CS765 Assignment 3

Nishant - 24M0743
Dipanshu Garg - 24M0755
Harsh Kumar - 24M0804

April 2025

Contents

1 Security and Validations

The smart contract `DEX.sol` has been made with many safety checks, validations, and clean coding rules to make sure it works well, gives correct results, and stays safe from common attacks. Below is a full list of the safety features and checks used:

1. **Input Validation:** Important functions like `addLiquidity`, `removeLiquidity`, and `swap` start with strict checks to make sure user inputs (like token amounts and LP tokens) are not zero and make sense. For example, swaps only happen if both token reserves are more than zero.

2. **Sanity Checks on Ratios:** When users add liquidity, the amount of tokens they give is checked to match the current reserve ratio, within a small allowed difference called `TOLERANCE`. This keeps prices steady and avoids cheating.

3. **Safe Arithmetic:** Since Solidity version 0.8.0 and above, the code has built-in checks to stop number overflows and underflows. So, we don't need extra libraries like SafeMath, and math bugs are avoided.

4. **Reentrancy Protection:** All functions that change state and move tokens (like `addLiquidity`, `removeLiquidity`, and both `swap` types) use OpenZeppelin's `ReentrancyGuard`. This stops repeated call attacks (reentrancy).

5. **ERC-20 Compliance Checks:** Token sending functions (transfer and transferFrom) are wrapped in `require` checks so that they succeed. This avoids silent token transfer failures and keeps the contract's state correct.

6. **Transparency through Events:** All big actions (adding liquidity, removing liquidity, and swapping tokens) trigger events with clear names. This helps with tracking, debugging, and showing what's happening outside the contract.

In short, the DEX contract is built for safe token use, defense against known smart contract bugs, and checks for fairness in every action. All of this makes the system strong, even when it's still being tested.

2 Testing

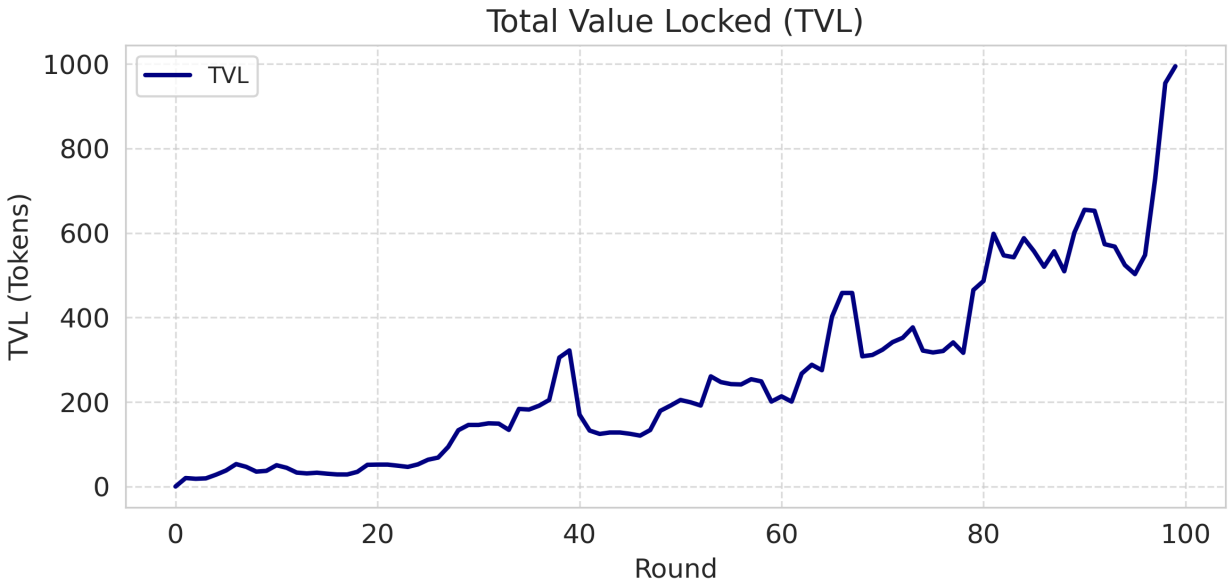

Figure 2.0.1: TVL

The TVL chart is rising, starting close to 0 and increasing to around 1000 tokens close to round 100. This shows that liquidity is gradually added to the pool over time, most likely due to successful LP deposits greater than withdrawals. The spikes are massive liquidity injections.

