CS765 Homework 3 Report

22B0956 22b0989 22b1205

April 16, 2025

Contents

1 Task 1: Deploying Tokens

1.1 Implementation Details

We deploy three types of tokens as part of our DEX ecosystem:

- **TokenA and TokenB:** These are ERC-20 tokens implemented in `Token.sol`, based on OpenZeppelin's standard. They support:
  - `transfer`: For sending tokens between accounts.
  - `approve` and `allowance`: For granting delegated spending.
  - `transferFrom`: Used by DEX and Arbitrage contracts to facilitate token movements during swaps and liquidity operations.

  Two separate instances of this contract were deployed — one each for TokenA and TokenB. A large initial supply is minted at deployment, and the deployer distributes tokens to traders and LPs at the start of the simulation.

- **LPToken:** This is a custom ERC-20 token defined in `LPToken.sol`. It represents a user's share in the DEX liquidity pool. Key features include:
  - Minting and burning are restricted to the DEX contract (via `onlyOwner` modifier).
  - Ownership is transferred to the DEX upon deployment using `transferOwnership()`, so that only the DEX can issue or destroy LP tokens in response to liquidity changes.
  - LP balances are later used to calculate each user's entitlement during withdrawals and to track fee distributions.

  Each DEX instance is paired with its own LPToken contract to isolate accounting across liquidity pools.

1.2 Deployment and Transfer Verification

All token deployments were executed via JavaScript using `web3.eth.Contract` and `.deploy().send()` in the Remix IDE. TokenA and TokenB were deployed first, and their instances were passed to the DEX contract during its deployment.

Once deployed:

- TokenA and TokenB were distributed to all users (LPs and traders) using `transfer()`.
- Users used `approve()` to authorize the DEX to spend tokens on their behalf during swaps and liquidity provisioning.
- LP tokens were minted and assigned to users upon adding liquidity, and their balances were verified using `balanceOf()`.

All token behaviors, including transfers, approvals, and LP token mint/burn, were validated during simulation through log outputs and reserve comparisons.

2 Task 2: Implementing the DEX

2.1 Contract Overview and Functional Design

- The core decentralized exchange logic is defined in `DEX.sol`.
- Each DEX contract is associated with a liquidity pool (TokenA, TokenB) pair and its own LPToken contract.
- LP token logic is abstracted in `LPToken.sol`, a custom ERC20 token with a transferOwnership function to restrict mint/burn privileges to the owning DEX after the DEX is deployed.

2.2 Functionality in `DEX.sol`

`DEX.sol` implements the following critical functions:

- **Liquidity Management:**
  - `addLiquidity(uint256 amountA, uint256 amountB)`: Accepts tokens from the user and mints LP tokens only if tokens are provided in the correct ratio.
  - `removeLiquidity(uint256 lpAmount)`: Burns LP tokens and returns proportional reserves to the user. This also returns all the swap fee an LP has accumulated since his last `removeLiquidity()` call.
- **Swapping Mechanism:** The DEX allows users to swap between TokenA and TokenB using a constant product AMM model that maintains $x \cdot y = k$. A 0.3% fee is applied on every trade. The mechanism is supported by the following core functions:
  - `swapAforB(uint256 amount)` and `swapBforA(uint256 amount)`: These are the primary user-facing functions for swapping tokens. The contract deducts a 0.3% fee from the input, computes the output amount using the AMM formula, and updates the internal reserves accordingly.
  - `getRequiredAforB(uint256 amountB)` and `getRequiredBforA(uint256 amountA)`: These public view functions are useful for liquidity providers. They allow users to query how much TokenA they need to deposit to match a given amount of TokenB, and vice versa, in order to maintain the current reserve ratio. This ensures that LPs add liquidity in the correct proportion and helps preserve the spot price.
- **Fee Tracking and LP Rewards:** Our DEX implementation includes a mechanism to track and distribute swap fees back to liquidity providers (LPs) in a fair and proportional manner. This is handled via the following structure:
  - **Fee Accumulation:** For every swap, a 0.3% fee is collected and not immediately distributed. Instead, it is tracked using:

        - `mapping(address => uint256) feeA;` — stores pending TokenA fees per LP.
        - `mapping(address => uint256) feeB;` — stores pending TokenB fees per LP.

