thus protecting the user from unexpectedly high slippage.

5.7 Q7: Slippage vs. Trade Lot Fraction Plot

Question: 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.

Answer: We need to analyze the relationship between slippage (S) and the trade lot fraction ($f = \frac{\Delta x}{x}$), where Δx is the amount of token X deposited, and x is the reserve of token X before the swap. Let y be the reserve of token Y. The constant product is $k = x \times y$.

To incorporate the DEX's 0.3% fee, we define:

- Fee factor: $\gamma = 0.997$ (representing 99.7% of the input amount after the 0.3% fee)
- Effective input amount after fee: $\gamma \cdot \Delta x$

After swapping Δx of token X, the new reserve of X is $x' = x + \gamma \cdot \Delta x$ (after fee). The new reserve of Y is y' such that $x' \times y' = k$, so $y' = \frac{k}{x'} = \frac{x \times y}{x + \gamma \cdot \Delta x}$.

The amount of Y received is $\Delta y = y - y' = y - \frac{x \times y}{x + \gamma \cdot \Delta x} = y \times \frac{\gamma \cdot \Delta x}{x + \gamma \cdot \Delta x}$.

The slippage formula (relative difference between actual price and expected price) is: $S = \left(\frac{\text{Actual Price}}{\text{Expected Price}} - 1 \right) \times 100\%$ Where:

- Expected Price (Y per X) = $\frac{y}{x}$
- Actual Price (Y per X) = $\frac{\Delta y}{\Delta x} = \frac{y \times \frac{\gamma \cdot \Delta x}{x+\gamma \cdot \Delta x}}{\Delta x} = \frac{y \cdot \gamma}{x+\gamma \cdot \Delta x}$

So, $S = \left(\frac{y \cdot \gamma/(x+\gamma \cdot \Delta x)}{y/x} - 1\right) \times 100\% = \left(\frac{\gamma \cdot x}{x+\gamma \cdot \Delta x} - 1\right) \times 100\%$

$S = \left(\frac{\gamma \cdot x - x - \gamma \cdot \Delta x}{x+\gamma \cdot \Delta x}\right) \times 100\% = \left(\frac{(\gamma-1) \cdot x - \gamma \cdot \Delta x}{x+\gamma \cdot \Delta x}\right) \times 100\%$

Now substitute the trade lot fraction $f = \frac{\Delta x}{x}$, which means $\Delta x = f \times x$:

$S(f) = \left(\frac{(\gamma-1) \cdot x - \gamma \cdot f \cdot x}{x+\gamma \cdot f \cdot x}\right) \times 100\% = \left(\frac{(\gamma-1) - \gamma \cdot f}{1+\gamma \cdot f}\right) \times 100\%$

With $\gamma = 0.997$:

$S(f) = \left(\frac{(0.997-1) - 0.997 \cdot f}{1+0.997 \cdot f}\right) \times 100\% = \left(\frac{-0.003 - 0.997 \cdot f}{1+0.997 \cdot f}\right) \times 100\%$

Plot of $S(f) = \frac{-0.003 - 0.997 \cdot f}{1+0.997 \cdot f} \times 100\%$:

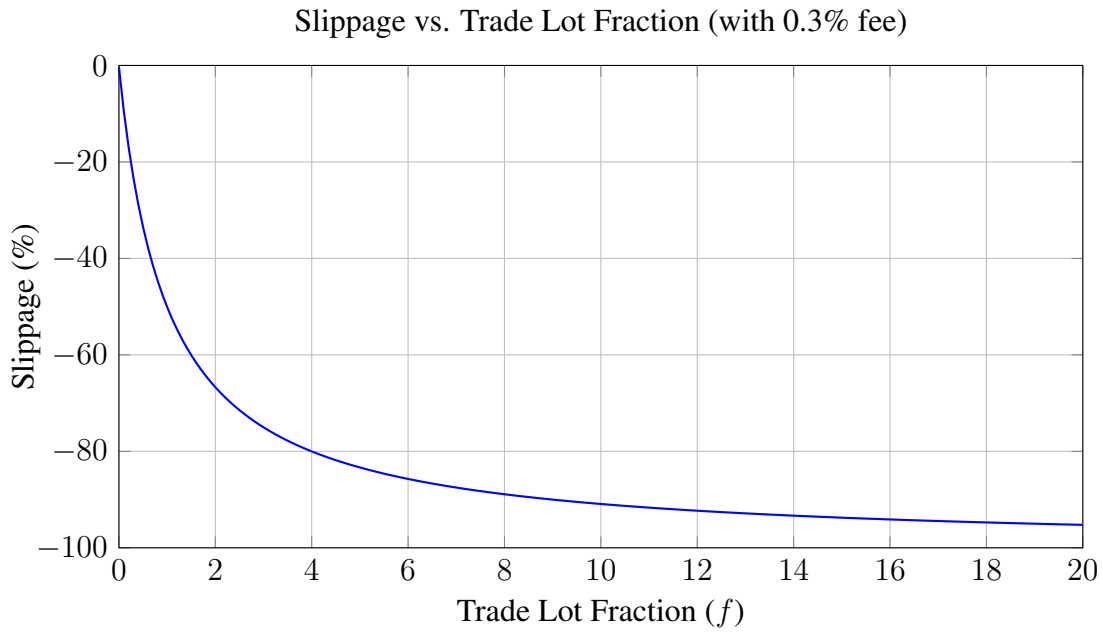

Figure 7: Plot of Slippage (%) vs. Trade Lot Fraction ($f = \Delta x/x$)