Philosophy Of Design and exploration of financial theory for An Automated Market Maker De-Fi Arbitrage Bot Which execute Flash Loans to Capture The Spread Between Price Mismatches On A Variety of The Uniswapv2 Implementations

Evan McGrane

Abstract

In this paper I explain the design philosophy behind my Defi automated market maker arbitrage bot. The bot uses the uniuswapV2 core and periphery smart contracts in order to fetch price data from Uniswap AND any other Uniswap fork through the V2Router so that we can capture the spread between any existing price misstates among the exchanges in order to simulate trades via the execution of a robust maximum profit algorithm and flash swap smart contract which uses Uniswap to call the flash swap. We will begin by exploring the mathematics and theory behind automated market makers as well as looking into the core functionality of the Uniswap contracts by highlighting key features that I have implemented in my bot to find arbitrage. This paper is concluded with a small section focussed on highlighting some of the limitations and shortcoming of my bot in a competitive environment as well as proposing some possible strategies and techniques that could easily be employed to greatly enhance the overall effectiveness of the bot to maximize its profitability.

1.0 Introduction: What is an automated market maker

Automated market makers are one of the most exciting things that have come intto fruition in the DEFI space,.However, at the same time they can simultaneously be one of the least understood primitives for many, especially when you get into the technicals. AMMs trace their origins to a Reddit post by Vitalik Buterin a few years ago. From there, we had Bancor being the first AMM live on Ethereum. However due to the incorrect use of their native BNT token, it failed to get much traction. This was until Uniswap came along and delivered on the simple promise of a properly functioning automated market maker with no extras. Since Uniswap got traction when it launched, the AMM space took off and is has many large players in the space today. Today there are a variety of different AMM's, but Uniswap is what's called a constant product AMM. As a market maker, providing liquidity is a tedious task, it often involves locking up significant capital, programming a trading bot and building up a feasible strategy. It is often in the hands of institutional providers where they have the capability and incentives to build such initiatives. The AMM structure creates possible ways for any individual to become passive market makers without worrying about the technical details for market makings. There are a few different strategies in creating that AMM structure, we call it the constant function market makers (CFMMs), under the CFMMs, there are a few different substructures with their unique characteristics.

$$V = \prod_t b_t^{W_t}$$

To understand more how CPMM's work Lets imagine we have a AMM that exchanges between token A and token B. Now say the AMM currently has 20 of token A in its reserve, and 40 of token B. This means that the Product is $20 \times 40 = 800$. This is the number that needs to stay constant (hence the name constant product market maker). Now lets turn our concentration because this is where we will explore the mathematics behind how this plays out.

$$x \times y = k$$

$$(x) = quanity\ of\ token1, \qquad (y) = quantity\ of\ token2, \qquad (k) = fixed\ product\ value$$

To create a uniswap market you need to provide two tokens in equal amounts. In this example lets say that we create an DAI/ETH pair such that we provide 10 ether at 4000 dollars and 40,000 DAI at 1 dollar per dai. Effectively we now have satisfied the constant product rule

$$E_S = 10, \quad E_P = 4000\$$$

$$D_S = 40{,}000, \quad D_P = 1\$$$

$$where\ E_S = Eth\ Supply, \qquad E_P = Eth\ price, \qquad D_s = Dai\ supply, \qquad D_p = Dai\ price$$

The in order to calculate the constant product we use the formula above. Thus, in this scenario here the the ETH/DAI pair we would have

$$x = 10 \times 4000 = 40{,}000$$

$$\gg y = 1 \times 40{,}000 = 40{,}000$$

$$\gg x \times y = k$$

$$\gg k = 40{,}000 \times 40{,}000 = 1{,}600{,}000{,}000$$

So if we just saw this equation, we could determine the ratio of ETH/DAI = 1,600,000,000 / 40,000 (which is the price of Ethereum in this scenario)! Now let's say someone would like to purchase $4000 worth of ETH from the pool. What they're essentially doing is increasing (y) by 4000 (depositing DAI) and decreasing (x) by 1 (withdrawing ETH). Thus our new equation for the their price has the form:

$$(E_P \times (E_S - EthOut)) \times ((D_P \times D_s) + DaiIn) = 1{,}600{,}000{,}000$$

$$(E_P \times (E_S - 1)) \times ((D_P \times D_s) + 4000) = 1{,}600{,}000{,}000$$

$$\left(E_p \times 9\right) \times (1 \times 40{,}000) + 4000) = 1{,}600{,}000{,}000$$

$$\left(E_p \times 9\right) \times (40{,}000 + 4000) = 1{,}600{,}000{,}000$$

$$\left(E_p \times 9\right) = \frac{1{,}600{,}000{,}000}{40{,}000 + 4000}$$

$$E_p = \frac{9 \times 1{,}600{,}000{,}000}{5000} = 4040.40\$$$

So we can see in the constant product AMM the price is always determined by supply and demand if one assets reserves decreases then it means its being sought after and from the maths above we can see that its price increases relative to the other asset in the pair. This is a high level overview on how a typical constant product AMM works. One of the first AMMs in the space that got extreme traction. Uniswap v1 was innovative due to the simplicity of the protocol and user interface. Provide a token and ETH of equal value and you're good to go as an Liquidity provider (LP). Uniswap proved that orderbook-less models are viable on Ethereum. Uniswap released v2 earrly last year (2020) year with the ability to create a pair between two tokens rather than forcing quotes against ETH only.

