1 Solution

The guide on how to run the code is located in the `README.md` file. The implementation of the `amm` class is located in `defi/amm.py`, with unit tests in the `tests.py` ensuring validity of the results. The solution to the second problem can be reproduced by running the command

```
WEIGHTS=0.001,0.332,0.332,0.332,0.001,0.001 SEED=4294967143 python main.py
```

Our optimal weights for the second problem are:

```
[0.13236412, 0.20837845, 0.1833777, 0.21926841, 0.2356687, 0.02094261]
```

This solution attains a CVaR_α of `0.00400993`. This solution satisfies the chance constraint, yielding for $\mathbb{P}(r_T \geq \zeta)$ the value `0.842`. This solution was obtained with $N_{batch} = 2000$ and seed number `4294967143`, as specified in the `params.py` file. Since there are only 5 fields in the submission form, we submitted the values for the first 5 pools, whereby the value for the sixth pool is implied by the condition $\sum_i w_i = 1$.

2 Problem setup

One way for market makers to provide liquidity in the world of cryptocurrencies, is to invest in a liquidity pool. We consider a market with two coins, denoted by $\mathcal{X}$ and $\mathcal{Y}$. When other market players trade $\mathcal{X}$ for $\mathcal{Y}$, or vice versa, via the liquidity pool, the market maker cashes a fee. In case multiple pools exists, a market maker has to decide how to distribute his wealth over the set number of pools.

In the used set-up, there are N_{pools} liquidity pools, all providing liquidity in $\mathcal{X}$ and $\mathcal{Y}$. $\mathcal{X}$ is set as numeraire, and we start with the initial stock x_0. Based on the sizes of the pools, and the pools activity, we have to choose a pool allocation $w \in \mathbb{R}^{N_{pools}}$. We invest a value of $x_0 w_i$ in the i-th pool. This is done by swapping an appropriate portion of $x_0 w_i$ into $\mathcal{Y}$, such that the ratio of our invested $\mathcal{X}$ and $\mathcal{Y}$ matches that of the pool. In exchange for investing, we receive liquidity tokens, which describe our share of the pool. After a predetermined time T has passed, we convert our liquidity tokens back into $\mathcal{X}$ and $\mathcal{Y}$, and then swap all the $\mathcal{Y}$ collected over all the pools back into $\mathcal{X}$ using pool which can give us the most $\mathcal{X}$, such that in the end we have a total of x_T in $\mathcal{X}$. The goal is to choose the distribution $w \in \mathbb{R}^{N_{pools}}$ such that for some $\alpha \in (0,1)$ the CVaR_α of the log-returns, i.e. $r_T := \log(x_T/x_0)$, is minimised, under the constraint that the probability of log-returns exceeding $\zeta \in (0,1)$ is larger than $q \in (0,1)$. In other words:

$$\min_w \quad \text{CVaR}_\alpha(r_T) \tag{2.1}$$

$$\text{s.t. } \mathbb{P}(r_T > \zeta) > q \tag{2.2}$$

3 Methodology and algorithm

Note that our methodology and algorithm allow to solve for a larger set of problems than the one sample problem provided, as we leave all parameters, including the amount of pools, abstract. We will use overline notation $\overline{x}$ to emphasize that the element runs over N_{batch}, i.e. $\overline{x} \in \mathbb{R}^{N_{batch} \times \cdots}$.

3.1 Parameters

We consider the parameters from the original setting, taking the following values specified in `params.py`:

Parameter	Value	Interpretation
N_{pools}	6	Number of pools
N_{batch}	1000	Number of paths
R_0^X	[100, 100, 100, 100, 100, 100]	Initial stock of $\mathcal{X}$ in each pool
R_0^Y	[1000, 1000, 1000, 1000, 1000, 1000]	Initial stock of $\mathcal{Y}$ in each pool
ϕ	[0.03, 0.03, 0.03, 0.03, 0.03, 0.03]	Proportional fee in each pool
x_0	10	Initial holdings of the LP
α	0.9	CVaR quantile
q	0.8	Profitability threshold
ζ	0.05	Profitability quantile
κ	[0.25, 0.5, 0.5, 0.45, 0.45, 0.4, 0.3]	Jump intensity
σ	[1., 0.3, 0.5, 1., 1.25, 2, 4]	Jump volatility
p	[0.45, 0.45, 0.4, 0.38, 0.36, 0.34, 0.3]	Directionality of swaps
T	60	Timeframe

