Swap and Mint Troubleshooting Mock [UniswapV3]

Wayne Wignes

February 14, 2025

Introduction

The aim of this project is to help users learn the UniswapV3 protocol. Being the worlds largest decentralized exchange, Uniswap is of course a protocol worh learning. The protocol relies on four fundamental contracts:

- **NonfungiblePositionManager.sol** [**NFPM**]: used for minting and managing liquidity positions.
- **UniswapV3Factory.sol**: used for deploying pools
- **SwapRouter.sol**: used for swap transactions.
- **UniswapV3Pool.sol**: main contract: NFPM and SwapRouter call this contract in order to execute minting/ swapping.

So a user interacts with the pool contract indirectly through the SwapRouter and NFPM contracts. Hence, though these contracts are in the 'v3-periphery' rather than in 'v3-core', they are the primary contracts which investors/ developers ultimately interact with. There are dozens of peripheral contracts and libraries which the above contracts rely on, and this makes the UniswapV3 protocol reasonably complex.

Uniswap lists mainnets and testnets to which the above contracts have been deployed. So, aside from the value of learning the protocol in painstaking detail, what then is the value of mocking the protocol? This is more specifically a troubleshooting mock, the aim of which is to aid users in troubleshooting failed transactions.

Troubleshooting the mint(...) function

The two functions of interest are `swap(...)` and `mint(...)`. Compared to the actual protocol, an additional parameter called `revert_option` (or in some contracts/scripts just `option`) is accepted by these functions. This is an `int8` variable which triggers a simple revert message at selected points in the call when the matching value is input. For example the `_modifyPosition(...)` function is used when minting. At the beginning of this function is the revert statement,

```
        if (option == 21){ revert(" --- [_mod]  21 ");}
```

so if the user inputs `revert_option = 21` and the call reverts with the message " — [_mod] 21 " the user at least knows the call made it to the beginning of the `_modifyPosition(...)` function. Adding more similar revert statements allows users to hone in to a fairly specific degree to where they are having issues getting a transaction to go through.

Miniting new positions is accomplished by selecting option 0 in the liquidityManagement.py script. Lines 55-84 of the liquidityManagement.py script look like,

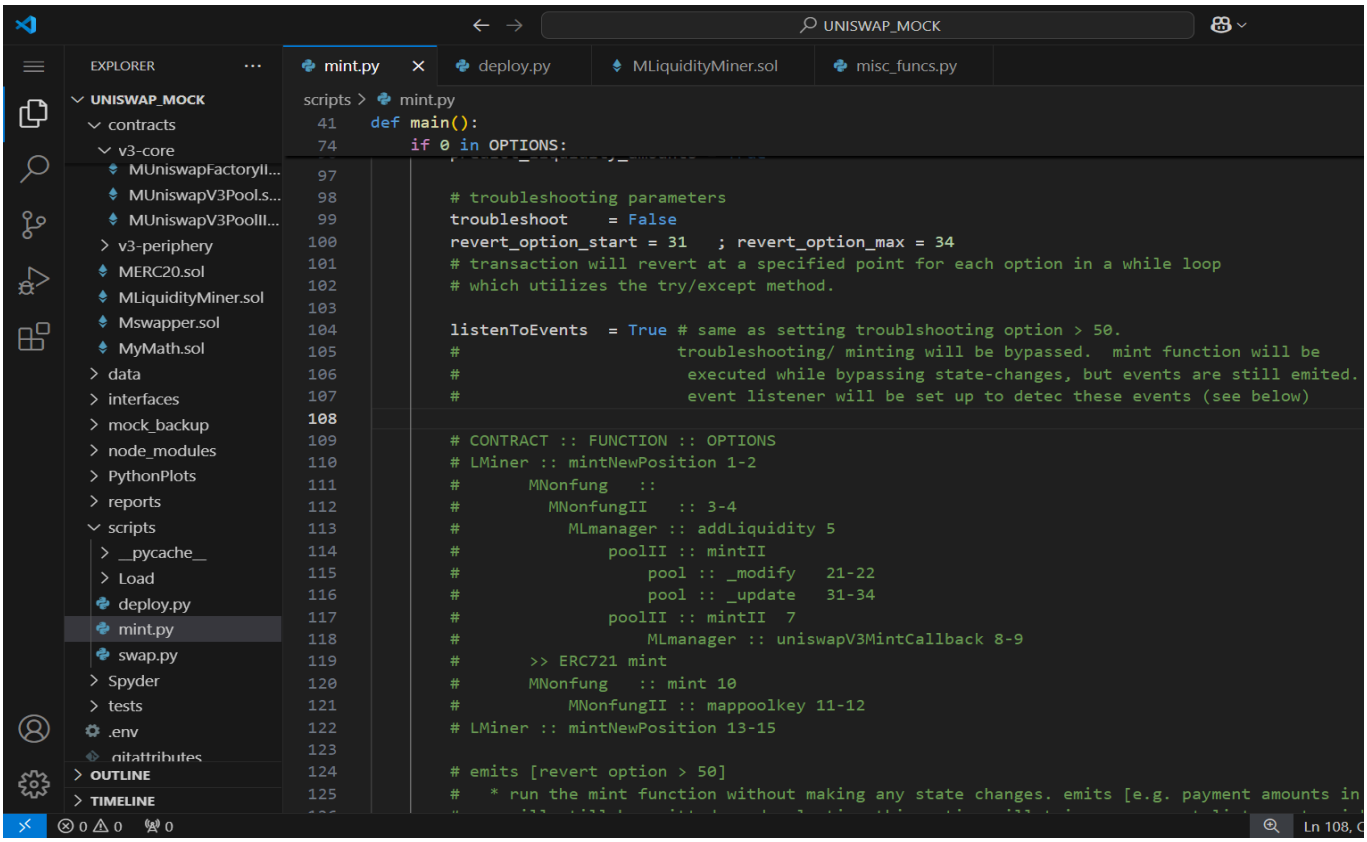

Figure 1: Troubleshooting [revert] options for the mint(...) function.

