Simulating RMMS - Documentation

Primer on CFMMs and arbitrage

Trading functions

Constant functions market makers (CFMMs)for two assets are functions of the reserves of each asset in the pool of the form $\varphi(x, y)$ that define a trading rule. In this case, x and y designate the reserves of each asset. The trading rule is such that if a trader adds some amount of tokens in Δ, they will get an amount out Δ' such that:

$$\varphi(x, y) = \varphi(x + \Delta, y + \Delta')$$

where Δ' is negative to express the fact that it's an amount out. Hence the name constant function. In the case of Uniswap v2 for example, the function is:

$$\varphi(x, y) = x \cdot y$$

This is often written as:

$$x \cdot y = k$$

where k is called the "invariant", so valid trades are those that will keep the product of the reserves after the trade equal to that invariant. Sometimes, a fee is charged to the trader. The way it works is that the calculation of the amount out is done with only a fraction of the amount in, which means the trader receives a lower amount out, but in the end the entire amount in is actually added to the pool.

Let's illustrate this again in the case of Uniswap. Let's say there is some fee $f \in [0, 1]$. Define $\gamma = 1 - f$. We start with a state of the liquidity pool (x, y) where the invariant is initially $k = x \cdot y$. In the present document, x will always designate some "risky asset" and y some "riskless asset". The trader requests a trade with some amount in Δ. To find the amount out Δ', we solve the following equation:

$$(x + \gamma\Delta)(y - \Delta') = k$$

The reserves are then updated to $(x', y') = (x + \Delta)(y - \Delta')$. Since we found a Δ' with a γ factor, obviously the product of the new reserves is not equal to k anymore, and so there is a new "invariant" k' that will be used for the next trade. Since the trader is getting a lower amount out with the same amount in compared to the no-fee case, we know that this new invariant will be greater, and it can be shown to always be the case for any quasiconcave, strictly increasing function. So we must have the relation:

$$\forall\ \varphi,\ k' > k$$

For more details on this, see Angeris and Chitra (2020).

Reported price, marginal price

This section is dedicated to understanding the price the traders get when they request a trade against a CFMM and the logic behind arbitrage, in the simple case of a two assets CFMM.

First let's consider the case of a no-fee CFMM. This CFMM can be understood as having a "spot price", which is that price that the trader would get if they were to swap an infinitesimal amount of one of the tokens. When there are two tokens, the trading function can often be rewritten as $y = f(x)$. If the trader adds some infinitesimal amount dx of token x, the new amount of the other token must satisfy the new equation, so we have:

$$y' = f(x') = f(x + dx)$$

Thus the trader was given some infinitesimal amount out $y' - y = dy$ (defined as negative because this amount is getting out of the pool). If we want to know the marginal price of that trade denominated in unit riskless per risky, we must compute the ratio:

$$-\frac{dy}{dx} = \frac{y' - y}{x' - x}$$

where there is a negative sign because the price must be positive. We can rewrite this as:

$$-\frac{f(x + dx) - f(x)}{dx}$$

Which when $dx \to 0$ is the definition of $-f'(x)$, the opposite of the derivative of f at x. So the "spot price" of a no-fee CFMM at some given reserves is the slope of the tangent at those reserves.

In the case of Uniswap, we see that the price denominated in riskless per risky goes up as the reserves of the risky get exhausted, and conversely, as expected.

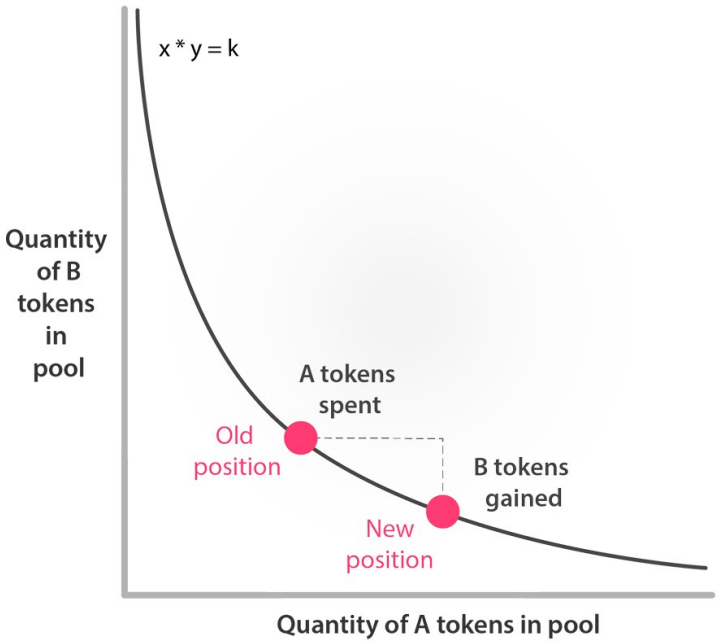


Figure 1. An illustration of the Uniswap v2 CFMM from this blog post.

This spot price is also sometimes called the *reported price* of the CFMM. That reported price can also be found in the following way:

The marginal price of an infinitesimal trade when swapping one of the assets can be found in the following way.

For any amount in of the risky asset Δ, we can find some amount out Δ' (taken positive here for simplicit). We can thus express the amount out as a function of the amount in $\Delta'(\Delta)$. Now we would like to know, given that a trader requests to swap some amount in Δ, what would be the marginal price of adding an infinitesimal amount $d\Delta$ to that request? The difference in the amount the trader would be $\Delta'(\Delta + d\Delta) - \Delta'(\Delta)$, while the difference in the amount in is $d\Delta$. So the price of that small infinitesimal trade is:

$$g(\Delta) = \frac{\Delta'(\Delta + d\Delta) - \Delta'(\Delta)}{d\Delta} = \frac{d\Delta'}{d\Delta}$$

or the derivative of the function $\Delta'(\Delta)$. Thus the marginal price of an infinitesimal trade is $g(0)$, which can be shown to be equal to $f'(x)$ as defined above.

No actual trade will have exactly that price, the marginal price of a non infinitesimal trade is of course Δ'/Δ.

$g(\Delta)$ represents what the marginal price of an infinitesimal trade *would be* after a trade of size Δ. The same argument must be mirrored to reason about swapping in the other asset.

When the CFMM has fees, the above argument doesn't work anymore. The marginal price of an infinitesimal trade is not the same depending on whether we're swapping in the risky or the riskless asset. This can be understood as a "spread", similar to an orderbook. This is due to the γ factor discussed above which is applied to the amount in, and so there is a dependence on which asset we're swapping. If the CFMM is asymmetrical in its arguments (i.e. x and y can't be interchanged without changing the value of the trading function), the spread will also be asymmetrical.

It can be shown that in the case when there are fees, the marginal price of a an infinitesimal trade after a trade of size Δ is given by $\gamma g(\gamma\Delta)$ where g is the function previously defined in the no-fee case. The marginal price of swapping some infinitesimal amount is thus given by $\gamma g(0)$. Since the function g is not necessarily the same depending on which asset we're swapping, in particular when the function is asymmetric, **the price of buying and selling the risky asset in the pool denominated in riskless per risky is *not* necessarily the same.**

Optimal arbitrage

An arbitrager seeks to exploit a difference between the price on the reference market and the marginal price in the pool. Assuming an infinitely liquid reference market with price m, the goal of the optimal arbitrage problem is to swap as much amount in such that after the swap, the marginal price is equal to the reference price. This can be expressed neatly as:

$$\gamma g(\gamma\Delta^*) = m$$

This is the equation we have to solve for Δ^*, the optimal amount in. Depending on whether the reference price m is above the price of buying, or selling the risky asset in the pool, the function g might be different as discussed above. If the reference price is between the price

of buying and the price of selling the risky asset in the pool, there is no profitable arbitrage possible.

Theory of the Covered Call CFMM

Here we are specifically looking at the covered call CFMM defined in the RMMS paper. That CFMM is:

$$y - K\Phi(\Phi^{-1}(1-x) - \sigma\sqrt{\tau}) = k$$

where the invariant $k = 0$ in the zero fees case. It can be shown indeed that assuming no-arbitrage, the value of the reserves of that CFMM (and hence the value of the LP shares) tracks the value of a covered call of a given time to maturity, strike price, and implied volatility, as illustrated below.