    The fee is proportionally split among LPs based on their LP token share at the time of the swap. Each LP's share is calculated as:

$$\text{LP Share} = \left(\frac{\text{lpBalance}}{\text{totalSupply}}\right) \times \text{feeCollected}$$

    The value is then added to the LP's corresponding fee mapping.

    - **Distribution on Withdrawal:** When LPs call `removeLiquidity()`, the contract:
        - Burns their LP tokens.
        - Returns their proportional reserves of TokenA and TokenB.
        - Adds their accumulated `feeA` and `feeB` to the final withdrawal.
        - Resets their fee balances to zero.
    - **Benefits of the Design:**
        - No gas cost for distributing fees mid-operation — only paid on exit.
        - Ensures fair distribution of fee as it should be proportional to how much liquidity an LP is providing.
        - Fully deterministic and front-running resistant.

- **Reserve Queries:**
    - `reserveA()`, `reserveB()` provide internal token reserves.
    - `spotPrice()` returns both reserves as a tuple for use by the arbitrage contract.
    - `getTVL()` returns the total value locked across both reserves.

2.3 Functionality in `LPToken.sol`

This file defines a basic ERC20 token with minting and burning restricted to the DEX owner.

- The constructor accepts the initial owner (DEX address).
- `mint()` and `burnFrom()` are only callable by the owner.
- Ownership is transferred using `transferOwnership()` after deployment to the DEX.

2.4 Simulation with JS

The entire DEX ecosystem was deployed and simulated using the `deployAndSimulateDEX()` script. This script:

- Deploys `TokenA`, `TokenB`, `LPToken`, and `DEX`.
- Distributes tokens to LPs and Traders.
- Adds initial liquidity using LP accounts.
- Executes 50-100 randomized user actions (swaps, adds/removes) to emulate real DEX activity.

This ensured robust testing of all contract features under dynamic conditions.

2.5 Security and Validations

Security was a key consideration in the design of our smart contracts. We implemented several layers of checks and safe practices to ensure correctness, avoid vulnerabilities, and maintain user trust. Below are the major strategies used:

- **Safe Arithmetic:** All contracts were written using Solidity `^0.8.0`, which includes built-in overflow and underflow protection by default. This removes the need for external libraries like SafeMath and ensures all arithmetic operations revert on error.
- **Zero-value Checks:** Swap and liquidity functions include explicit checks to reject zero-value inputs, which helps prevent unexpected behavior or gas-wasting calls.
- **Liquidity Ratio Validation:** In `addLiquidity()`, we ensure that the ratio of TokenA to TokenB matches the existing pool to maintain price consistency and prevent pool manipulation. This is precisely done by using getAforB() or getBforA(). These ensure the correct token ratio for adding liquidity.
- **Ownership Control:** LPToken minting and burning is restricted via an `onlyOwner` modifier, and ownership is transferred to the DEX contract after deployment. This guarantees only the DEX can control LP supply.
- **Fee Accounting Isolation:** Each LP's accumulated fees are tracked using individual mappings (`feeA`, `feeB`). These are updated only on swap and claimed upon withdrawal, minimizing the attack surface (as the LPs cannot manipulate to get more fees than what they deserve) and computation overhead (which would occur if we released fees for every swap).
- **Fail-safes in Transfers:** All ERC20 `transfer()` and `approve()` calls are wrapped in `require()` to ensure they succeed. If any token operation fails (due to allowance issues, low balance, etc.), the transaction reverts.

