SIAG/FME Code Quest 2023

DeFi & RoboAdvising Challenge

Sponsored by World Business Chicago and SIAM

Organized by SIAM Activity Group on Mathematical Finance and Engineering (SIAG/FME)

The SIAG/FME Code Quest is a programming challenge dedicated to solving mathematical and finance modeling problems using computational tools. Its primary objective is to inspire and harness the problem-solving capabilities of students around the world. The competition invites individuals and teams to collaborate and devise innovative solutions that not only tackle real-world challenges but also advance the frontiers of financial engineering. By promoting teamwork, creativity, and practicality, the quest aims to create a vibrant community where diverse perspectives converge to engineer efficient and inventive solutions to interesting financial mathematics motivated problems.

The Quest, as a standalone event, originated from the SIAM Financial Mathematics & Engineering 2021 Student Programming Competition, as part of the SIAM FM21 conference. The 2023 Quest combines two FinTech themes: Decentralized Finance (DeFi) and Automated Asset Management (RoboAdvising). This document describes the how the prototypical market operates and the instructions for the competition.

1 The competition

The two main objectives of the *DeFi & RoboAdvising 2023* challenge organized by **SIAG/FME** are:

i. Grasp and code the essential operations of constant product market makers in Python to accurately reproduce the processes of liquidity taking and provision.

ii. Implement a liquidity provision strategy across multiple pools that minimizes predefined risk measures and attain a certain investment goal.

1.1 Introduction

Traditional electronic financial markets are organized around Limit Order Books (LOBs) and Over-The-Counter (OTC) markets to exchange securities. LOBs allow traders to submit different types of orders that indicate the price, the volume, and the intention to buy or sell a security. LOBs rely on financial institutions and trusted third-parties that facilitate trading by collecting orders and matching buyers and sellers. OTC markets are based on a network of financial institutions and market makers that allow clients to buy and sell securities.

In contrast to traditional electronic exchanges, DeFi is a collection of blockchain-based financial services that do not rely on intermediaries such as brokers and banks. The blockchain is a peer-to-peer network, where users can develop and share immutable programs that are referred to as *smart contracts*. DeFi protocols are smart contracts that encode the rules that govern how users interact without relying on intermediaries such as brokers or banks. Remarkably, these services are permissionless, so anyone can participate in the market.

Within DeFi, automated market makers (AMMs) is a DeFi protocol that is based on two distinct coins (like Bitcoin and Etherium). They allow traders to execute trades (swap one coin for another), provide liquidity (add both coins to the 'pool'), or remove liquidity (take both coins from the 'pool'). Currently, most AMMs are what are known as constant product market makers (CPMMs), with Uniswap v3 being a prime example where billions of dollars are exchanged weekly. In CPMMs, **liquidity providers (LPs)** deposit (`mint`) coins in a liquidity pool, and **liquidity takers (LTs)** exchange coins (`swap`) directly with the pool. When LPs `mint` coins, they receive `LP coins` from the pool that represent the portion of the pool that they hold. LPs that hold coins in the pool can `burn` their `LP coins` to withdraw their liquidity.

Consider a pool for a pair of coins X and Y. The pool has a certain number of each coin, called the *reserves*, which are the total quantity of coins that are currently in the pool – and we denote these reserves by R^X and R^Y, respectively.

For example, consider the coin pair USDC (`coin-`X) and ETH (`coin-`Y). The coin ETH represents the cryptocurrency *Ether*, which is the native cryptocurrency of the Ethereum blockchain, and the coin USDC represents *USD coin*, a cryptocurrency that tracks parity with the U.S. Dollar and is fully backed by real assets. Consider a pool for this pair where LPs have minted $R^X = 2\times10^6$ USDC and $R^Y = 10^3$ ETH. These reserves imply a marginal price of $Z = 2000$ USDC/ETH in the pool (i.e., if a trader swaps a small amount x of ETH, they will receive approximately $2000\,x$ of USDC); see Section 1.2. LTs can swap USDC for ETH (buy ETH) or swap ETH for USDC (sell ETH) directly with the pool, and LPs can mint new coins or burn existing reserves; see Figure 1. More details on how these various events affect the state of the pool is described in Section 1.2 below.