So, rather than a single `revert_option` input, a user is asked to input a range of values. The `try/except Exception as e` [python] method will then be used to cycle through the mint transaction for each `revert_option` in range. This allows the user to quickly hone in on where a mint call is reverting. As soon as the error message received does not match the message embedded in the revert statement of the contract the user knows the problem area has been reached.

Readers may wonder if there is a more efficient means of incorporating the troubleshooting functionality. To my knowledge, no – not when using eth-brownie/solidity anyway[1]. Furthermore, protocols such as

[1]There are potential exceptions to this if one understands the behind-the-scenes workings of node providers .

Uniswap are often not designed to maximize their user-friendliness with respect to troubleshooting. However utopian one may see the concept of decentralized finance, it is still finance, and Uniswaps aim is enable users to make $$. Maximizing security, efficiency, and transparency might understandably take precedence over educational design or its level of ease to developers who are new to the Uniswap protocol and just trying to learn to use it.

The input parameter `listenToEvents` will set `revert_option=51`. Any `revert_option > 50` will cause all state-changing lines of code to be bypassed in the solidity contracts. Meanwhile, events of interest are still emitted, and when `listenToEvents` is set to True an event listener will be triggered to detect these events. For example the `MintPayAmounts(payer, token0, token1, amount0, amount1)` event will still be emitted in the `uniswapV3MintCallback` function (see contracts/v3-periphery/base/MLiquidityManger.sol). This is useful because it gives users an empirical method to *see* rather than track and/ or calculate the variables going into the payment (or transfer) being executed in the mint callback function – variables which were extrapolated from packed data rather than being input directly no less (it can all be a bit much for a humble coder to keep track of).

A similar design is given to the `swap(...)` function and the swap.py script used to call it.

Adding in simple revert statements unfortunately has the effect of causing the already packed contracts to exceed the exceeds EIP-170 limit of 24577 bytes. This seemingly trivial issue was cause for significant hair-pulling for me, with the end result that there are not one but two factory contracts, two NFPM contracts, and two pool contracts As is the way coding goes, one thing led to another thing which led to another thing, etc. with the result that dual contracts were necessary in order to accommodate the troubleshooting functionality. I won't bother writing a block-diagram of the resulting design, partly because it was never my aim to optimize the design so much as it was to get *a* working mock up – one which, despite have design differences, functions more or less the same as the V3 protocol (while I make no claim the resulting design is the most efficient, I did go to reasonable lengths to ensure that V3 protocol was mocked realistically). Then there is the fact that the V4 protocol is anticipated to be released any day/month...year from now???

**Note : there are three options to choose from in liquidityManagement.py. These options are,*

1. : **mint a position**
2. : **add liquidity** [to an existing position]
3. : **price conversion** : use Chainlink price-feeds to convert amount A to amount B.

Deployment

The .py scripts presume eth-brownie is being used. While not as fast as industry standard tools like *Foundary [Rust]*, python is good for learning as well as for mocking purposes. Once eth-brownie has been installed users will want to create a node_modules folder *then* run,

```
npm add @uniswap/v3-periphery @uniswap/v3-core
```

deploy.py script automates the deployment process so that users only need to select from the options

```
    # 0 ERC20 Tokens
    # 1 V3 Periphery Libraries
    # 2 V3 Core Contracts
    # 3 Swapper contract
    # 4 Liquidity Miner contract
    deployments = [4]
```

Deployments are meant to occur in numerical order. Selecting option 0 will deploy a set of mock Tokens as well as mint 100 tokens to the user. The Token symbol will be whatever the user enters in `token_list = [tokenA_name, tokenB_name]` (see deploy.py :: option 0). Just make sure this same symbol is used in liquidityManagement.py for loading the pool, and also update the config file under the deployment network as,

```
    tokenA_name = ""
    tokenB_name = ""
```

where blank quotations are where the deployment address will automatically be filled in after deployment. The function `UpdateConfigAdresses(...)` will follow every deployment. This function automatically updates the brownie-config.yaml file with the address of the contract just deployed. If users have their own token they wish to use, skip option 0 in deploy.py and just A) enter the name of the token in liquidityManagement.py at the very beginning of the script just before the pool is loaded, and B) enter token name and address in the config file under the active network.

Running deploy.py with option 1 set will then deploy the V3-libraries. However, the user needs to ensure that all conditional variables under option 1 are set to 'True'. For example, the deployment of the libraries [option 1] looks like,