Table 1: Model Parameters

Moreover, we consider the following numerical parameters corresponding to our implementation:

Parameter	Value	Interpretation
β	0.01	Learning rate of the GD
$(\theta_1, \theta_2, \theta_3, \theta_4)$	[10, 1, 1, 1]	Weights of the loss components
w	[0.001, 0.332, 0.332, 0.332, 0.001, 0.001]	Initial weights
N_{iter}	20	# iterations in the main loop
N_{GD}	2000	# steps in the GD iteration

Table 2: Numerical Parameters

3.2 Methodology

Essentially, our algorithm uses generated paths of the market to find the optimal weights, and in order to account for the market impact, we alternate between market generation and gradient descent.

Main loop consists of N_{iter} iterations, where each of these iterations includes a market generation step and a gradient descent loop. At the beginning of each iteration $i \in \{1, \dots, N_{iter}\}$ we use the weights w_i obtained from the last iteration to generate the market paths, and within this iteration we use these paths in the gradient descent iteration of N_{GD} steps to obtain new weights w_{i+1}. The idea behind alternation between market generation and gradient descent is that once the weights have changed, the market impact of the pool allocations has changed as well, which affects profitability of the pools.

In order to account for this, at each step $j \in \{1, \dots, N_{GD}\}$ of the gradient descent iteration we reweigh the holdings $\overline{x}$, $\overline{y}$ using the weight iterates $\widetilde{w}_j$, and use the characteristics of the best pool $(\overline{R}^X, \overline{R}^Y, \overline{\phi}) \in (\mathbb{R}^{N_{batch}})^3$ – which will be different for each $k \in \{1, \dots, N_{batch}\}$ – to compute the ultimate holdings denominated in $\mathcal{X}$ after the exchange of $\mathcal{Y}$ into $\mathcal{X}$ in the best pool. This allows us to compute returns for each path $k \in \{1, \dots, N_{batch}\}$, which we will use to compute objectives in the optimization routine.

3.3 Algorithm description

The algorithm proceeds as follows. At the beginning of each step $i \in \{1, \dots, N_{iter}\}$, we start with some weights $w_i \in \mathbb{R}^{N_{pools}}$. We then invoke the function

$$\textsc{GenerateMarket}(w_i) = \left(\overline{x}, \overline{y}, \overline{R}^X, \overline{R}^Y, \overline{\phi}\right),$$

which generates N_{batch} paths of the market evolution starting from initial pool allocation $x_0 \cdot w_i$, such that $\overline{x}, \overline{y} \in \mathbb{R}^{N_{batch} \times N_{pools}}$ correspond to the amounts of $\mathcal{X}$, $\mathcal{Y}$ in each pool obtained after burning the liquidity tokens at the end of the period; $\overline{R}^X, \overline{R}^Y \in \mathbb{R}^{N_{batch}}$ correspond to the stock of the $\mathcal{X}$, $\mathcal{Y}$ in the pool offering the best price; and $\overline{\phi} \in \mathbb{R}^{N_{batch}}$ corresponds to the fee parameter of the pool offering the best price (in our case, always constant and equal to 0.03 for all pools).