However, relative to the competition Uniswap has fallen behind due to the inflexibility in fees, pool customisation, formula and pairings. Every trade incurs a 0.3% trade fee but pool owners can't set fees and the fee is always the same. V3 does inch up to the competition and the introduction/launch of the Uniswap ERC20 Token brought much popularity back to Uniswap.

2.0 A Closer Look At UniswapV2

The marginal price offered by Uniswap (not including fees) at any given time (t) can be computed by dividing the reserves of an asset (a) by the reserves of asset (b)

$$p_t = \frac{r_t^a}{r_t^b}$$

Since arbitrageurs will trade with Uniswap if this price is incorrect (by a sufficient amount to make up for the 0.03% fee), the period offered by Uniswap tends to track the relative market price of the assets. This means that it can be used as an approximate price oracle. Uniswap V2 builds upon the oracle functionality of uniswapV1 by introduction the concept of cumulative prices. Basically, this attributes to the measurement of the price exactly before the first trade on each block. This price is more difficult to manipulate than prices during a block. Specifically, uniswapv2 accumulates thus price, by keeping track of the cumulative sum of the prices at the very beginning of each block. Each price is weighted by the amount of time has elapsed since the last block in which it was updated. This means that the accumulator value at any time should be the sum of the spot price at each second in the history of the smart contract

$$a_t = \sum_{i=1}^{t} p_t$$

$$(where\ \boldsymbol{p_t}\ is\ the\ marginal\ price\ from\ above)$$

To estimate the time weighted average price from time (t_1) to time (t_2), an external Calle can checkpoint the accumulator's value at (t_1) tand (t_2). The we subtract the first value from the second and divide this result by the total number of elapsed seconds

$$pt_1, t_2 = \frac{\sum_{i=t_1}^{t_2} p_i}{t_2 - t_1} = \frac{\sum_{i=1}^{t_2} p_i - \sum_{i=1}^{t_1} p_i}{t_2 - t_1} = \frac{a_{t_2} - a_{t_1}}{t_2 - t_1}$$

Users of the Uniswap oracle smart contract can choose when to start and end the period. However, it is important to note. The key selling point of AMM arbitrage is the ability to use flash loans or flash swaps. In Uniswap v1, a user purchasing ABC with XYZ needs to send the XYZ to the contract before they could receive the ABC. This is inconvenient if that user needs the ABC they are buying in order to obtain the XYZ they are paying with. For example, the user might be using that ABC to purchase XYZ in some other contract in order to arbitrage a price difference from Uniswap, or they could be unwinding a position on Maker or Compound by selling the collateral to repay Uniswap. Uniswap v2 adds a new feature that allows a user to receive and use an asset before paying for it, as long as they make the payment within the same atomic transaction. The swap function makes a call to an optional user-specified call back contract in between transferring out the tokens requested by the user and enforcing the invariant. Once the call back is complete, the contract checks the new balances and confirms that the invariant is satisfied. The last noteworthy thing to mention about Uniswap is their ***sync()*** event. To protect against bespoke token implementations that can update the pair contract's balance, Uniswap v2 has an important bail-out function called ***Sync()***. In essence, ***Sync()*** functions as a recovery mechanism in the case that a token asynchronously deflates the balance of a pair. In this case, trades will receive sub-optimal rates, and if no liquidity provider is willing to rectify the situation, if he pair is stuck. ***Sync()*** exists to set the reserves of the contract to the current balances, providing a somewhat graceful recovery from this situation.

3.0 The UniswapV2 Forks

One other thing that is very useful is the creation of multiple other exchanges that are effectively "Uniswap clones", where the code base is an exact copy of that of Uniswap's. Some well known UniswapV2 forks include SushiSwap, Pancake Swap, Quick swap, Crow Swap, Sake Swap and more. If we have such a case where and exchange is build on top of forking the Uniswap codebase then we have an exciting way to preform arbitrage across these Uniswap clones in a relatively easy way. This is because Uniswap has two "main" smart contracts. Those are the Uniswap Router and the Uniswap Factory. Liquidity providers (LP) can create token pair markets (called pools) by deploying them using the Uniswap factory contract. Once a pool is created, anyone in the ecosystem can provide liquidity to it. Liquidity providers earn a flat percentage fee according to their stake in the pool's total liquidity. Shares of the pooled liquidity are accounted for using a liquidity token, commonly known as an LP token. You can think of each LP token as a share unit in the pool. The router is the smart contract used to interact with a token pair pool. On the other hand, Routers are stateless, meaning they don't hold token balances. Therefore, they can be replaced safely and in a trust less way for a more efficient router in the future. For example, the current V2 router (02) is an upgrade from the first iteration (01). The Router mainly has functions to do with the "swapping" and "transportation" of tokens.

