CS765 Homework 3

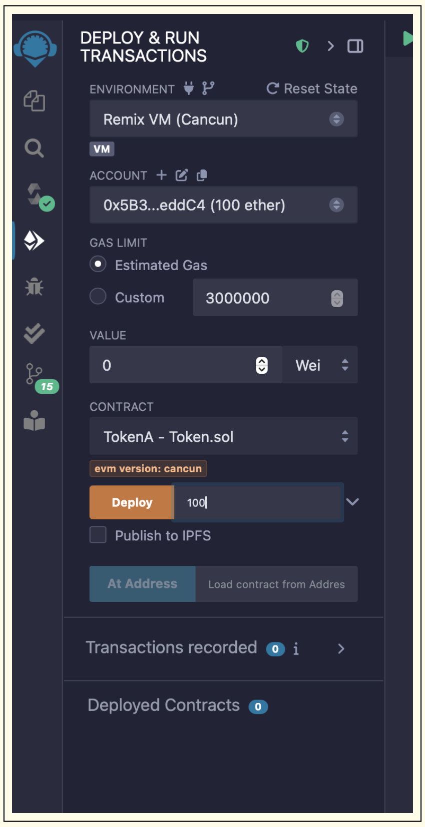
Building your own Decentralized Exchange Report

Saksham Rathi (22B1003), Kavya Gupta (22B1053), Mayank Kumar (22B0933)
Department of Computer Science,
Indian Institute of Technology Bombay

Task 1

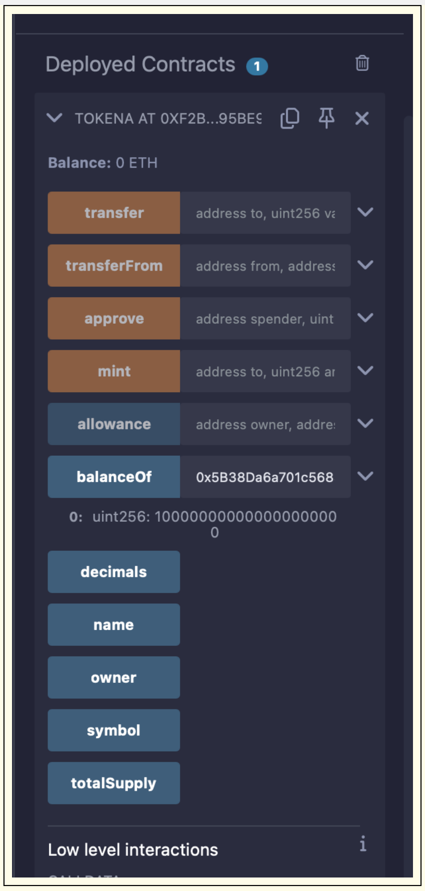
The token classes have been created in the file `Token.sol` using the ERC20 template. In each of the classes, the constructor and the mint functions have been defined. The constructor accpets the initial supply of the token, and multiples by a large number, to convert the amount into the smallest indivisible units of ERC20. The mint function, accepts the addres of the receiver, and the amount to be sent. Here, are the steps which we had followed to check and run our implementations in this task:

- Firstly, we compile the file `Token.sol`.
- We then add an initial supply of 100, and deploy our implementation. (The console shows the confirmation message.)



ENVIRONMENT: Remix VM (Cancun)
ACCOUNT: 0x5B3...eddC4 (100 ether)
GAS LIMIT: Estimated Gas; Custom 3000000
VALUE: 0 Wei
CONTRACT: TokenA - Token.sol
Deploy 100

- We then check the balance of the account from which we had deployed the token, which shows correctly:



Deployed Contracts 1

TOKENA AT 0XF2B...95BE9

Balance: 0 ETH

transfer address to, uint256 va

transferFrom address from, address

approve address spender, uint

mint address to, uint256 ar

allowance address owner, addre

balanceOf 0x5B38Da6a701c568

0: uint256: 100000000000000000000

decimals

name

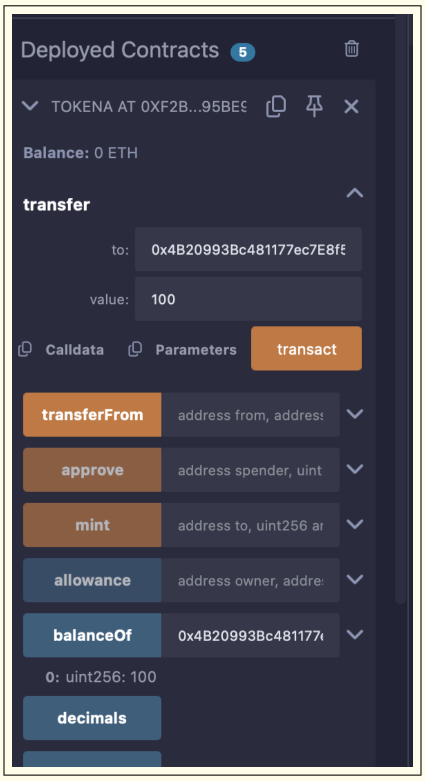
owner

symbol

totalSupply

Low level interactions

- After this, we tested the transfer function, which accepts a `to` address (chosen from the accounts remix gives us). After this, we checked the balance of both the sender and receiver accounts, and they reflected the transfer correctly. Similar to this, we have also tested TokenB.



Deployed Contracts 5

TOKENA AT 0XF2B...95BE9

Balance: 0 ETH

transfer

to: 0x4B20993Bc481177ec7E8f5

value: 100

Calldata Parameters transact

transferFrom address from, address

approve address spender, uint

mint address to, uint256 ar

allowance address owner, addre

balanceOf 0x4B20993Bc481177

0: uint256: 100

decimals

Task 2

Implementation:

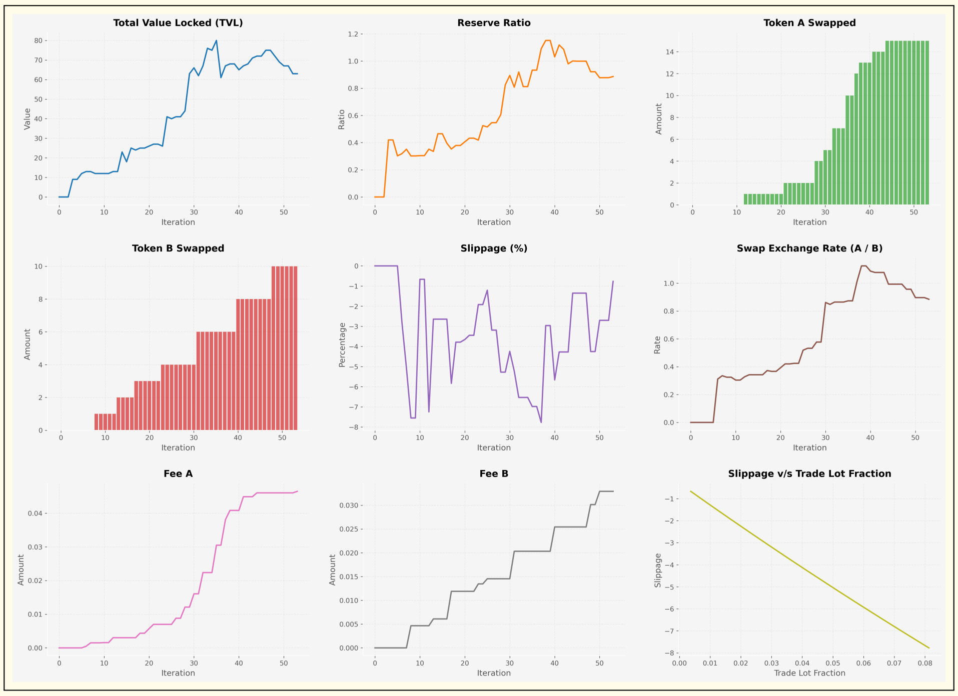
- We have implemented the LPToken contract in the file `LPToken.sol`. It has the constructor, mint and burn as the primary functions.
- We have implemented our decentralized exchange in the file `DEX.sol`. It accepts the addresses of tokens A, and B, in its constructor. The function `getSpotPriceBperA()` returns the ratio of the current reserves (multiplied by a large number to avoid floating point issues).
- The function **addLiquidity** takes the amounts of tokens A, B to be deposited. `amountA` to be added is picked uniformly from what the LP has at that moment, then we find `amountB` to be added using the current reserve ratio of the DEX. If the LP has less than `amountB` of token B, then the transaction is skipped.
- Similar to this, we have the function **removeLiquidity** function which takes the number of lpTokens to be taken out, and finds the number of tokens of A and B to be returned, and modifies the reserves accordingly (and transfers these tokens to the message sender). Again the amount of lpTokens to be removed is picked uniformly from the LP's balance.
- Then, we have two swap functons ((A for B) and (B for A)). Both of them accept the amount of the input tokens. We then find the corresponding amountB to be returned, using the current reserve ratio. After this we deduct the swap fees, from the output tokens. The reserves are modified, and tokens are transferred to the trader who had requested for the swap.
- The function `getReserves()` just returns the number of tokens in the reserve of both the categories. Also, the amount of lpTokens are calculated as the square root of the amount of tokens A and B, to preserve the ratios.