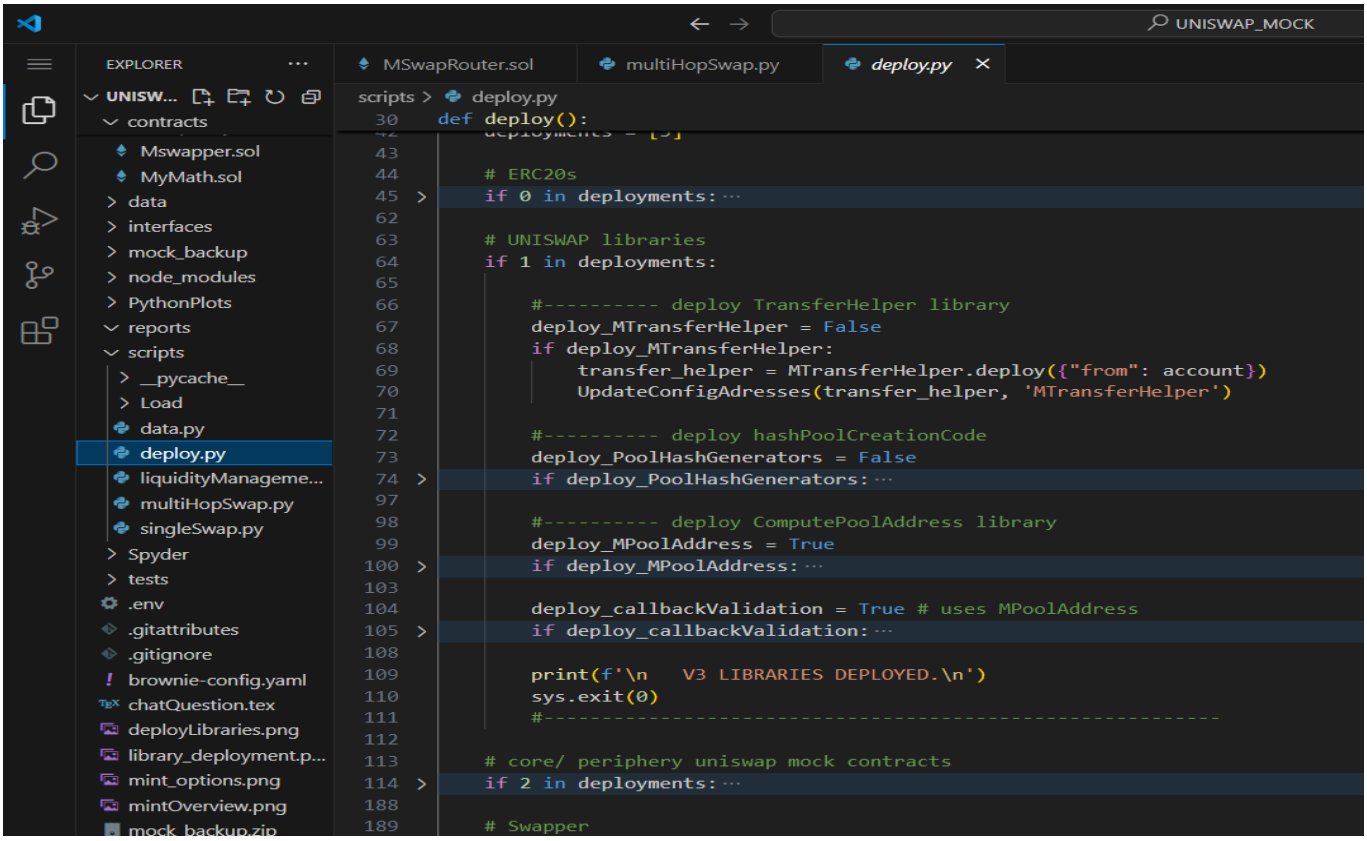

Figure 2: deploy.py :: option 1 [deploy v3 libraries]

In addition to 1 being in the OPTIONS list it is required that `deploy_MTransferHelper=True` in order to deploy MTransferHelper.sol. As can be inferred from the fact that 1¡2 the v3 libraries [option 1] need to be deployed prior to deploying the v3-core contracts. Furthermore, assume the order in which contracts/ libraries are deployed within the same option are meaningful. For example, if the deployment of MTransferHelper occurs before other libraries, it can be assumed that this is because subsequent libraries/ contracts make use of this library in some fashion. Hence the need for added 'conditional variables' for each deployment; because sometimes we may only want to deploy one single contract/ library out of a number which fall under deployment option 1 (e.g. when we changed something in one contract which was subsequent to others and we only wish to re-deploy the contract which was altered).

The most significant complication [more like an inconvenience] to the deployment process comes when deploying the libraries. Noting lines 72-101 of deploy.py [see above], the poolHashGenerator contracts serve the purpose of generating and then hashing the creation code of the pool contracts (MUniswapPool.sol and MUniswapPoolII.sol). Notice that these creation code hashes (`POOL_INIT_CODE_HASH` and `POOL_INIT_CODE_HASHII`) are printed along with a detailed message telling the user they need to copy and past BOTH of these hash values into the MPoolAddress.sol library. The code then performs a system exit. *MAKE SURE THESE HASHES ARE SAVED PRIOR TO CONTINUING WITH DEPLOYMENT OF MPoolAddress.sol.* This is essential because these two hashes are immutable, meaning they cannot be

changed after MPoolAddress.sol library has been deployed. A number of contracts will make use of MPoolAddress.sol to compute the contract addresses of the pools, and the immutable values of `POOL_INIT_CODE_HASH` and `POOL_INIT_CODE_HASHII` which have been stored in MPoolAddress.sol library will be essential to this calculation[2]. After these hash values have been saved, re-run deploy.py, still with option 1 [libraries] selected, only this time a `deploy_MTransferHelper` and `deploy_ PoolHashGenerators` should both be set to False as they've already been deployed. The deployment of MPoolAddress.sol will ensue along with the remaining libraries.

After the v3-libraries have been deployed the system will exit and the user will need to remove the number 1 in the options list and include the number 2 if its not already there). Now re-run deploy.py to proceed with the deployment of the v3-core and periphery contracts. The system will exit again after this. Now – while the deployments are still visible on the users command line and before moving onto subsequent deployments which might make use of contracts/ libraries which have just been deployed – is a good time to ensure the address which the config file has been updated with matches the actual address of deployment visible on the command line[3]

The remaining contracts to deploy are example contracts from uniswap. They are not designed with security in mind and should be modified if used on a live network.

