Philip Kendler
CS1951L Capstone Project
Professor Herlihy

For this Capstone Project I researched and coded functions that analyzed the different costs that both market makers and customers incur in a transaction. Specifically, I calculated the linear slippage, angular slippage, divergence cost, and load associated with a transaction with a specified input or output corresponding to the quantity of either Eth or tokens wished to be sold or bought. These functions calculate the aforementioned costs for an automated market maker providing conversions under xy = k for some constant k, where x and y are the reserves of Eth and tokens. Below I have included the code for each cost for both inputs and outputs as well as explanations for each one.

Linear Slippage:

```
/**
* @notice Public linear slippage function for ETH to Token trades with an exact input.
* @param input_amount Amount of ETH or Tokens being sold.
* @param input_reserve Amount of ETH or Tokens (input type) in exchange reserves.
* @param output_reserve Amount of ETH or Tokens (output type) in exchange reserves.
* @return linear slippage of proposed trade.
*/

function getInputLinearSlippage(uint256 input_amount, uint256 input_reserve, uint256 output_reserve) public view returns (uint256) {
  require(input_reserve > 0 && output_reserve > 0, "INVALID_VALUE");
  uint256 spot_price = input_reserve.mul(997) / output_reserve.mul(1000);
  uint256 input_price = getInputPrice(input_amount, input_reserve, output_reserve);
  uint256 linear_slippage = ((spot_price.mul(input_amount)).sub(input_price));
  return linear_slippage;
}


/**
* @notice Public linear slippage function for ETH to Token trades with an exact output.
* @param output_amount Amount of ETH or Tokens being bought.
* @param input_reserve Amount of ETH or Tokens (input type) in exchange reserves.
* @param output_reserve Amount of ETH or Tokens (output type) in exchange reserves.
* @return linear slippage of proposed trade.
*/

function getOutputLinearSlippage(uint256 output_amount, uint256 input_reserve, uint256 output_reserve) public view returns (uint256) {
  require(input_reserve > 0 && output_reserve > 0);
  uint256 spot_price = input_reserve.mul(1000) / (output_reserve.mul(997));
  uint256 output_price = getOutputPrice(output_amount, input_reserve, output_reserve);
  uint256 linear_slippage = (output_price.sub(spot_price.mul(output_amount)));
  return linear_slippage;
}
```

Linear slippage measures the loss incurred by the buyer or seller transacting with the market maker. The functions above find the difference between the spot price – the current market price as dictated by the ratio of the reserves of Eth and tokens – and the actual price of the transaction. The customer cannot trade at the spot price because each marginal size of their trade moves the spot price along the curve and as the customer trades a larger size, the price they are getting gets worse and worse due to the convex nature of the curve.

Angular Slippage:

```
/**
* @notice Public angular slippage function for ETH to Token trades with an exact input.
* @param input_amount Amount of ETH or Tokens being sold.
* @param input_reserve Amount of ETH or Tokens (input type) in exchange reserves.
* @param output_reserve Amount of ETH or Tokens (output type) in exchange reserves.
* @return angular slippage of proposed trade.
*/

function getInputAngularSlippage(uint256 input_amount, uint256 input_reserve, uint256 output_reserve) public view returns (int128) {
  require(input_reserve > 0 && output_reserve > 0);
  uint256 spot_price = input_reserve.mul(997) / (output_reserve.mul(1000));
  uint256 input_price = getInputPrice(input_amount, input_reserve, output_reserve);
  output_reserve = output_reserve.sub(input_price);
  input_reserve = input_reserve.add(input_amount);
  uint256 new_spot_price =  input_reserve.mul(997) / (output_reserve.mul(1000));
  uint256 angle = (spot_price.sub(new_spot_price))/(spot_price.mul(new_spot_price));
  int128 angle_1 = int128(angle);
  int128 angular_slippage = atanSmall(angle_1);
  return angular_slippage;
}
```

```
 /**
 * @notice Public angular slippage function for ETH to Token trades with an exact output.
 * @param output_amount Amount of ETH or Tokens being sold.
 * @param input_reserve Amount of ETH or Tokens (input type) in exchange reserves.
 * @param output_reserve Amount of ETH or Tokens (output type) in exchange reserves.
 * @return angular slippage of proposed trade.
 */

 function getOutputAngularSlippage(uint256 output_amount, uint256 input_reserve, uint256 output_reserve) public view returns (int128) {
   require(input_reserve > 0 && output_reserve > 0);
   uint256 spot_price = input_reserve.mul(1000) / (output_reserve.mul(997));
   uint256 output_price = getOutputPrice(output_amount, input_reserve, output_reserve);
   output_reserve = output_reserve.sub(output_amount);
   input_reserve = input_reserve.add(output_price);
   uint256 new_spot_price =  input_reserve.mul(1000) / (output_reserve.mul(997));
   uint256 angle = (spot_price.sub(new_spot_price))/(spot_price.mul(new_spot_price));
   int128 angle_1 = int128(angle);
   int128 angular_slippage = atanSmall(angle_1);
   return angular_slippage;
 }
```