Simulation:

- Our simulation is present in the file `simulate_dex.js`. For running, we need to deploy both the tokens and DEX, and modify the addresses in these files accordingly. There are 5 liquidity providers and 8 traders.
- We initialize the balances of LP with random numbers (all the token numbers are multiplied by 1e10, to get rid of floating point issues). Then we run our simulation for N number of iterations, where N is chosen random (uniformly) from $[50, 100]$.
- We then pick an index amongst the actors randomly. Based on the nature of the actor (whether he is an LP or a trader), we pick the corresponding actions (adding or removing liquidity and swapping respectively). We then simulate these actions through the functions present in `DEX.sol`.
- Each LP is assigned an initial balance of tokens A and B, picked uniformly from fixed values which are parameters of the simulation. Before each swap, some amount of money is added to the account of the trader to simulate the real world scenarios. The amount to be added to trader's account is also picked uniformly from a fixed range (parameter).
- After each iteration, we calculate various metrics, such as reserves, spot prices and TVL. We also calculate the slippage if it was a swap action. At the end, we print the balances of all the liquidity providers.

Analysis of Simulation

- Parameters: top_up_trader = 10 | initTokenA = 20 | initTokenB = 30

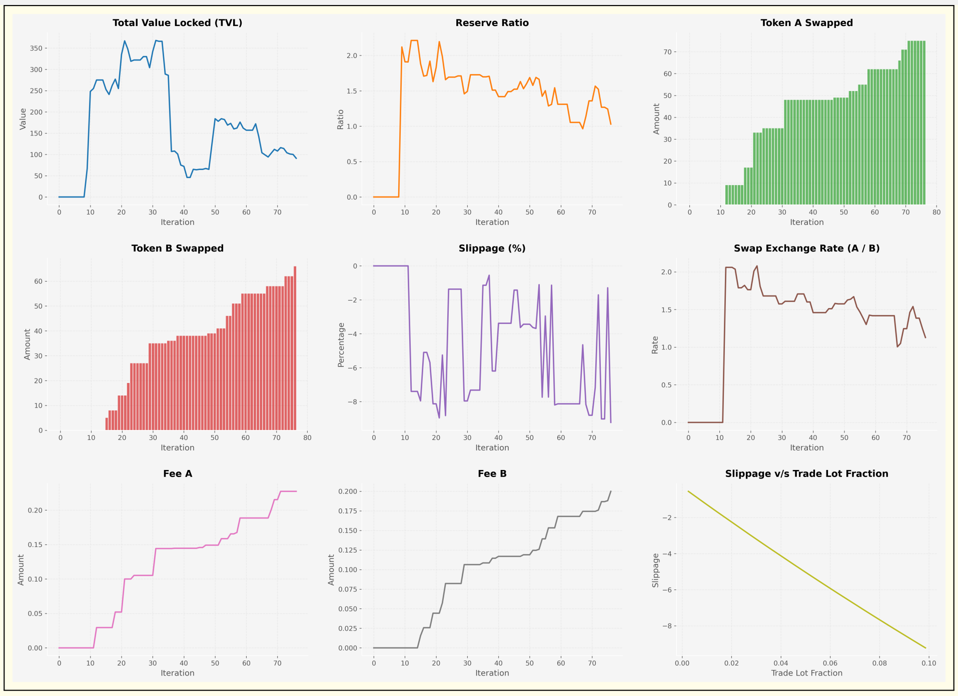


Liquidity Provider	TokenA	TokenB	LP Tokens
LP 1	5.1754456846	10.5640529108	4.1652749836
LP 2	2.3795822495	3.0504153854	7.0176560908
LP 3	2.168353112	3.8306966207	0.6463583539
LP 4	2.0400567793	2.0378392928	19.2435202491
LP 5	13.9399356394	11.6636450539	1.2182765932

Table 1: Liquidity Provider Holdings

  - TVL shows modest growth to approximately 80 units with gradual stabilization, reflecting limited capital inflow consistent with the low `top_up_trader` value.
  - The reserve ratio shifts from 0.4 to just above 1.0, indicating that TokenB becomes relatively more valuable over time.
  - Trading volumes remain small (maximum of 15 for TokenA and 10 for TokenB), with fee accumulation minimal (around 0.045 for TokenA).