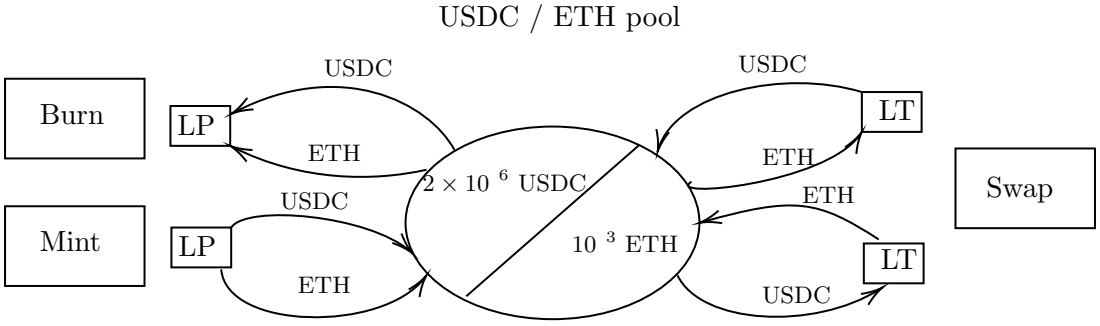

Figure 1: A pool for coins X and Y interacts with LTs that swap one coin for the other, and LPs that mint or burn liquidity in the pool.

1.2 Constant product market making

CPMMs rely on a set of rules to determine how LTs and LPs interact with the pool, and we refer to these rules as the *LT trading condition* and *the LP trading condition*. For LTs, the LT trading condition and the available reserves (R^X, R^Y) in the pool determine how coins are swapped. For

LPs, the LP trading condition determines the quantity of coins X and Y that they deposit when they mint liquidity or the quantity they withdraw when they burn it.

LT trading condition LT transactions involve swapping a quantity y of `coin-`Y for a quantity x of `coin-`X, and vice-versa. The LT trading condition links the state of the pool before and after a *swap* is executed. The condition requires that the product of the reserves in both coins remains constant before and after the transaction.

In practice, CPMMs charge a proportional fee ϕ to LTs. More precisely, when the swap of an LT is for a quantity x in the pool, then the swap is executed for a quantity $x\ (1-\phi)$. Similarly, when the swap of an LT is for a quantity y in the pool, the swap is executed for a quantity $y\ (1-\phi)$.

Below, we describe this more precisely.

`[swap X to Y]` ▷ When a trader deposits x of `coin-`X they receive y of `coin-`Y , (we call this action `swap X to Y`). The amount y of `coin-`Y that they receive is determined s.t.

$$R^X\, R^Y = \left(R^X + (1-\phi)\ x\right)\ \left(R^Y - y\right) \qquad \Longrightarrow \qquad y = x\,\frac{(1-\phi)\ R^Y}{R^X + (1-\phi)\ x}\ . \tag{1}$$

Even though the trade is executed for a proportion $(1-\phi)x$ of the quantity sent to the pool, the all x of `coin-`X are still put in the pool; see Figure 3. Therefore, after the swap event the reserves in the pool are updated according to

$$R^X \longrightarrow R^X + x \qquad \text{and} \qquad R^Y \longrightarrow R^Y - y \tag{2}$$

`[swap Y to X]` ▷ When a trader deposits y of `coin-`Y they receive x of `coin-`Y , (we call this action `swap Y to X`). The amount x of `coin-`X that they receive is determined s.t.

$$R^X\, R^Y = \left(R^X - x\right)\ \left(R^Y + (1-\phi)\ y\right) \qquad \Longrightarrow \qquad x = y\,\frac{(1-\phi)\ R^X}{R^Y + (1-\phi)\ y}\ . \tag{3}$$

Similar to the previous case, the reserves of the pool are updated according to

$$R^X \longrightarrow R^X - x \qquad \text{and} \qquad R^Y \longrightarrow R^Y + y\ . \tag{4}$$

It is worthwhile investigating the above relationships when the amount being swapped is very small. To this end, note that for an infinitesimal quantity $y \ll 1$ of `coin-`Y swapped, the amount x of `coin-`X the trader receives is (from (3))

$$x = y\frac{R^X}{R^Y + y} = y\frac{R^X}{R^Y}\left(1 + \frac{y}{R^Y}\right)^{-1} = y\frac{R^X}{R^Y}\left(1 - \tfrac{y}{R^Y} + o(\tfrac{y}{R^Y})\right)$$

where the last equality follows from the Taylor expansion $(1+a)^{-1} = 1 - a + o(a)$. Therefore, the ratio of x to y has limit

$$\lim_{y\to 0}\frac{x}{y} = \frac{R^X}{R^Y}\ .$$

The ratio $\frac{R^X}{R^Y}$ is referred to as *marginal price* of the pool, which is akin to the midprice in traditional markets. The marginal price is a reference price – the difference between its value and the actual ratio $\frac{x}{y}$ is called the *execution cost*.