Thereafter, in the i-th iteration of the main loop we enter a gradient descent loop of N_{GD} steps, which yields the next weight iterate w_{i+1} based on the generated paths. Specifically, for $j \in \{1, \dots, N_{GD}\}$, in

the j-th step of the gradient descent loop, we consider the weight iterate $\widetilde{w}_j \in \mathbb{R}^{N_{pools}}$ and approximate the returns of the reweighed portfolios by

$$\overline{x}_{burn} := \overline{x}\left(\widetilde{w}_j \odot w_i^{-1}\right) \in \mathbb{R}^{N_{batch}}, \qquad \overline{y}_{burn} := \overline{y}\left(\widetilde{w}_j \odot w_i^{-1}\right) \in \mathbb{R}^{N_{batch}},$$

$$\overline{x}_{swap} := \frac{\overline{y}_{burn} \odot (1-\overline{\phi}) \odot \overline{R}^X}{\overline{R}^Y + (1-\overline{\phi}) \odot \overline{y}_{burn}} \in \mathbb{R}^{N_{batch}}, \qquad \overline{r} := \log(\overline{x}_{burn} + \overline{x}_{swap}) - \log(x_0) \in \mathbb{R}^{N_{batch}}, \tag{3.1}$$

where the inverse w_i^{-1} and division are considered elementwise, and we denote elementwise multiplication by $\odot$. For $w \in \mathbb{R}^{N_{pools}}$ and $\overline{r} \in \mathbb{R}^{N_{batch}}$ we introduce the following loss functions

$$\ell_1(w,\overline{r}) = \left(q - \frac{1}{N_{batch}} \sum_{i=1}^{N_{batch}} G(1000 \cdot (r_i - \zeta))\right)_+^2, \qquad \ell_2(w,\overline{r}) = \frac{1}{N_{pools}} \sum_{i=1}^{N_{pools}} (-w_i)_+,$$

$$\ell_3(w,\overline{r}) = \left(\sum_{i=1}^{N_{pools}} w_i - 1\right)^2, \qquad \ell_4(w,\overline{r}) = \mathrm{CVaR}_\alpha = \frac{\sum_{i=1}^{N_{batch}} (-r_i) \mathbb{1}_{\{-r_i > q_\alpha(-\overline{r})\}}}{|\{i \in \{1,\dots,N_{batch}\} : -r_i > q_\alpha(-\overline{r})\}|},$$

where G is the sigmoid function $G(z) := \frac{1}{1+e^{-z}}$, $(z)_+ = z\mathbb{1}_{\{z>0\}}$ is the ReLU function, and $q_\alpha : \mathbb{R}^{N_{batch}} \to \mathbb{R}$ denotes the empirical quantile function. We define the total loss function as

$$\ell(w,\overline{r}) := \theta_1\ell_1(w,\overline{r}) + \theta_2\ell_2(w,\overline{r}) + \theta_3\ell_3(w,\overline{r}) + \theta_4\ell_4(w,\overline{r}).$$

where we used the loss weights θ which are used to ensure that the loss components are of the same order. The interpretation is straightforward: ℓ_1 corresponds to the chance constraint (2.2), where we implement $\mathbb{E}[\mathbb{1}_{r>\zeta}]$ with sigmoid function G to maintain differentiability of the loss; ℓ_2 ensures that the weights are not negative; ℓ_3 ensures that the weights sum to one; ℓ_4 is the CVaR objective (2.1) to be minimized. The gradient descent is implemented in `pytorch` with gradients computed by backpropagation. Note that our algorithm crucially uses the fact that `torch.quantile` function, which is used for the CVaR implementation, is differentiable.

We present in pseudocode the algorithm described above in Algorithm 1. Note that log, division and inverse are taken elementwise, whenever applicable. Also note that in the algorithm we make small adjustments which we don't include in the pseudocode, such as flooring weights at 0 to ensure that they are non-negative, adding a small number $\varepsilon = 0.0001$ to each weight to ensure that they are not zero (since zero swap is not allowed), and normalize them so that they sum up to 1. This is necessary, since these constraints may be violated. However, the violation is being minimized due to being encoded in the loss.