All of the Uniswap forks have identical Router and Factory contracts to Uniswap. Therefor it becomes incredibly easy to preform arbitrage between all of these cloned Uniswap AMM's because we don't have to concern ourselves with writing different custom logic or taking different precautions to either manipulate data or to preform swaps on the different exchanges. We can effectively use the uniswapV2 Router contract to initialise the state of all the other exchanges whiles passing in the exact addresses in order to access the data of a particular Uniswap fork AMM.

4.0 Algorithm Design

In this section I will explain my design philosophy for my Bot implementation. I just want to mention that I was designed my arbitrage algorithm around the problem requirements and if I was to make another Bot for myself this would not be the approach I would take. What I mean by this is sometimes the Uniswap reserves take some time to update via the ***Sync()*** event emission. This can effectively be bad for the arbitrage algorithm as we may run into the scenario where our algorithm calculates a profitable arb but by the time we execute the flash swap the prices actual market prices on the exchanges may not be exactly the same. In the ideal scenario I would write my own price oracle, perhaps base it off of the Uniswap price oracle to get much better and consistent price data. I also only used Infura as my provider for this bot. When forking main net with ganache-cli, it seems as if ganache is even worse at listening to these sync events. I should ideally have used the flashBots RPC node endpoint in my project to prevent frontrunning etc but to be honest I didn't even think of it at the time. These two things are areas I will focus on in my own personal project. Restricting myself to using the Uniswap factory and router contracts prevents me from arbitraging on other dexes that are not Uniswap forks. So, I think an ideal option would to be source my own price data via a custom built price oracle, use a flashBots' RPC node and use Aave as my means to perform flash swaps and preform one extra trade if needs be to get the funds back to Aave.

The first step in the algorithm is to obviously get access to price data. As explained in section 2.0 this can be done by fetching the reserves of a given pair from Uniswap's pair contract and dividing them to get the marginal price. Recall the formula

$$p_t = \frac{r_t^a}{r_t^b}$$

If we want to arbitrage with a pair such as DAI/WETH, then we would divide $reserve_W$ by $reserve_D$ to get the price of WETH per in terms of DAI. W can find the rate for DAI by taking the reciprocal of this value. Once we have the price data we need to calculate our expected return that we get out for providing and input of amount (x). Uniswap has a very good function in the UniswapV2Library contract called $getAmountOut()$. Given an input asset amount, this function returns the maximum output amount of the other asset (accounting for fees) that we can expect to receive in return given. The calculate amount out is given by the formula

$$amountOut = \frac{A_F \times R_O}{(R_{IN} \times 1000) + A_F}$$

$$where\ A_F = AmountIn \times 997, \quad R_{IN} = ReserveIn, \quad R_O = ReserveOut$$

Say we choose to trade with 1 WETH, then we would pass in the Wei denominated value of 1 WETH into this function in order to get the expected return in DAI. However, we can employ a powerful algorithm which will calculate the maximum input value that we should provide in order to get a maximum return. Consider the initial state of pair0 and pair1 in the table below

	Pair0	Pair1
Base Token Amount	a1	b1
Quote Token Amount	a2	b2

Table1: table showing 2 pools of the same token pair

Note in the example of DAI/WETH, from the table above here, DAI is the quote token and WETH is the Base token. In this case we can calculate the $amountIn$ to borrow to maximise our profit as:

$$\Delta a_1 = \frac{\Delta b1\ \times a_1}{b_2 - b_2} \qquad \Delta a_2 = \frac{\Delta b_2\ \times a_2}{b_2 - b_2}$$

The amount of the borrowed Quote token are some so that $\Delta b_1 = \Delta b_2$, and we can let $x = \ \Delta b$, then usimg this we can calculate our expected profit as a function of (x). Thus we now have:

$$f(x) = \Delta a_2 - \Delta a_1 = \frac{a_2 x}{b_2 + x} - \frac{a_1 x}{b_1 + x}$$

The whole point of this algorithm is to calculate the maximum profit. So, if our profit estimations are given as this function of (x) then if we recall some simple calculus, we know that the derivative of any function can be used to calculate the local maxima and local minima. Thus if we take the derivative of the above expression then we can calculate our profit by evaluating the derivative at $x = 0$, wghich is the local maxima.