- **Arbitrage Guardrails:** In the arbitrage contract, swaps only occur if the computed profit exceeds a defined threshold. This prevents unnecessary gas spending and guards against edge cases with minimal or rounding-error profit.
- **Decimal Handling without Floating Points:** Since Solidity does not support floating point arithmetic, all percentage-based calculations (e.g., LP share of fees, token ratios) are handled using integer math scaled by a large factor (typically 10^{18}). For example, to compute a spot price or fee share ratio, we multiply numerators by 10^{18} before division to preserve precision and emulate fixed-point math.
- **Initial LP Token Minting with Geometric Mean:** When the first liquidity provider seeds the DEX with amounts a of TokenA and b of TokenB, we mint LP tokens equal to $\sqrt{a \cdot b}$. This choice ensures:
  - The LP token supply reflects the geometric depth of the initial liquidity, not just the sum.Sum might be misleading as it dosent represent the composition of the reserve.
  - It treats TokenA and TokenB symmetrically — neither token dominates the minting formula.
- **JavaScript Simulation Safety with `try/catch`:** In the simulation scripts ('simulate_DEX.js' and 'simulate_arbitrage.js'), all user actions (e.g., swaps, liquidity operations) are wrapped in 'try/catch' blocks. This prevents the simulation from crashing due to individual transaction failures, such as:
  - Insufficient token balances.
  - Zero-value operations.
  - Failed 'approve' or 'transfer' calls.

  By catching and logging these exceptions, the simulation continues to execute and provides meaningful feedback for debugging or edge case detection.

These security measures collectively ensure that both user operations and internal accounting remain robust under simulation and edge-case conditions.

3 Task 2: Testing

3.1 Simulation

The simulation script simulate_DEX.js is a JavaScript script designed to run on the Remix IDE. It automates the deployment of the DEX and token contracts, initializes user accounts as liquidity providers (LPs) and traders, and performs a series of randomized transactions (swaps, liquidity additions, and withdrawals). Throughout the simulation, it collects key metrics such as TVL, reserve ratios, swap volumes, fees, spot prices, and slippage, then outputs CSV files for further analysis and plotting.