Algorithm 1: OPTIMIZE

Data: N_{pools}, N_{batch}, R^X, R^Y, x_0, α, q, ζ, κ, σ, w, θ, β, N_{iter}, N_{GD}.
Result: Optimal weights.
$w_1 := w$
for $i \leftarrow 1$ **to** N_{iter} **do**
  $\left(\overline{x}, \overline{y}, \overline{R}^X, \overline{R}^Y, \overline{\phi}\right) :=$ GENERATEMARKET(w_i)
  $\widetilde{w}_1 = w_i$
  for $j \leftarrow 1$ **to** N_{GD} **do**
    $\overline{x}_{burn} := \overline{x}\left(\widetilde{w}_j \odot w_i^{-1}\right)$
    $\overline{y}_{burn} := \overline{y}\left(\widetilde{w}_j \odot w_i^{-1}\right)$
    $\overline{x}_{swap} := \frac{\overline{y}_{burn} \odot (1-\overline{\phi}) \odot \overline{R}^X}{\overline{R}^Y + (1-\overline{\phi}) \odot \overline{y}_{burn}}$
    $\overline{r} := \log(\overline{x}_{burn} + \overline{x}_{swap}) - \log(x_0)$
    Compute the loss $\ell(w_j, r_j) := \theta_1\ell_1 + \theta_2\ell_2 + \theta_3\ell_3 + \theta_4\ell_4$
    Compute gradients $\nabla\widetilde{w}_j$ with respect to the loss ℓ.
    $\widetilde{w}_{j+1} := \widetilde{w}_j - \beta \cdot \nabla\widetilde{w}_j$
  $w_{i+1} := \widetilde{w}_{N_{GD}+1}$
return $w_{N_{iter}+1}$

3.4 Choice of the initial value

In order to obtain a good initial value for the weights, we consider profitability of the respective pools in isolation, where we invest all initial stock in a single pool. That is, for each $i \in \{1,\dots,N_{pools}\}$ we

generate returns according to the formula (3.1) with initial weights $w^i \in \mathbb{R}^{N_{pools}}$ defined as

$$w^i_j = \begin{cases} 1, & i = j \\ 0, & i \neq j. \end{cases}$$

The profitability metrics of the pools, i.e. mean return, CVaR_α and $\mathbb{P}(r_T \geq \zeta)$ are displayed in the Figure 1.

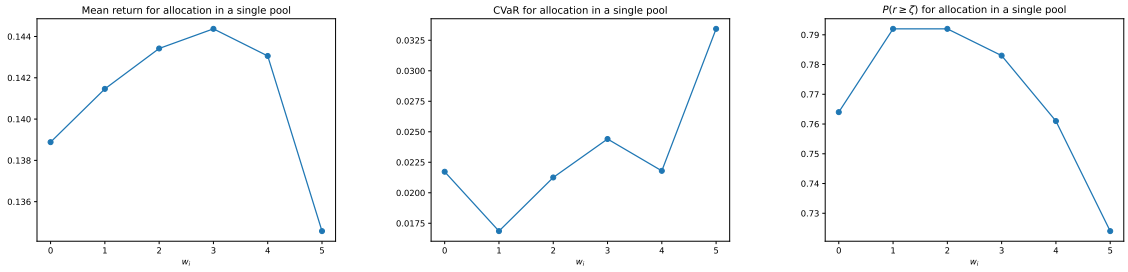

Figure 1: Mean return, CVaR_α and $\mathbb{P}(r_T \geq \zeta)$ whenever allocating all initial stock in a single pool i, for $i \in \{1, \ldots, N_{pools}\}$.

In Table 3.4 we summarize ranking of the pools across different metrics. It suggests that Pool 6 is the worst performing pool, Pool 1 and Pool 5 are slightly better, followed by Pool 4, while Pool 2 and Pool 3 are among the best performers. This leads us to the choice of the initial allocation corresponding to the equal distribution between the 3 best pools, i.e.:

$$w := \left(0, \frac{1}{3}, \frac{1}{3}, \frac{1}{3}, 0, 0\right).$$

Metric \ Pool	1	2	3	4	5	6
Mean return	5	4	2	1	3	6
CVaR_α	3	1	2	5	4	6
$\mathbb{P}(r_T \geq \zeta)$	4	1	1	3	5	6

Table 3: Ranking of the pools for individual performance metrics.