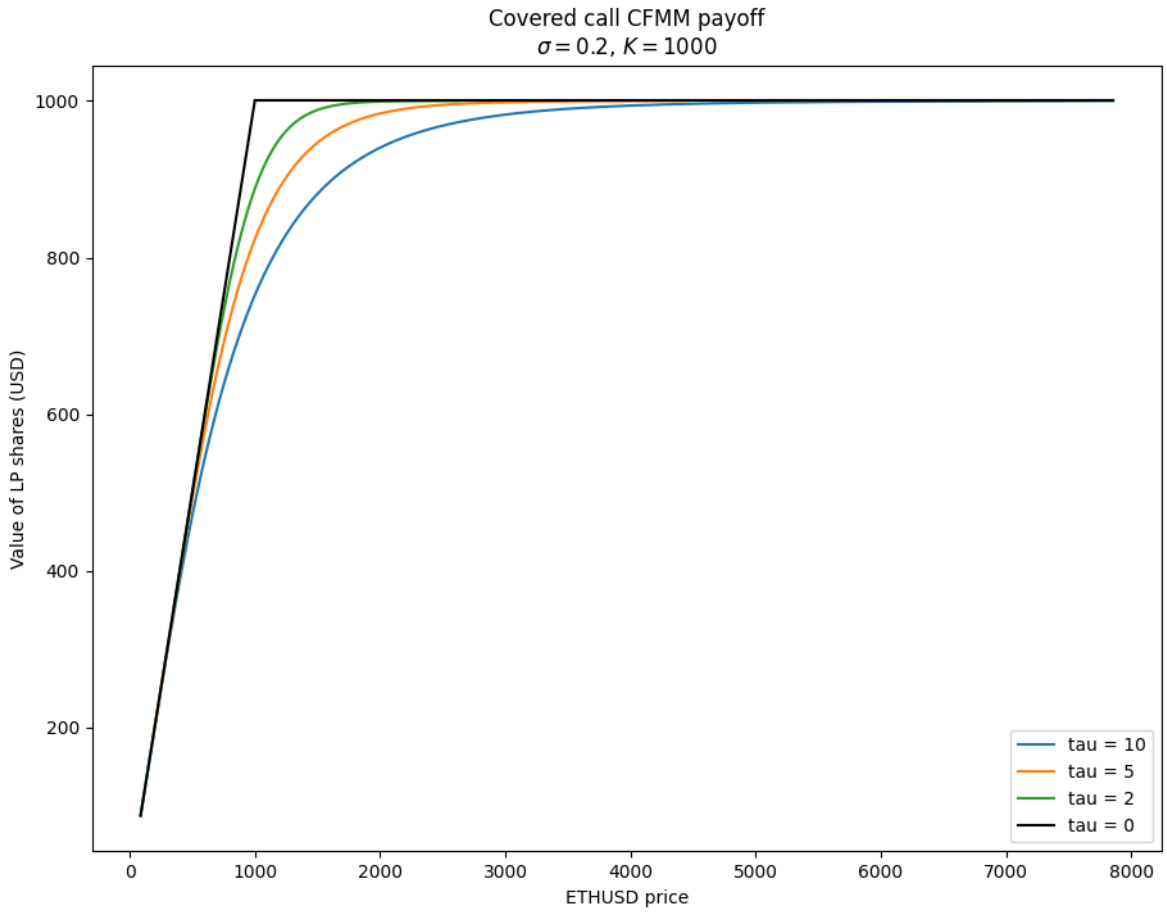


Figure 2. The value of the LP shares in the covered call CFMM, assuming no-arbitrage, as a function of the risky asset price and τ.

For every γ, τ, k, we can calculate the different quantities of interest for that CFMM, with and without fees:

Reported price

We write the CFMM as:

$$y = f(x) = K\Phi(\Phi^{-1}(1-x) - \sigma\sqrt{\tau}) + k$$

Taking the derivative gives us the reported price S as a function of the risky asset reserves. Using the chain rule:

$$S(x) = K\phi(\Phi^{-1}(1-x) - \sigma\sqrt{\tau}) \times (\Phi^{-1})'(1-x)$$

Where ϕ is the derivative of the standard CDF Φ, the standard normal PDF. Using the inverse function theorem, we can get an analytical expression for the derivative of the inverse CDF:

$$\forall x \in]0,1[,\ (\Phi^{-1})'(x) = \frac{1}{\phi(\Phi^{-1}(x))}$$

Such that we can rewrite:

$$S(x) = K\frac{\phi(\Phi^{-1}(1-x) - \sigma\sqrt{\tau})}{\phi(\Phi^{-1}(1-x))}$$

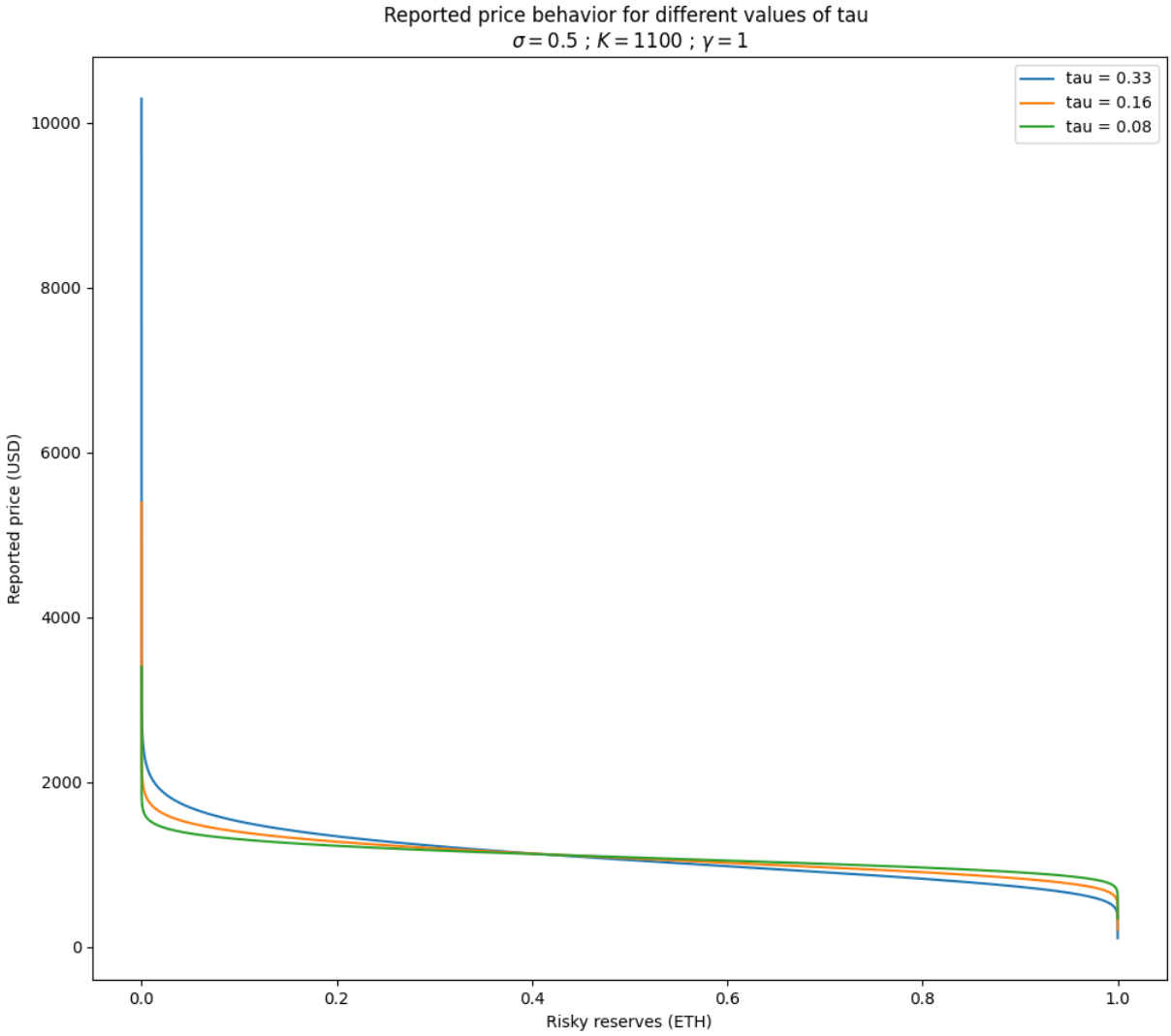
Using the definition of $\phi(x) = \frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}$, we can further simplify by noting that:

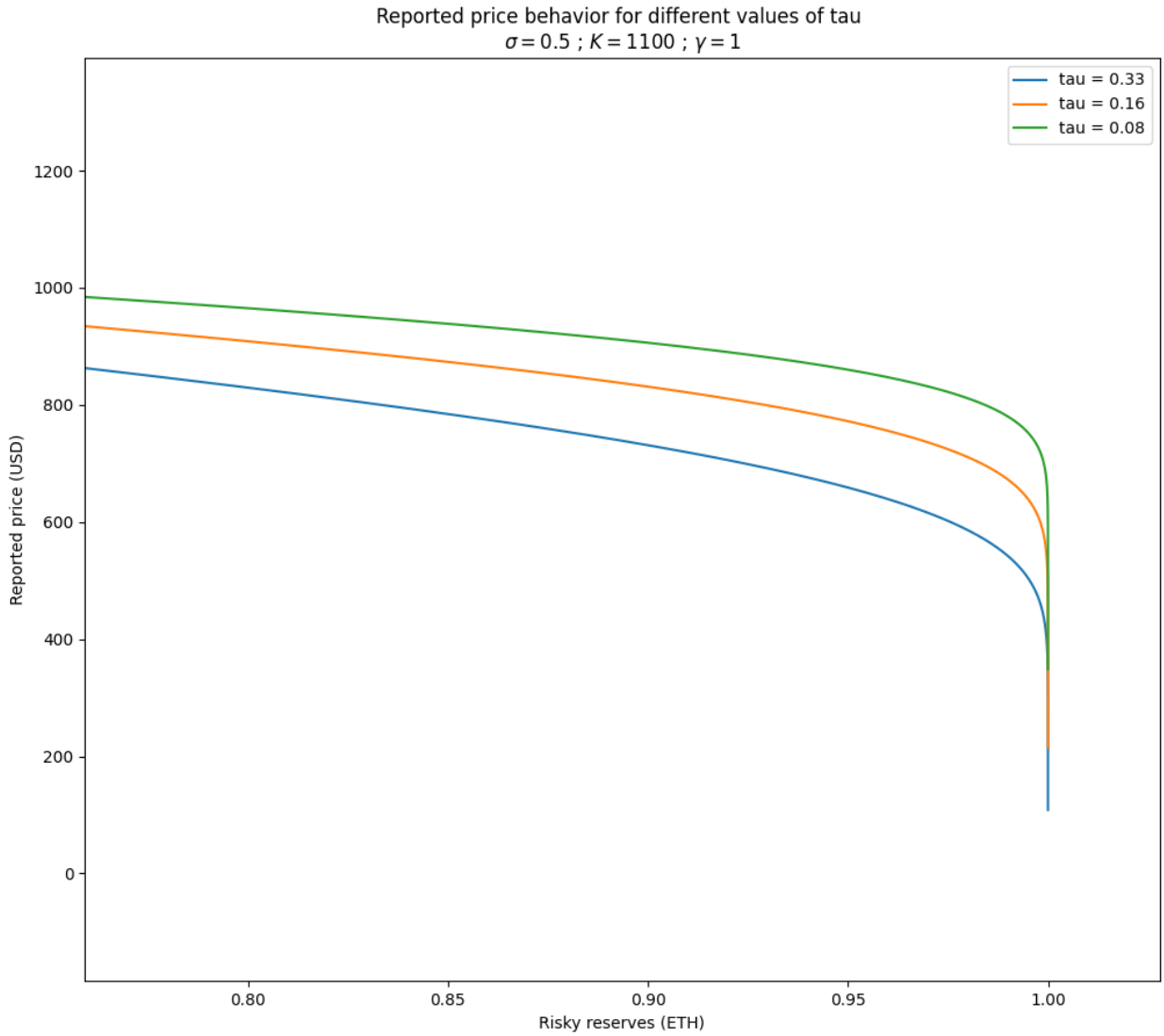
$$\phi(\Phi^{-1}(1-x) - \sigma\sqrt{\tau}) = \phi(\Phi^{-1}(1-x))e^{2\Phi^{-1}(x)\sigma\sqrt{\tau}}e^{-\sigma^2\tau}$$

And in the end:

$$S(x) = \frac{K}{2\pi}e^{2\Phi^{-1}(1-x)\sigma\sqrt{\tau}}e^{-\sigma^2\tau}$$

We can see that all prices are supported by this AMM for any value of σ or τ. Indeed, $\lim_{x\to 0} S(x) = +\infty$ and $\lim_{x\to 1} S(x) = 0$. However, because of the presence of the standard normal quantile function, also known as the probit function, there are significant kinks at the boundaries as seen in the plots below:


Reported price behavior for different values of tau
$\sigma = 0.5$; $K = 1100$; $\gamma = 1$
tau = 0.33
tau = 0.16
tau = 0.08
Reported price (USD)
10000
8000
6000
4000
2000
0
0.0
0.2
0.4
0.6
0.8
1.0
Risky reserves (ETH)



The effect of this is that while the AMM does theoretically support any price between 0 and $+\infty$, some prices cannot practically be reached in a real world setting. For example for the value $\tau = 0.33$ in the figure above, a decrease of a factor of 1000 from risky reserves of 1e-13 ETH to 1e-16 ETH only moves the reported price up by 14% to 3978 USD per ETH. This means that above some threshold, the pools will behave as if they can be emptied above a certain price threshold, at which point the CFMM will not report any meaningful price other than a lower / upper bound.

Swap risky in

Let's assume that a trader requests a swap of an amount Δ of the risky asset in a pool that initially has reserves $x,\ y$. The new amount of risky assets in the reserves is $x' = x + \Delta$. Assuming a fee regime γ, we can obtain the new riskless reserves as follows:

$$y' = K\Phi(\Phi^{-1}(1 - (x + \gamma\Delta)) - \sigma\sqrt{\tau}) + k$$

And so the amount out to give to the traders is $\Delta' = y - y'$.

Of course k needs to be updated according to the new amounts in the pool as:

$$k = y' - K\Phi(\Phi^{-1}(1 - x') - \sigma\sqrt{\tau})$$

Swap riskless in

Similarly, if a trader requests a swap of an amount Δ of the riskless asset, the new reserves of the riskless asset are $y' = y + \Delta$. The relationship between x' and y' is, again:

$$y' = K\Phi(\Phi^{-1}(1 - x') - \sigma\sqrt{\tau}) + k$$

We can inverse that relationship with some algebra and successive application of the inverse CDF and the CDF on both sides:

$$x' = 1 - \Phi\left(\Phi^{-1}\left(\frac{y + \gamma\Delta - k}{K}\right) + \sigma\sqrt{\tau}\right)$$

Such that the amount out to give to the trader is $\Delta' = x - x'$, and k needs to be appropriately updated as above.

Price of buying the risky asset

In the no fees case, we can obtain the price of buying the risky, ie swapping some amount in Δ of the riskless asset into the pool, using the procedure described in the previous section. The invariant equation for the trade:

$$y + \Delta - K\Phi(\Phi^{-1}(1 - (x - \Delta)) - \sigma\sqrt{\tau}) = k$$

We can then write the amount out Δ' as a function of the amount in Δ:

$$\Delta' = x - 1 + \Phi\left(\Phi^{-1}\left(\frac{y + \Delta - k}{K}\right) + \sigma\sqrt{\tau}\right)$$

And we obtain:

$$\frac{d\Delta'}{d\Delta} = g(\Delta) = \frac{1}{K}\phi\left(\Phi^{-1}\left(\frac{y + \Delta - k}{K}\right) + \sigma\sqrt{\tau}\right) \times (\Phi^{-1})'\left(\frac{y + \Delta - k}{K}\right)$$

With a fee regime of γ, that price becomes $\gamma g(\gamma\Delta)$.

