Research

How UniswapV2 Swaps Work

Alex R. Mead

Abstract

This paper walks through the operations of a UniswapV2 swap, with the goal of expediting the learning process for those interested in UniswapV2 engagement. To begin, a high level overview of the UniswapV2 architecture is given: the *Core* and *Periphery*. Next, automated market maker (AMM) *token pairs*, governed by the *constant product formula*, and their main use case, token *swaps*, are discussed. Throughout the paper, mathematics, Solidity code, and on-chain transactions are used for clear explanation of the topic.

Uniswap is a group of decentralized exchanges (DEX's) originally created on Ethereum by Hayden Adams in 2018. DEX's are on-chain, *automated market makers*, that use pooled capital to allow for *counter-party free* and *permissionless* swapping between on-chain assets. UniswapV2 came online in 2020 and added several new features, including arbitrary ERC20-ERC20 pairs. The latest DEX, UniswapV3, came online in 2021 and added further functionality, most notably, *concentrated liquidity*. This paper focuses exclusively on version 2, specifically a deep dive into swapping between ERC20 tokens.

1 UniswapV2 - Architectural Overview

The UniswapV2 protocol is comprised of two smart contract clusters: the Core and the Periphery. The Core has two basic smart contract types, a single Factory (0x5C69bEe701ef814a2B6a3EDD4B1652CB9cc5aA6f) which was deployed by the Uniswap developers and many pairs, see Figure 1.

Each pair is a unique one-to-one mapping of ERC20 token types used for *swaps*. Pair contracts are deployed by protocol users via Factory functions. There can only be one pair for any given ERC20 token combination. For example, if you want to swap WETH-LINK, there is only one UniswapV2 pair in which to do that, located at 0xa2107FA5B38d9bbd2C461D6EDf11B11A50F6b974.

The Periphery contracts are not essential to the UniswapV2 protocol, however, offer convenience and safety. Only advanced users should interact directly with Core smart contracts. The periphery will be discussed briefly below.

2 UniswapV2 - Basic Swap Explained

The main purpose of the UniswapV2 protocol is to offer a convenient on-chain system for converting one ERC20 token into another. As introduced above, this is accomplished using a smart contract known as a pair, which is simply two pools of ERC20 tokens which users can trade between. As of this writing there are more than 147,000 pairs in the UniswapV2 protocol.

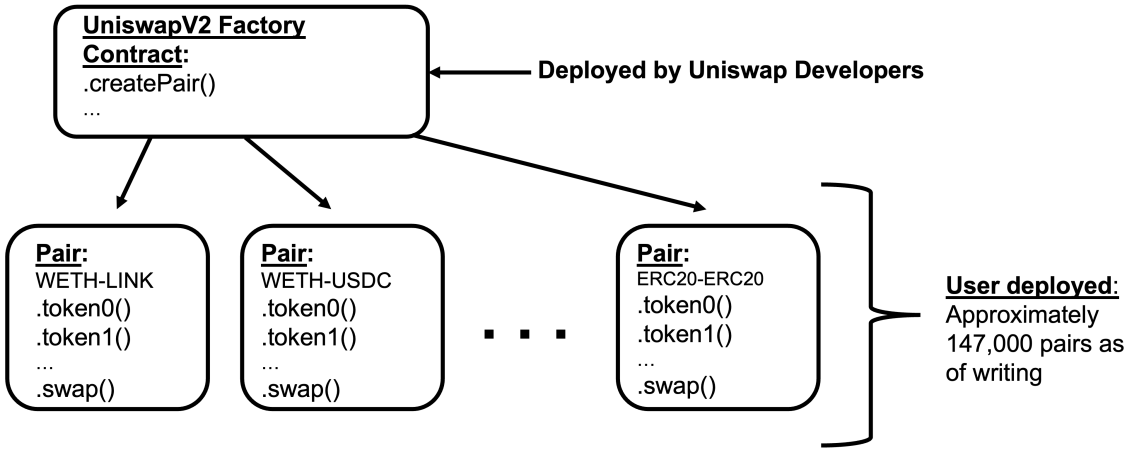

Figure 1: Relationship of UniswapV2 Factory and Token Pair contracts.