Angular Slippage is the price incurred by a future customer transacting with the market maker in the same direction. Angular slippage is calculated by finding the angular distance between the two slopes corresponding to the spot price before and after a transaction is made. After a transaction occurs, the equilibrium shifts along the curve meaning that a future customer wishing to trade in the same direction will have to do so at a worse price.

Divergence Loss:

```
/**
* @notice Public divergence loss function for ETH to Token trades with an exact input.
* @param input_amount Amount of ETH or Tokens being sold.
* @param input_reserve Amount of ETH or Tokens (input type) in exchange reserves.
* @param output_reserve Amount of ETH or Tokens (output type) in exchange reserves.
* @return divergence loss of proposed trade.
*/
  function getInputDivergenceLoss(uint256 input_amount, uint256 input_reserve, uint256 output_reserve) public view returns (uint256) {
    require(input_reserve > 0 && output_reserve > 0);
    uint256 input_price = getInputPrice(input_amount, input_reserve, output_reserve);
    uint256 new_output_reserve = output_reserve.sub(input_price);
    uint256 new_input_reserve = input_reserve.add(input_amount);
    uint256 new_market_price_in = new_output_reserve.mul(997) / (new_input_reserve.mul(1000));
    uint256 new_market_price_out = new_input_reserve.mul(997) / (new_output_reserve.mul(1000));
    uint256 prev_wealth = output_reserve.mul(new_market_price_out).add(input_reserve.mul(new_market_price_in));
    uint256 curr_wealth = new_output_reserve.mul(new_market_price_out).add(new_input_reserve.mul(new_market_price_in));
    uint256 divergence_loss = prev_wealth.sub(curr_wealth);
    return divergence_loss;
}
```

```
/**
* @notice Public divergence loss function for ETH to Token trades with an exact output.
* @param output_amount Amount of ETH or Tokens being bought.
* @param input_reserve Amount of ETH or Tokens (input type) in exchange reserves.
* @param output_reserve Amount of ETH or Tokens (output type) in exchange reserves.
* @return divergence loss of proposed trade.
*/
  function getOutputDivergenceLoss(uint256 output_amount, uint256 input_reserve, uint256 output_reserve) public view returns (uint256) {
    require(input_reserve > 0 && output_reserve > 0);
    uint256 output_price = getOutputPrice(output_amount, input_reserve, output_reserve);
    uint256 new_output_reserve = output_reserve.sub(output_amount);
    uint256 new_input_reserve = input_reserve.add(output_price);
    uint256 new_market_price_in =  new_output_reserve.mul(1000) / (new_input_reserve.mul(997));
    uint256 new_market_price_out =  new_input_reserve.mul(1000) / (new_output_reserve.mul(997));
    uint256 prev_wealth = output_reserve.mul(new_market_price_out).add(input_reserve.mul(new_market_price_in));
    uint256 curr_wealth = new_output_reserve.mul(new_market_price_out).add(new_input_reserve.mul(new_market_price_in));
    uint256 divergence_loss = prev_wealth.sub(curr_wealth);
    return divergence_loss;
}
```