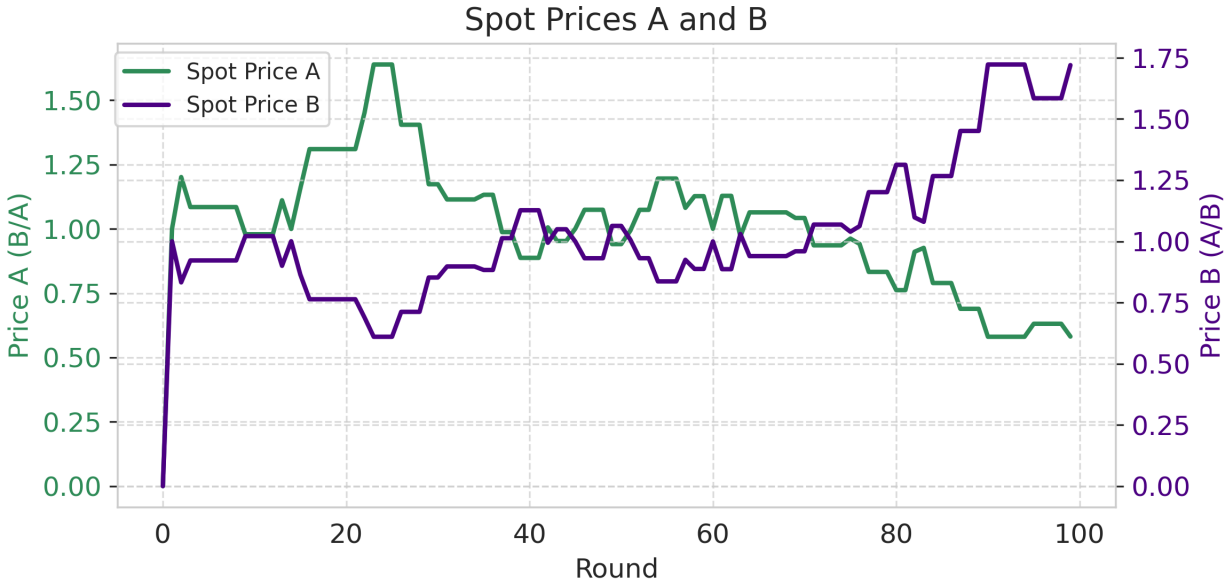

Figure 2.0.2: Spot Price

The "Spot Prices A and B" chart shows the price of Token A (as Token B) fluctuating between 0.25 and 1.75 and the price of Token B (as Token A) between 0.25 and 1.5. These inverse relationships are what the constant product formula ($x \cdot y = k$) would resemble, with reserve adjustments from swapping and liquidity activity dictating price.

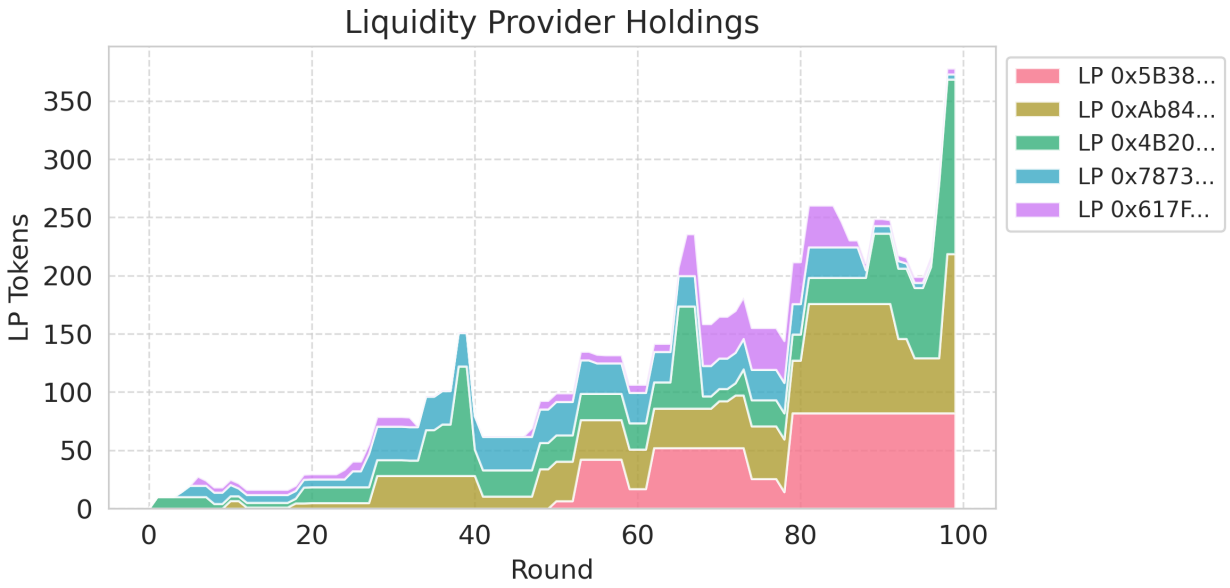

Figure 2.0.3: Holdings

The "Liquidity Provider Holdings" plot shows varying token holdings across LPs (e.g., LP 0x5B38... to LP 0x617F...), with some LPs (e.g., LP 0x4B20...) holding significantly more tokens (up to 350) by round 100. This suggests unequal participation, possibly due to random selection favoring certain LPs or successful deposits.

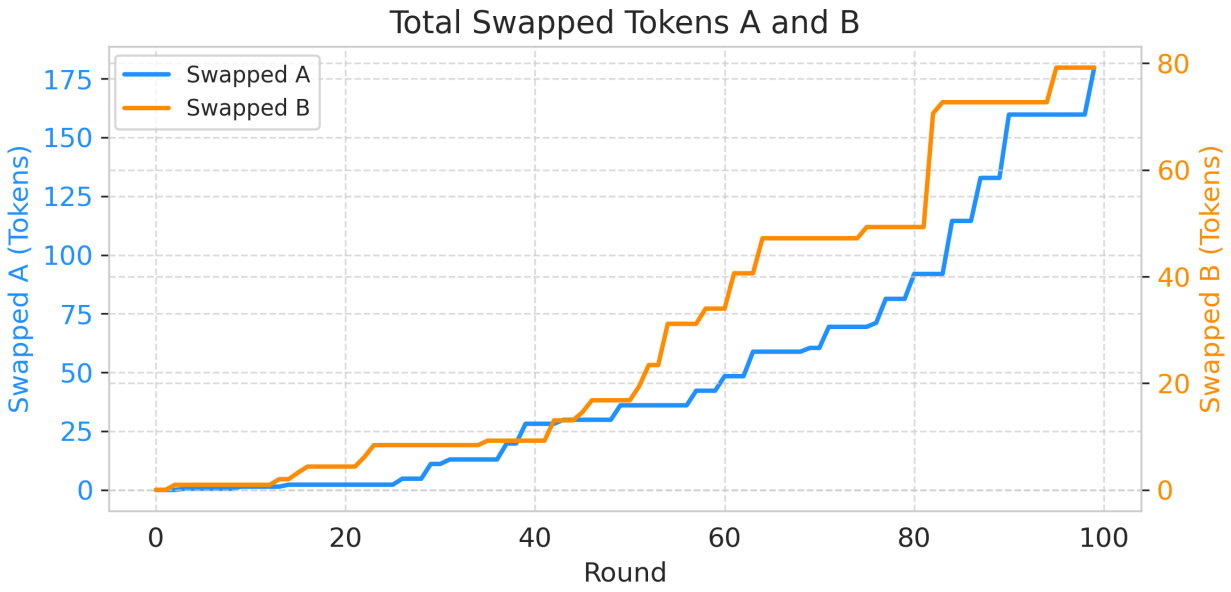

Figure 2.0.4: Swap Volume

The "Total Swapped Tokens A and B" graph indicates swapped tokens increasing consistently, with Token A at 175 tokens and Token B 80 tokens in round 100. The spread suggests more swapping or larger trades of Token A.