$$f'(x) = \frac{a_2 b_2}{(b_2 + x)^2} - \frac{a_1 b_1}{(b_1 - x)^2}$$

$$\frac{a_2 b_2}{(b_2 + x)^2} - \frac{a_1 b_1}{(b_1 - x)^2} = 0$$

We can now do some simple algebraic manipulation to split this expression up a 2nd order quadratic equation. So we will get each of the coefficients in the expression below be (a), (b) and (c) respectively In order to simplify the maths

$$(a_1 b_1 - a_2 b_2)x^2 + 2b_1 b_2 (a_1 + a_2)x + b_1 b_2 (a_1 b_2 - a_2 b_1) = 0$$

$$\begin{cases} a = a_1 b_1 - a_2 b_2 \\ b = 2b_1 b_2 (a_1 + a_2) \\ c = b_1 b_2 (a_1 b_2 - a_2 b_1) \end{cases}$$

Now we have a reduced general quadratic equation as seen below and if we solve this we get the following solution, also given below.

$$ax^2 + bx + c = 0$$

$$\begin{cases} x = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a} \\ 0 < x < b_1 \\ x < b_2 \end{cases}$$

The solution (x) represents the amount that we need to borrow from pair0 in order to get the maximum returns in an arbitrable case. Uniswap then provides another powerful function called $getAmountIn()$ that given a specified desired output amount of an asset, returns the required minimum input amount of the other asset to get the specified amount. The formula for this calculation is as follows

$$amountIn = \left(\frac{R_{IN} \times A_O) \times 1000}{(R_O - A_O) \times 997}\right) + 1$$

$$where\ A_0 = AmountOut, \quad R_{IN} = ReserveIn, \quad R_O = ReserveOut$$

However before we can use these prices to calculate the difference and hence our expected profit in any case we need to first figure out whether to trade in the DAI/WETH direction or the WETH/DAI direction. I have written a smart contract called $TradeOrder.sol$ which uses the reserves for the pair in both pools on two given exchanges in order to determine the following criteria

1. Which of the two tokens is smaller (by the value of its address not by price)
2. Which of the pools is smaller by the value of their reserves?
3. Order the reserves in every scenario such that $[a1, b1, a2, b2]$, where $(a1, b1)$ =the pool with the lower price denominated in the quote token and $(a2, b2)$ = the base tokens in the two pools

In order to figure out what direction we need to trade in we must consider the base token. Recall that the base token is TOKENY such that the pair is in the form TOKENX/TOKENY. We can calculate which token in the pair should be the base token by using the following conditions.

```
if (pool0token0 is in the predefined baseToken array) then {
    baseTokenSmaller = true, the pair is ordered TOKEN0/TOKEN1

} else if (pool0token0 is in the predefined baseToken array) then {
    baseTokenSmaller = false, the pair gets ordered TOKEN1/TOKEN0
}
```

Once we have established whether or not the base token that we predefine is smaller in order to correctly choose the trade direction then we have what we need to order our reserves. We need to order the token reserves in both pools in order to fix it so that we are always borrowing form the lower price pool and selling to the higher price pool. To establish which pool is the lower price pool we need to first calculate the pair margin price using (1) for each case. One for the case when the base token is smaller and two for the case when the base token is not smaller. The formulas for both bases are as follows:

$$(if\ base\ token\ is\ smaller) \quad p_{token_0} = \frac{pool_0 Reserve_0}{pool_0 Rserve_1}, \quad p_{token_1} = \frac{pool_1 Reserve_0}{pool_1 Reserve_1}$$

$$(if\ base\ token\ is\ not\ smaller) \quad p_{token_0} = \frac{pool_0 Reserve_1}{pool_0 Rserve_0}, \quad p_{token_1} = \frac{pool_1 Reserve_1}{pool_1 Reserve_0}$$

Essentially if the base token is smaller, we divide the reserves as normal where we divide $Reserve0$ by $Reserve1$. However, recall that if the base token is not smaller than the quote token, we reverse the trade direction. Therefore, in order to get the price in this case we need to divide the reserves in the opposite manner. Once we have determined the price of the tokens in the pair, we finally have all of the information we need to effectively order the reserves such that the order is $(lowerPricePool,\ higherPricePool)$. We can use the price information calculated above to order the reserves for both pools. Let's say that the ordered reserves need to take following form

$$Reserves = [a1, a2, b1, b2]$$

$$where\ (a1, b1) = the\ poolwith\ the\ lower\ price\ denominated\ in\ the\ quote\ token$$

$$and\ (a2, b2) = the\ base\ tokens\ in\ the\ two\ pools$$

In the case where p_{token_0} from above is less than p_{token_1} then we know that the lower pool is going to be $pool0$ and the higher pool is going to be $pool1$. And for the reserves they will be ordered the following was.