3.2 Metrics Tracked

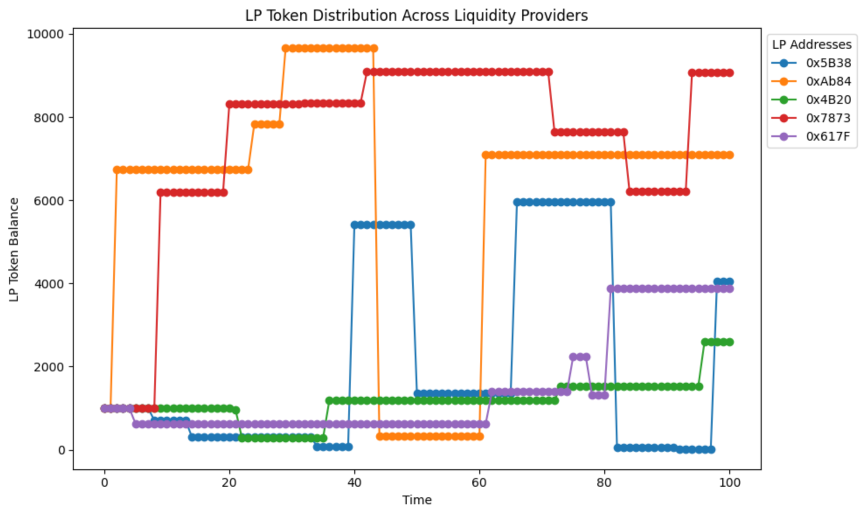


Figure 1: LP Distribution

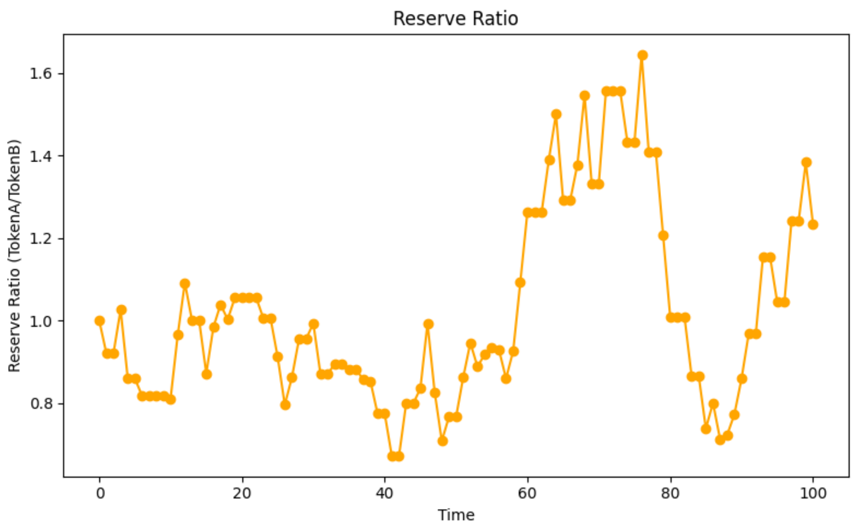


Figure 2: Reserve Ratio

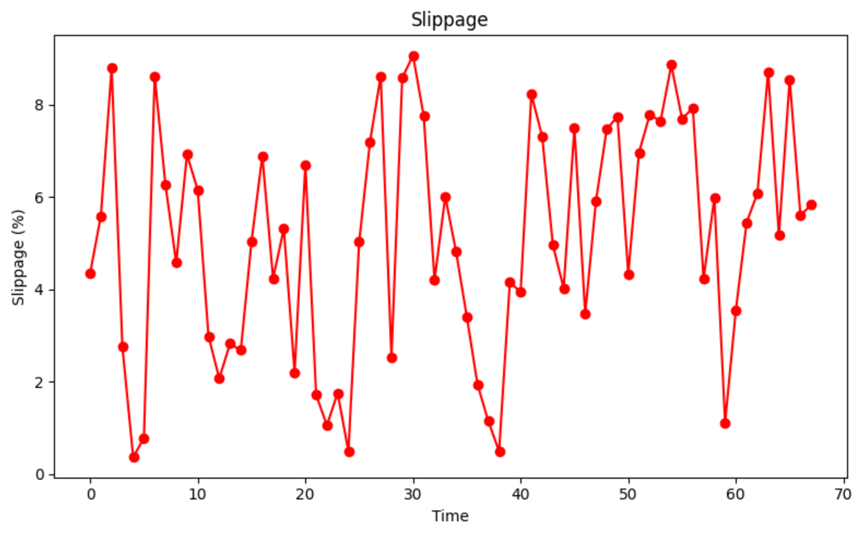


Figure 3: Slippage

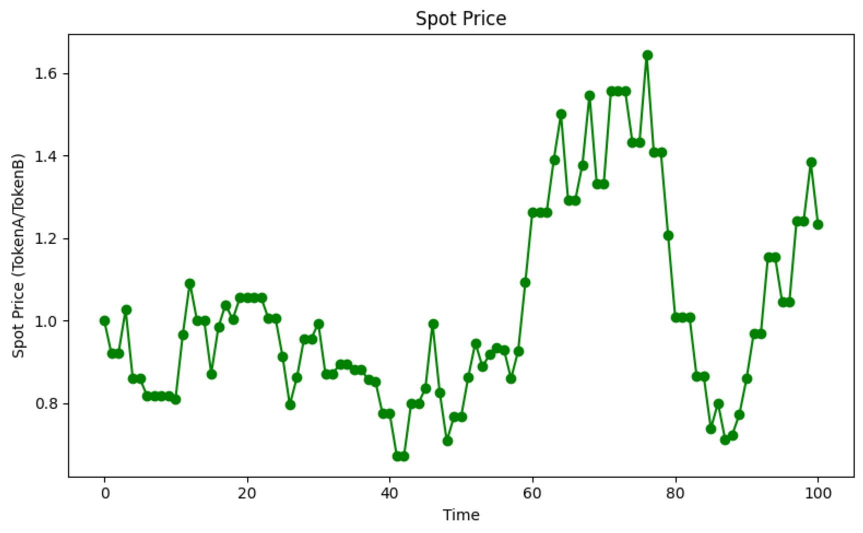


Figure 4: Spot Price

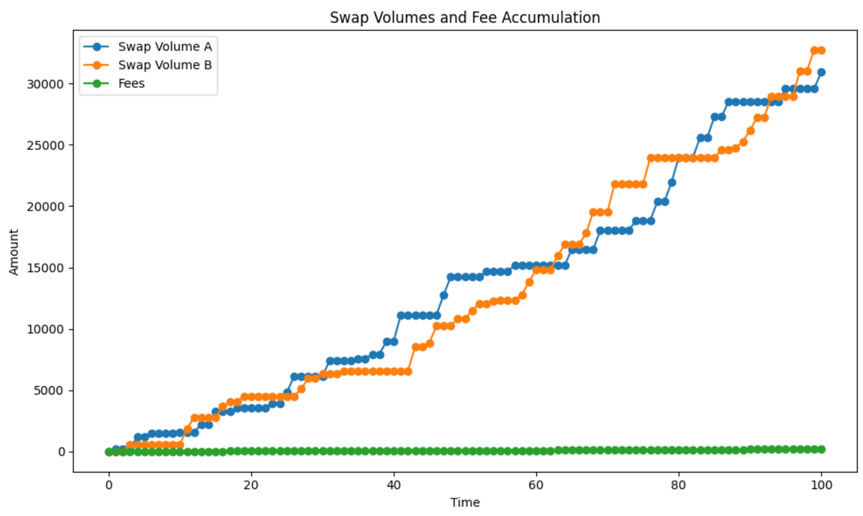


Figure 5: Swap Volume Fees

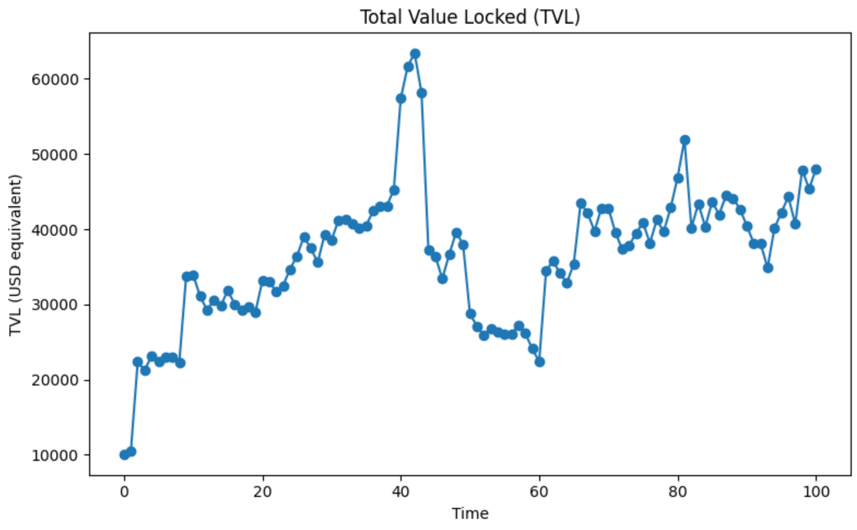


Figure 6: TVL

4 Task 3: Arbitrage

In this task, we basically had to show that if there is a trader who is capable of trading in two different DEXs, he can exploit arbitrages if such an opportunity arises.

4.1 Arbitrage.sol Details

The `Arbitrage.sol` smart contract is responsible for identifying and executing arbitrage opportunities between two DEXs.

It uses the following key functions:

- `constructor(address dex1, address dex2, address tokenA, address tokenB)`
  Initializes the arbitrage contract with references to both DEX contracts and the token contracts. It stores these addresses for internal operations.

- `spotPrice(address dex) public view returns (uint)`
  Returns (Reserve A, Reserve B) present in the DEX. This is further used to calculate the spotprice.

- `getAmountsOut(uint amountIn, uint reserveIn, uint reserveOut)`
  A pure helper function that implements the AMM swap pricing logic using the constant product formula:

  $$\text{amountOut} = \frac{\text{amountIn} \cdot 997 \cdot \text{reserveOut}}{1000 \cdot \text{reserveIn} + 997 \cdot \text{amountIn}}$$

  This accounts for a 0.3% fee and is used to simulate potential profit from swaps.