Constant Product Formula

The basic mechanics of an UniswapV2 pair are governed by the *Constant Product Formula* as shown in equation 1

$$x * y = k, \tag{1}$$

where x and y each represent the reserves of the ERC20 tokens, token0 and token1 respectively. k is a constant value[1] that does not change throughout contract life-cycle[2].

To better understand how a swap actually plays out, consider some block height, $b \in B_{pair}$, where $B_{pair} \subset N$ is the set of all blocks of the Ethereum Mainnet since the given *pair* was initialized. One such *pair* might be WETH-LINK as mentioned above. Considering this *pair* at height b, there exists some initial *reserves* r_0^b and r_1^b of *token0* and *token1*, such that,

$$r_0^b * r_1^b = k^b. \tag{2}$$

Now suppose further at block height b, a user would like to swap a certain amount of token0 (denoted, $token0_{in}$) for token1 which will be executed at the top of block $b+1$. How much of token1 (denoted, $token1_{out}$) would they receive?

To calculate this value, we first deduct the *fee* of 0.3% from the $token0_{in}$ value, leaving an effective input of $0.997 * token0_{in}$. Understanding the constant product formula must always hold for a given pair, we can use equation 1 to show that,

$$r_0^b * r_1^b = r_0^{b+1} * r_1^{b+1}, \tag{3}$$

where r_0^{b+1}, r_1^{b+1} are the new *reserve* values for token0 and token1 after the swap. Understanding the reserve of token0 has only increased from the user input, we know,

$$r_0^{b+1} = r_0^b + 0.997 * token0_{in}. \tag{4}$$

Further, we can express the token1 output as the difference between initial and final state as,

$$token1_{out} = r_1^b - r_1^{b+1}. \tag{5}$$

Using substitution for r_0^{b+1} from equation 4 into equation 3 we get,

$$r_0^b * r_1^b = (r_0^b + 0.997 * token0_{in}) * r_1^{b+1}, \tag{6}$$

which with simple algebra gives us,

$$r_1^{b+1} = \frac{r_0^b * r_1^b}{r_0^b + 0.997 * token0_{in}}, \tag{7}$$

[1]Note, all mathematical symbols can be found in the Symbols section on page 6.
[2]Technically with the 0.3% fee k does change, but this is not necessary to understand trade mechanics.

we finish the calculation by substitution and algebra using equations 5 and 7 to get $token1_{out}$ as,

$$token1_{out} = r_1^b - \frac{r_0^b * r_1^b}{r_0^b + 0.997 * token0_{in}}. \tag{8}$$

Notice, all values on the right hand side of the equation are known, thus it is a closed from calculation for $token1_{out}$ given some input $token0_{in}$ and the initial pair conditions.

Solidity Smart Contract Code

The above mathematical expressions are realized in Solidity code in a function called, swap. The code can be found in the Uniswap Github repository v2-core, in the file v2-core/contracts/UniswapV2Pair.sol. The swap function itself has been extracted and reproduced in Appendix A. Let's look through the function line-by-line and check if it matches our reasoning above.

First thing to notice, is the user specifies the amount of *both* token0 and token1 they wish to withdrawal (i.e. $token0_{out}, token1_{out}$), meaning typically one of these will equal zero. As is often the case in code, the implementation of the swap does not follow the *intuitive* steps one might expect. We'll now outline the process, offering a line-by-line explanation using the same notation above mapped to the variables in the code.

Beginning on line 160 and running through 169, several steps occur unrelated to the swap directly. First, some "sanity checks" on the input values ensure at least some tokens are being requested, almost in reverse order as compared to above,

$$token0_{out} = amount0Out \quad and \quad token1_{out} = amount1Out. \tag{9}$$

Then current token reserves for the *pair* itself are queried, meaning line 161 effectively gives us,

$$r_0^b = \_reserve0 \quad and \quad r_1^b = \_reserve1, \tag{10}$$

Next, lines 162-169 perform "house keeping," checking there are enough reserves within the pair to even perform the swap (lines: 161, 162), that the destination address is valid (line: 169), and also initializes variables (lines: 164, 165, 167, 168).

Now, in lines 170 and 171, the tokens are in fact sent to the destination address *optimistically*. Recall for our example above,

$$token0_{out} = 0 \quad and \quad token1_{out} = ?, \tag{11}$$

as the "input" token is the independent variable, more on this below when discussing on-chain data and usage of the UniswapV2 Periphery code. Spoiler: $token0_{out}, token1_{out}$ are calculated for the user based on whatever type of swap they want, hence this step makes sense when considering the whole system, but admittedly is confusing at this stage. Note, for a simple swap, line 172 is not executed.