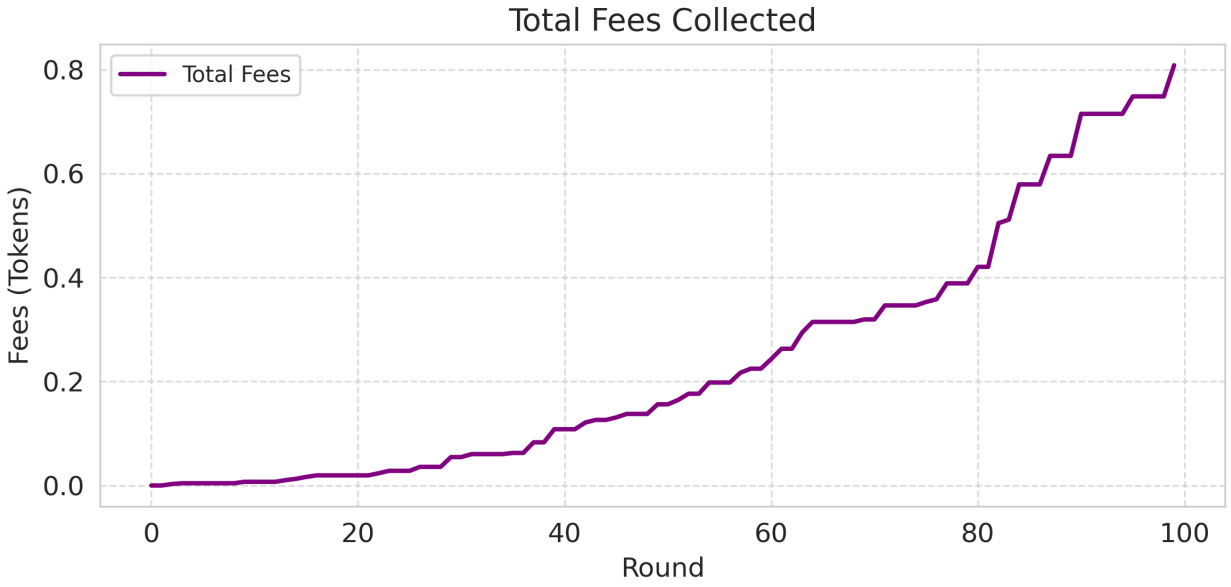

Figure 2.0.5: Fees

The "Total Fees Collected" graph rises steadily to 0.8 tokens, the 0.3% fee (997/1000) on swaps. The consistent rise reflects moderate activity in trading, as would be expected with the random transaction distribution.

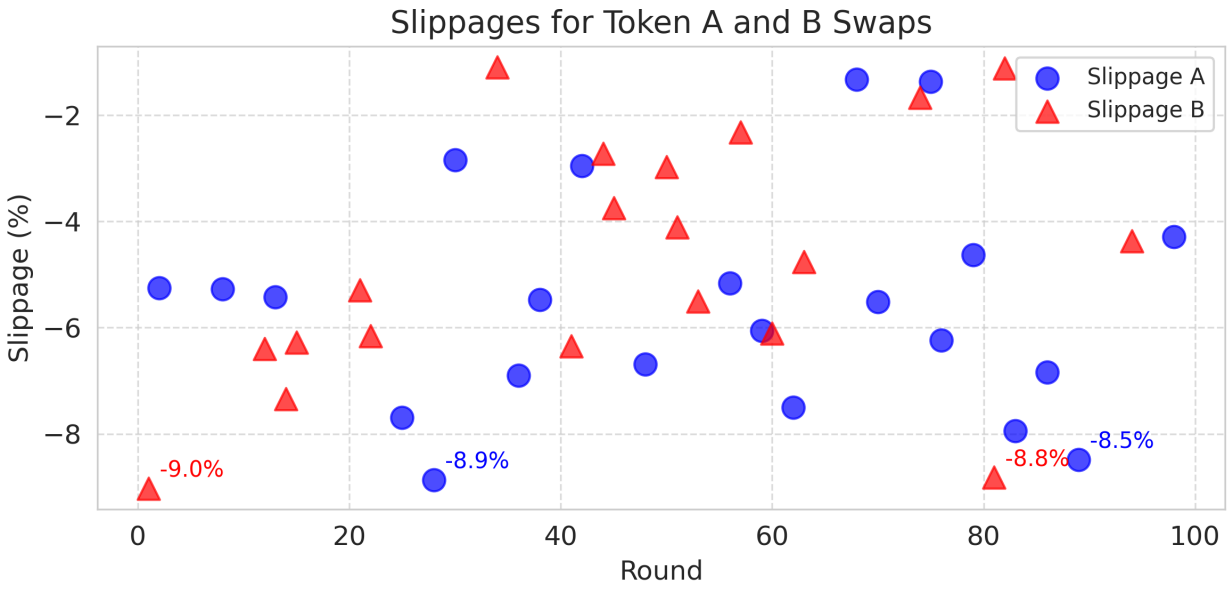

Figure 2.0.6: Slippage

The "Slippages for Token A and B Swaps" chart shows that slippage ranges between -9% and -2%, with some outliers (like -8.9% to -8.5%). Negative slippage means that traders get less than they expected. This is common in AMMs because big trades affect the price. The spread in slippage happens because we randomly changed reserve values and trade sizes, which affect how much the price moves.

Insights

Liquidity Imbalance and LP Participation: The "Liquidity Provider Holdings" chart shows that some LPs (like LP 0x4B20...) have much more tokens than others. This might be because LPs are picked randomly to deposit tokens, and some of them get lucky with more profitable trades — especially if they already have enough tokens. Also, people who deposit early get more LP tokens when the pool is still small.