- `executeArbitrage(uint amount, uint minProfit) public returns (bool)`
  This is the main arbitrage function. I consider all 4 possible arbitrage cycles possible. If there were no swap fee, and the reserves were much greater than the swap amounts, then considering 2 of these four would be sufficient. However, to be safe, I am considering all the possibilities. We start with the 'amount' of tokens of the respective token as the initial amount which is passed as a parameter. It:

  1. Computes potential returns for all four paths:
     - A $\rightarrow$ B $\rightarrow$ A via DEX1 then DEX2
     - A $\rightarrow$ B $\rightarrow$ A via DEX2 then DEX1
     - B $\rightarrow$ A $\rightarrow$ B via DEX1 then DEX2
     - B $\rightarrow$ A $\rightarrow$ B via DEX2 then DEX1
  2. Select the most profitable, provided the profit is greater than `minProfit`.
  3. Executes the swaps by calling `swapAforB` or `swapBforA` on the appropriate DEX contracts.
  4. Returns true if DEX1 was used first and then DEX2 otherwise returns false. We dont care about the return value when there was no successful arbitrage.

- `approveTokens()`
  An internal utility function used to approve token spending by DEX contracts. This ensures that the arbitrage contract can transfer tokens on the fly.

This contract is stateless across calls and is designed to return all swapped capital and profits to the caller (arbitrageur). It requires that the arbitrageur pre-fund the contract with an amount of TokenA and/or TokenB before calling `executeArbitrage()`.

4.2 Arbitrage Detection Logic

We are using the `simulate_arbitrage.js` script to check our arbitrage detector. The details of the setup are as follows.

1. **Setup:** Two DEX contracts are deployed with slightly different liquidity conditions. Each DEX is seeded with liquidity from two LPs, and 3 traders. We also have our special trader (account[11]) which is the arbitrager. Now 10 random swaps are executed on each DEX to simulate market dynamics and change the market reserve ratio in these markets.
2. **Arbitrage Contract Deployment:** An `Arbitrage` smart contract is deployed with knowledge of the two DEX addresses and token contracts. This contract will check for arbitrage opportunities and execute profitable trades.
3. **Profit Calculation:** After executing the trade path, the balance of TokenA and TokenB in the arbitrageur's wallet is compared to the initial balance. The deltas in both tokens (`delta_A` and `delta_B`) determine the net profit.
4. **Decision Logic:**
   - If $\Delta_A + \Delta_B > 0$, arbitrage is deemed successful.
   - Based on the direction of gain (`delta_A` or `delta_B`), the console logs whether it was an A→B→A or B→A→B arbitrage and based on the return value of the executeArbitrage function it is determined in which DEX order.

Example Output

```
Arbitrage Found: 10 tokenA of A->B->A gives finally: 10.082 TokenA via DEX1, DEX2
```

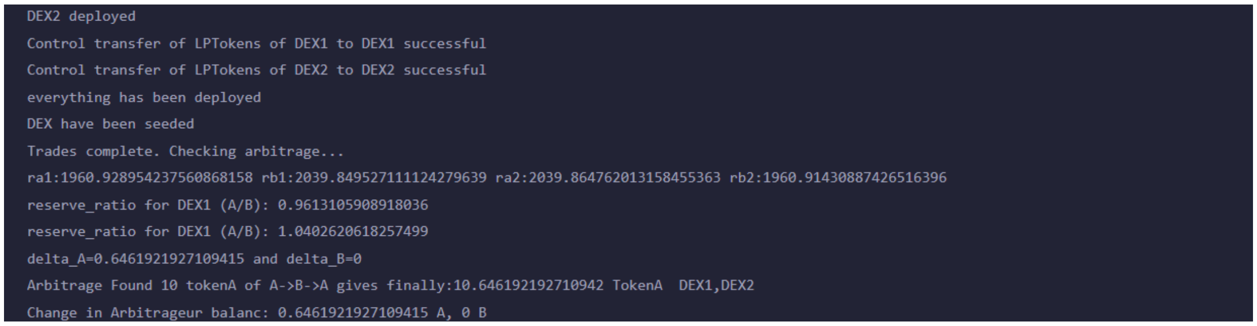
This result implies that 10 TokenA were swapped to TokenB and back, resulting in 0.082 TokenA profit, and the first swap was in DEX1 (A-¿B) and the second was in DEX2 (B-¿A).

4.3 Testing

We have limited the amount involved in the arbitrage to 10 and the threshold to 0.1 in our simulation. The reserves have initially 2000 tokens of each. Therefore the 1% condition is taken care of.