LP trading condition LPs interact with the pool by submitting both coins into the pool (called `minting liquidity`) and receiving liquidity providing coins (`LP coins`). The amount of these `LP coins` they receive reflects the percentage of liquidity they have provided to the pool – the precise definition is below. LPs may subsequently return `LP coins` and receive a certain amount of `coin-`X and `coin-`Y. This act is called `burning liquidity`, and the precise amount they receive is also described below.

In either LP event `mint` or `burn` , the LP trading condition requires that LP operations do not impact the marginal price $\frac{R^X}{R^Y}$.

`[mint]`▷ More precisely, when the LP wishes to `mint LP coins`, the quantities of x and y they submit must respect the equality of marginal price before and after the `mint` event, so that x and y must satisfy the relationship:

$$Z = \frac{R^X}{R^Y} = \frac{R^X + x}{R^Y + y} \qquad \Rightarrow \qquad \boxed{\frac{x}{y} = \frac{R^X}{R^Y}}\,. \tag{5}$$

Upon submitting the correct amount of coins, the LP trader will then receive `LP coins` in the amount equal to:

$$\boxed{\ell = \frac{x}{R^X}\,L = \frac{y}{R^Y}\,L}$$

where L is the outstanding amount of `LP coins` issued by the pool prior to trader's `LP mint` event. Further, after the `LP mint` event the reserves and outstanding `LP coins` are updated as follows

$$\boxed{R^X \longrightarrow R^X + x, \quad R^Y \longrightarrow R^Y + y, \quad \text{and} \quad L \longrightarrow L + \ell}\,.$$

`[burn]`▷ Similarly, when the trader wishes to `burn LP coins`, the quantities of x of `coin-`X and y of `coin-`Y they receive respects the equality of marginal price before and after the `burn` event. As well, they receive an amount equal to proportion of `LP coins` they hold relative to the total outstanding coins. In particular, they receive the following amount of `coin-`X and `coin-`Y :

$$\boxed{x = \frac{\ell}{L}\,R^X \text{ and } y = \frac{\ell}{L}\,R^Y}\,, \tag{6}$$

where all quantities on the right hand side of the equalities are the quantities just prior to the `burn` event. Furthermore, after the texttttLP burn event, the the reserves and outstanding `LP coins` are updated as follows

$$\boxed{R^X \longrightarrow R^X - x, \quad R^Y \longrightarrow R^Y - y, \quad \text{and} \quad L \longrightarrow L - \ell}\,.$$

2 Code

2.1 The amm class

The participants are provided with a Python class amm that models a list of pools; see below. The attributes Rx and Ry are arrays with the reserves of coin-X and coin-Y , respectively, in each pool, phi is an array with the fee rate in each pool, L represents the total number of LP coins available in the pool (i.e., $\sqrt{R^X \times R^Y}$), and finally l represents the coins of liquidity held by the LP in each pool. The number of coins owned by the LP is modified when calling the liquidity provision operations mint and burn (see Sections 2.4 and 2.5). The methods of the amm class mimic the basic functioning of liquidity taking and liquidity provision. Participants are expected to complete the code of these methods according to the descriptions below. Each method description is accompanied with a code snippet with the expected results when the correct code is used and the missing functionality are filled in as per described in Sections 2.3, 2.4, 2.5, 2.6, and 2.7. A notebook with the code snippets is provided.

```
class amm():
    def __init__(self, Rx, Ry, phi):
        """ Reserves in each pool """
        self.Rx  = 1*Rx
        self.Ry  = 1*Ry

        """ Fee rate in each pool """
        self.phi = 1*phi

        # number of LP coins for each pool
        self.L = np.sqrt(self.Rx*self.Ry)

        # the LP begins with no LP coins
        self.l = np.zeros(len(self.L))
```