So, in short, to deploy the uniswap troubleshooting mock all the user need to do is cycle through options 0-4, being sure to save the init [creation] code hashes of the hash generator libraries into MPoolAddress.sol prior to deploying this library.

**Note: the 'M' prefix to contract names stands for 'Mock'.*

Managing Data With data.py

data.py is a script used for managing data and creating visual displays of ticks and/ or liquidity positions. The available options to choose from in this script are,

1. **update tick infor data file**: update tick data for a given pool (updates saved to data/poolTickData folder)
2. : **update liquidity position info**: two functions are ran:

   `updateNFPMPositionsFile(...)`: get position info from NFPM contract (updates saved to data/NFPM_positions folder)

   `updatePoolPositionsFile(...)`: get position info from pool contract (updates saved to data/pool_positions

[2] A quirk of the uniswap protocol is that the pool contract is deployed by running a function in a solidity contract which has already been deployed [the factory contract] rather than by directly deploying the pool contract itself. Even if a user happens to know the address of a given pool which has already been deployed, still the protocol will at a number of points utilize the MPoolAddress.sol library library to calculate the address of the pool. Upon consideration, this is a subtle yet significant security feature because the hash value of the pool contracts creation code (the value of`POOL_INIT_CODE_HASH`) is unique, and this provides a means for interested users to verify that the contract they are interacting with on the blockchain is in fact the contract they are looking at.

[3] by now I've made use of `UpdateConfigAdresses(...)` function to update my config file with deployment addresses hundreds of times. Still, it may save the user valuable time/ energy knowing the address in their config file is at least correct.

folder) so we are updating the same position in two different ways. Overkill? Maybe. But the UniswapV3Pool contract does track the position in two different wants – through the pool contract and through the NonfungiblePositionManger (NFPM) contract. If we are going to mock this, then it is of use to be able to see that the two positions are in alignment with one another.

3. : **plot pool tick info**: plot various selected parameters for tick data in a pool. Uses tickBitmap.sol logic to efficiently identify initialized ticks so that we don't have to cycle through ALL ticks.

4. : **plot NFPM position**: plot NFPM tracked liquidity position

5. : **PLOT NFPM position + ticks**: plot NFPM tracked liquidity position AND pool tick data [most useful] OPTIONS = [0]

The functions which do the bulk of the work for the above options in data.py are contained in the scripts/Load/manageLiquidityFuncs.py file.

the initial lines which account for loading the pool aside, by using the left-hand arrows for contracting/-expanding sections of code the entire data.py script can be condensed to,

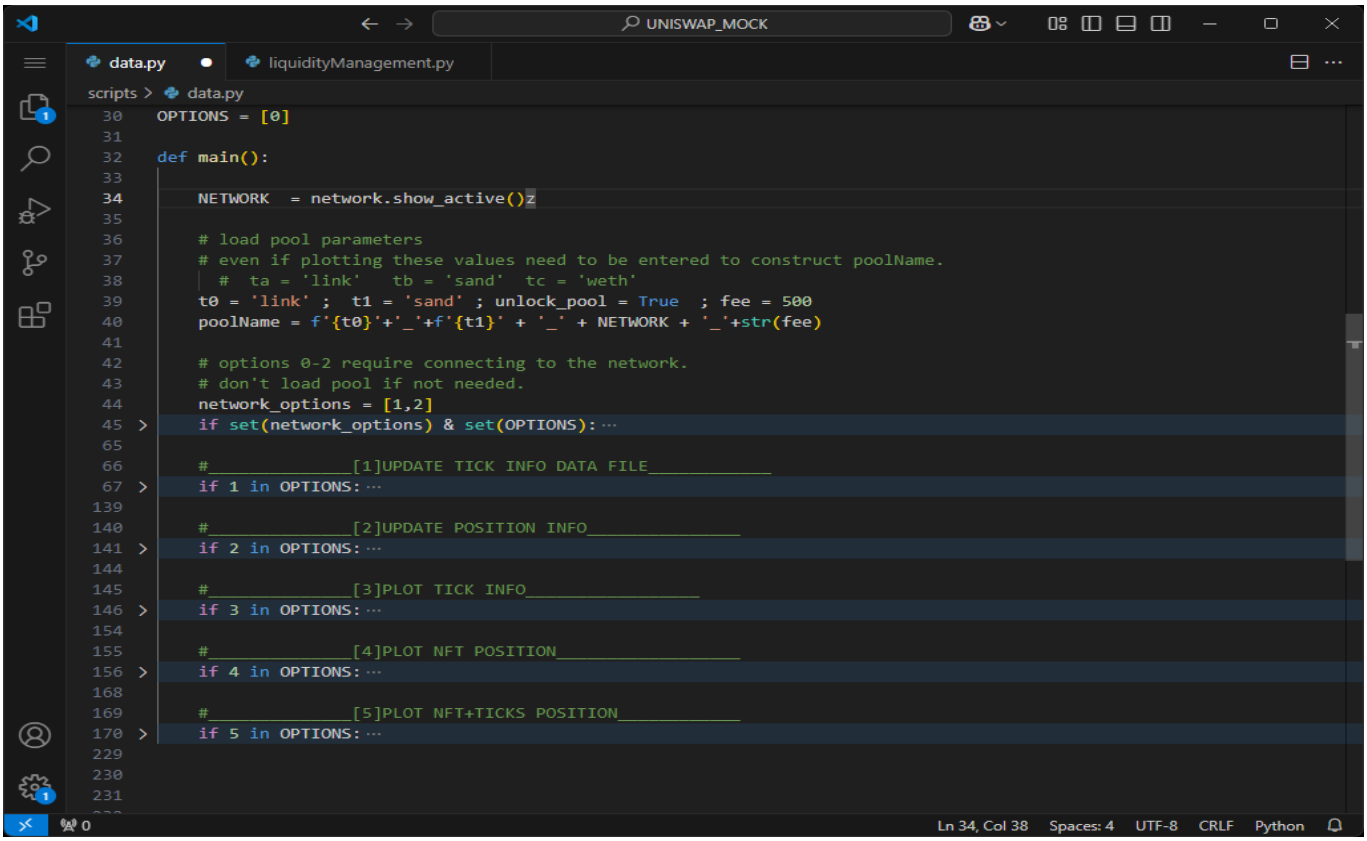

Figure 3: options to choose from in data.py.