- Parameters: top_up_trader = 30 | initTokenA = 100 | initTokenB = 80

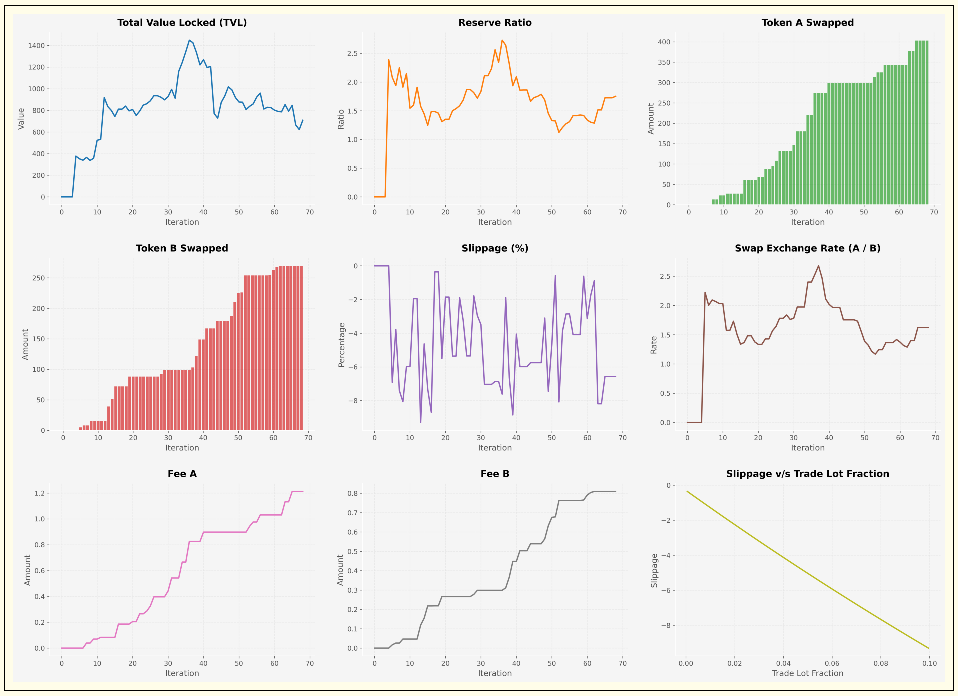


Liquidity Provider	TokenA	TokenB	LP Tokens
LP 1	25.6164670888	69.5504544782	4.4882560234
LP 2	21.4857045605	11.2727575831	26.7101230874
LP 3	37.8929316075	29.5091600677	5.6611821029
LP 4	102.1928345161	66.2254862988	7.4419151472
LP 5	73.3895539296	47.8399550284	18.2130972465

Table 2: Liquidity Provider Holdings

- The distribution of LP tokens is more balanced than in Run 1, though still showing significant variation.
- TVL exhibits high volatility, with dramatic rises and falls—peaking at roughly 350 before declining to about 100—suggesting significant liquidity additions and removals.
- Token swap volumes show consistent growth, with TokenA reaching approximately 70 units and TokenB about 60 units by iteration 75.