$case\ where\ baseToken\ \boldsymbol{IS}\ smaller\ and\ p_{token_0} < p_{token_1}$:

$$a1 = pool_0Reserve_0$$

$$a2 = pool_0Reserve_1$$

$$b1 = pool_1Reserve_0$$

$$b_2 = pool_1Reserve_1$$

$case\ where\ baseToken\ \boldsymbol{IS\ NOT}\ smaller\ and\ p_{token_0} < p_{token_1}$

$$a1 = pool_0Reserve_1$$

$$a2 = pool_0Reserve_0$$

$$b1 = pool_1Reserve_1$$

$$b_2 = pool_1Reserve_0$$

$case\ where\ baseToken\ \boldsymbol{IS}\ smaller\ and\ p_{token_0} > p_{token_1}$

$$a1 = pool_1Reserve_0$$

$$a2 = pool_1Reserve_1$$

$$b1 = pool_0Reserve_0$$

$$b_2 = pool_1Reserve_1$$

$case\ where\ baseToken\ \boldsymbol{IS\ NOT}\ smaller\ and\ p_{token_0} > p_{token_1}$

$$a1 = pool_1Reserve_1$$

$$a2 = pool_1Reserve_0$$

$$b1 = pool_0Reserve_1$$

$$b_2 = pool_0Reserve_0$$

This might seem confusing but its not really to bad. Basically, we are using the state of the base token in the pairs and also the price of the pairs to order the reserves such that they are ordered from the lower price pool to the higher price pool. The reason we have four check is because we have two cases for when $p_{token_0} < p_{token_1}$ and comversley two cases for when $p_{token_0} > p_{token_1}$. Those cases being when the base tojen is and isn't smaller. When we have our reserves ordered we can be confident that based on the token address and the pair price data that we are always going to be borrowing from the lower pool and selling to the higher. Once we have the reserves ordered we can use the maximum profit algorithm described above to calculate the In and Out amounts from the Uniswap $getAmountIn()$ and $getAmountOut()$ functions. When this step is complete, and we have the expected by and sell price of the pairs on both pools we can calculate the difference to estimate the potential profit if any

We can efficiently use these two functions to determine if there is an arbitrage opportunity for a given trade on two Uniswap-esque exchanges. For example, lets say we want to take a flash loan off Uniswap to trade with and preform arbitrage on another exchange to keep the profits. Well, using the uniswapRouterContract, we could call the $getAmountIn()$ on Uniswap and specify 1 ether as the amount we want to get as a return. Then the formula above would return us the minimum number of DAI that we would have to trade in order to get 1 WETH. Once we have this DAI amount, we can conversely call the $getAmountOut()$ function on say sushiSwap for example by passing in this 1 WETH. According to the formula above this would return to us the maximum amount that we can expect to receive for the trade. With these two values if the amount of DAI that we receive back on sushiSwap is greater than the amount of DAI we originally spend on Uniswap, then we have profit which will be equal to the difference after we pay back the Uniswap flash loan that we borrowed. To estimate this difference we apply the following formula

$$difference = AmOut_{uniswap} - AmIn_{sushiswap}$$

$$totalDifference = difference \times amIn_{uniswap}$$

If this total difference is greater than zero, we have a potential profitable opportunity. However, in order to calculate out final profit we must take into account the gas costs that are required in order to execute all of the trades that we need to capture this arbitrage. There are a few different functions that we need to call from different smart contracts. Firstly, we need to approve the uniswapRouter Contract to spend tokens on our behalf. The we need to estimate the gas for a Uniswap flash loan, a transfer of this loan to the sushiSwap exchange, another swap on sushiSwap and then one more transfer back to Uniswap to pay back the loans. We can estimate the gas for all of this by using the $Web3.js$ libraries' ***.estimateGas()*** function. Thus we have the following pseudo Code:

```
GasForApproval = DAI.approve().estimateGas()
GasFaorFlashLoan = FlashBot.flashSwap().estimateGas()
combinedGas = GasForApproval + gasForFlashLoan
gasPrice = web3.eth.getGasPrice()
TotalGas = gasPrice x combinedGas / 10 ** 18
```

These estimations roughly estimate that the gas price to execute the entire trade will be 0.0072 ETH. But this price may vary depending on the current gas Price at the time of the trade. When we have the total estimated gas we can subtract this from the total difference above in order to calculate our final estimated profit margin

$$totalProfit = totalDifference - totalGas$$

If this number is grater than zero then we are expected to make a profit and thus we call my FlashBot smart contract in order to execute the trade. This is effectively the main algorithm its simple and concise. There is more detailed logic in the smart contract themselves. I have written some logic that takes in the two pair pools and depending on the size of the reserves we reorder the token pool reserves for doing the price calculations in such a way that we always borrow from the lower price pool and sell to the higher price pool. As you will see in my code doing this prevents me from

having to potentially running my algorithm for DAI/WETH and WETH/DAI simultaneously on both exchanges, as by ordering the token pair and reserves in such a ways that if the base token is smaller we reverse the per order and also that we always order to reverse to borrow from the lower pool means we will always be executing the trade for the most profitable case. The flash swap contract itself is also simple. I have written a custom flash swap function which was inspired from the example given in the Uniswap docs but I felt that mine is more robust and the call back function is also similar but I have added my own custom logic. All of the code is very well commented for you to read so I will leave this section at that.