Slippage and Trade Size Relationship: In the "Slippages for Token A and B Swaps" chart, the slippage starts at around -9% and later goes up to about -8.5%. This probably happens because there is very little liquidity in the beginning (so big trades change the price a lot), but as more tokens are added, the price becomes more stable. Big trades — like those that are 10% of the reserve — usually cause more slippage, especially when the two tokens aren't balanced.

Price Volatility and Reserve Dynamics: The "Spot Prices A and B" graph shows sudden jumps, like Token A going from 20 to 40. This is likely caused by very large swaps or people taking out a lot of liquidity at once. When the price of Token A goes down and Token B goes up, it matches the AMM rule (constant product formula), where changing reserves automatically shifts prices.

Fees Accumulation Lag: The "Total Fees Collected" chart shows that fees slowly go up over time. Since there's a 0.3% fee on each trade, the total depends on how big the trades are. But because trade sizes change randomly (and can be small), the DEX might not be collecting as many fees as it could. This could mean people are not trading much, or most trades are very small.

3 Theory Questions

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

Design Principle: On a secure decentralized exchange (DEX) platform, only certain addresses should be allowed to create (mint) or destroy (burn) LP (Liquidity Provider) tokens. This prevents people from creating fake ownership or removing others' shares. The only address that should have this power is the DEX contract itself. This ensures that LP tokens are only minted when someone adds liquidity and only burned when someone withdraws.

Implementation: In this deployment, the LPToken contract uses OpenZeppelin's `ERC20` and `Ownable` contracts. The functions `mint` and `burn` are protected using the `onlyOwner` modifier. This means only the contract owner can call them:

```
// LPToken.sol
function mint(address _account, uint256 _amount) external onlyOwner {
    _mint(_account, _amount);
}

function burn(address _account, uint256 _amount) external onlyOwner {
    _burn(_account, _amount);
}
```

Who is the Owner? In `DEX.sol`, the LPToken is created in the constructor:

```
// DEX.sol
lpToken = new LPToken();
```

When the LPToken contract is deployed like this, the `msg.sender` — which is the DEX contract — becomes the owner of the LPToken contract.

This setup ensures:

- The DEX contract can mint LP tokens when users add liquidity:

  ```
  lpToken.mint(msg.sender, lpAmount);
  ```

- The DEX contract can burn LP tokens when users remove liquidity:

  ```
  lpToken.burn(msg.sender, lpAmount);
  ```

Security Assurance: This design makes sure that no outside contract or random user can mint or burn LP tokens. All minting and burning actions happen only when liquidity is added or removed, so the system stays safe and fair.

Conclusion: Only the **DEX contract itself** (because it is the owner of `LPToken`) is allowed to mint and burn LP tokens. This is enforced by using the `onlyOwner` modifier in the code. The current implementation does this correctly.

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!)?

Centralized Finance (CeFi) – Not So Fair: In regular centralized exchanges, big trading firms get a lot of special benefits:

- They can see market data faster than others.
- They can place their computers really close to the exchange servers, so their trades go through quicker.
- They get access to special types of trades and cheaper fees because they trade in big volumes.

Because of these things, big traders can buy or sell before normal people even get a chance. This lets them take advantage of small price changes and make more profit.

Decentralized Finance (DeFi) – More Fair: On the other hand, Decentralized Exchanges (DEXs) are built in a way that's open and equal for everyone:

- **Everyone uses the same smart contract**, so the rules are the same for all.
- **The blockchain decides the order of trades**, not the exchange. Some people (like miners or validators) can still try to get ahead, but this is a known issue and people are working on fixing it.
- **No one gets early access to price info or trades**. Everyone uses the same pool and pricing method.
- **Anyone can join in**. You just need a wallet—no ID checks or special permission needed.

Why AMMs Are Fair: Most DEXs (including ours) use something called an Automated Market Maker (AMM) instead of a traditional order book. That means:

- Trade prices are based on math formulas and how much of each token is in the pool.
- Big traders can't cut in line or trade faster than others.
- Everyone sees the same price impact when they trade, no matter how big or small their trade is.

Things That Still Need Work: Even though DEXs are more fair than regular exchanges, they're not perfect:

- Gas fees (fees to use the blockchain) can be high, especially for small traders.
- Some smart people still try to sneak in front of others (called front-running), using tricks based on how the blockchain works.
- Big firms can still use fancy trading algorithms to try to get an edge.

Conclusion: Even with some problems, DEXs do a good job of **making things fair** for everyone. There's no special treatment—just one set of open, clear rules written in code that anyone can see and use. Whether you're trading $10 or 10,000, the system treats you the same.

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?

The Problem – MEV (Miner/Validator Extractable Value): When someone makes a trade on a DEX, the request first goes to a public waiting area called the **mempool**. This is like a queue of transactions waiting to be added to the blockchain. But since it's public, miners or validators can peek at the queue and try to make money unfairly by changing the order of transactions. This trick is called **MEV – Miner Extractable Value**.

Example – Front-running Attack: Let's say someone wants to trade a big amount of token A for token B. A miner can see this trade coming and do the following:

1. First, make their own trade to buy token B at the lower price.
2. Then, let the original big trade happen, which increases token B's price.
3. Finally, sell their token B at the now higher price to make profit.

This trick is called a **sandwich attack** — the honest user's trade gets sandwiched between two trades by the miner.

What Happens Because of This?

- Normal users get worse prices for their trades.
- People may start losing trust in DEXs and blockchains.
- Instead of helping the system, some miners just focus on making profits from these tricks.

Can We Stop This? It's really hard to stop MEV completely on public blockchains, but some clever ideas can reduce the problem:

- **Private Transaction Relays:** Tools like Flashbots let people send their trades directly to miners in private, so others can't see them in the mempool.
- **Commit-Reveal Method:** First, users send a hidden version of the trade (just a hash), and later they reveal the full details. This hides the trade for a while and prevents others from copying or jumping ahead.
- **Batch Auctions:** Instead of handling trades one by one, the DEX waits for a bunch of trades and processes them all together. This way, order doesn't matter much.
- **Encrypted Mempools:** Some researchers are working on hiding transactions in the mempool by encrypting them until they're added to the blockchain.
- **Smart DEX Designs:** Some newer DEXs (like CowSwap or SUAVE) are being designed in ways that reduce MEV or make it fairer for everyone.