2.2 Initializing a list of pools

A pool is instantiated with initial reserves $\left(R_0^X, R_0^Y\right)$ of coin-X and coin-Y , respectively, and a fee rate, i.e., the proportional fee charged to LTs when they trade in the pool. As a result, the initial marginal price in the pool is $R_0^X \,/\, R_0^Y$. The following code creates 3 pools with initial reserves $(100, 1000)$, i.e., with initial marginal price $Z_0 = 0.1\, X/Y$, and a fee rate 0.3%; see Figure 2.

```
Rx0   = np.array([100,   100,   100],   float)
Ry0   = np.array([1000,  1000,  1000],  float)
phi   = np.array([0.003, 0.003, 0.003], float)

pools = amm(Rx=Rx0, Ry=Ry0, phi=phi)

print('Available LP coins:', pools.L)
```

```
Available LP coins: [316.228 316.228 316.228]
```

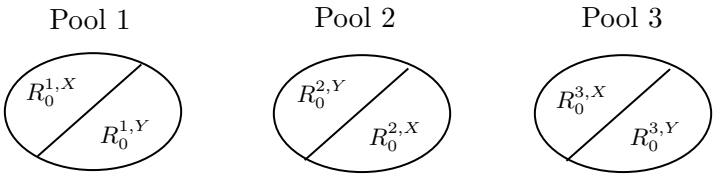

Figure 2: A list of 3 pools initialized with reserves $\left(R_0^{1,X}, R_0^{1,Y}\right)$, $\left(R_0^{2,X}, R_0^{2,Y}\right)$, and $\left(R_0^{3,X}, R_0^{3,Y}\right)$.

Once the pools are instantiated, there are two ways of interacting with them. One is through liquidity taking (swapping `coins`), and the other is through liquidity provision (minting and burning `LP coins`).

2.3 Swapping

To swap `coin-`X for `coin-`Y (i.e., buy `coin-`Y from the pool), use the method `swap_x_to_y`, and to swap `coin-`Y for `coin-`X (i.e., sell `coin-`Y to the pool), use the method `swap_y_to_x`. The two methods return the amount that is received from the pool, and update the reserves according to equations (2) and (4).

The code snippet in the panel to the right swaps 1, 0.5, and 0.1 units of X for `coin-`Y in the three pools and returns the quantity y obtained by the LT from each pool. Figure 3 shows details for Pool 1. Note that when `quote` is set to `True`, the method does not change the state of the pool and only returns the amount of `coin-`Y that the LT would potentially receive. Finally, the method `swap_y_to_x` is similar and uses the LT trading condition (3).

```
y = pools.swap_x_to_y([1, 0.5, 0.1], quote=False)

print('Obtained Y coins :', y)
print('Reserves in X    :', pools.Rx)
print('Reserves in Y    :', pools.Ry)
```

```
Obtained Y coins : [9.87   4.96   1.   ]
Reserves in X    : [101.   100.5  100.1]
Reserves in Y    : [990.13 995.04 999. ]
```

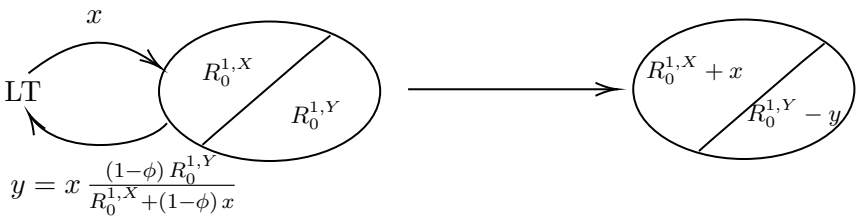

Figure 3: A swap transaction in the first pool.

2.4 Minting

To `mint` a quantity (x, y) of both coins in the pool, recall that the pair has to satisfy the LP trading condition (5). The method `mint` of the class `amm` allows the trader to `mint` liquidity, and returns the number of `LP coins` that have been minted; see Figure 4. These coins are then added to the LP's coins `self.l`. The ratio of the trader's `LP coins self.l` to the total number of `LP coins self.L` represents the portion of the pool held by the trader. The `mint` method first checks if the LP trading condition is satisfied and then executes the operation, including updating the reserves according to the LP conditions.

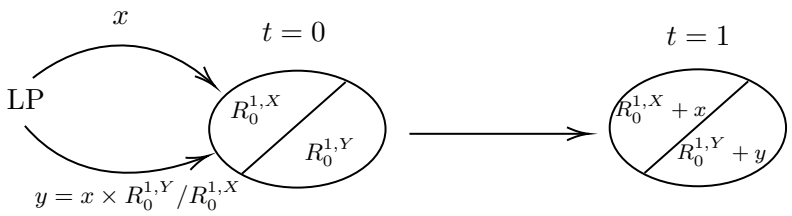

Figure 4: A mint operation in the pool.