A more in-depth overview of each option will now be given.

[1]update pool tick info :

The maximum/ minimum values for ticks in the pool are +/- 887272, respectively. This would give 1,774,544 ticks to a pool As far as upper and lower bounds for liquidity positions goes however, there are less. This is because `flipTick(...)` function (used whenever liquidity is added) of the TickBitMap.sol library only allows ticks which are evenly divisible by the tick spacing of the pool. Specifically it is the require statement in this function which creates the restriction,

```
    require(tick \% tickSpacing == 0)
```

where % represents modulo division. Hence, while `slot0.tick` can take on whatever integer tick value it pleases within the range [-887272,887272] ticks which can be used for upper/ lower bounds of liquidity positions are more constrained. the function `find_modulo_zero_tick(...)` in the scripts/Load/misc_funcs.py file automates finding such ticks.

The reason for such a large spread in possible tick values is so that `slot0.tick` (which sets the pool price) can vary over a wide range of prices. The price corresponding to a tick is given by the formula,

$$price = 1.0001^{tick}$$

The number of ticks which can be used for creating liquidity positions is `int(1,774,544/tick_spacing)`. Further divide this number by 256 and you have the number of possible 'words' which can be mapped via the tickBitmap mapping in the pool contract,

```
    mapping(int16 => uint256) public override tickBitmap;
```

the `int16` variable represents the word position – a number with magnitude ¡ 2^{15} (it may be positive or negative because it is declared `int16` instead of `uint16`). The `uint256` value represents the word value which will be 256 bits long – all zeros or ones. The ones will represent ticks which have been initialized, and this can only occur when using a tick as a liquidity position boundary, i.e. through minting.

so we 'just' need to ...

(A) determine the possible word position values:

$$|wordPosition| = int\left(\frac{1,774,544}{tick\_spacing} * \frac{1}{256}\right) \quad \longrightarrow \quad wordPosition = ...-10,-9,-8,...,8,9,10,...$$

(B) call wordValue = pool.tickBitmap(wordPosition) for each possible wordPostion

(C) examine the returned word value – a 256 bit binary number of zeros and ones where ones represent the initialized ticks – to determine the bit position of initialized ticks within the word.

(D) calculate the value of the initialize tick(s):

$$initialized\_tick = (word\_position * 256 + bit\_position) * tick\_spacing$$

(E) query pool.ticks)initialized_tick

From here it is all python magic – store the queried tick info into a an array, repeat for all initialized ticks, then save it to a .csv file which can later be used for plotting purposes (see options 3 and 5).

[2]update NFPM position info :

This option queries a list of users tokenIds and stores all associated parameters (see Position.sol :: Info struct) into a .csv file. Options 4 and 5 will make use of this .csv file to create a visual of these positions.

[3] plot pool tick info :

pull the .csv file created in option 1 and plot it. Users can adjust the input parameters,

```
keyList    = {
    'liquidityGross': ['blue', 0.5],
    'liquidityNet': ['green', 0.5]
}
tick_margin = 10000
bar_width    = 1000
```

`keyList`: first input is parameter to plot (see Tick.sol library: Info struct for all available parameters.). The next input is a list;[bar color, opacity]

`tick_margin`: add a margin to the left and right of minimum/ maximum tick.

`bar_width`: adjust tick bar width.

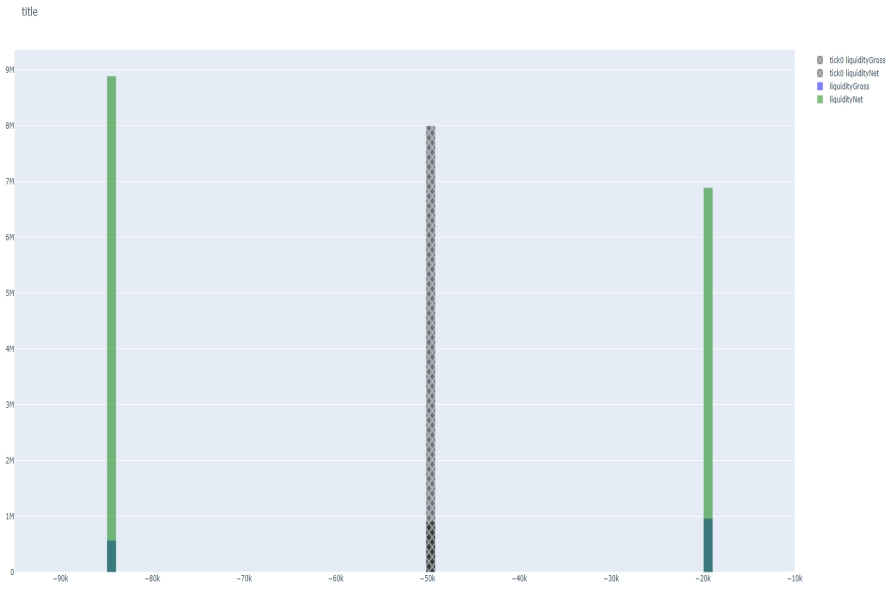

Figure 4: plot pool ticks

**Note 1: that values are scaled, but hovering over the bar [tick] of interest will show the real data. **Note 2: tick0 will be show in black with a specialized pattern.*

