Brandon Sohn
May 10th, 2022

# AMM Project

Overall, learning about AMMs and primarily looking into Uniswap's docs and whitepapers has been a great learning experience. It was also extremely interesting to see the smart contracts act as a liquidity pool. AMMs in the defi space seem like an extremely scalable project due to the permissionless nature of smart contracts. If there was a customary, universal contract that could allow people to deploy liquidity pools extremely easily, I feel like this would be a cool niche and defi that web3 developers can take advantage of. I thought that looking into deeper calculations embedded in AMMs really helped me learn and understand how prices work in this exchange mechanism. The following screenshots are my interpretations of divergence loss, linear slippage, angular slippage and load in the context of AMMs.

**Divergence Loss**: This value helps create equilibrium in the liquidity pools since ratio balance is quintessential. Reading through Uniswap's smart contracts, it shows that token pairs must contain equal total values and this is the cost to liquidity providers.

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

  // precalculate some values
  uint256 eth_supply_sq = eth_supply.mul(eth_supply);
  uint256 eth_sold_sq = eth_sold.mul(eth_sold);
  uint256 eth_sold_cubed = eth_sol_sq.mul(eth_sold);

  uint256 numerator = eth_supply_sq;
  uint256 denominator = eth_supply.mul(2).mul(eth_sold_sq) + eth_sold_cubed + eth_supply_sq.mul(eth_sold) + eth_sold;

  // TODO: we have to subtract the input eth sold and output token price?
  return numerator.div(denominator);

}
```

```
/**
 * @notice Public divergence loss function for Token to ETH trades with an exact input.
 * @param amount Amount of the token sold.
 * @return Divergence loss of proposed trade.
 */
function getTokenToEthDivergenceLoss(uint256 tokens_sold) public view returns (uint256) {
  // get supply of tokens in the pool
  uint256 token_supply = token.balanceOf(address(this));

  // precalculate some values
  uint256 token_supply_sq = token_supply.mul(token_supply);
  uint256 tokens_sold_sq = tokens_sold.mul(tokens_sold);
  uint256 tokens_sold_cubed = tokens_sold_sq.mul(tokens_sold);

  uint256 numerator = token_supply_sq;

  // TODO: we have to subtract the input tokens sold and output eth price?
  uint256 denominator = token_supply.mul(2).mul(tokens_sold_sq) + tokens_sold_cubed + token_supply_sq.mul(tokens_sold) + tokens_sold;

  return numerator.div(denominator);
}
```

**Slippage**: The reason that slippage is important to consider in liquidity pools, every trade makes an impact on the expected price after the transaction. This includes linear slippage, which calculates the direct impact to the buyer's price difference. The second is angular slippage, which characterizes how that buyer affects prices for later buyers.

```
/**
 * @notice Public linear slippage function for ETH to Token trades with an exact input.
 * @param tokens_sold Amount of the token sold.
 * @return Linear Slippage of proposed trade.
 */
function getTokenToEthLinearSlippage(uint256 tokens_sold) public view returns (uint256) {
  // get the supply of tokens in the pool
  uint256 token_supply = token.balanceOf(address(this));

  //precalculate some values
  uint256 token_supply_sq = token_supply.mul(token_supply);
  uint256 tokens_sold_sq = tokens_sold.mul(tokens_sold);

  uint256 numerator = token_supply_sq.mul(token_supply.add(tokens_sold));
  uint256 denominator = tokens_sold_sq.mul(token_supply_sq.add(tokens_sold_sq).add(2.mul(tokens_sold).mul(token_supply)).add(1));

  return numerator.div(denominator).mul(-1);
}
```

```
/**
 * @notice Public linear slippage function for ETH to Token trades with an exact input.
 * @param eth_sold Amount of ETH sold.
 * @return Linear Slippage of proposed trade.
 */
function getEthToTokenLinearSlippage(uint256 eth_sold) public view returns (uint256) {
  // get the supply of eth in the pool
  uint256 eth_supply = address(this).balance;

  // precalculate some values
  uint256 eth_supply_sq = eth_supply.mul(eth_supply);
  uint256 eth_sold_sq = eth_sold.mul(eth_sold);

  uint256 numerator = eth_supply_sq.mul(eth_supply.add(eth_sold));
  uint256 denominator = eth_sold_sq.mul(eth_supply_sq.add(eth_sold_sq).add(2.mul(eth_sold).mul(eth_supply)).add(1));
  return numerator.div(denominator).mul(-1);
}
```

```
/**
 * @notice Public angular slippage function for ETH to Token trades with an exact input.
 * @param eth_sold Amount of ETH sold.
 * @return Angular Slippage of proposed trade.
 */
function getEthToTokenAngularSlippage(uint256 eth_sold) public view returns (uint256) {
  // get the supply of eth in the pool
  uint256 eth_supply_0 = address(this).balance;
  // get the supply of eth in the pool after tx
  uint256 eth_supply_1 = eth_supply_0 - eth_sold;

  // get the supply of tokens in the pool
  token_supply_0 = token.balanceOf(address(this));


  // compute k
  uint256 k = eth_supply_0.mul(token_supply_0);

  // get the supply of tokens in the pool after tx
  token_supply_1 = k / eth_supply_1;

  // calculate valuation before tx
  uint256 v_0 = eth_supply_0.div(token_supply_0.add(eth_supply_0));

  // calculate valuation after tx
  uint256 v_1 = eth_supply_0.div(token_supply_0.add(eth_supply_0));

  uint256 numerator = v_0.sub(v_1);

  uint256 denominator = v_0.mul(v_1).add(1.sub(v_0)).add(1.sub(v_1));

  return atanSmall(numerator.div(denominator));
}
```

```
/**
 * @notice Public angular slippage function for ETH to Token trades with an exact input.
 * @param tokens_sold Amount of the token sold.
 * @return Angular Slippage of proposed trade.
 */
function getTokenToEthAngularSlippage(uint256 tokens_sold) public view returns (uint256) {
  // get the supply of tokens in the pool
  uint256 token_supply_0 = token.balanceOf(address(this));
  // get the supply of tokens in the pool after the tx
  uint256 token_supply_1 = token_supply_0 - tokens_sold;

  // get the supply of eth in the pool
  uint256 eth_supply_0 = address(this).balance;

  // compute k
  uint256 k = eth_supply_0.mul(token_supply_0);

  // get the supply of eth in the pool after tx
  eth_supply_1 = k / token_supply_1;

  // calculate valuation before tx
  uint256 v_0 = token_supply_0.div(eth_supply_0.add(token_supply_0));

  // calculate valuation after tx
  uint256 v_1 = token_supply_0.div(eth_supply_0.add(token_supply_0));

  uint256 numerator = v_0.sub(v_1);

  uint256 denominator = v_0.mul(v_1).add(1.sub(v_0)).add(1.sub(v_1));

  return atanSmall(numerator.div(denominator));
}
```

**Load**: This value is simply just a measure of balancing provider and traders costs.

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

```
/**
 * @notice Public load function for ETH to Token trades with an exact input.
 * @param tokens_sold Amount of the token sold.
 * @return Load of proposed trade.
 */
function getTokenToEthLoad(uint256 tokens_sold) public view returns (uint256) {
   return getEthToTokenDivergenceLoss(tokens_sold) * getEthToTokenLinearSlippage(tokens_sold);
}
```

We propose the following measure to balance provider-facing and trader-facing costs.