In Our DEX Right Now: Our DEX does not yet have MEV protection. Here's how it works at the moment:

- Trades are sent openly through the public mempool.
- We use the `nonReentrant` rule to prevent reentrancy bugs, but this doesn't stop front-running.
- No commit-reveal or batch processing is used yet.

Conclusion: MEV is a big issue when it comes to making DEXs fair. Right now, our DEX is still open to these kinds of tricks. But in the future, we can improve it by adding features like private transactions, commit-reveal, or batch auctions to protect users and make trading safer for everyone.

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?

What Are Gas Fees? On Ethereum (and similar blockchains), every function call in a smart contract uses **gas**, which is a fee paid to the network to run your code. The more complex or storage-heavy your transaction, the more gas it consumes. Users pay gas using ETH (or the native token), and the final cost depends on:

- How much gas the transaction needs.
- The **gas price**, which users can set based on how fast they want it processed.

Gas fees aren't just a technical detail—they directly impact whether using a DEX makes financial sense.

How Gas Fees Affect Different Users:

- **Liquidity Providers (LPs):**
  - Have to pay gas to add or remove liquidity.
  - If gas is expensive, LPs may avoid adjusting their positions frequently—even if it would be profitable.
- **Traders:**
  - Each trade involves a gas fee *on top of* the swap fee.
  - If the gas fee is more than the expected gain (like saving $1 on a trade but paying $5 gas), the trade is a loss.
- **Arbitrageurs:**
  - They exploit price differences between DEXs.
  - Their profits must be greater than the gas fee to make it worthwhile.
  - This means small opportunities are ignored unless gas is low.

Impact on the DEX Ecosystem:

- **Fewer Small Users:** High gas prices make DEXs harder to use for retail users with small trades or liquidity amounts.
- **Network Congestion:** During busy times, gas prices spike—making things even worse for small users.
- **Arbitrage Gets Centralized:** Only fast bots or large players (with custom contracts or Flashbots access) can afford to do arbitrage when gas is high.

In Our DEX Smart Contract: Our DEX is written using standard Solidity practices. We avoid reentrancy bugs and try to keep things simple for gas efficiency. But:

- Every call—like `addLiquidity`, `removeLiquidity`, `swapAForB`, and `swapBForA`—costs gas.
- Things like transferring tokens, minting or burning LP tokens, and updating storage cost extra gas.

How Can We Make It Better?

- **Use Layer 2 Solutions:** Move the DEX to a rollup (e.g., Optimism, Arbitrum) or a sidechain like Polygon to cut gas costs drastically.
- **Batch Transactions:** Let users combine multiple operations (e.g., swap + liquidity add) into one to save on base gas costs.
- **Gas Subsidies or Rebates:** Give LPs some kind of gas refund or rewards to offset their costs and encourage participation.

Conclusion: Gas fees are often overlooked, but they can make or break the real-world usability of a DEX. They affect traders, LPs, and arbitrage bots—and if not managed properly, can reduce adoption, fairness, and decentralization. Our current implementation works on Ethereum Layer 1, but for serious deployment, exploring Layer 2 or gas optimizations is essential.

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

Yes, gas fees can introduce significant inequalities in who can participate effectively in DEX trading and arbitrage. These asymmetries often benefit well-capitalized and technically sophisticated users over average or retail participants.

Key Ways Gas Fees Lead to Undue Advantage:

- **Transaction Priority via Higher Gas Prices:**
  In Ethereum and similar systems, validators prioritize transactions offering higher gas prices. This means:
  - **Front-running:** A malicious actor can observe a pending transaction (e.g., a large swap) and submit their own with a higher gas price to be executed first.
  - **Back-running and Sandwich Attacks:** An attacker places transactions both before and after a target transaction to manipulate token prices and extract risk-free profit.
- **Retail vs. Institutional Participation:**
  Institutional traders and bots enjoy several advantages:
  - Use gas-optimized smart contracts,
  - Access private channels like Flashbots to avoid public mempool exposure,
  - Dynamically adjust gas prices in real time based on mempool data,
  - Operate with large capital reserves.

  In contrast, typical users:
  - Use public wallets or DEX frontends with static or delayed gas estimates,
  - Cannot adapt to real-time network conditions,
  - Are often priced out of trades during network spikes.
- **Network Congestion and Exclusion:**
  During periods of heavy demand (e.g., NFT mints, DeFi yield farming), gas prices can become prohibitively high. This leads to:
  - Exclusion of low-value users from trading,
  - Unfair access to new token launches or arbitrage trades.

- **Arbitrage Becomes Exclusive:**
  While arbitrage opportunities theoretically exist for all, they typically disappear within seconds. To exploit them consistently, users need:

  - High-speed bots,
  - The ability to submit transactions with high gas,
  - Preferential access to mempool data or Flashbots.

  Hence, arbitrage—intended as a fair market mechanism—becomes dominated by elite actors.

Conclusion: While gas fees serve important purposes—like preventing spam and compensating validators—they also introduce systemic inequities. High fees skew DEX participation in favor of those with better tools, faster access, and deeper pockets. This undermines the open and permissionless nature that decentralized finance aspires to achieve, and should be considered when designing DEXs for broader inclusivity.

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

Slippage is the difference between the expected price of a trade and the actual execution price. It becomes significant especially in low-liquidity pools or large trades. Below are some strategies to minimize slippage during a swap:

- **Use Pools with High Liquidity:** Slippage is inversely proportional to the pool's size. A pool with larger reserves can absorb larger trades with minimal price impact. Traders should prefer high-liquidity pools to reduce slippage.
- **Split Large Trades:** Breaking a large trade into smaller chunks and executing them over time can avoid sudden price impacts, though it might increase gas costs.
- **Set a Slippage Tolerance:** Most DEX interfaces allow users to specify a maximum acceptable slippage. If the price impact exceeds this tolerance, the transaction will revert. This does not reduce slippage directly but protects the trader from executing under worse prices.
- **Use Limit Orders (if supported):** Unlike market orders, limit orders execute only at the specified price or better. Some DEX aggregators or Layer 2 DEXs support such features to avoid slippage altogether.
- **Choose Optimal Time:** Executing swaps during periods of lower network congestion or lower trading volatility can help reduce price fluctuations and slippage.
- **DEX Aggregators:** Tools like 1inch or Matcha split a trade across multiple liquidity sources to get the best execution price with minimal slippage. They intelligently route trades through optimal paths.