[4] plot NFPM position :

much the same as option 3, only the parameters to choose from vary – see contracts/v3-periphery/interfaces/IMNonfungiblePositio

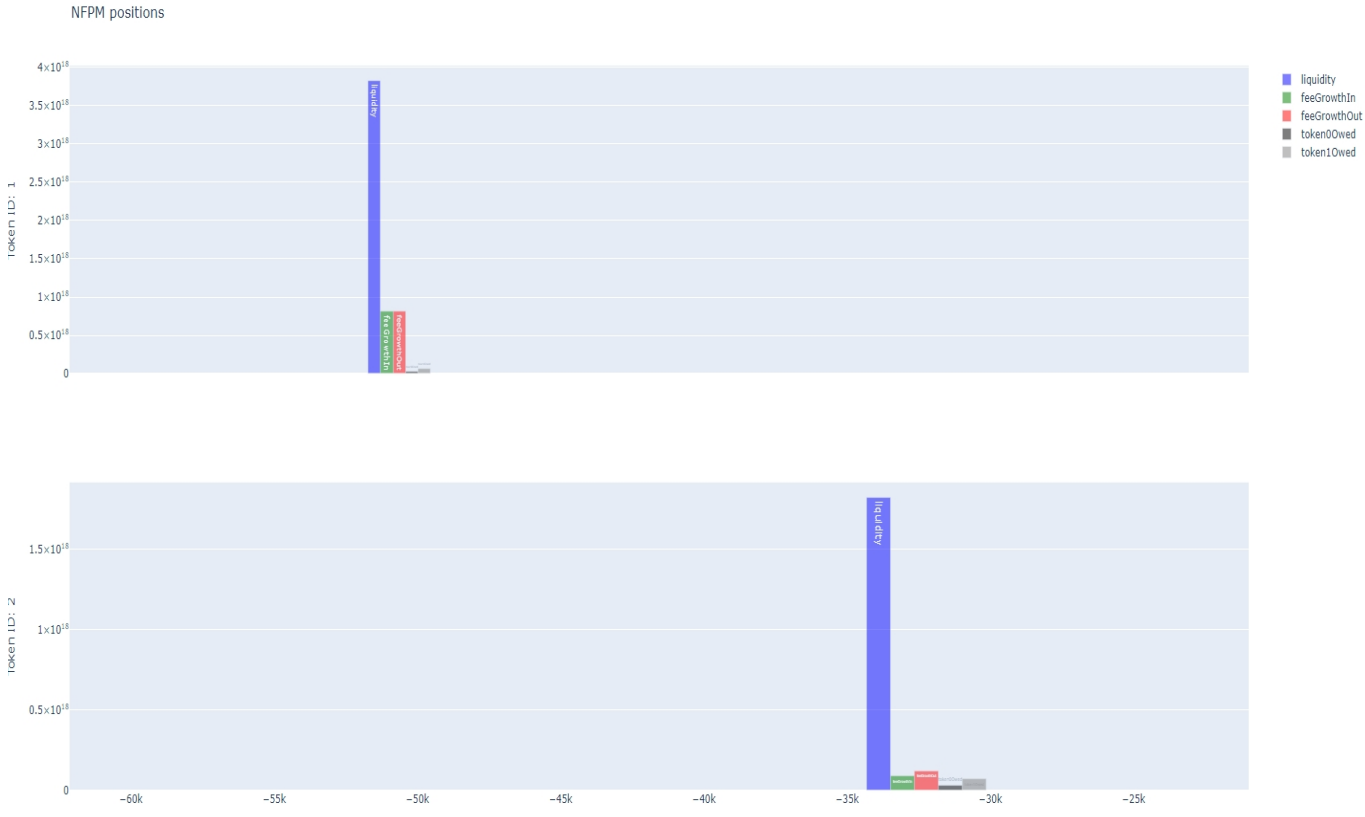

Figure 5: plot NFPM position.

**Note that values are scaled, but hovering over the bar [tick] of interest will show the real data.*

[5] plot NFPM position + ticks :

A combination of options 3 and 4: if the tokenId list contains two tokens then three subplots will be created. The bottom-most subplot will be a plot of pool ticks and their selected associated values. Subplots above this will be for liquidity positions. Subplots will have a shared axis with the tick plot so this gives a visual of how held positions relate to the pool.