- **Profitable Arbitrage Case:**

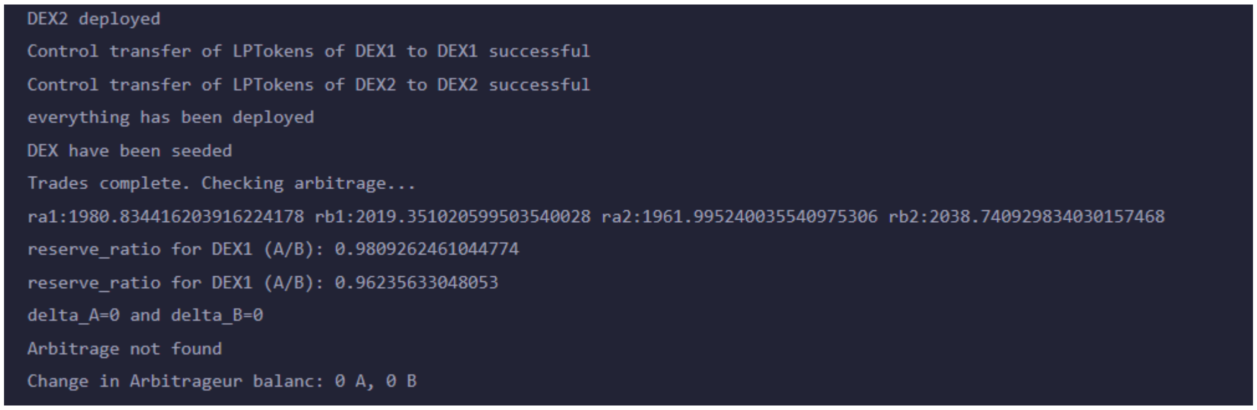
  The following simulation demonstrates a successful arbitrage opportunity. The arbitrageur starts with 10 TokenA, and by performing A → B on DEX1 and B → A on DEX2, ends up with more than 10 TokenA, satisfying the profit threshold.



```
DEX2 deployed
Control transfer of LPTokens of DEX1 to DEX1 successful
Control transfer of LPTokens of DEX2 to DEX2 successful
everything has been deployed
DEX have been seeded
Trades complete. Checking arbitrage...
ra1:1960.928954237560868158 rb1:2039.849527111124279639 ra2:2039.864762013158455363 rb2:1960.914308874265l6396
reserve_ratio for DEX1 (A/B): 0.9613105908918036
reserve_ratio for DEX1 (A/B): 1.0402620618257499
delta_A=0.6461921927109415 and delta_B=0
Arbitrage Found 10 tokenA of A->B->A gives finally:10.646192192710942 TokenA  DEX1,DEX2
Change in Arbitrageur balanc: 0.6461921927109415 A, 0 B
```

- **Failed Arbitrage Case:**

  In this case, the arbitrage contract scans all possible trade paths but determines that the profit does not exceed the required threshold. Therefore, no trade is executed and the function returns false.



```
DEX2 deployed
Control transfer of LPTokens of DEX1 to DEX1 successful
Control transfer of LPTokens of DEX2 to DEX2 successful
everything has been deployed
DEX have been seeded
Trades complete. Checking arbitrage...
ra1:1980.834416203916224178 rb1:2019.351020599503540028 ra2:1961.995240035540975306 rb2:2038.740929834030157468
reserve_ratio for DEX1 (A/B): 0.9809262461044774
reserve_ratio for DEX1 (A/B): 0.96235633048053
delta_A=0 and delta_B=0
Arbitrage not found
Change in Arbitrageur balanc: 0 A, 0 B
```

5 Theory Questions

1. Only the DEX contract (or a designated authority within it) should mint and burn LP tokens to ensure that they accurately represent a user's share of the liquidity pool.

2. DEXs use an algorithm (like the constant product formula) that sets transparent, rule-based prices and removes the need for order books, giving equal access to all traders regardless of their resources

3. Miners can exploit decentralized exchange (DEX) transactions in the mempool through various strategies such as front-running, where they spot profitable trades and insert their own transactions ahead by offering higher gas fees. Another common tactic is the sandwich attack, which involves placing one transaction before and another after a large swap to benefit from the resulting price movement. Additionally, miners may engage in transaction reordering to maximize their own gains or even practice transaction censorship by deliberately excluding specific transactions from being included in blocks.