- **Use Layer 2 or Sidechains:** On Layer 1 (e.g., Ethereum), congestion and gas competition can affect execution timing and slippage. Layer 2 solutions like Optimism, Arbitrum, or zkSync offer faster and more stable pricing environments with less slippage risk.
- **Flash Loans and MEV Protection:** Advanced users or bots might use flash loans or private mempool services (like Flashbots) to pre-calculate and execute arbitrage-efficient swaps with minimal slippage and front-running risk.

Conclusion: While slippage is inherent to AMM-based DEXs due to their pricing curve, informed strategies and tooling can significantly reduce its impact on trades.

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.

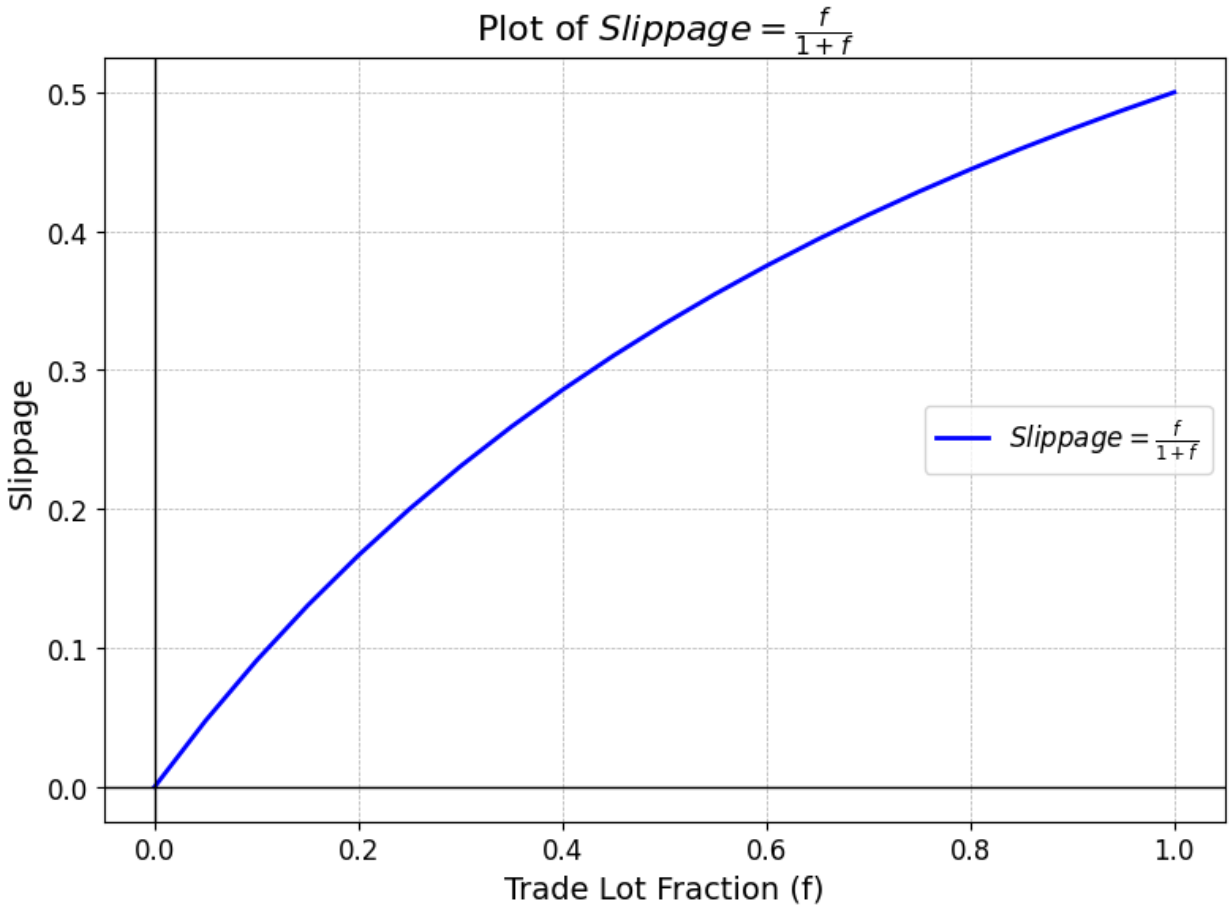

Figure 3.0.7: Spot Price

4 Implementation

4.1 DEX.sol

Overview

The `DEX.sol` contract is a simple decentralized exchange (DEX) that lets users trade between two ERC-20 tokens using an automated market maker (AMM) model. It is inspired by Uniswap V2 but designed to be easier to understand. The contract allows users to add/remove liquidity, swap tokens, and check token prices. Internal balances keep everything consistent and fair.

Token Setup and Security

The DEX works with two fixed ERC-20 tokens set at deployment. To prevent reentrancy attacks, it uses `ReentrancyGuard`. Liquidity providers (LPs) receive custom `LPToken` tokens, representing their share of the pool.

Adding Liquidity

LPs add both tokens to the pool to enable swaps. The first deposit sets the token ratio. Future deposits must keep roughly the same ratio, within a 1% error margin ($\epsilon = 10^{-2}$).

The LP tokens minted are calculated as:

$$\text{LP Amount} = \begin{cases} \sqrt{A \cdot B} & \text{if the pool is empty} \\ \min\left(\frac{A \cdot T}{R_A}, \frac{B \cdot T}{R_B}\right) & \text{otherwise} \end{cases}$$

Where T = total LP tokens, R_A, R_B = token reserves.