- Parameters: top_up_trader = 30 | initTokenA = 200 | initTokenB = 50

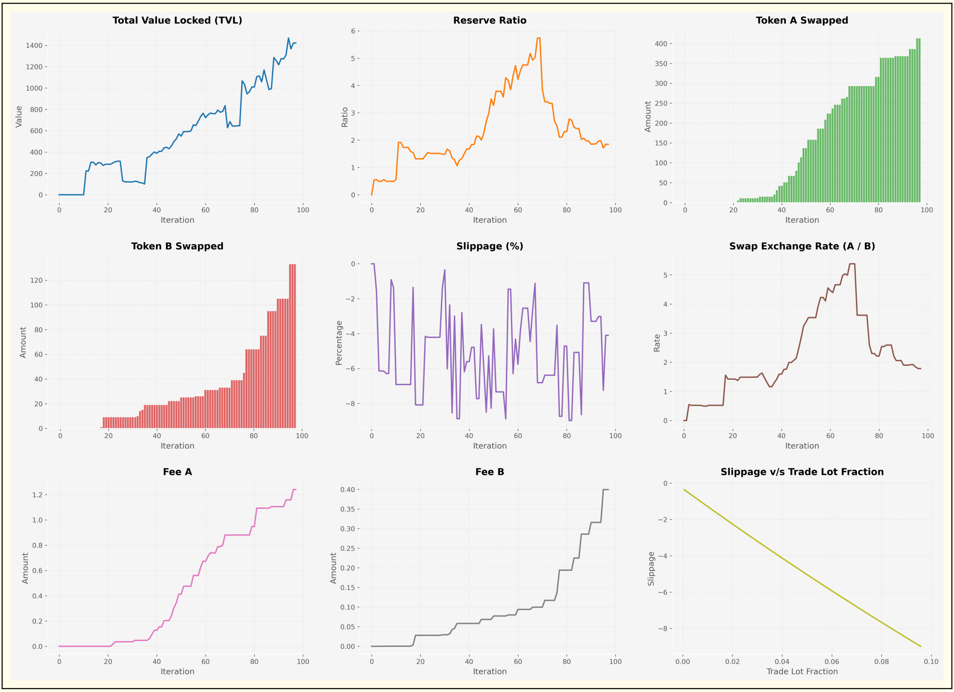


Liquidity Provider	TokenA	TokenB	LP Tokens
LP 1	224.0532091266	119.556011448	0.0717284559
LP 2	156.3045386631	79.2674286907	35.6346872088
LP 3	25.5358890367	11.0430324706	139.8975542293
LP 4	151.7413347438	86.8236794033	8.8904508712
LP 5	4.0847616454	3.6203343561	74.9860734094

Table 3: Liquidity Provider Holdings

- LP3 holds the most LP tokens (139.90) despite having relatively modest token balances, indicating they likely entered early at advantageous prices.
- TVL reaches significantly higher values—peaking around 1400—compared to previous runs, demonstrating the impact of a higher initial TokenA ratio on total pool value.
- The reserve ratio fluctuates between 1.0 and 2.5, indicating more volatility, consistent with the initial 4:1 TokenA:TokenB imbalance.
- Trading volumes are substantially higher (TokenA reaching around 400 and TokenB about 250), resulting in more fee accumulation (approximately 1.2 for TokenA and 0.8 for TokenB).

• Parameters: top_up_trader = 100 | initTokenA = 40 | initTokenB = 80



Liquidity Provider	TokenA	TokenB	LP Tokens
LP 1	12.8877859269	6.2303099251	356.0028893394
LP 2	250.2322450291	120.270245915	98.7049065441
LP 3	37.1845437813	20.7871126896	131.8751285895
LP 4	17.9212970132	25.8963970129	0.8806163605
LP 5	1.2510564453	1.4864916277	31.7637575949

Table 4: Liquidity Provider Holdings

- The Reserve Ratio (TokenB/TokenA) exhibits high volatility, beginning below 1, climbing to nearly 6 at iteration 60, then sharply declining to around 2 by iteration 100, demonstrating how dramatically the pool composition can shift during trading activity.
- Both Token A and Token B swap volumes increase exponentially over time, with Token A swaps reaching over 400 units and Token B reaching approximately 130 units by iteration 100, reflecting increased trading activity as the pool gained liquidity.

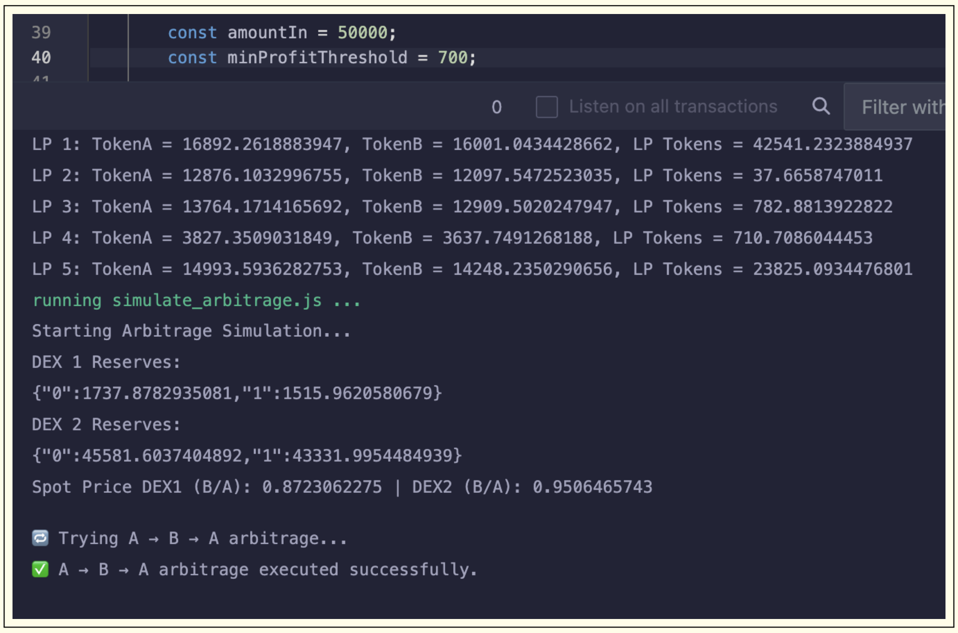
  - Fee accumulation accelerates over time for both tokens, with Fee A growing more steadily while Fee B shows a sharp exponential increase after iteration 70, corresponding to the period when Token B swap activity intensified.

