Capstone Project

CSCI 1951L Herlihy

Robert Wang

Introduction:

For this project, I worked with Professor Herlihy and extended the codebase of an existing Uniswap AMM implementation.[1] To begin, let's discuss a little bit about what an AMM (automated market maker) is, how they came to be, and why they were developed. Traditionally, the way a centralized market works is that there are two sides, buyers and sellers. On the buyer side, for example, we would have one trader, trader A, who wants to sell x units of some good or stock. On the other side, we would have trader B who wants to BUY the good that the first trader is selling. In a traditional market, there exists some centralized authority that works as a middleman to facilitate trades between buyers and sellers. The central authority works to match bids with asks as fast as possible to maintain liquidity for the assets.

With this in mind, let's turn to Automated Market Makers. Essentially, the goal of an AMM is to remove the centralized authority. Code and automatically executed smart contracts act to facilitate interactions between buyers and sellers and removes the need for a middleman. The capstone that I worked on involved a very basic Uniswap AMM and targeted an Ethereum-Token exchange. Essentially, it was an AMM with two liquidity pools: One for ether and one for whatever token we wanted to use. In the following photo snippets, I've attached the functions that I coded up for calculating specific statistics (Divergence Loss, Load, Linear Slippage, and Angular Slippage).

[1] https://github.com/PhABC/uniswap-solidity

The Functions:

1. Divergence Loss: Eth → Token

```
/**
 * @notice Public divergence loss function for ETH to Token trades with an exact input.
 * @param eth sold Amount of ETH sold.
 * @return Divergence loss of proposed trade.
 */
function getEthToTokenDivergenceLoss(uint256 eth_sold)
    public
    view
    returns (uint256)
{
    uint256 eth_lp = address(this).balance;

    uint256 numerator = eth_sold.mul(eth_sold);

    uint256 t1 = eth_lp.mul(2).mul(eth_sold.mul(eth_sold));
    uint256 t2 = eth_lp.mul(eth_lp.mul(eth_lp));
    uint256 t3 = eth_sold.mul(eth_sold.mul(eth_lp));
    uint256 t4 = eth_lp;
    uint256 denominator = t1.add(t2).add(t3).add(t4);

    return numerator / denominator;
}
```

2. Divergence Loss: Token → Eth

```
/**
 * @notice Public divergence loss function for Token to eth trades with an exact input.
 * @param token_sold sold Amount of the token.
 * @return Divergence loss of proposed trade.
 */
function getTokentoEthDivergenceLoss(uint256 token_sold)
    public
    view
    returns (uint256)
{
    uint256 token_lp = token.balanceOf(address(this));

    uint256 numerator = token_sold.mul(token_sold);

    uint256 t1 = token_lp.mul(2).mul(token_sold.mul(token_sold));
    uint256 t2 = token_lp.mul(token_lp.mul(token_lp));
    uint256 t3 = token_sold.mul(token_sold.mul(token_lp));
    uint256 t4 = token_lp;
    uint256 denominator = t1.add(t2).add(t3).add(t4);

    return numerator / denominator;
}
```

3. Linear Slippage: Eth → Token

```
/**
 * @notice Public linear slippage function for ETH to Token trades with an exact input.
 * @param eth_sold Amount of ETH sold.
 * @return LinearSlippage loss of proposed trade.
 */
function getEthToTokenLinearSlippage(uint256 eth_sold)
    public
    view
    returns (uint256)
{
    // Get the current amount of ETH in the liquidity pool
    uint256 eth_lp = address(this).balance;

    // Calulate the numerator
    uint256 delta_sq = eth_sold.mul(eth_sold);
    uint256 numerator = delta_sq.mul(eth_sold.add(eth_lp));

    // Calulate the denominator
    uint256 eth_lp_sq = eth_lp.mul(eth_lp);
    uint256 denom = eth_lp_sq.mul(
        delta_sq.add(eth_lp_sq.add(eth_sold.mul(2).mul(eth_lp)).add(1))
    );

    return -numerator / denom;
}
```