Removing Liquidity

LPs can remove their share by burning LP tokens. The amount of tokens they receive is based on their pool share:

$$A = \frac{R_A \cdot \text{LP}}{T}, \quad B = \frac{R_B \cdot \text{LP}}{T}$$

Token Swaps

Users can swap Token A for B and vice versa using `swapAForB` and `swapBForA`. The contract maintains the constant product rule:

$$x \cdot y = k$$

With a 0.3% fee, the output is:

$$\text{Output} = \frac{\text{Input} \cdot 997 \cdot R_{\text{out}}}{1000 \cdot R_{\text{in}} + \text{Input} \cdot 997}$$

This formula helps adjust prices after each swap and rewards LPs from trading activity.

Price Queries

Two functions, `getPriceAInB` and `getPriceBInA`, return the current exchange rate with fixed-point precision:

$$\text{Price}_{A\to B} = \frac{R_B \cdot 10^{18}}{R_A}, \quad \text{Price}_{B\to A} = \frac{R_A \cdot 10^{18}}{R_B}$$

Design Considerations

This contract is built for learning. It avoids complex features like front-running protection or slippage limits. Still, its simple design and clear events make it useful for testing and educational demos on how AMMs work.

4.2 Arbitrage.sol

Introduction

The `Arbitrage.sol` contract enables automated arbitrage between two decentralized exchanges (DEXs) that use constant product AMM models. It looks for price differences between DEX1 and DEX2 to identify and execute profitable token swap paths. This contract helps demonstrate how real-time arbitrage works in DeFi.

Architecture and Components

The contract works with two deployed `DEX` contracts that support two ERC-20 tokens (Token A and Token B). On deployment, it saves the DEX addresses and sets up the token interfaces.

Reserves and Spot Prices

Two structs store key information:

- `Reserves`: token balances in both DEXs (A1/B1 and A2/B2).
- `SpotPrices`: token price ratios, scaled by 10^{18} for precision.

Arbitrage Pathways

The contract checks four trade routes for arbitrage:

1. A $\to$ DEX1 $\to$ B $\to$ DEX2 $\to$ A
2. A $\to$ DEX2 $\to$ B $\to$ DEX1 $\to$ A
3. B $\to$ DEX1 $\to$ A $\to$ DEX2 $\to$ B
4. B $\to$ DEX2 $\to$ A $\to$ DEX1 $\to$ B

Each path is simulated to find the most profitable one.

Profit Estimation Strategy

The function `getProfit` uses a hill-climbing search to find the input amount that gives the highest profit without checking every value.

$$\text{Profit}(x) = \text{getAmountOut}(x) - x$$

Swap Simulation

`getAmountOut` simulates a two-step swap including the 0.3% fee:

$$a_1 = \left\lfloor \frac{997 \cdot \text{in} \cdot R_{\text{out1}}}{1000 \cdot R_{\text{in1}} + 997 \cdot \text{in}} \right\rfloor$$

$$\text{out} = \left\lfloor \frac{997 \cdot a_1 \cdot R_{\text{out2}}}{1000 \cdot R_{\text{in2}} + 997 \cdot a_1} \right\rfloor$$

This accurately reflects real trade outcomes.

Execution of Arbitrage

Once the best path is found, `executeTrade` runs the full swap. Token approvals and transfers follow standard ERC-20 practices. The user's token balance is updated at the end, and logic is kept clean and atomic.

Threshold Filter

Users can set a minimum profit threshold. If the profit is below this value, no trade is made, and `txnType` is set to 6 to show the opportunity was skipped.

Event Logging

After a successful trade, the contract emits an `ArbitrageDone` event with:

- **profit**: net profit from the arbitrage
- **txnType**: number (1 to 4) showing which path was used
- **amountIn**: tokens used for the trade

Conclusion

The `Arbitrage` contract offers a simple yet effective way to test or simulate DeFi arbitrage. It smartly picks the best path, handles tokens securely, and can be a strong base for research or automated trading bots.

4.3 LPToken.sol

Introduction

The `LPToken.sol` contract defines a simple ERC-20 token used to represent a user's share in a liquidity pool. When users add liquidity to a DEX, they receive these LP tokens as proof of ownership.

It uses OpenZeppelin's ERC-20 and Ownable contracts for security and easy access control. Only the contract owner can mint or burn tokens.

Token Specification

- **Name:** LP Token
- **Symbol:** LPT
- **Decimals:** 18

These tokens track how much of the pool a user owns and can be used to withdraw their share later.

Constructor Logic

On deployment:

- Sets the name and symbol using `ERC20("LP Token", "LPT")`.
- Assigns ownership to the deployer with `Ownable(msg.sender)`.

This ensures only the owner can mint or burn tokens unless ownership is transferred.

Minting Mechanism

The `mint` function:

```
mint(address _account, uint256 _amount)
```

- Only the owner can call it.
- Called when users add liquidity to the pool.
- Mints LP tokens based on how much they contributed.

Burning Mechanism

The `burn` function:

```
burn(address _account, uint256 _amount)
```

- Only callable by the owner.
- Used when users remove liquidity from the pool.
- Burns their LP tokens as they reclaim their share.

Ownership Control

Using OpenZeppelin's `Ownable`, only the owner can mint or burn. Ownership can be changed securely with:

```
transferOwnership(address newOwner)
```

Use Case Integration

This LP token can be:

- Shown in user wallets.
- Used as collateral or in staking.
- Transferred or locked in liquidity mining programs.

Conclusion

`LPToken` is a simple but key contract in DeFi. It safely tracks liquidity shares using standard ERC-20 features and secure ownership control. It's a building block for DEXs, AMMs, and yield farming setups.

4.4 Token.sol

Introduction

The `Token.sol` contract is a simple, flexible ERC-20 token built using OpenZeppelin's trusted library. It allows setting the name, symbol, and total supply during deployment, making it useful for DeFi apps like DEXs, staking, and governance.