Now, with the token transfer completed, the contract queries each token for the new pair balance ${}^*r_0^{b+1} = balance0$, ${}^*r_1^{b+1} = balance1$. Please note, these balances include the 0.3% fee, hence the '$*$', but knowing the balances are directly related to deposits and withdrawals, we derive the implied formulas,

$${}^*r_0^{b+1} = r_0^b + token0_{in} - token0_{out} \quad and \quad {}^*r_1^{b+1} = r_1^b + token1_{in} - token1_{out}. \tag{12}$$

The new balances are then used in lines 176 and 177 to "back calculate" the tokens the user deposits to the pair, $token0_{in}, token1_{in}$.

Considering our example from above, equation 11 and 12, and lines 176 and 177, we can see the following:

$${}^*r_0^{b+1} \overset{?}{>} r_0^b - token0_{out} \tag{13}$$

$$r_0^b + token0_{in} \overset{?}{>} r_0^b - token0_{out} \tag{14}$$

$$r_0^b + token0_{in} \overset{?}{>} r_0^b - 0, \tag{15}$$

and knowing the user deposited a non-zero amount of token0, thus $token0_{in} > 0$, which forces the inequality to be true,

$$r_0^b + token0_{in} > r_0^b \equiv True, \tag{16}$$

thus finally, the token input is,

$$token0_{in} = {}^{*}r_0^{b+1} - (r_0^b - token0_{out}). \tag{17}$$

Moving along, line 178 checks to make sure at least some token0 or token1 is deposited. Now, because all unknown values have been satisfied, a simple check to ensure the values are consistent with the *constant product formula* is necessary. As mentioned earlier, this logical sequence is somewhat counter intuitive[3]. First in checking compliance, the fee component component must be removed in lines 181 and 182, giving

$$r_0^{b+1} = {}^{*}r_0^{b+1} * 1000 - token0_{in} * 3 \quad and \quad r_1^{b+1} = {}^{*}r_1^{b+1} * 1000 - token1_{in} * 3. \tag{18}$$

This fee removal is necessary because the fee accumulates with the reserves, but is not part of the actual constant product function calculation. The final step now requires checking an inequality for compliance, as shown in line 182:

$$r_0^{b+1} * r_1^{b+1} \overset{?}{\geq} r_0^b * r_1^b * 1000^2. \tag{19}$$

Examining equation 19 one can see the constant product formula, with the two pool balances on each side representing the initial state and final state[4]. Notice the inequality, not equality. Regardless, by substituting equation 18 into 19 we get,

$$({}^{*}r_0^{b+1} * 1000 - token0_{in} * 3) * ({}^{*}r_1^{b+1} * 1000 - token1_{in} * 3) \overset{?}{\geq} r_0^b * r_1^b * 1000^2. \tag{20}$$

Noting $token1_{in} = 0$ and expanding the terms results in,

$${}^{*}r_0^{b+1} * {}^{*}r_1^{b+1} * 1000^2 - {}^{*}r_1^{b+1} * 3000 * token0_{in} \overset{?}{\geq} r_0^b * r_1^b * 1000^2. \tag{21}$$

While equation 21 is a bit unwieldy, each term is known at this stage in the swap, hence if the proposed trade is valid, the trade will execute. If not, the final require statement on line 183 will fail and the entire state, including the optimistic transfer from above, will be rolled back.

Further manipulation of equation 21 using substitution and algebra can prove helpful to develop intuitions about swap dynamics. It is encouraged for the reader to experiment along these lines to gain more intuition.

In real-world swapping conditions, however, few traders actually use the pair contract directly. More likely, they will use the Periphery contracts mentioned above. Next, we will examine an actual swap on-chain which does just this.

UniswapV2 Swap On-Chain Example

UniswapV2 is divided into two clusters of smart contracts: Core and Periphery. For most users, the Periphery is the recommended interaction interface. Examining the Periphery, a smart contract refereed to as the Router (0x7a250d5630b4cf539739df2c5dacb4c659f2488d) is the point of entry at which user should start.