General Observations:

- Early liquidity providers typically receive disproportionately high LP token counts relative to their final token holdings, demonstrating a significant first-mover advantage in DEX liquidity provision.
- The `top_up_trader` parameter dramatically influences market dynamics, with higher values creating larger reserves and more extreme price movements throughout the simulation.
- Initial token ratios (TokenA:TokenB) tend to shift substantially during trading activity, with some simulations showing complete reversals, illustrating how market forces reshape liquidity pool composition.
- LP token distribution remains highly uneven across all simulation runs, highlighting the inherent volatility and unpredictability of returns for liquidity providers in DEX systems.
- The constant product formula ($x \times y = k$) creates predictable but sometimes counterintuitive distributions of assets after multiple trading operations, as evidenced by LP positions that don't correlate linearly with token balances.

Task 3

We have implemented the arbitrage in the file `arbitrage.sol`. The constructor accepts the addresses of the two DEXes. Then there are two functions, which are responsible for executing A-B-A, and B-A-B arbitrages. Firstly, we ensure the execution was profitable (greater than a profit threshold). After this we execute those transactions (an actual transfer). The DEXes are already simulated using the previous `.js` file, so their spot prices would be different. The other functions check for profitability, and the amount we would get out from some transaction, after deducting the swap fees.
The real simulation is present in the file `simulate_arbitrage.js`. It takes the addresses of the deployed DEXes, and Arbitrage. It then tries both A-B-A and B-A-B directions, and prints appropriate messages (both in cases of success and failure). The `minProfitThreshold` is set in this file, which when changed, can lead to failure/success of an arbitrage.
Here are the screenshots of successful and failed arbitrage executions:
First Run (Successful Arbitrage)



```
const amountIn = 50000;
const minProfitThreshold = 700;
```

```
LP 1: TokenA = 16892.2618883947, TokenB = 16001.0434428662, LP Tokens = 42541.2323884937
LP 2: TokenA = 12876.1032996755, TokenB = 12097.5472523035, LP Tokens = 37.6658747011
LP 3: TokenA = 13764.1714165692, TokenB = 12909.5020247947, LP Tokens = 782.8813922822
LP 4: TokenA = 3827.3509031849, TokenB = 3637.7491268188, LP Tokens = 710.7086044453
LP 5: TokenA = 14993.5936282753, TokenB = 14248.2350290656, LP Tokens = 23825.0934476801
running simulate_arbitrage.js ...
Starting Arbitrage Simulation...
DEX 1 Reserves:
{"0":1737.8782935081,"1":1515.9620580679}
DEX 2 Reserves:
{"0":45581.6037404892,"1":43331.9954484939}
Spot Price DEX1 (B/A): 0.8723062275 | DEX2 (B/A): 0.9506465743

🔁 Trying A → B → A arbitrage...
✅ A → B → A arbitrage executed successfully.
```

Initial Reserves:

$$\text{DEX 1: } \{"0" : 1737.8782935081, "1" : 1515.9620580679\}$$
$$\text{DEX 2: } \{"0" : 45581.6037404892, "1" : 43331.9954484939\}$$

Step 1: Swap A $\rightarrow$ B on DEX1 with 0.3% fee

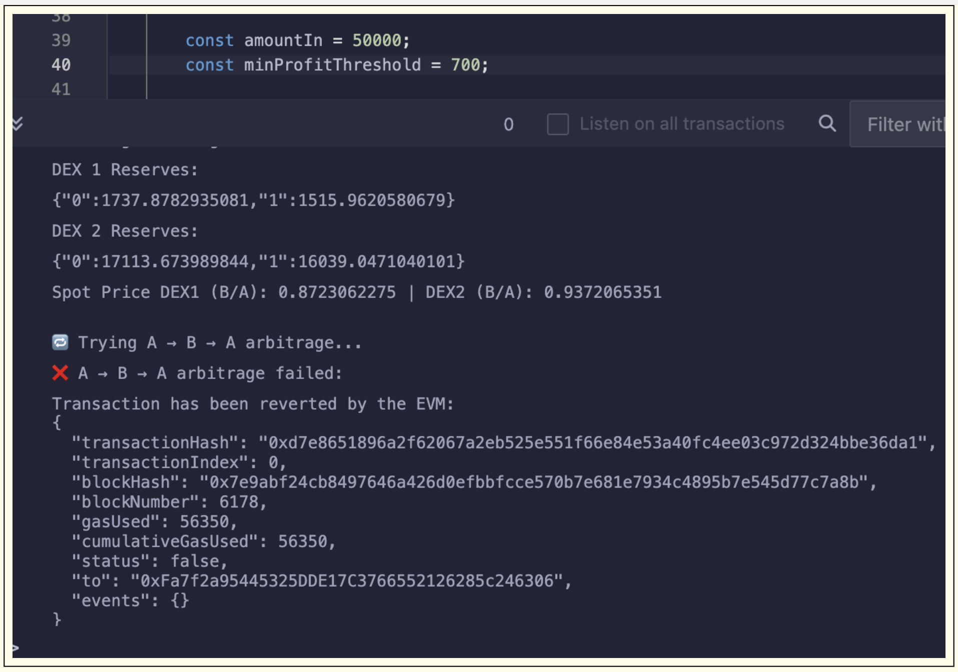
$$\text{Input: } 50,000 \text{ A}$$
$$\text{After fee: } 50,000 \times (1 - 0.003) = 49,850 \text{ A}$$
$$\text{Constant product: } (1737.87 + 49,850) \times (1515.96 - y) = 1737.87 \times 1515.96$$
$$\text{Solving for } y \approx 1,408.64 \text{ B received}$$