The following code shows how to mint a pair of coins (x, y) in the three pools of our previous example. Note that in first attempt below, the amounts of coins submitted are not correctly specified, and the code raises an error.

```
x = [1, 1, 1]
l = pools.mint(x = x, y = np.random.rand(3))
```

```
AssertionError: pool 0 has incorrect submission of coins
```

```
y = x * pools.Ry / pools.Rx
l = pools.mint(x = x, y = y)

print('Trader LP coins :', pools.l)
print('Pool LP coins   :', pools.L)
```

```
Trader LP coins : [3.13   3.15   3.16]
Pool LP coins   : [319.36 319.37 319.39]
```

2.5 Burning

The method `burn` of the `amm` class burns some quantity `l` of LP coins. The amounts (x, y) that are withdrawn from the pool when burning satisfy the LP trading condition (6).

The method `burn` first checks if the LP has enough coins to burn in each pool, executes the operation, updates the reserves, and finally returns the amounts (x, y) withdrawn from the pool. The following code shows how to burn a quantity of l, `LP coins`. Note that first, the amount to burn is not correctly specified.

```
x, y = pools.burn(l+1)
```

```
AssertionError: you have insufficient LP coins
```

```
x, y = pools.burn(l)

print('Trader LP coins        :', pools.l)
print('Trader X coins         :', x)
print('Trader Y coins         :', y)
print('Pool LP coins          :', pools.L, 2)
```

```
Trader LP coins         : [0. 0. 0.]
Trader X coins          : [1. 1. 1.]
Trader Y coins          : [9.8  9.9  9.98]
Pool LP coins           : [316.23 316.23 316.23]
```

2.6 Swap then mint

In the competition's example below (and in practice), a liquidity provider holds some initial quantity $\tilde{x}$ of the reference `coin-`X and wishes to provide liquidity in a pool. To use all the available wealth to mint, the liquidity provider must first swap `coin-`X in the pool in order to hold the right amounts (x, y) to be able to mint.

Assume the LP has an initial number of coins $\tilde{x}$ of `coin-`X and that the pool has initial reserves $\left(R^X, R^Y\right)$. First, the LP swaps $x = (1-\theta)\ \tilde{x}$ in the pool, for $\theta \in (0,1)$, and receives a quantity y of `coin-`Y in exchange. Then, the LP uses the remaining $\theta\,\tilde{x}$ coins of `coin-`X and the y coins of `coin-`Y to mint `LP coins`. From the LT conditions, the amount of `coin-`Y received after the swap is

$$y = x\,\frac{(1-\phi)\ R^Y}{R^X + (1-\phi)\ x}\,, \tag{7}$$

where $x = (1-\theta)\ \tilde{x}$. Next, observe that the new marginal price in the pool is

$$\frac{R^X + (1-\theta)\ \tilde{x}}{R^Y - y}\,,$$

and to ensure the LP trading condition is enforced, the following equality should be satisfied to be able to mint $(\theta\,\tilde{x}, y)$ in the pool:

$$\frac{\theta\,\tilde{x}}{y} = \frac{R^X + (1-\theta)\ \tilde{x}}{R^Y - y}. \tag{8}$$

Solving for θ that satisfies equations (7)–(8) yields the following formula for θ

$$\theta = 1 + \frac{(2-\phi)\,R^X}{2\ (1-\phi)\ \tilde{x}}\left(1 - \sqrt{1 + 4\,\frac{\tilde{x}}{R^X}\,\frac{1-\phi}{(2-\phi)^2}}\right). \tag{9}$$

from which one obtains the quantity $(1-\theta)\ \tilde{x}$ of `coin-`X to swap.