3.5 Convergence and performance analysis

We executed the algorithm for 3 different starting values w, for 30 different values of seed ranging from 4294967143 to 4294967162, with $N_{iter} = 20$ iterations of the main loop, each encompassing $N_{GD} = 2000$ of the gradient descent steps with learning rate $\beta = 0.01$. For the initial values we considered

$$w_{011100} := \left(0, \frac{1}{3}, \frac{1}{3}, \frac{1}{3}, 0, 0\right), \qquad w_{000001} := (0, 0, 0, 0, 0, 1), \qquad w_{111111} := \left(\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}\right).$$

The value w_{011100} corresponds to the heuristic described in the previous section. The value w_{111111} corresponds to the uniformly distributed portfolio, which one would start from without any additional information about the dynamics of the system, and we demonstrate that our algorithm delivers good results for this starting point as well (but not as good as our educated guess). The value w_{000001} was chosen as a bad initial guess, which violates the chance constraint and attains rather large CVaR value, and we demonstrate that our algorithm yields roughly the same results as for the starting point w_{111111}, despite a rather bad initial guess.

Convergence of the CVaR objective as a function of iteration is displayed in the Figure 2. We observe that the algorithm already provides a very good candidate after 1 iteration (even for the bad initial guess w_{000001}), and converges after roughly 5 iterations, while attaining the minimum after the first iteration for roughly half of the paths.

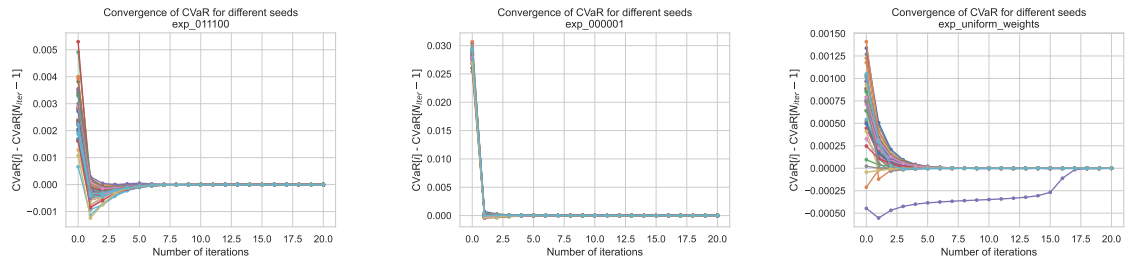

Figure 2: Convergence of the CVaR for different seeds as function of iteration for w_{011100} (left), w_{000001} (right) and w_{111111} (bottom).

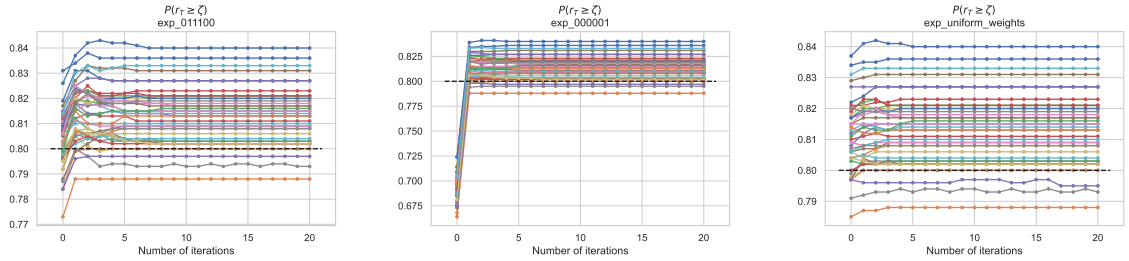

Figure 3: Convergence of chance constraint $\mathbb{P}(r \geq \zeta)$ for different seeds as function of iteration for w_{011100} (left), w_{000001} (right) and w_{111111} (bottom), with the threshold value $q = 0.8$.

Behavior of the chance constraint as a function of iteration is displayed in the Figure 3. We observe that for those paths that satisfied the constraint from the beginning, the algorithm never violates it again, and for those which violated the constraint in the beginning, the algorithm moves swiftly towards the barrier $q = 0.8$, which for the bad initial value w_{000001} happens right away after 1 iteration. Concerning the paths which never cross the barrier $q = 0.8$ and always stay below it, in all likelihood samples corresponding to those seeds do not admit enough profitability in the first place.