Price of selling the risky asset

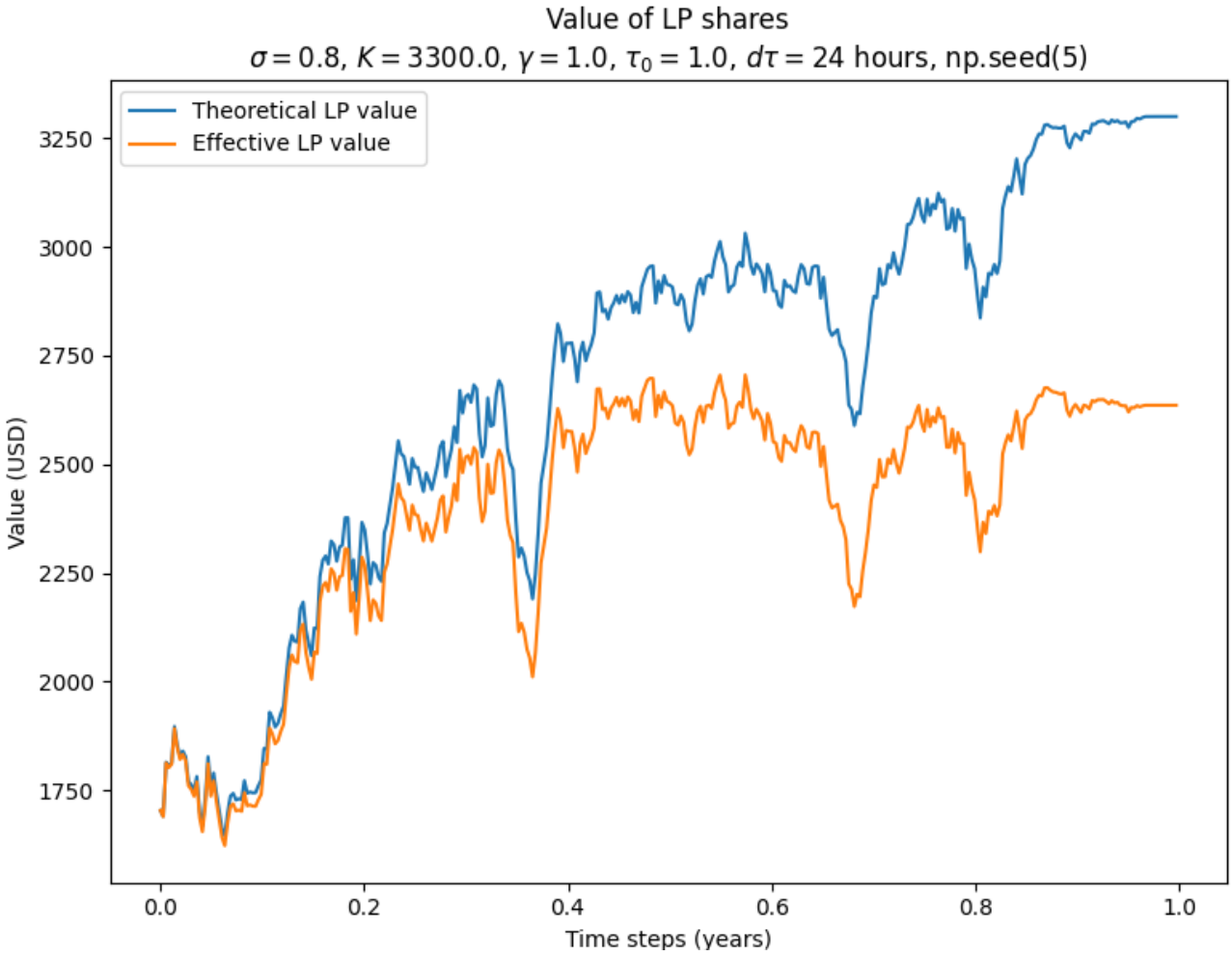
A similar procedure gives us when swapping some amount in Δ of the risky asset:

$$\frac{d\Delta'}{d\Delta} = g(\Delta) = K\phi(\Phi^{-1}(1 - x - \Delta) - \sigma\sqrt{\tau}) \times (\Phi^{-1})'(1 - x - \Delta)$$

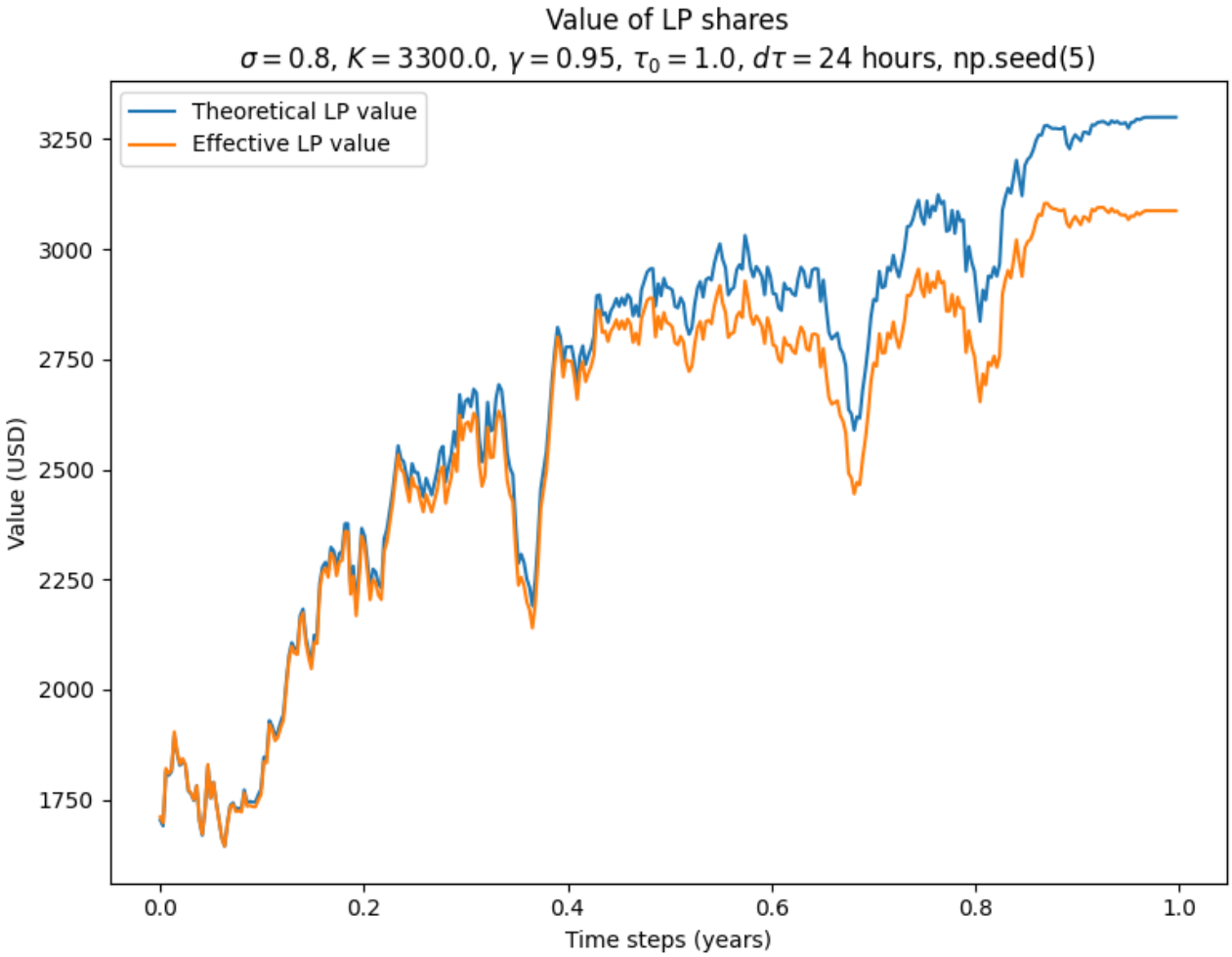
Fee optimization

In the case of the Covered Call CFMM, replication fails because of the increase in value of the instrument as we get closer to expiry. It was conjectured in the RMMS paper that implementing a fee might allow to recover the tau decay.