Step 2: Swap B $\rightarrow$ A on DEX2 with 0.3% fee

$$\text{Input: } 1,408.64 \text{ B}$$
$$\text{After fee: } 1,408.64 \times (1 - 0.003) = 1,404.42 \text{ B}$$
$$\text{Constant product: } (45581.60 + 1,404.42) \times (43331.99 - x) = 45581.60 \times 43331.99$$
$$\text{Solving for } x \approx 50,754.87 \text{ A received}$$

Profit $= 50,754.87 - 50,000 = 754.87$ token A

Second Run (Failed Arbitrage)



```
const amountIn = 50000;
const minProfitThreshold = 700;
```

0 Listen on all transactions Filter wit

```
DEX 1 Reserves:
{"0":1737.8782935081,"1":1515.9620580679}
DEX 2 Reserves:
{"0":17113.673989844,"1":16039.0471040101}
Spot Price DEX1 (B/A): 0.8723062275 | DEX2 (B/A): 0.9372065351

🔁 Trying A → B → A arbitrage...
❌ A → B → A arbitrage failed:
Transaction has been reverted by the EVM:
{
  "transactionHash": "0xd7e8651896a2f62067a2eb525e551f66e84e53a40fc4ee03c972d324bbe36da1",
  "transactionIndex": 0,
  "blockHash": "0x7e9abf24cb8497646a426d0efbbfcce570b7e681e7934c4895b7e545d77c7a8b",
  "blockNumber": 6178,
  "gasUsed": 56350,
  "cumulativeGasUsed": 56350,
  "status": false,
  "to": "0xFa7f2a95445325DDE17C3766552126285c246306",
  "events": {}
}
>
```

Initial Reserves:

$$\text{DEX 1: } \{"0" : 1737.8782935081, "1" : 1515.9620580679\}$$
$$\text{DEX 2: } \{"0" : 17113.6739989844, "1" : 16039.0471040101\}$$

Step 1: Swap A $\rightarrow$ B on DEX1 with 0.3% fee

$$\begin{aligned} \text{Input: } & 50,000 \text{ A} \\ \text{After fee: } & 50,000 \times (1 - 0.003) = 49,850 \text{ A} \\ \text{Constant product: } & (1737.87 + 49,850) \times (1515.96 - y) = 1737.87 \times 1515.96 \\ \text{Solving for } & y \approx 1,408.64 \text{ B received} \end{aligned}$$

Step 2: Swap B $\rightarrow$ A on DEX2 with 0.3% fee

$$\begin{aligned} \text{Input: } & 1,408.64 \text{ B} \\ \text{After fee: } & 1,408.64 \times (1 - 0.003) = 1,404.42 \text{ B} \\ \text{Constant product: } & (17113.67 + 1,404.42) \times (16039.05 - x) = 17113.67 \times 16039.05 \\ \text{Solving for } & x \approx 50,487.25 \text{ A received} \end{aligned}$$

Expected Profit $= 50,487.25 - 50,000 = 487.25$ token A

Theory Questions

1. Only the DEX contract itself should be allowed to mint/burn LPTokens. In our own DEX constructor, we are transferring the ownership of the LPToken to the DEX contract. The DEX contract mints LP tokens when users add liquidity and burns them when liquidity is removed. This design is appropriate because only the DEX contract should control the supply of LP tokens based on liquidity provisions. It prevents unauthorized minting/burning that could compromise the relationship between LP tokens and underlying reserves. Moreover, the DEX contract ensures proper ratios and products, thus avoiding malicious liquidity providers.