The method `swap_and_mint` of the `amm` class implements this functionality, which we schematize in Figure 5. Finally, the following code shows the result for the example discussed above.

```
l = pools.swap_and_mint([10, 10, 10])

print('Minted LP coins       :', l)
print('Total trader LP coins :', pools.l)
print('Available LP coins    :', pools.L)
```

```
Minted LP coins       : [15.26.   15.34   15.4 ]
Total trader LP coins : [15.26   15.34   15.4 ]
Available LP coins    : [331.49  331.56  331.62]
```

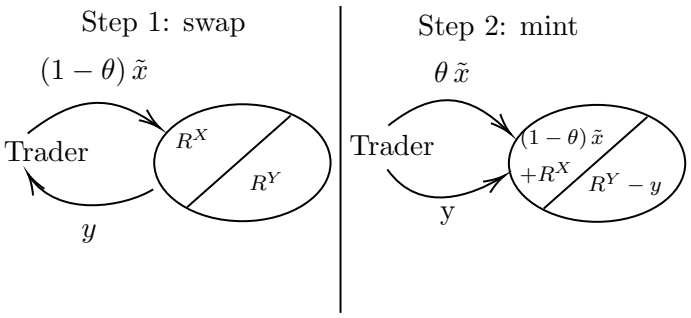

Figure 5: Swapping then minting.

2.7 Burn then swap

In the competition's example below, the liquidity provider `mints LP coins` across n pools at an initial time $t = 0$ and burns the positions at the end of a trading window $[0, T]$. To compute the performance, the trader swaps all the withdrawn `coin-`Y (from the `burn` trades) into `coin-`X. To execute the swap, the LP first checks which pool offers the best price, i.e., the pool that returns the maximum number of `coin-`X by swapping all of the `coin-`Y they have. The method `burn_and_swap` in the `amm` class implements this functionality, which is schematized in Figure 6 and showed in the following code.

```
total_x = pools.burn_and_swap(l)

print('Number of X coins received   :', total_x)
print('Total trader liquidity coins :', pools.l)
print('Available liquidity coins    :', pools.L)
```

```
Number of X coins received   : 28.796
Total trader liquidity coins : [0. 0. 0.]
Available liquidity coins    : [316.23 316.23 316.23]
```

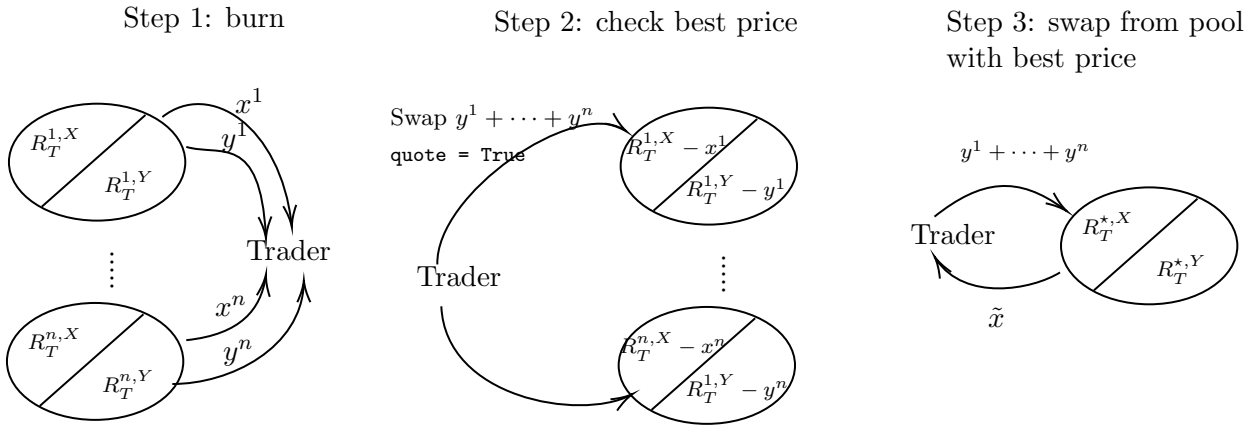

Figure 6: Burning then swapping.

2.8 Simulating trading in the pool

The method `simulate` in the `amm` class simulates a trading environment based on the set of parameters (κ, p, σ, T) and the parameter `batch_size`. The parameters κ and p set the level of

trading activity in the pools, σ effectively represents the volatility of the marginal price in each pool, T is the trading horizon in days, and `batch_size` defines the number of paths to generate[1].

The method returns the results of the simulation, which we describe in Table 1 for n pools and a `batch_size` of m.