5.0 Considerations & Improvement Proposals

Although I have had one successful arbitrage flash swap with my bot on a main net forked environment most of the time there is no arbitrage available. In this section I will go over some improvement proposals to maximise the chances of getting arbitrage. Firstly, there is too many bots running in the wild, especially on Ethereum and binance smart chain. Although I think my Bot code is rather good, in the grand scheme of things a bot that I made I a week is too simple to be competitive with production bots that are employed by different collectives. However, there are many different strategies and technique that we could employ to be more competitive. Some of which include

1. Lower the network latency by using out own custom node
2. Set higher gas prices to make sure out transactions get settled quickly enough to take profit. This is like a competition between bots if they find a profitable trade at the same time
3. Although this is not a problem here, monitoring les tokens can help. The more we monitor the lesser frequency the bot is looping. A good work around is to run multiple scripts at once to monitor separate pairs
4. Going to other lesser used blockchains like Fantom, Matic, Polygon etc. These blockchains will have lesser bots running on them so it would be easier to be more competitive
5. Do some other works such as liquidation bots, arbitrage in balancer or curve or 0x for example.

The above examples are just a few of many different techniques that we could employ to make out bot more efficient. Also to reiterate what I said in section 1, writing our own oracle to get better quality and more up to date rice data would hugely benefit in finding slim arbitrage opportunities. The fact that in my bot I have to reply on the Uniswap **Sync()** event to get price updates from the reserve is a big point of failure because the reserves don't update frequently enough at all to be competitive. Using chain-link would be desirable but sadly there is no way to get specific price data on specific exchanges using chain-link. This the best option would be to write a custom oracle.

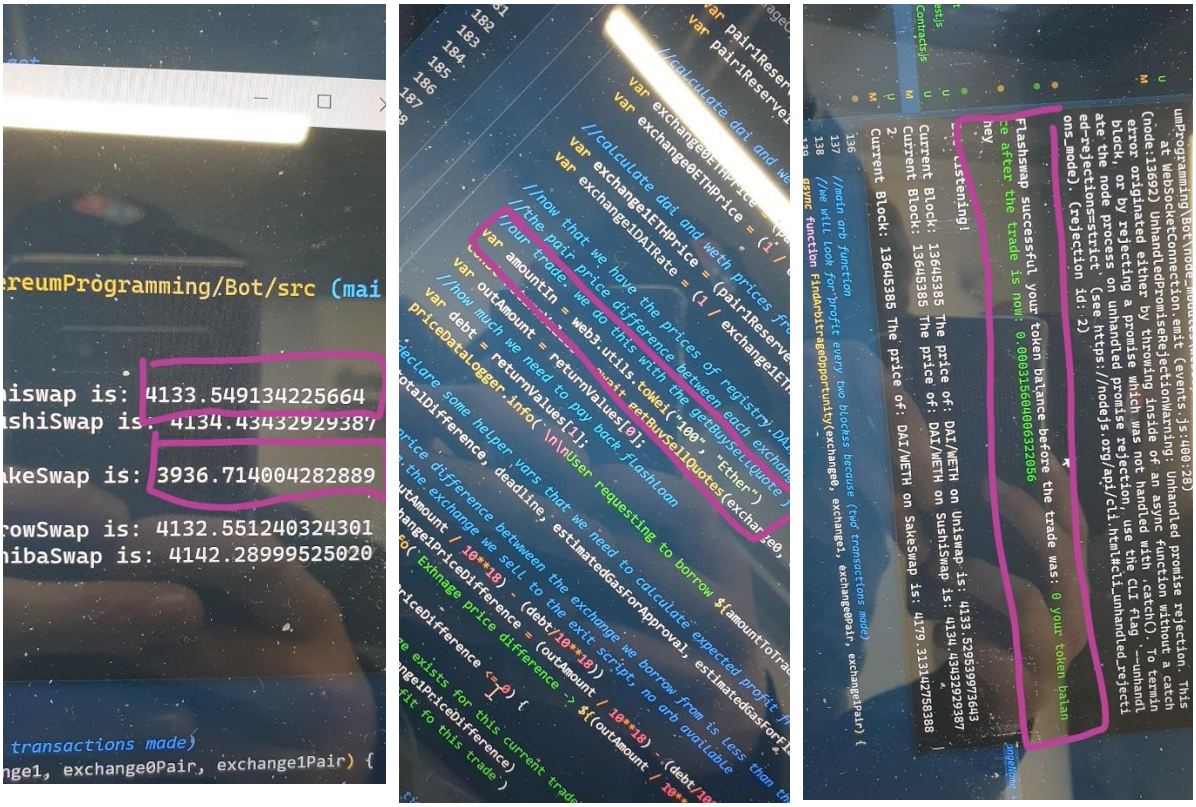
Also, In the Ethereum me pool, ex predators take the form of these more sophisticated "arbitrage bots" which can front run our transactions. Sophisticated Arbitrage bots can monitor pending transactions and attempt to exploit profitable opportunities created by them. No white hat knows more about these bots than Phil Daian, the smart contract researcher who, along with his colleagues, wrote the Flash Boys 2.0 paper and coined the term "miner extractable value" (MEV). In theory MEV accrues entirely to miners because miners are the only party that can guarantee the execution of a profitable MEV opportunity. In practice, however, a large portion of MEV is extracted by independent network participants referred to as "searchers." Searchers run complex algorithms on blockchain data to detect profitable MEV opportunities and have bots to automatically submit those profitable transactions to the network. With that, for some highly competitive MEV opportunities, such as DEX arbitrage which is what we are concerned with, searchers may have to pay 90% or even more of their total MEV revenue in gas fees to the miner because so many people want to run the same profitable arbitrage trade. This is because the only way to guarantee that their arbitrage transaction runs is if they submit the transaction with the highest gas price.