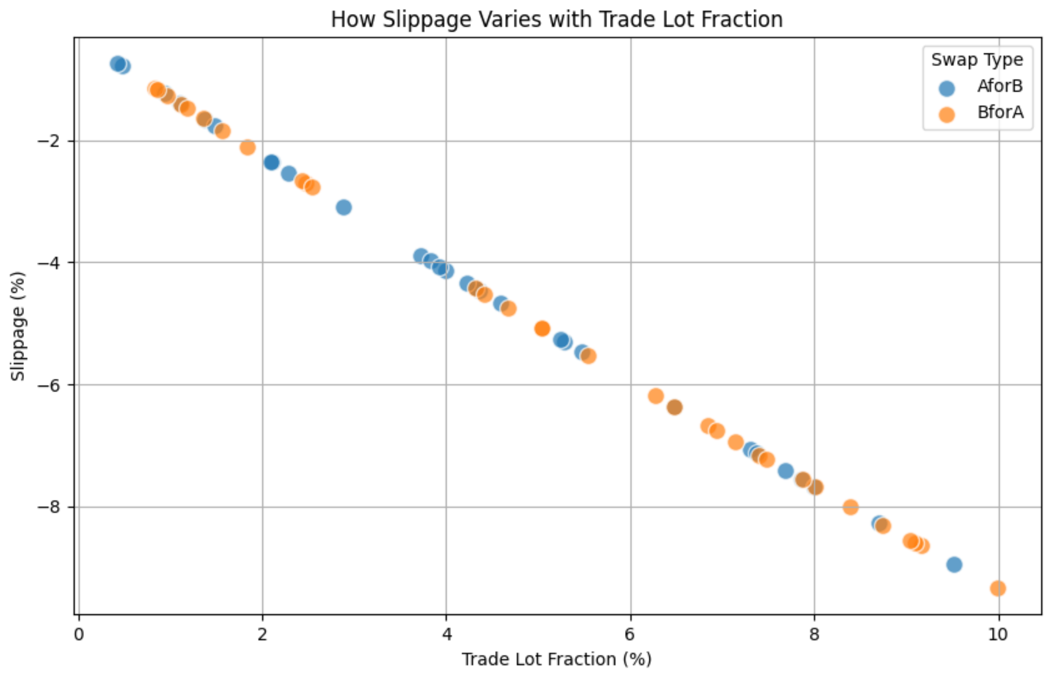
To enhance the resilience of DEXs against such manipulations, techniques like commit-reveal schemes, batch auctions, and time-weighted average price (TWAP) mechanisms are employed.

4. High gas fees can make trades and arbitrage unprofitable if fees eat into potential gains. This can deter small traders and require that arbitrage opportunities must be large enough to cover these additional costs.

5. Traders who can afford to pay higher gas fees or use faster transaction methods (e.g., private channels) can prioritize their trades, effectively gaining a timing advantage over less-resourced traders.

6. Slippage in decentralized exchanges can be reduced through several strategies. Deeper liquidity pools help by minimizing the price impact of individual trades, while multi-route swaps spread large orders across multiple pools to achieve better rates. Concentrated liquidity, as seen in platforms like Uniswap v3, enables liquidity providers to allocate funds within specific price ranges, enhancing efficiency. Batch auctions further mitigate slippage by aggregating multiple trades into a unified price discovery process. Additionally, the use of limit orders allows traders to define acceptable slippage levels or wait for more favorable prices. Finally, time-weighted execution strategies help by breaking large trades into smaller portions and executing them gradually to reduce market impact.

7. For a constant product AMM, as the trade lot fraction (amount traded/reserve) increases, slippage decreases.



6 Appendix

- **Contracts:** `Token.sol`, `LPToken.sol`, `DEX.sol`, `Arbitrage.sol`
- **Simulations:** `simulate_DEX.js`, `simulate_arbitrage.js`
- **README:** Description of each file and how to run simulations.