Return variable	Data type & shape	Description
`end_pools`	`deque m x 1`	List of `amm` objects with terminal state of the pools for each simulation path.
`Rx_t`	`deque m x 1`	List of `K x n numpy` objects with historical values of the reserves in `coin-`X for each simulation path. (K is the number of transactions)
`Ry_t`	`deque m x 1`	List of `K x n numpy` objects with historical values of the reserves in `coin-`Y for each simulation path.
`v_t`	`deque m x 1`	List of `K x n numpy` objects with historical transaction sizes in the pools.
`event_type_t`	`deque m x 1`	List of `K x n numpy` objects with historical transaction types in the pools.
`event_direction_t`	`deque m x 1`	List of `K x n numpy` objects with historical transaction directions; 0 for a swap $X \to Y$, and 1 for a $Y \to X$.

Table 1: Description of simulation method return variables.

Consider an LP who swaps and mints a fixed number x of `coin-`X in each of the three pools of our previous example. The following code simulates $1,000$ paths for the three pools for a fixed trading environment defined by the tuple (κ, p, σ, T). Moreover, we show the final reserves in both coins in each pool for one of the simulation scenarios.

```
""" Fix the seed """
np.random.seed(999983)

""" Initialise the pools """
Rx0   = np.array([100,   100,   100], float)
Ry0   = np.array([1000,  1000,  1000], float)
phi   = np.array([0.003, 0.003, 0.003], float)

pools = amm(Rx=Rx0, Ry=Ry0, phi=phi)

""" Swap and mint """
xs_0 = [10, 10, 10]
l    = pools.swap_and_mint(xs_0)

""" Simulate 100 paths of trading in the pools """
batch_size = 1_000
T          = 60
```

[1]More details are provided in the implementation code of the method. These parameters are set to fixed values for the purposes of the competition.

```
kappa      = np.array([0.6,  0.5,  1,     2])
p          = np.array([0.85, 0.3,  0.2,   0.3])
sigma      = np.array([0.05, 0.01, 0.025, 0.05])

end_pools, Rx_t, Ry_t, v_t, event_type_t, event_direction_t =\
        pools.simulate( kappa = kappa, p = p, sigma = sigma, T = T, batch_size =
                                                    batch_size)

print('Reserves in coin X for scenario 0:', end_pools[0].Rx)
print('Reserves in coin Y for scenario 0:', end_pools[0].Ry)
```

```
Reserves in asset X for scenario 0: [ 96.491   118.366  121.083]
Reserves in asset Y for scenario 0: [1142.267  931.521  912.311]
```

To compute the performance of the LP in each simulation scenario, one must first burn the LP `coins` and then compare the final wealth with the initial one. Recall that the method `burn_and_swap` burns the LP's liquidity in all pools, then swaps all the coins into `coin-`X.

```
""" Burn and swap all coins into x """
x_T = np.zeros(batch_size)
for k in range(batch_size):
    x_T[k] = np.sum(end_pools[k].burn_and_swap(1))

x_0      = np.sum(xs_0)
log_ret  = np.log(x_T) - np.log(x_0)

print('Average performance     :', np.mean(log_ret)/T*100)
print('Std. Dev. of performance:', np.std(log_ret)/np.sqrt(T)*100)
```

```
Average performance     : 0.2629
Std. Dev. of performance: 1.1604
```

Finally, Figure 7 shows the Conditional Value at Risk of the distribution of the final performance and is generated by the code below. Note that the companion notebook provides visualization code to study simulation results.

```
fig, ax = plt.subplots(1, 1)

x_0      = np.sum(xs_0)
log_ret  = np.log(x_T) - np.log(x_0)
mean_ret = np.mean(log_ret)/T*100
std_ret  = np.std(log_ret)/np.sqrt(T)*100

#histogram plot
sns.histplot(ax=ax, x=100*log_ret, kde=True)

#Compute cvar
alpha = 0.95 # 95%
qtl   = np.quantile(-log_ret, alpha)
cvar  = np.mean(-log_ret[-log_ret>=qtl])
zeta  = 0.05 # 5%
```