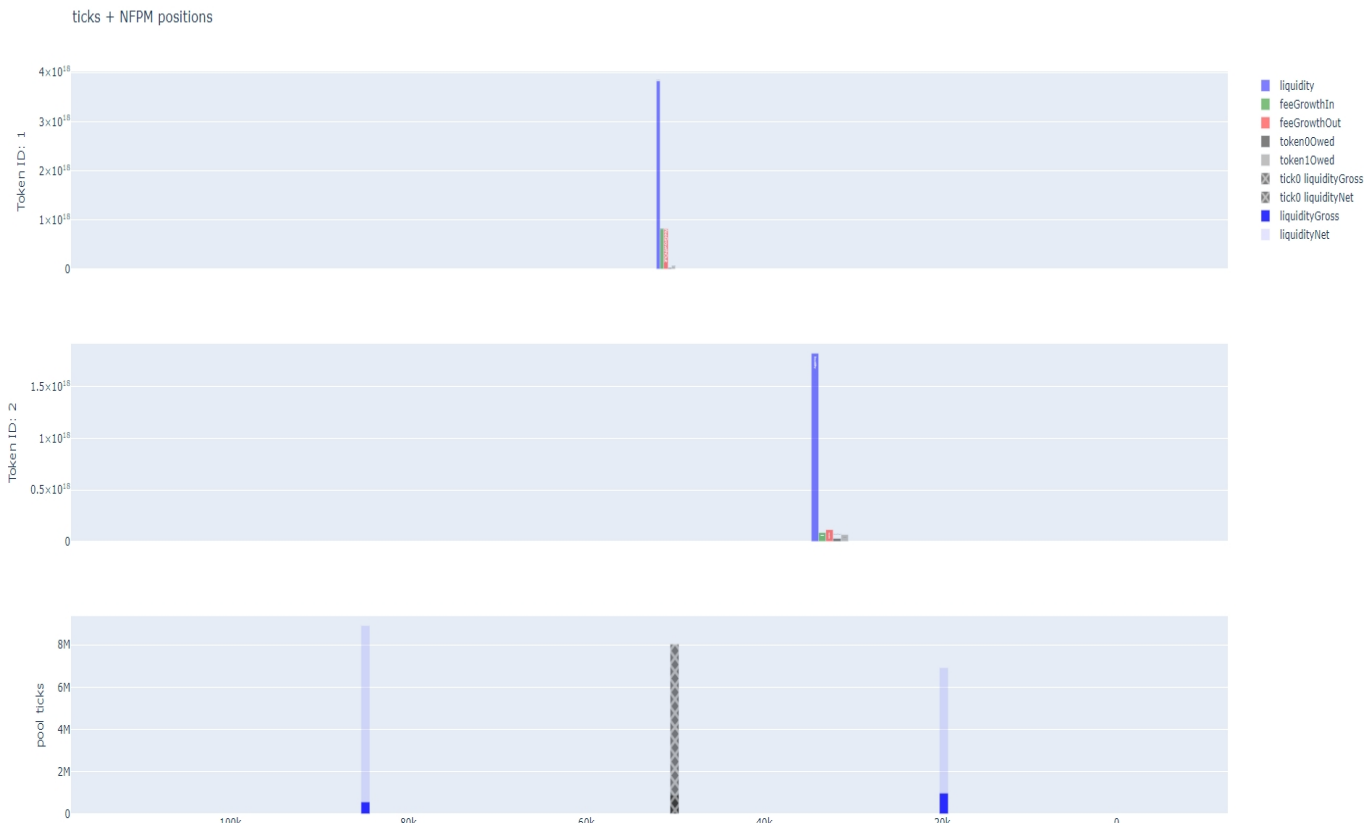

Figure 6: plot pool ticks + NFPM position

**Note 1: NFPM position bars spread out evenly over the tick values which the position spans. *Note 2: the values are scaled, but hovering over the bar [tick] of interest will show the real data.*

Swaps

triggering reverts:

There are two scripts for swapping; multiHopSwap.py and singleSwap.py. multiHopSwap.py interacts with the MSwapper.sol contract which in turn executes the `exactInput`/ `exactOutput` functions of the router. singleSwap.py interacts with the router directly by calling `exactOutputSingle`/ `exactInputSingle`.

A very similar troubleshooting functionality as is found in liquidityManagement.py is found in either of these scripts. Here however, a more advanced method is needed than revert statements that are conditional on an integer input. Because multihop swapping involves consecutive swaps in two different pools, it is naturally of interest to be able to pinpoint in which pool the error occurred. This is accomplished by inserting the function `revertOption(...)` into the router and pool contracts[4]. `revertOption(...)` is conditional on an

[4]this came at the expense of `increaseObservationCardinality` of the pool contract – attempting to include it will bypass

integer input, but also on a boolian input which is presumed true for triggering reverts in the first pool and false for those in the second.