2. DEX plays the role of a playing field in the following ways:
   - It is an automated market making system. The implemented constant product formula provides deterministic pricing regardless of the trader's size or resources.
   - Anyone can trade without requiring approval or minimum capital requirements, unlike traditional exchanges.
   - Moreover, from our simulations we had observed, that larger trades observed higher slippage, which ensures that markets can't be exploited by larger groups.
   - The swap fees is the same across all sizes of trades.
   - The prices are deterministic, unlike traditional exchanges, where the order books are visible to only selective number of groups.

3. As we had studied in one of our lectures, here are the ways, the miner can take undue advantage of this information:
   - Front-running: Miners can see profitable trades in the mempool and insert their own transactions before them to profit from the expected price movement.
   - Sandwich Attacks: Miners can place transactions both before and after a large swap to profit from price slippage.
   - Transaction Reordering: Miners can reorder transactions to maximize their profit or minimize their loss.

   Here are a few possible steps, which we can take against this:
   - One possible way is to first create a commit transaction, which the user can place on the blockchain, and then reveal the actual parameters (quantities of tokens) later (so that the miner has access to less information).
   - We can create a way to submit transactions through channels that bypass the public mempools.
   - We can process multiple transactions in a batch with a single clearing price (so that the miner cannot take any advantage of the information).
   - We can skip transactions, that modify the slippage too much (through some pre-decided threshold).

4. Here are some of the ways, in which gas fees can influence economic viability of the entire DEX and arbitrage:
   - High gas fees create a minimum threshold for profitable trades. Small trades may become unprofitable due to fixed gas costs.
   - Arbitrageurs must factor gas costs into their profit calculations (so some of the arbitrages might not be profitable after this).
   - Functions with complex logic demand more gas fees, thus making them economically infeasible as compared to the other functions.
   - Gas costs reduce LP returns when adding/removing liquidity, potentially discouraging smaller liquidity providers.

5. Here are some of the ways, in which gas fees can lead to undue advantages to some transactions over others:
   - Traders with larger capital can afford higher gas fees, enabling them to execute profitable trades when smaller traders are priced out (because of fixed gas fees costs).
   - Larger entities can amortize gas costs across multiple operations or use batching techniques unavailable to average users.
   - Traders with lower latency connections to the network have advantages in competitive gas price bidding situations (because gas fees increases during network congestion).

6. Here are some of the ways to minimize slippage:
   - Breaking larger orders into smaller chunks, to reduce significant price movements (although this can mean increased gas fees).
   - Dividing trades into smaller chunks across various pools or different DEXs to minimize price impact.
   - We can increase the total liquidity at the pools, to reduce slippage.
   - We can time such large transactions, to happen later when the volatility is lower.
7. The required plots for two runs are given below:

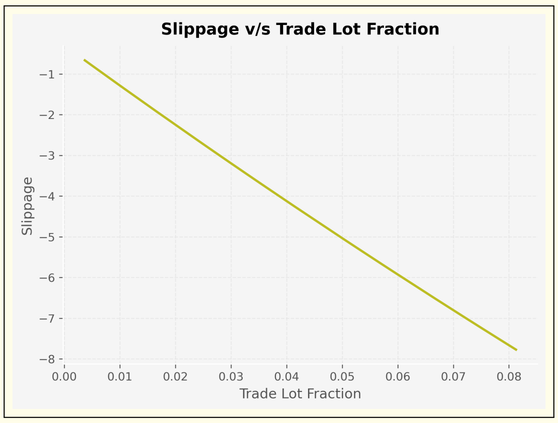


Figure 1: top_up_trader = 10 | initTokenA = 20 | initTokenB = 30

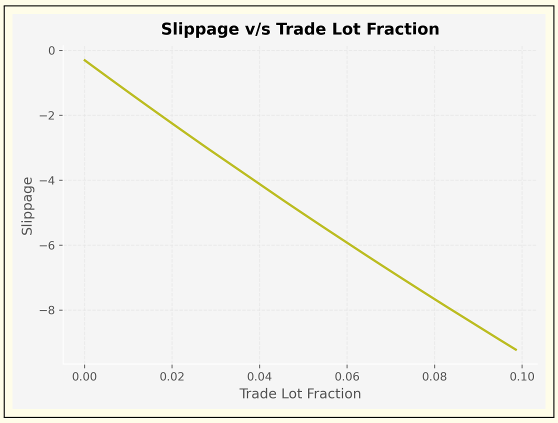


Figure 2: top_up_trader = 30 | initTokenA = 200 | initTokenB = 50

The plot remains consistent across all parameter configurations