The Router has several functions that represent all possible combinations of swaps, each with the proper safe guards for modern smart contract interaction. For example, swapping some exact number of tokenA for another tokenB (swapExactTokensForTokens()), or getting some exact number of tokenB for tokenA (swapTokensForExactTokens()), along with combinations of ether and tokens as well.

To explore these functionalities, let's examine a real swap between WETH and LINK using the UniswapV2 Pair via the UniswapV2 Router02,
transaction: 0x135953b064429bd41403de10c0dcb39612455a0774bdaed371ae67e61254a0e3. Beginning our examination, we see this transaction is sent to the UniswapV2 Router02, calling function, swapExactTokensForTokens(), see Appendix B. Meaning, the swap intends to send in an *exact amount* of WETH tokens and receive the resulting amount of LINK tokens. The number of LINK tokens the transaction will generate is defined by the constant product function. To protect against manipulation

[3] In smart contracts, this is often done to save gas fees.

[4] Note, this notation is assuming state changes block-to-block for simplicity, however, in real swaps these states could change several times per block and would instead be before and after any given transaction within a single block.

of the pair, both a minimum amount of tokens out and a deadline are included. Simply put, if the executed trade results in less than the minimum number of tokens or is not executed by a certain time in history, the swap will fail to execute.

Examining this specific swap (tx hash: 0x1359..e3) we see the WETH deposited is 0.03 eth, with a minimum output of 6.0 LINK out with a time limit of about 10 minutes after initial submission. Diving into the code, we see on line 231 a helper function is called to build a array of price data. In this case, the helper function is solving equation 8 from above to determine the output of LINK given WETH. This is a array because it is possible to link multiple swaps together, however, only one is shown here. Notice now, we have both $token0_{out} = 0$ and $token1_{out}$. Which is exactly what the UniswapV2 pair is expecting as we learned above.

Next, line 232 checks the minimum condition of the LINK out of the swap. Next, on line 233, a subtle smart contract requirement occurs, in that WETH is transferred on behave of the user to the pair. This is needed because if it were placed on the contract in a separate transaction than the swap, it would be arbitraged away itself. This is the first transfer of ERC20 tokens in a possible series. But for us here, this will be only one swap. Next, another private function on the Router is called, _swap(), see Appendix B.

As expected, the first control logic in _swap() is a loop for each possible token swap, but is only one swap here, WETH-LINK. Going further, we see lines 214-218 are purely "book keeping" to order the tokens in the proper sequence for a swap with the pair and ensuring each destination address is properly setup, including the final address of the calling account (i.e. Externally Owned Account (EOA), or Smart Contract). Finally, lines 219-221 are the given token pair itself and it's respective .swap() function. This, is the same function as detailed above, with code snippet in Appendix A.

Checking the transaction details on Etherscan we can see each of these smart contracts clearly being interacted with, from the UniswapV2 Router02, the WETH ERC20, Uniswap WETH-LINK pair, then finally LINK ERC20. Examining the emitted event logs of the transaction make this clear.

To conclude, we can see the final transfer of LINK was a value of 6.93, which is clearly above the minimum requested value of 6.0[5].

3 Conclusion

This paper attempts to shorten the "learning curve" for new users to UniswapV2. While constant product formula based automated market makers (AMM) are simple in theory, following the logic on-chain and also finding the execution code can be quite daunting for the beginner. Here, these steps have been outlined for the reader in as clear and brief a manner as possible.

Acknowledgements

Thank you to Matías Andrade, Kyle Waters, Tanay Ved, Nate Maddrey, and Mudabbir Kaleem, all of Coin Metrics, for their helpful reviews and comments. Any remaining errors or omissions are my responsibility.

[5]Recall, smart contracts on the EVM use integer math. Further, ERC20's have a decimal place value, that for LINK is 18. Hence, the actual balance will appear as 6931371159474691587 ≅ 6.93 LINK.

Symbols

$b \in B$ - a specific block height to consider

B - the set of all block heights for Mainnet

B_{pair} - the set of all block heights for which a given pair has existed.