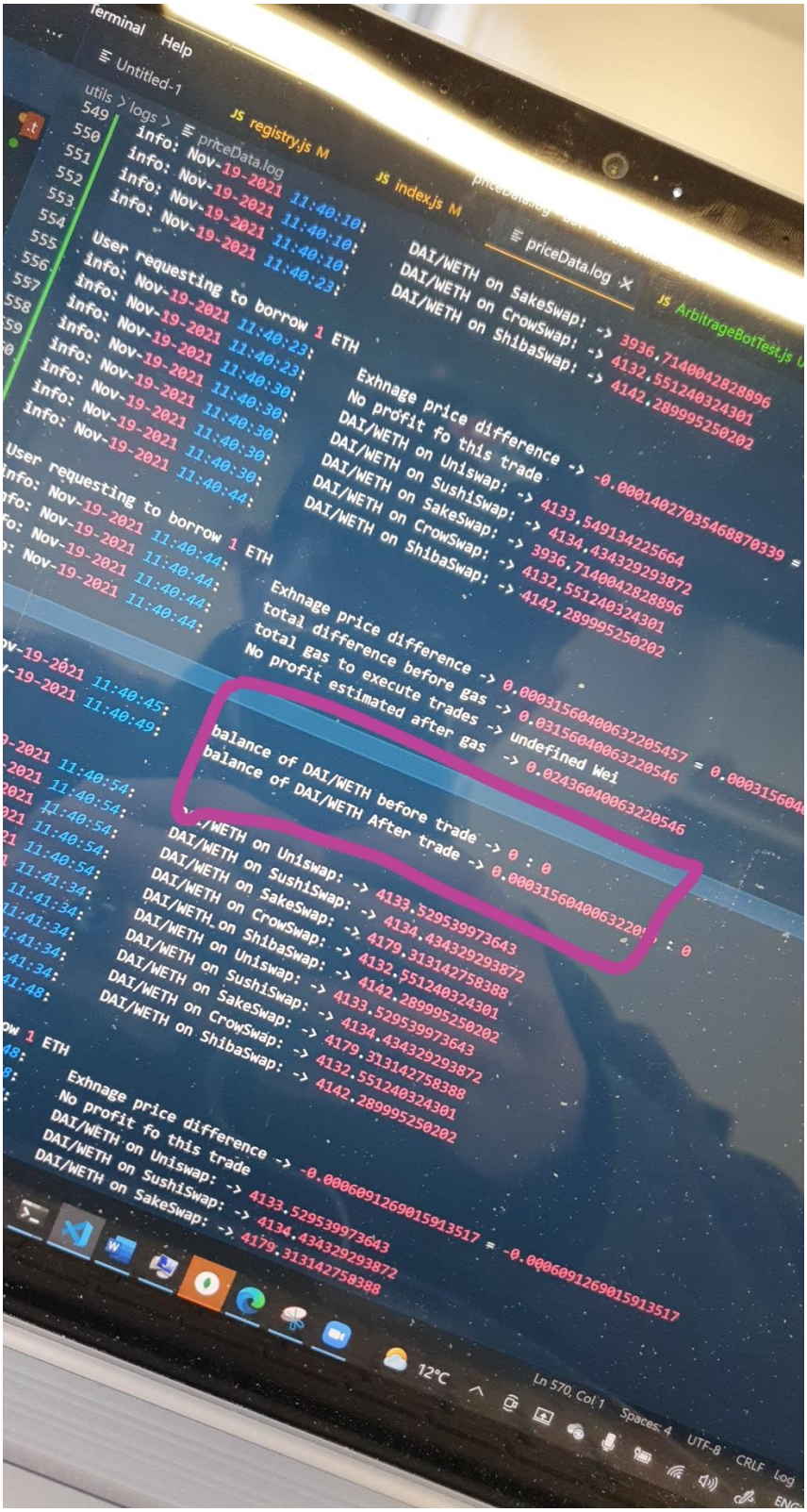
One last strategy then, that we could then employ is to interact with the flashBots relays through one of their RPC endpoints. The great thing about flashBots is that they connect searchers directly to miners and allows them to avoid the public tx pool. Searchers with transactions they would like to send miners first craft what we call "bundles" and send these to FlashBots' MEV-Relay. MEV-Relay is a gateway that FlashBots runs today which simulates searchers' bundles, and if there are no errors then forwards them on to miners. Miners then receive bundles and include them in blocks if it is profitable for them to do so. FlashBots offer frontrunning protection which is very desirable on Uniswap and other AMM's