Divergence Loss is the cost suffered by the market maker as a result of their assets being worth less after the trade than prior to the trade. It is important to note that this does not imply that the market maker is losing money on the trade. Instead, a market maker has their assets converted during a transaction – for this example, let's assume a customer is converting Eth to tokens. This transaction results in the market maker gaining Eth and losing tokens and the price of Eth relative to tokens decreasing. As a result, at the new equilibrium market price, the market maker's assets were worth more prior to the transaction than after the transaction – this difference is the divergence loss of the transaction.

Load:

```
/**
* @notice Public load function for ETH to Token trades with an exact input.
* @param input_amount Amount of ETH or Tokens being sold.
* @param input_reserve Amount of ETH or Tokens (input type) in exchange reserves.
* @param output_reserve Amount of ETH or Tokens (output type) in exchange reserves.
* @return load of proposed trade.
*/

function getInputLoad(uint256 input_amount, uint256 input_reserve, uint256 output_reserve) public view returns (uint256) {
  require(input_reserve > 0 && output_reserve > 0);
  uint256 lin_slip = getInputLinearSlippage(input_amount, input_reserve, output_reserve);
  uint256 div_loss = getInputDivergenceLoss(input_amount, input_reserve, output_reserve);
  uint256 load = lin_slip.mul(div_loss);
  return load;
}


/**
* @notice Public load function for ETH to Token trades with an exact output.
* @param output_amount Amount of ETH or Tokens being bought.
* @param input_reserve Amount of ETH or Tokens (input type) in exchange reserves.
* @param output_reserve Amount of ETH or Tokens (output type) in exchange reserves.
* @return load of proposed trade.
*/

function getOutputLoad(uint256 output_amount, uint256 input_reserve, uint256 output_reserve) public view returns (uint256) {
  require(input_reserve > 0 && output_reserve > 0);
  uint256 lin_slip = getOutputLinearSlippage(output_amount, input_reserve, output_reserve);
  uint256 div_loss = getOutputDivergenceLoss(output_amount, input_reserve, output_reserve);
  uint256 load = lin_slip.mul(div_loss);
  return load;
}
```

Load is calculated by taking the product of the linear slippage (incurred by the customer) and the divergence loss (incurred by the market maker). For a transaction to occur both the trader and the provider must be willing to incur the costs associated with the transaction. Load measures the relationship between the costs on each side of the transaction; load is higher in cases where there is more uncertainty regarding the true market valuation of the two assets and lower when there is more certainty.

Summary:

I found this project to be incredibly interesting. As someone who will be working at a market making firm (albeit not in the decentralized finance space), analyzing the costs in a customer - market maker transaction is something I am used to doing but is very different in this space. Market makers must be very skeptical of the implied market valuation of their assets as dictated by their algorithms; in cases where their valuation differs greatly from true market prices, there is an opportunity for traders to profit at the expense of the provider in the form of divergence loss. However, in reality, traders must pay fees in addition to suffering from linear slippage in the price at which they are able to transact with the market maker, and must therefore assess if moving the implied market price to the true market price is actually profitable.

For my functions I assumed a cost curve of the form $xy = k$. This is reflected in the way I calculated the spot price prior to and after a transaction. Rather than using formulas to calculate the various costs, I chose to do so in a way that I found intuitive. I ran into some troubles with

my code, specifically getting the arctan function to work properly with the change in int types and overall testing with the absence of decimals in solidity. Lastly, I find the concept of load to be quite interesting as a metric for the cost trade-off between trader and provider but I don't think that its usefulness was entirely fleshed out in this project. I would love to delve deeper into this metric from a market maker's perspective to analyze how to optimize profits bearing in mind the costs the trader is bearing.