N - the set of Natural numbers.

pair - a specific UniswapV2 trading pair, with ERC20 tokens *token*0 and *token*1

r_0^b, r_1^b - reserve of token0, token1 at block height b for some *pair*

r_0^{b+1}, r_1^{b+1} - reserve of token0, token1 at block height $b+1$ for some *pair*, after a swap

${}^*r_0^{b+1}, {}^* r_1^{b+1}$ - the true token balance after a swap, it includes the accumulated fee of 0.3%

*token*0 - the ERC20 of a UniswapV2 pair with the lower numerical sort address

$token0_{in}$ - the quantity of token0 some user deposits into the UniswapV2 pair

$token0_{out}$ - the quantity of token0 some user receives from their UniswapV2 swap

*token*1 - the ERC20 of a UniswapV2 pair with the higher numerical sort address

$token1_{in}$ - the quantity of token1 some user deposits into the UniswapV2 pair

$token1_{out}$ - the quantity of token1 some user receives from their UniswapV2 swap

x, y, k - constant product formula variables

k^b - constant product value at block height b

Appendix A

Source code from UniswapV2Pair.sol, the *swap* function.

```
159     function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external lock {
160         require(amount0Out > 0 || amount1Out > 0, 'UniswapV2: INSUFFICIENT_OUTPUT_AMOUNT');
161         (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
162         require(amount0Out < _reserve0 && amount1Out < _reserve1, 'UniswapV2: INSUFFICIENT_LIQUIDITY');
163 
164         uint balance0;
165         uint balance1;
166         { // scope for _token{0,1}, avoids stack too deep errors
167         address _token0 = token0;
168         address _token1 = token1;
169         require(to != _token0 && to != _token1, 'UniswapV2: INVALID_TO');
170         if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); // optimistically transfer tokens
171         if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); // optimistically transfer tokens
172         if (data.length > 0) IUniswapV2Callee(to).uniswapV2Call(msg.sender, amount0Out, amount1Out, data);
173         balance0 = IERC20(_token0).balanceOf(address(this));
174         balance1 = IERC20(_token1).balanceOf(address(this));
175         }
176         uint amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0 - amount0Out) : 0;
177         uint amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1 - amount1Out) : 0;
178         require(amount0In > 0 || amount1In > 0, 'UniswapV2: INSUFFICIENT_INPUT_AMOUNT');
179         { // scope for reserve{0,1}Adjusted, avoids stack too deep errors
180         uint balance0Adjusted = balance0.mul(1000).sub(amount0In.mul(3));
181         uint balance1Adjusted = balance1.mul(1000).sub(amount1In.mul(3));
182         require(balance0Adjusted.mul(balance1Adjusted) >= uint(_reserve0).mul(_reserve1).mul(1000**2), 'UniswapV2: K');
183         }
184 
185         _update(balance0, balance1, _reserve0, _reserve1);
186         emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
187     }
```

Appendix B

Source code from UniswapV2Router02.sol, the *swapExactTokensForTokens()* function.

```
224     function swapExactTokensForTokens(
225         uint amountIn,
226         uint amountOutMin,
227         address[] calldata path,
228         address to,
229         uint deadline
230     ) external virtual override ensure(deadline) returns (uint[] memory amounts) {
231         amounts = UniswapV2Library.getAmountsOut(factory, amountIn, path);
232         require(amounts[amounts.length - 1] >= amountOutMin, 'UniswapV2Router: INSUFFICIENT_OUTPUT_AMOUNT');
233         TransferHelper.safeTransferFrom(
234             path[0], msg.sender, UniswapV2Library.pairFor(factory, path[0], path[1]), amounts[0]
235         );
236         _swap(amounts, path, to);
237     }
```


Source code from UniswapV2Router02.sol, the *_swap()* function.

```
212     function _swap(uint[] memory amounts, address[] memory path, address _to) internal virtual {
213         for (uint i; i < path.length - 1; i++) {
214             (address input, address output) = (path[i], path[i + 1]);
215             (address token0,) = UniswapV2Library.sortTokens(input, output);
216             uint amountOut = amounts[i + 1];
217             (uint amount0Out, uint amount1Out) = input == token0 ? (uint(0), amountOut) : (amountOut, uint(0));
218             address to = i < path.length - 2 ? UniswapV2Library.pairFor(factory, output, path[i + 2]) : _to;
219             IUniswapV2Pair(UniswapV2Library.pairFor(factory, input, output)).swap(
220                 amount0Out, amount1Out, to, new bytes(0)
221             );
222         }
223     }
```