Token Initialization

The contract uses OpenZeppelin's ERC20. During deployment, it takes:

- **name** — full token name.
- **symbol** — short symbol like "DAI" or "UNI".
- **initialSupply** — total tokens to mint at start.

It calls:

```
ERC20(name, symbol)
```

to set token metadata and:

```
_mint(msg.sender, initialSupply * 10^decimals())
```

to mint the full supply to the deployer, using 18 decimals.

Minting Logic

This contract doesn't allow minting or burning after deployment. All tokens are created once and given to the deployer. This keeps things simple and secure, ideal for fixed-supply tokens.

Use Case Scenarios

The token can be used for:

- Liquidity pools on DEXs.
- Staking or reward systems.
- Governance or as a wrapped asset.

Different tokens like "Token A" and "Token B" can be made by redeploying with different names and supplies.

Security and Standards

Using OpenZeppelin's ERC-20 ensures:

- Safe math and overflow protection.
- Compatibility with wallets and DeFi tools.
- Gas-efficient internal functions.

Conclusion

The `Token` contract is a simple and reliable way to create standard ERC-20 tokens with custom name, symbol, and supply. It's great for DEXs and DeFi systems, especially where token interactions like arbitrage or staking are needed.

4.5 simulator_DEX.js

Introduction

The `simulator.js` script simulates how a decentralized exchange (DEX) works. It models real actions like adding/removing liquidity, swapping tokens, generating fees, and tracking market behavior. It helps measure slippage, spot prices, and Total Value Locked (TVL) over multiple trades.

Architecture and Components

The simulation uses smart contracts and ERC-20 tokens. It involves:

- `Token A` and `Token B`: Swappable ERC-20 tokens.
- `DEX`: Custom contract for swaps and managing liquidity.
- `LPToken`: Issued to liquidity providers to show their pool share.
- `Accounts`: Simulated LPs and traders.

Key Variables and Constants

Important constants in the simulation:

- `Num_LP`: Number of liquidity providers.
- `Num_Traders`: Number of traders.
- `N`: Total simulation rounds.
- `SCALE`: 10^{18} used for accurate calculations.

Simulation Process

The simulation runs in stages:

Token Distribution

Tokens are created and equally distributed among all LPs and traders. Each gets approval to interact with the DEX.

Transaction Execution

In each round, a random action happens:

1. **ADD Liquidity:** LP adds tokens to the pool (if they have enough).
2. **REMOVE Liquidity:** LP removes a random portion of their LP tokens.
3. **SWAP Token:** Trader swaps Token A $\leftrightarrow$ Token B. Swap size depends on user balance and 10% of DEX reserves. Fees and slippage are applied.

Spot Price and Slippage

Spot prices are calculated from current reserves:

$$\text{Price A} = \frac{\text{Reserve B} \times \text{SCALE}}{\text{Reserve A}}, \quad \text{Price B} = \frac{\text{Reserve A} \times \text{SCALE}}{\text{Reserve B}}$$

Slippage is calculated based on how much the price changes after a trade.

Fee Calculation

Each swap includes a 0.3% fee:

$$\text{Fee} = \text{Swap Amount} \times 0.003$$

Fees are added to total liquidity pool earnings.

Performance Metrics

Metrics tracked in each round:

- **TVL:** Combined value of Token A and B in the pool.
- **Total Fees:** Accumulated swap fees.
- **Swapped A/B:** Total tokens swapped.
- **Slippage A/B:** How much token prices changed.
- **Spot Prices:** Updated prices of each token.
- **Txn Types:** Type of action in the round.
- **Holdings:** LP token balance per account.

Event Logging

Each round logs:

- **Transaction Type and Status:** What happened and if it succeeded.
- **Token Amounts:** How much was added, removed, or swapped.
- **Spot Prices and Slippage:** Price updates and deviations.

Conclusion

The `simulator.js` script provides a realistic setup to test how a DEX reacts to liquidity changes and trading. It helps study fee impacts, slippage, and token price trends, making it useful for optimizing DEX performance and liquidity strategies.

4.6 simulate_arbitrage.js

Introduction

The `simulate_arbitrage.js` script tests an arbitrage strategy using two DEXes and an arbitrage smart contract. It checks three scenarios: (1) no arbitrage profit, (2) profit below a set threshold, and (3) profit above the threshold. The goal is to ensure the contract correctly detects and performs only profitable trades.

Setup and Initialization

Scaling Factor:

$$\texttt{SCALE} = 10^{18}$$

Used for handling 18-decimal precision typical of ERC-20 tokens.

Contracts Deployed:

- **TokenA** and **TokenB:** ERC-20 tokens with 10,000 supply each.
- **DEX1** and **DEX2:** Two DEX contracts initialized with the tokens.
- **Arbitrage:** Contract that executes trades between DEX1 and DEX2.

Approvals and Liquidity Setup

The deployer account:

1. Approves tokens for both DEXes and the arbitrage contract.
2. Adds 2000 TokenA and 2000 TokenB to each DEX as liquidity.

Arbitrage Threshold

$$\texttt{threshold} = 100 \times 10^{17}$$

Only profits above this value trigger arbitrage execution. Smaller profits are ignored.

Test Cases

Three scenarios are tested:

Case 1: No Profit

- DEX prices are equal.
- Arbitrage function is called.
- No profit is detected, so no trade is made.

Case 2: Profit Below Threshold

- 20 TokenA is swapped on DEX1 to create a small price difference.
- Arbitrage is run but ignored due to low profit.

Case 3: Profit Above Threshold

- 200 TokenA is swapped on DEX1 to create a big price gap.
- Arbitrage is executed successfully since profit exceeds the threshold.

Logging and Output

Each arbitrage result is logged from the `ArbitrageDone` event:

- **AmountIn:** Tokens used in the trade.
- **Profit:** Earned profit.
- **TxnType:** Trade direction (A→B→A or B→A→B).

Results are printed in Ether units for readability.

Conclusion

This simulation confirms:

- No trade occurs if there's no profit.
- Trades with small profits are skipped.
- Profitable trades above the threshold are executed.

It provides a solid test setup to validate arbitrage strategies with controlled price changes and liquidity.