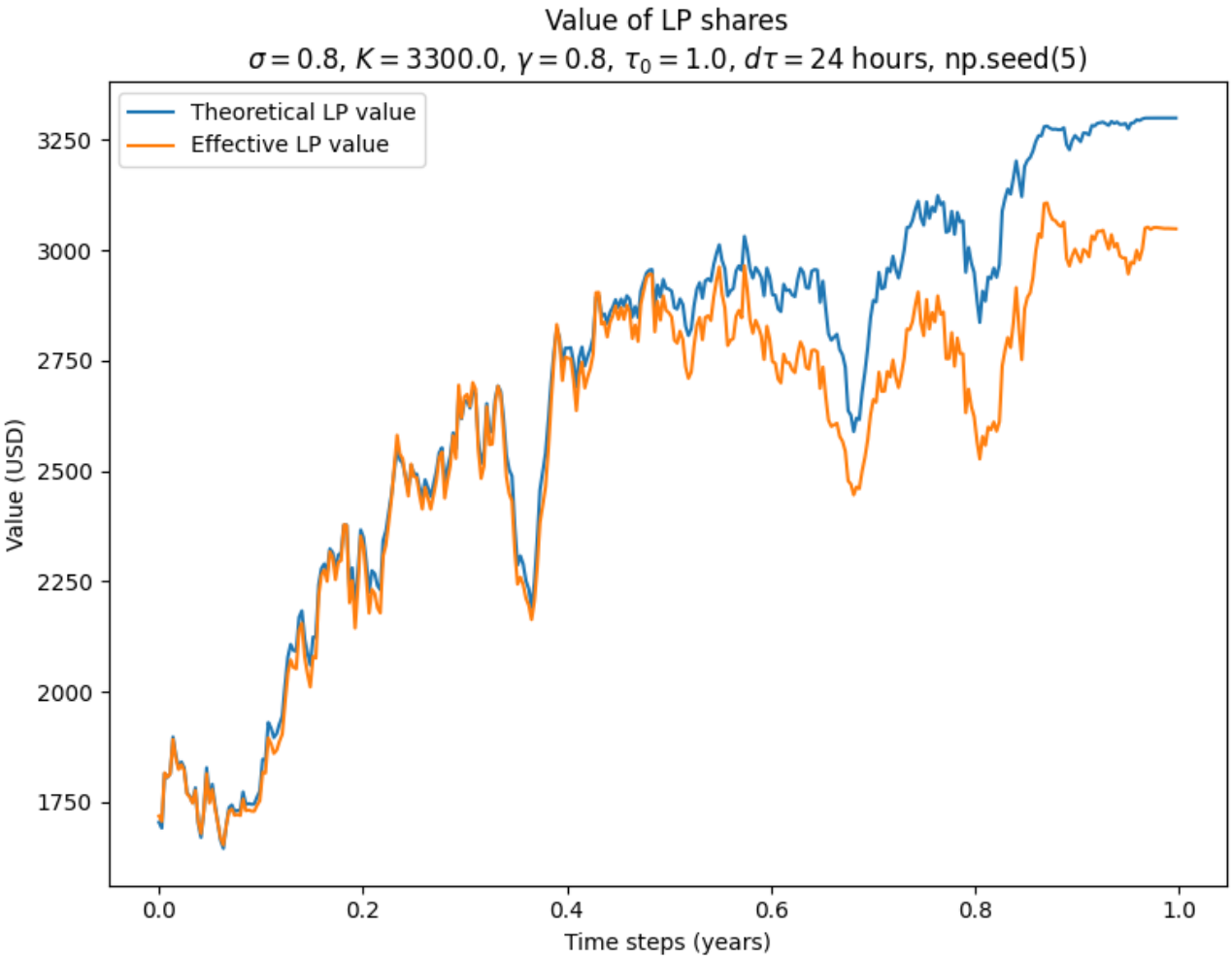
Below is the evolution of the payoff in the zero fee case with a 24 hours arbitrage period.



If we increase the fee to 5% in the same market conditions, we will get closer to the theoretical payoff as seen below:



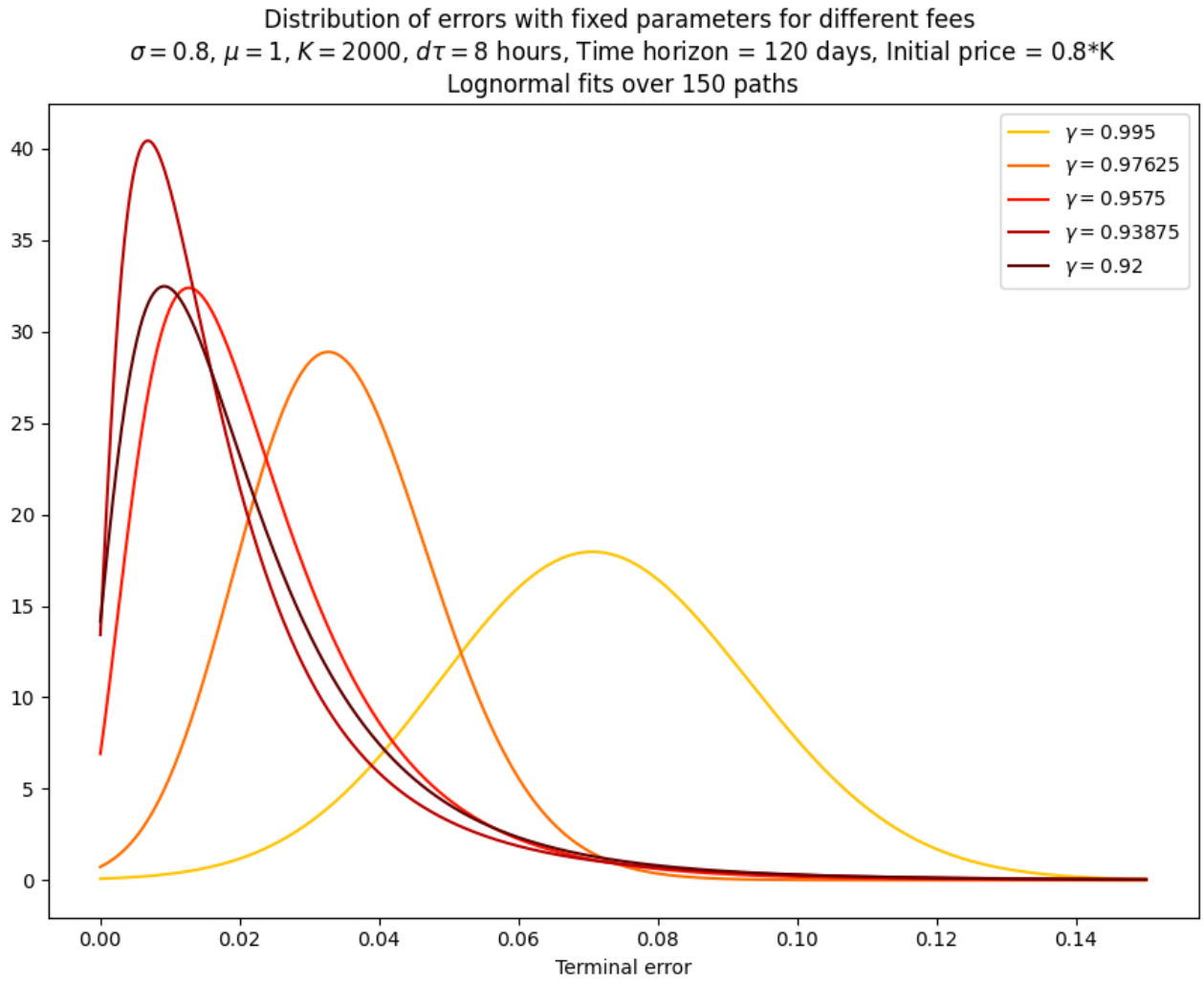
However, if we increase the fees too much, we will start going lower again because the price will spend too much time in the no arbitrage bound (which are made large because of the large fee), as seen below with an exaggerated fee of 20%.



In this case, we're only looking at a single price path. One might ask, given some parameters for the pool and some assumed market conditions, what is the fee that minimizes the error between the effective and theoretical LP values at the time of expiry? To do this, we need to generate a large number of price paths and average over them to get an expected error for each static fee chosen, and then run an optimization routine.

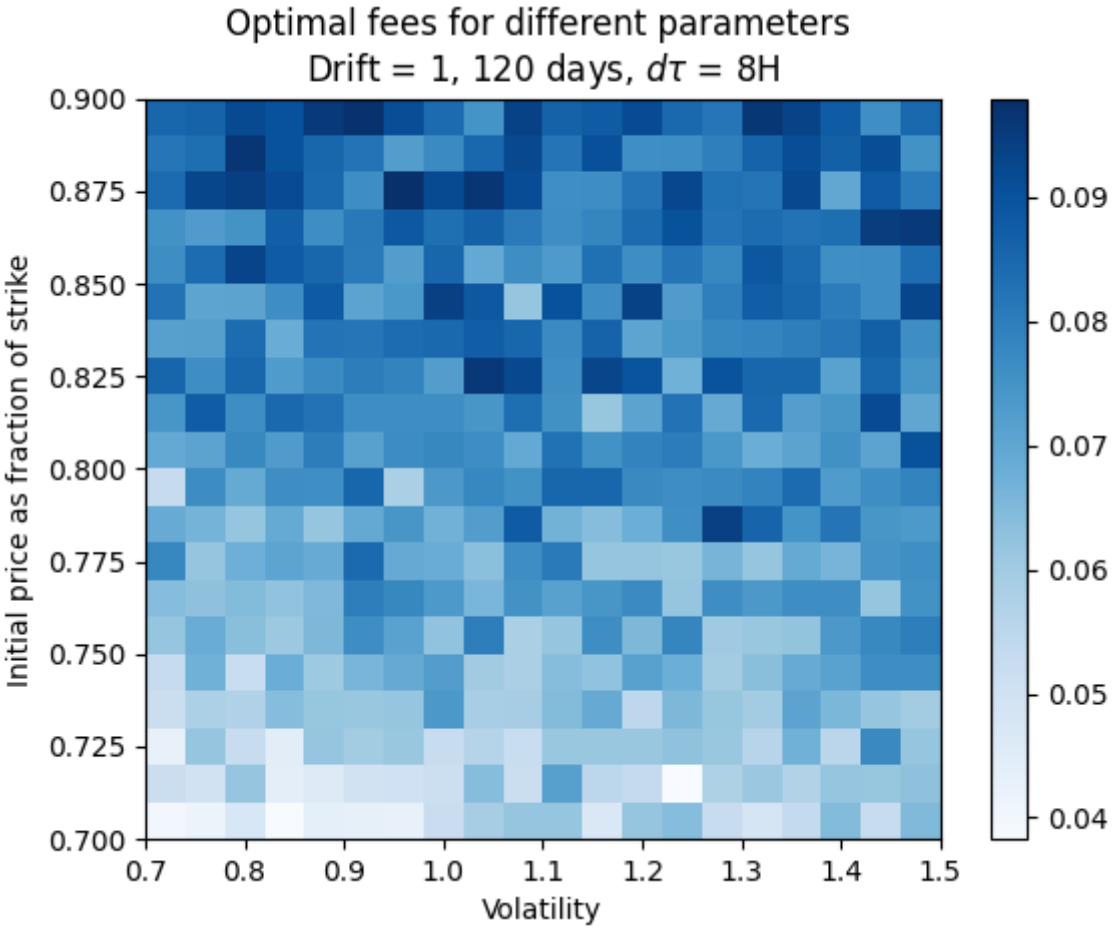
Of note is that the result should be highly dependent on arbitrage frequency. Indeed, if arbitrage happens more often, that means that more fees are accrued to the pool.