4. Linear Slippage: Token → Eth

```
/**
 * @notice Public linear slippage function for Token to eth trades with an exact input.
 * @param token_sold sold Amount of the token.
 * @return LinearSlippage loss of proposed trade.
 */
function getTokentoEthLinearSlippage(uint256 token_sold)
    public
    view
    returns (uint256)
{
    // Get the current amount of ETH in the liquidity pool
    uint256 token_lp = token.balanceOf(address(this));

    // Calulate the numerator
    uint256 delta_sq = token_sold.mul(token_sold);
    uint256 numerator = delta_sq.mul(token_sold.add(token_lp));

    // Calulate the denominator
    uint256 token_lp_2 = token_lp.mul(token_lp);
    uint256 denom = token_lp_2.mul(
        delta_sq.add(token_lp_2.add(token_sold.mul(2).mul(token_lp)).add(1))
    );

    // Return
    return -(numerator / denom);
}
```

5. Load: Eth → Token

```
/**
 * @notice Public load function for ETH to Token trades with an exact input.
 * @param eth sold Amount of ETH sold.
 * @return Load loss of proposed trade.
 */
function getEthToTokenLoad(uint256 eth_sold) public view returns (uint256) {
    uint256 div_loss = getEthToTokenDivergenceLoss(eth_sold);
    uint256 lin_slip = getEthToTokenLinearSlippage(eth_sold);

    return div_loss.mul(lin_slip);
}
```

6. Load: Token → Eth

```
/**
 * @notice Public load function for Token to eth trades with an exact input.
 * @param token_sold sold Amount of the token.
 * @return Load loss of proposed trade.
 */
function getTokentoEthLoad(uint256 token_sold)
    public
    view
    returns (uint256)
{
    uint256 div_loss = getTokentoEthDivergenceLoss(token_sold);
    uint256 lin_slip = getTokentoEthLinearSlippage(token_sold);

    return div_loss.mul(lin_slip);
}
```

7. Angular Slippage: Eth → Token

```
/**
 * @notice Public angular slippage function for ETH to Token trades with an exact input.
 * @param eth_sold sold Amount of ETH sold.
 * @return AngularSlippage loss of proposed trade.
 */
function getEthToTokenAngularSlippage(uint256 eth_sold)
    public
    view
    returns (uint256)
{
    // Variables
    uint256 eth_lp = address(this).balance;
    uint256 eth_lp_0 = address(this).balance.sub(eth_sold);

    uint256 token_lp = token.balanceOf(address(this));
    // Get k and compute new token balance
    uint256 k = eth_lp.mul(token_lp);
    uint256 token_lp_0 = k / eth_lp_0; // TODO: is this correct

    // Numerator: v - v'
    // v_x = x / (x + y)
    uint256 v = eth_lp / token_lp.add(eth_lp);
    uint256 v_0 = eth_lp_0 / token_lp_0.add(eth_lp_0);
    uint256 numerator = v.sub(v_0);

    // Denominator: v * v' + (1 - v) * (1 - v')
    uint256 denominator = v.mul(v_0).add(1 - v).add(1 - v_0);

    return atanSmall(numerator / denominator);
}
```

8. Angular Slippage: Token → Eth

```
/**
 * @notice Public angular slippage function for Token to eth trades with an exact input.
 * @param token_sold sold Amount of the token.
 * @return AngularSlippage loss of proposed trade.
 */
function getTokentoEthAngularSlippage(uint256 token_sold)
    public
    view
    returns (uint256)
{
    // Variables
    uint256 token_lp = token.balanceOf(address(this));
    uint256 token_lp_0 = token_lp.sub(token_sold);
    uint256 eth_lp = address(this).balance;
    // Get k and compute new eth balance
    uint256 k = eth_lp.mul(token_lp);
    uint256 eth_lp_0 = k / token_lp_0; // TODO: is this correct

    // Numerator: v - v'
    uint256 v = eth_lp / token_lp.add(eth_lp);
    uint256 v_0 = eth_lp_0 / token_lp_0.add(eth_lp_0);
    uint256 numerator = v.sub(v_0);

    // Denominator: v * v' + (1 - v) * (1 - v')
    uint256 denominator = v.mul(v_0).add(1 - v).add(1 - v_0);

    return atanSmall(numerator / denominator);
}
```

A big thank you to Professor Herlihy for setting up the capstone project and guiding me through. Reading through the paper and understanding the different metrics that we wanted to calculate was no easy task, and Professor Herlihy was always available to meet with me and help me dissect the paper. I think the hardest parts of this was 1) understanding the existing code base for the AMM we were extending and 2) getting familiar with what Divergence Loss, etc really meant.