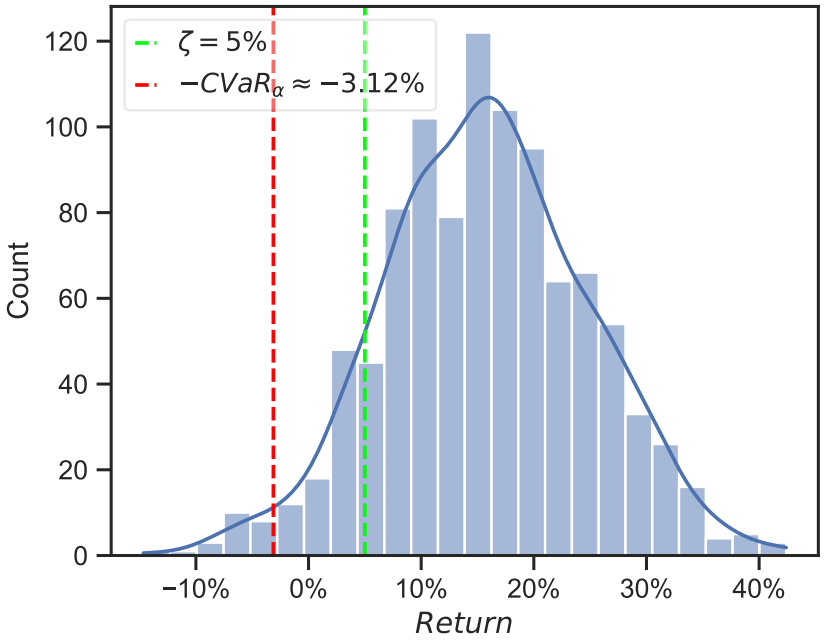

Figure 7: Distribution of the final performance.

3 Instructions for the competition

This competition aims at emulating the basic functioning of a CPMM with Python code and designing a strategy to supply liquidity in multiple liquidity pools.

3.1 Task #1

The first task is to complete the provided code of the Python `amm` class. The code for the methods `swap`, `mint` , `burn` , `swap_and_mint`, `burn_and_swap` must be completed according to the description in Sections 2.3, 2.4, 2.5, 2.6, and 2.7. More precisely, replace the following comment by the appropriate code:

```
# ********************
#      fill in code
# ********************
```

When the code is completed, it can be checked against the numerical results provided above by running the Google Colab notebook `test_notebook.ipynb`.

3.2 Task #2

Participants are provided with a fixed initial state of n pools, a fixed trading environment defined by the tuple (κ, p, σ, T), and a fixed initial number x_0 of X coins; see the companion file `params.py`.

The objective is to compute the amounts (in percentage of x_0) to put in each of the n pools.

The amounts should optimize the following performance criterion

$$\min_{\theta} \mathrm{CVaR}_{\alpha}[r_T] \tag{10}$$

$$\text{subject to } \mathbb{P}[r_T > \zeta] > q\,, \tag{11}$$

where $r_T = \log(x_T) - \log(x_0)$ is the final performance of the LP, θ is the distribution of the initial wealth across the n pools and α, ζ, and q are in the file `params.py`. The conditional Value-at-Risk (also called expected shortfall) at level $\alpha \in (0,1)$ for a variable r (regarded as the performance of a portfolio) is

$$\mathrm{CVaR}\,(r) = \frac{1}{1-\alpha}\int_{\alpha}^{1} \mathrm{VaR}_s\,(r)\; ds \;\approx \mathbb{E}[-r \mid -r \geq \mathrm{VaR}_{\alpha}], \tag{12}$$

where $\mathrm{VaR}_s\,(r)$ is the Value-at-Risk at level $s \in (0,1)$ of a real-valued random variable r (regarded as the performance of a portfolio) is

$$\mathrm{VaR}_{\alpha}\,(r) = -\inf\{z \in \mathbb{R} \mid \mathbb{P}\,(r \leq z) \leq 1-\alpha\}\,, \tag{13}$$

i.e., $\mathrm{VaR}_{\alpha}\,(r)$ is the $\alpha-$quantile of $-r$. Note that VaR can be computed using the standard numpy/Python quantile function `numpy.quantile` function. Respectively, CVaR can be approximated using (12) as sample conditional mean.

To obtain the performance r_T, one follows similar steps as those detailed in Section 2.8, i.e., the LP starts with x_0 coins of `coin-`X, decides on the proportion to mint in each pool, swaps and mints the coins, simulates trading using the trading environment provided above, and finally burns and swaps the `LP coins` at the end of the trading window.

3.3 Judging criteria

Participants are requested to provide the code they used to determine the initial weights θ. The final winner will be selected based on both the performance metric r_T and the methodology employed. Please note that straightforward grid or randomized search methods with extended run times may potentially be subject to a penalty in the overall evaluation of the work. We appreciate your understanding and adherence to these guidelines.

If you have specific questions or issues related to the problem, please do not hesitate to contact us at siagfmecodequest@gmail.com.