Below is a plot showing the log-normal distribution of errors given different static fees fitted from the data of 150 different paths.



We can see that as expected, the distributions are more skewed towards as the fee increases (i.e. γ decreases) until a certain point where the distribution seems to go back to higher errors (see deeper colored curve). Of note is also that above some threshold, the fitted distributions become hard to distinguish.

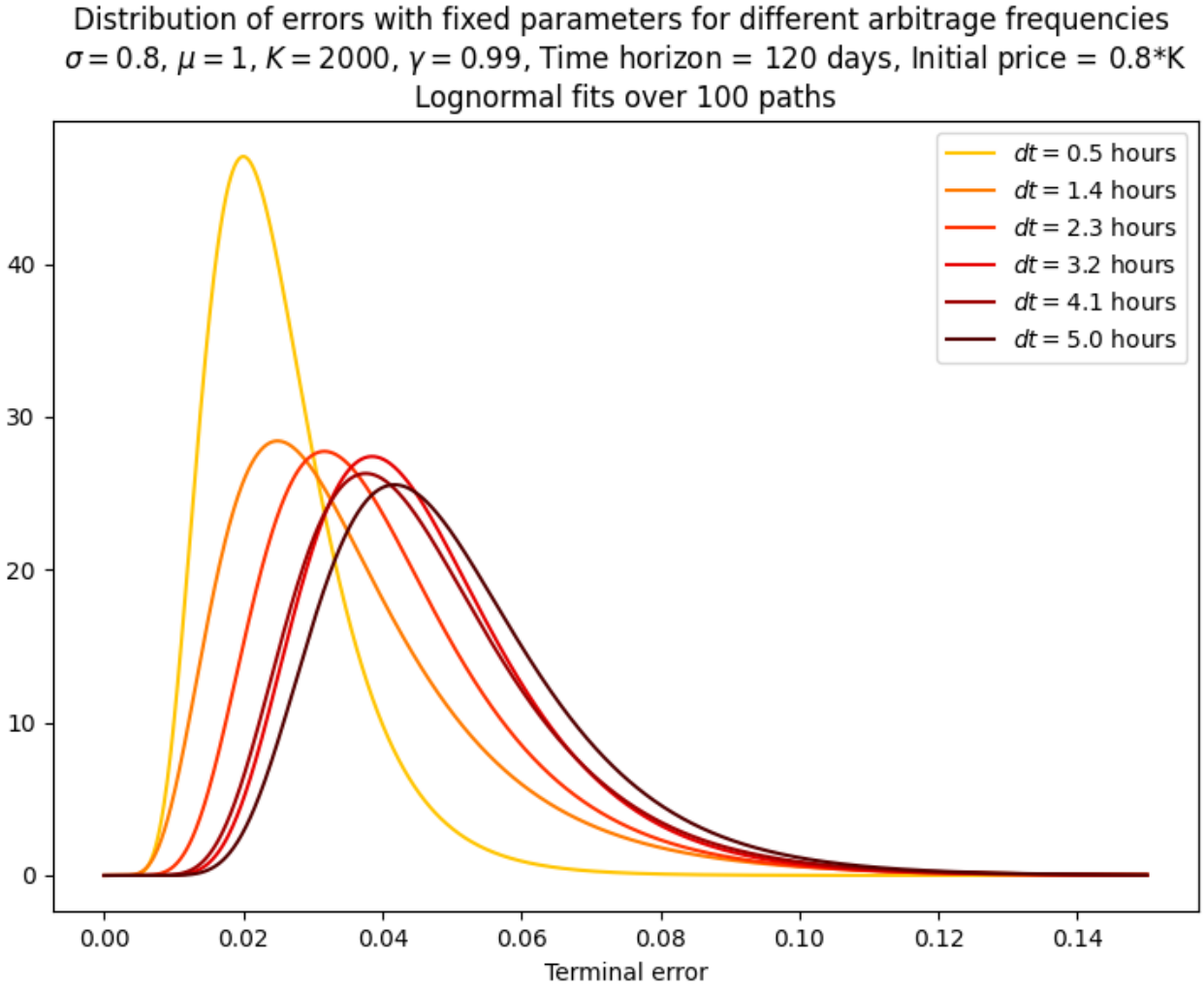
For each choice of time horizon, arbitrage frequency, it is possible to construct a mapping of (volatility, drift, strike price) to an optimal static fee that minimizes the expected error. The result of such an optimal fee search is given in the figure below:

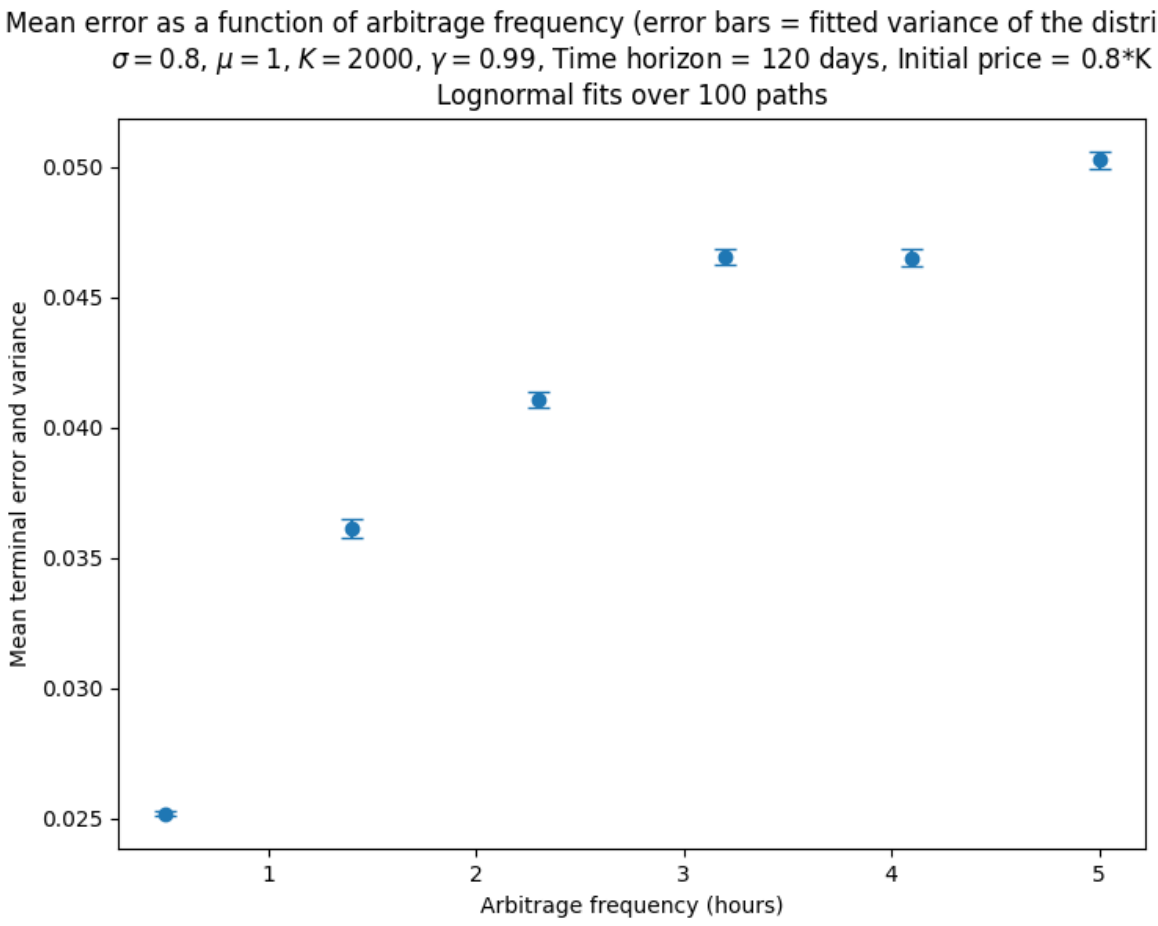


Unfortunately, this result is not quite as smooth as we might have expected. This is likely because, as discussed previously, it would appear that for a given choice of parameter, there is a *range* of static fees that produces a very similar error distribution. To resolve this problem and have a smoother mapping, one might significantly increase the number of paths used in the objetive function to determine the average error, such that the average of the runs would be very close to the actual mean and/or significantly decrease the tolerance search. Further analysis might be required to assess the rate of convergence of the mean of a sample to the "true" mean.

Effect of arbitrage frequency

The more frequently arbitrage is assumed to occur, the lower the average error given a fixed fee, and thus the lower the optial fee should be. This is demonstrated in the figures below where the distribution of error and mean errors are plotted for different arbitrage frequencies given a fixed fee of 1% in the same market conditions.


Distribution of errors with fixed parameters for different arbitrage frequencies
$\sigma = 0.8$, $\mu = 1$, $K = 2000$, $\gamma = 0.99$, Time horizon = 120 days, Initial price = 0.8*K
Lognormal fits over 100 paths
dt = 0.5 hours
dt = 1.4 hours
dt = 2.3 hours
dt = 3.2 hours
dt = 4.1 hours
dt = 5.0 hours
0
10
20
30
40
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
Terminal error


Mean error as a function of arbitrage frequency (error bars = fitted variance of the distribution)
$\sigma = 0.8$, $\mu = 1$, $K = 2000$, $\gamma = 0.99$, Time horizon = 120 days, Initial price = 0.8*K
Lognormal fits over 100 paths
Mean terminal error and variance
0.025
0.030
0.035
0.040
0.045
0.050
1
2
3
4
5
Arbitrage frequency (hours)

Implementation

`utils.py`: Utility functions used throughout

Contains simple utility functions such as the GBM generation algorithm, the derivative of the quantile function, or functions that allow to calculate reserves.

`cfmm.py`: CFMM pool implementation

The AMM pool is an object whose attributes are the reserves as well as all the other parameters that appear in the covered call trading function. The formulas outlined above are implemented using the `scipy.stats.norm` implementation of the normal distribution and related functions. To get the reserves that the AMM should be updated to (and thus deduce an amount out) given an amount in, the equations described in the theory section above are used. Whenever a price is returned by a function, it is denominated in riskless per risky.

```
class CoveredCallAMM():
    '''
    A class to represent a two-tokens AMM with the covered call trading func

    Attributes
    __________

    reserves_risky: float
        the reserves of the AMM pool in the risky asset
    reserves_riskless: float
        the reserves of the AMM pool in the riskless asset
    tau: float
        the time to maturity for this pool in the desired units
    K: float
        the strike price for this pool
    sigma: float
        the volatility for this pool, scaled to be consistent with the unit
    invariant: float
        the invariant of the CFMM
    '''

    def __init__(self, initial_x, K, sigma, tau, fee):
        '''
        Initialize the AMM pool with a starting risky asset reserve as an
        input, calculate the corresponding riskless asset reserve needed to
        satisfy the trading function equation.
        '''

    def getRisklessGivenRisky(self, risky):
        '''Get riskless reserves corresponding to the risky reserves with th

    def getRisklessGivenRiskyNoInvariant(self, risky):
        '''Get risky reserves corresponding to the current riskless reserves

    def getRiskyGivenRiskless(self, riskless):
```

```
def getRiskyGivenRiskless(self, riskless):
    '''Get riskless reserves corresponding to the riskless reserves witl

def swapAmountInRisky(self, amount_in):
    '''
    Swap in some amount of the risky asset and get some amount of the r

    Returns:

    amount_out: the amount to be given out to the trader
    effective_price_in_risky: the effective price of the executed trade
    '''

def virtualSwapAmountInRisky(self, amount_in):
    '''
    Perform a swap and then revert the state of the pool.

    Returns:

    amount_out: the amount that the trader would get out given the amou
    effective_price_in_riskless: the effective price the trader would pa
    '''

def swapAmountInRiskless(self, amount_in):
    '''
    Swap in some amount of the riskless asset and get some amount of the
    risky asset in return.

    Returns:

    amount_out: the amount to be given to the trader
    effective_price_in_riskless: the effective price the trader actuall
    '''

def virtualSwapAmountInRiskless(self, amount_in):
    '''
    Perform a swap and then revert the state of the pool.

    Returns:

    amount_out: the amount that the trader would get out given the amou
    effective_price_in_riskless: the effective price the trader would pa
    '''



def getSpotPrice(self):
    '''
    Get the current spot price (ie "reported price" using CFMM jargon) o
    the risky asset, denominated in the riskless asset, only exact in th
    no-fee case.
    '''

def getMarginalPriceSwapRiskyIn(self, amount_in):
    '''
    Returns the marginal price after a trade of size amount_in (in the
    risky asset) with the current reserves (in RISKLESS_RISKY_1)
```

```
        risky asset) with the current reserves (in RISKLESS.RISKY-1).
        See https://arxiv.org/pdf/2012.08040.pdf
        '''

    def getMarginalPriceSwapRisklessIn(self, amount_in):
        '''
        Returns the marginal price after a trade of size amount_in (in the
        riskless asset) with the current reserves (in RISKLESS.RISKY-1)
        See https://arxiv.org/pdf/2012.08040.pdf
        '''

    def getRiskyReservesGivenSpotPrice(self, S):
        '''
        Given some spot price S in the no-fee case, get the risky reserves
        spot price by solving the S = -y' = -f'(x) for x.
        '''
```

Arbitrager `arb.py`

This file provides a *function* that given a market price and a pool, acts on the pool to perform the exact trade that will bring the pool price in line with the reference market price.

```
def arbitrageExactly(market_price, Pool):
    '''
    Arbitrage the difference *exactly* at the time of the call to the funct

    Params:

    reference_price (float):
        the reference price of the risky asset, denominated in the riskless
    Pool (AMM object):
        an AMM object, for example a CoveredCallAMM class, with some curren
    '''
```

In this section, ε is always equal to 10^{-8}.

First we run a couple of checks from line 40 to 54. The intent here is to check whether the pools are almost empty or almost full to a precision of ε. If they are, the arbitrager does not perform any action in order to avoid having to deal with the kinks previously described.

The arbitrager first checks the marginal price of an ε swap of each asset. If the price differs from some reference price, they will try to find an optimal trade using the method described above.

In particular, if the price of selling the risky asset is above the market price to a precision of ε (line 58), the arbitrager looks for the amount of the risky asset that they should sell between ε and $1 - R_1 - \varepsilon$ (the maximum amount of the risky assets that we can add in so as to not) to bring the prices back in line, where R_1 are the risky reserves.

If the price buying the risky asset is below the market price to a precision of ε (line 75), the arbitrager looks for the amount of the riskless asset that they should swap in between ε and $\frac{K+k-R_2}{\gamma}$ to bring the prices back in line where R_2 are the riskless reserves.

`scipy.opitmize.brentq` is used to solve the root finding problem. The choice of this method is motivated by a benchmark that determined it is the fastest converging for this particular problem. After the problem is solved, the arbitrager checks that they are making a profit and if yes, executes the trade (lines 70 to 72 and 86 to 88).

Note on the bounds of the search:

With a negative invariant, the maximum amount of both the riskless and the risky goes down compared to K and 1 respectively because we're translating the curve downward by the value of the invariant.

In the case of looking for an amount of risky asset to swap in, all of the functions are still perfectly defined even when requesting a trade with an amount of risky in greater than those bounds and up to 1, it will just return a negative value of the riskless reserves. To take this into account, the `virtualSwapAmountInRisky` function checks whether the new reserves risky would be negative, and if yes returns 0 for the amount out to give to the trader. This makes sure that when checking for profit, the arbitrager always finds a negative profit and does not actually perform the trade.