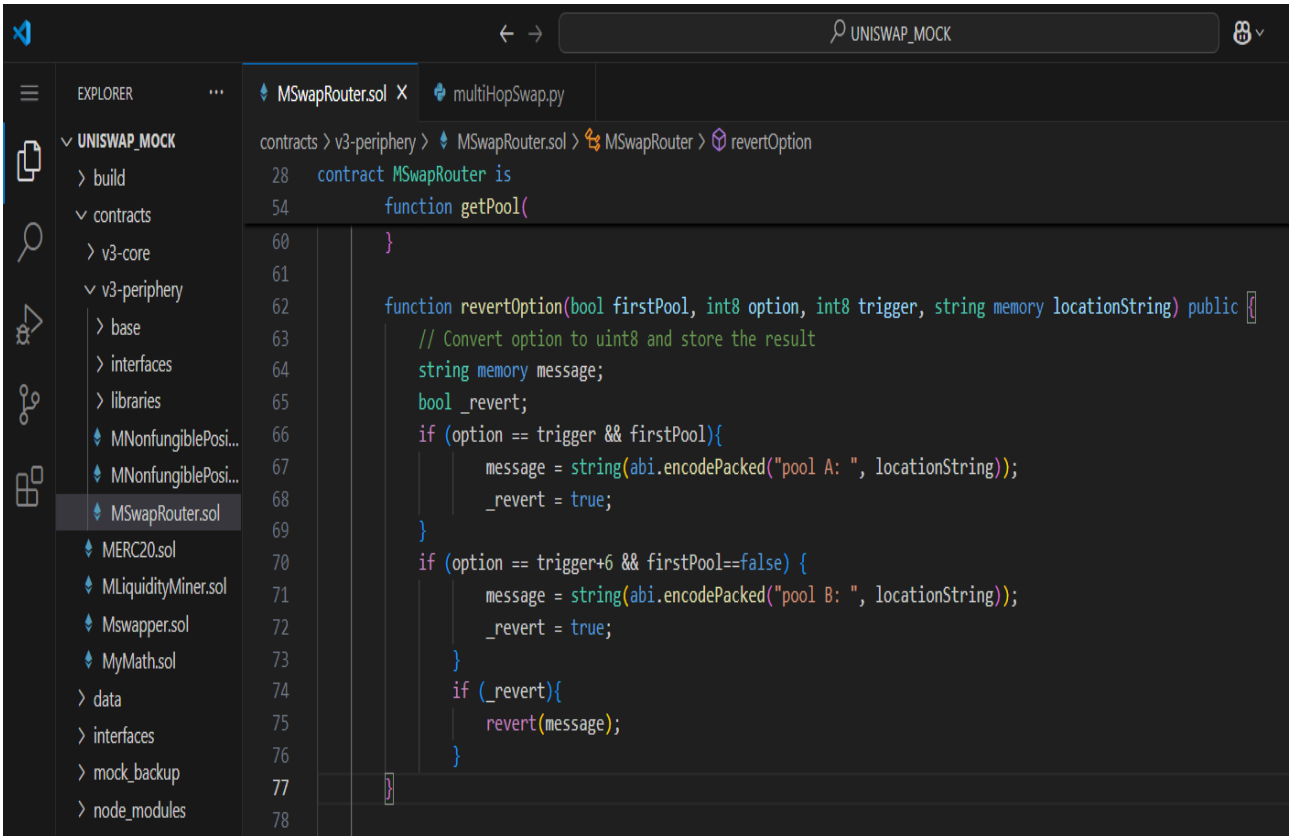

Figure 7: `revertOption(...)` defined in the router and pool contracts allows users to trigger a revert with a message that not only gives the location of where the call has made it to, but will also indicate which of the two pools involved in a multihop swap the function was triggered. Note that `option == trigger +6 && firstPool==false` is used to detect the second pool.

There are a total of 14 conditional revert statements/ functions from the start of a multiHop swap [two in MSwapper.py + 12 between the router and pool]. Again, the trigger value which the `option` parameter must match in order to trigger a revert increases along with the sequence of the call. The first two trigger statements are in MSwapper.sol and they just serve as an elementary check that transfer/ approval has been accomplished before moving into the router contract which will call By the time the call `pool::swap(...)` .

Option 3 is then in `exactInputInternal/ exactOutputInternal` of the router. before any swap occurs the call will go through these functions. By the time the call reaches the `pool::swap(...)` function the integer trigger value is 4. the highest integer trigger value in the path of a swap call is 8, and this trigger is placed in the callback function of the router (`uniswapV3SwapCallback`). Hence, there are a total of $8-3=5$ integer triggers which the call goes through whenever a swap occurs.

the byte limit.

When multiHop swapping inputting an option value of 3-8 will serve to troubleshoot the first swap. Once this call is done it then becomes tricky – how to trigger a revert in the same exact contract [deployed to a different address] the second time through but not the first?

The workaround was two-fold; A) detect a boolean value which is true for the first pool, but which the router sets to false before executing the second swap[5] and B) increase the value the trigger by 6. Together, these two conditions allow us to trigger reverts in the first swap by inputing an option value of 3-8, but option values 9-14 will not trigger a revert until the second swap is being executed. For convenience [and sanity] 'poolA' or 'poolB' is appended to the location string of the revert.

**Note 1: However, the location string is going to be the same for both pools Ergo, if option 10 causes the revert statement `"pool B: swap--4"` then just remember that 'poolB' tells us we are in the second pool, and if we subtract 10-6=4 we get the value for which this call would have reverted had poolB instead been poolA.*

event listening in swap transactions:

As far as event listening goes, a user is asked to input true or false for the variable `listenForSpecificEvents`. A true value will initialize the event listener. Just below this is the variable `multiSwapOption/ singleSwapOption` which allows a user to select a single troubleshooting/ event listening option. Choosing values ¿= 20 will cause state-changing lines of code to be bypassed, thus allowing a user to see parameter values of interest at various locations in the call without affecting the state of the router or pool contracts.

Conveniently, rather than contract or token addresses being printed off, the event listener is rigged to instead print contract names/ token symbols making it easy to get a feel for which contract is being asked to pay which other contract, in what token, and how much.

Option 20 will detect parameters used in the payment executed in the callback of the router. Options 21-23 will emit events pertinent to the calculations of `amount0`/`amount1` in the while loop in `pool::swap(...)` – just be aware there may be a large number of emitted events depending on how many iterations the loop goes through. `listenForRepeats = False` is a parameter intended to negate this behaviour in eth-brownies `eventWatcher` function, but at the moment it does not have the intended effect.

If for whatever reason a user wants to input a value lower than 20 for `multiSwapOption/ singleSwapOption`, a value of 0 [or 15-19] would be equivalent to attempting the full multiHop swap transaction without listening to events. options 1-14 would be equivalent to triggering a single revert option.

**Note 2: option 7 may unexpectedly cause a full unimpeded transaction to go through. This is because it is placed in the while loop of the swap function, and this trigger being reached is conditional on whether the next tick is initialized. If desired, users could either move this trigger (the `revertOption(...)` function) before the `if (step.initialized){...` statement before deployments, or just be careful to avoid using this option.*

A suggested list of [contract/ event_name] values have been filled in under the if statement `if listenForSpecificEvents:`. This is where the dictionary `eventOptions` is defined. It links the selected option to a list of contract/ event pairs to be 'watched' for/ listened to.

[5] the call returns to the router after the first swap.