6.0 Conclusion

In this paper we explored some of the underlying mathematics and theory associated with automated market makers as well as looking specifically at the inner workings of some of the key features of the Uniswap protocol in order to understand the reasons behind the strategy I used to develop my AMM arbitrage bot algorithm. I just want to highlight that I made a testing environment where I used open zeppelin to create an mint dummy ERC20 tokens which I then added liquidity to n Uniswap in such a sway that there was always a large price mismatch between the pairs on both pools. This way I was able to test my flashBot smart contract. In the real forked main net environment I have successfully picked up one successful arbitrage trade where I received a profit of 0.003 Ether from a flashloan of 100 ether. The arbitrage was captured between Uniswap and sake swap. I have had a few arbitrage trades but all of them were non profitable after the gas estimations. Do the conclusion is that my Bot does work and I do intend to deploy it on main net sometime soon once I can afford the crazy deployment costs.

Proof of my first successful Arb This is also recorded in a mongo DB logger I made for my bot




Terminal Help
Untitled-1
priceData.log
registry.js M
index.js M
ArbitrageBotTest.js
info: Nov-19-2021 11:40:10:
info: Nov-19-2021 11:40:10:
info: Nov-19-2021 11:40:10:
info: Nov-19-2021 11:40:23:
User requesting to borrow 1 ETH
info: Nov-19-2021 11:40:23:
info: Nov-19-2021 11:40:30:
info: Nov-19-2021 11:40:30:
info: Nov-19-2021 11:40:30:
info: Nov-19-2021 11:40:30:
info: Nov-19-2021 11:40:44:
User requesting to borrow 1 ETH
info: Nov-19-2021 11:40:44:
info: Nov-19-2021 11:40:44:
info: Nov-19-2021 11:40:44:
DAI/WETH on SakeSwap: -> 3936.71400428288896
DAI/WETH on CrowSwap: -> 4132.551240324301
DAI/WETH on ShibaSwap: -> 4142.289995250202
Exhnage price difference -> -0.000140270354688703339 =
No profit fo this trade
DAI/WETH on Uniswap: -> 4133.549134225664
DAI/WETH on SushiSwap: -> 4134.434329293872
DAI/WETH on SakeSwap: -> 3936.71400428288896
DAI/WETH on CrowSwap: -> 4132.551240324301
DAI/WETH on ShibaSwap: -> 4142.289995250202
Exhnage price difference -> 0.000315604006322054577 = 0.000315604
total difference before gas -> 0.031560400632205457
total gas to execute trades -> undefined Wei
No profit estimated after gas -> 0.024360400632205457
balance of DAI/WETH before trade -> 0 : 0
balance of DAI/WETH After trade -> 0.000315604006322054
DAI/WETH on Uniswap: -> 4133.529539973643
DAI/WETH on SushiSwap: -> 4134.434329293872
DAI/WETH on SakeSwap: -> 4179.313142758388
DAI/WETH on CrowSwap: -> 4132.551240324301
DAI/WETH on ShibaSwap: -> 4142.289995250202
Exhnage price difference -> -0.000609126901591351 = -0.000609126901591351
No profit fo this trade
DAI/WETH on Uniswap: -> 4133.529539973643
DAI/WETH on SushiSwap: -> 4134.434329293872
DAI/WETH on SakeSwap: -> 4179.313142758388
12°C
Ln 570, Col 1
Spaces: 4
UTF-8
CRLF
Log