Pool allocation among different values of the seeds is displayed in the Figure 4. We note that the allocation is relatively stable between the seeds, and would be more homogeneous if one would consider larger values of N_{batch}.

Comparison of CVaR values between different initial values w_{011100}, w_{000001} and w_{111111} is displayed in the Figure 5, i.e. for each seed we display

$$\mathrm{CVaR}_{000001} - \mathrm{CVaR}_{011100}, \qquad \mathrm{CVaR}_{uniform} - \mathrm{CVaR}_{011100},$$

so that positive values indicate that the initial guess w_{011100} is better than the other. We observe that our chosen heuristic for the initial value beats other initial values across most seeds.

3.6 Running time

One iteration of the algorithm takes roughly 2–3 seconds on MacBook Air with M1 processor. Majority of this time (more than 2 seconds) is used to generate paths of the market. One step of the gradient

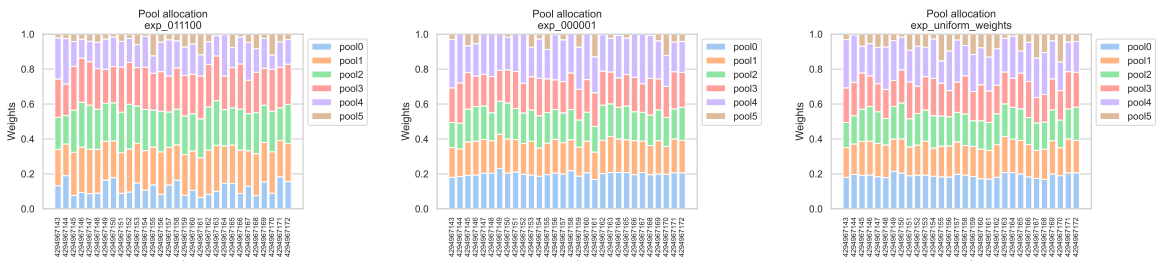

Figure 4: Optimal allocation among the pools for different seeds as function of iteration for w_{011100} (left), w_{000001} (right) and w_{111111} (bottom).

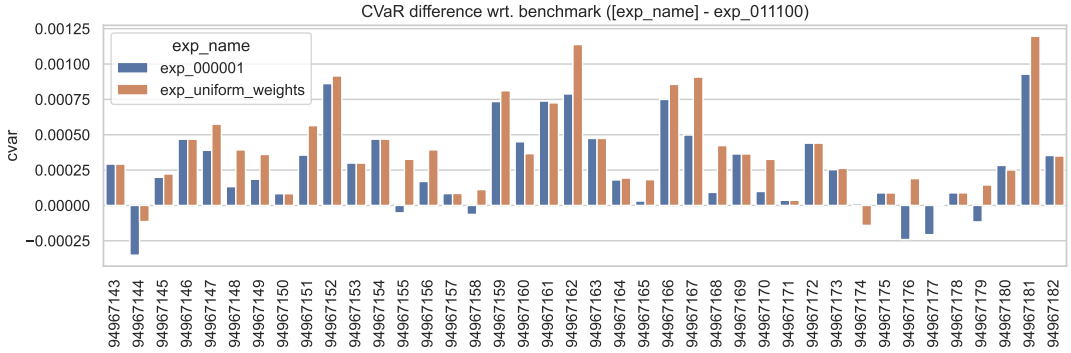

Figure 5: Comparison of CVaR values attained for different seeds with different initial weights.

descent takes roughly 50μs, and we are making $N_{GD} = 2000$ gradient descent steps for each iteration of the main loop. Therefore, each iteration of the algorithm (market generation + gradient descent) is taking roughly 2–3 seconds.

The plots above suggest that after 1 iteration one already gets a very good candidate for the weights. At this point, if the speed is highest priority, one can just use $N_{iter} = 1$ and obtain a reliable result after 2–3 seconds, otherwise one can obtain improvement of the CVaR of order 10^{-3} at the expense of some other 10 seconds.