In the case of looking for an amount of riskless asset to swap in, the `cfmm.getMarginalPriceSwapRisklessIn()` and `cfmm.getRiskyGivenRiskless()` are not defined for all values requested. In particular, the amount of riskless in appears as an argument in the following formulas.

```
norm.ppf((riskless - self.invariant)/self.K)
```

```
quantilePrime((R + gamma*amount_in - k)/K)
```

This is the reason why the amount of riskless in may never be greater than $\frac{K+k-R_2}{\gamma}$ as specified as the bounds for the root-finding algorithm, otherwise the `quantilePrime` function would not be defined.

Simulation routine `simulate.py`

The function `simulate()` takes in a Pool, a time array, and a corresponding GBM and runs the optimal arbitrage simulation under these conditions.

Every time step, this is what happens in the simulation:

1. Update the pool's τ.
2. Update the invariant k to reflect the change in τ (otherwise we get in the interior of the trading set and value can be extracted from the pool for free).
3. Run `arbitrageExactly()` on the pool. If a profitable optimal arbitrage is found, a swap occurs, which changes the reserves, and the invariant
4. Move to the next time step.

At every timestep, the value of the reserves in the pool *after arbitrage* and the theoretical desired payoff are recorded. At the end of the simulation, these are compared. The goal is for them to be as close as possible.

The simulation currently doesn't take into account any gas fee.

Fee optimization module `optimize_fee.py`

The `returnErrors()` function is simply a recasting of the `simulate()` function such that it returns only the mean error and terminal error given a number of parameters. It is used to define the objective function in the fee optimization routine.

`findOptimalFee()` is the actual fee optimization routine. It takes in all the parameters required to run a simulation except the fee regime, and returns the optimal fee for that regime. Within this routine, we define the objective function `ErrorFromFee()` which given a fee, runs a number of simulations with the given parameters and returns the average terminal error. These runs are parallelized using the `joblib` library as seen on line 50. To specify the number of paths one would like to average over, one should simply change the range of the for loop within the `Parallel` function call to the desired number of paths.

The optimal fee is found using the `scipy.optimize.fminbound` method. We look for an optimal fee between 0 and 10%.

Fee optimization script `optimal_fees_parallel.py` and data visualization `optimal_fee_visualization.py`

The above module is used in `optimal_fees_parallel.py` to construct a mapping of optimal fees such as the one discussed in the previous section. The user specifies a range of volatility, drift and an initial price as a fraction of the strike price. We then loop over all parameters to run `findOptimalFee()` with all of these parameters. The results are then recorded as an array in the JSON format in the `optimization_results` folder.

To visualize the results as a 2D color mapping, one needs to first go in `optimal_fee_visualization.py` and retrieve the `parameters` and `optimal_fees` arrays from the JSON. `x` is the range of volatility parameters and `y` the range of initial prices parameters. `drift_index` specifies the index of the drift value we want to plot the mapping for. The mapping is then plotted using Matplotlib's `imshow()`.

Observations regarding the behavior of the pool

Observation 1: negative invariant

Given a pool with set parameters and an initial invariant of 0, if τ is decreased while everything else remains equal and before any arbitrage occurs, the invariant will take on a negative value.

Explanation: as we update tau without changing the reserves, the invariant needs to be changed to keep the two sides of the equation equal. Graphically, decreasing tau would bring us to a higher curve, but because we haven't changed the reserves we're not

anymore in the reachable reserve set. Decreasing the value of the invariant translates the curve down so that we are again in the reachable reserve set.

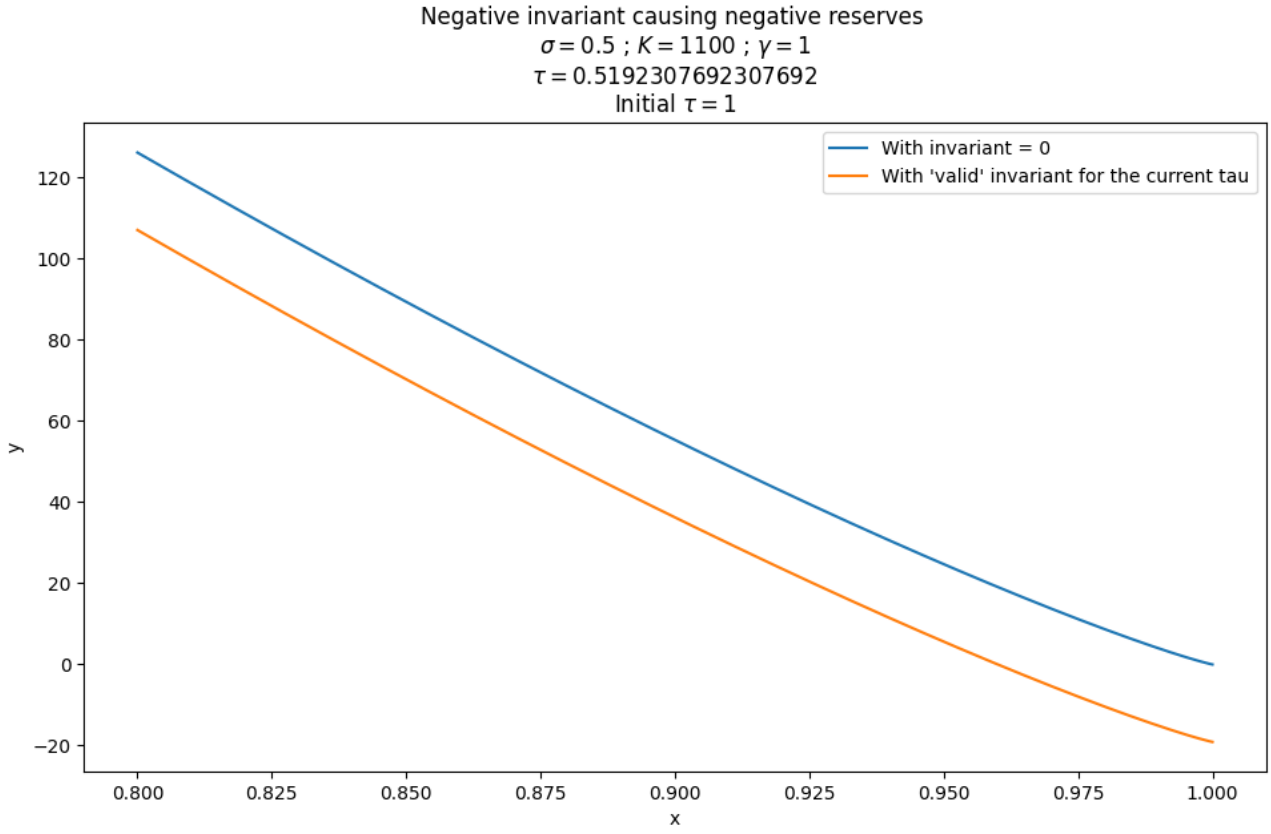
Observation 2: negative reserves

Similar to Observation 2, we start with a given fee-less pool, update τ, and then update k accordingly. We observe that on this new curve, it's some reported prices are associated with negative reserves of the riskless asset, which are practically unreachable. These are sometimes encountered in the simulations, especially for smaller values of τ where reasonable prices can get us exceedingly close to the right side of the curve. Because there is a negative k, which has the effect of translating the curve down, it may cross the x-axis. This means that some of the prices on the curve become practically inaccessible (reserves emptied at a higher price).

This is demonstrated in the test script on line 187 at commit `a6afd46` in the branch `use-config-file`. Below is a visual representation of this for parameters that were encountered in a simulation run.

Observation 3: fees don't always fill the theta gap

Even when choosing the optimal fee returned by the optimization routine, a static fee might still lead to a relatively large error for some price paths.



Appendix: Root finding benchmarking

Consider the following pieces of code at commit `a43263c` in `rmms-py` for terminal error comparison and runtime benchmarking:

Terminal error comparison:

```
import numpy as np
import time
import time_series
import cfmm
from simulate import simulate
fee = 0.05
strike = 2000
initial_price = 0.8*2000
volatility = 0.5
drift = 0.5
time_steps_size = 0.0027397260274
time_horizon = 1
initial_tau = 1
total_time = 0
np.random.seed(300)
Pool = cfmm.CoveredCallAMM(0.5, strike, volatility, initial_tau, fee)
t, gbm = time_series.generateGBM(time_horizon, drift, volatility, initial_pr
start = time.time()
_, _, _, d = simulate(Pool, t, gbm)
end = time.time()
print("RUNTIME: ", end-start)
print(d)
```

Runtime benchmarking:

```
import time
import time_series
import cfmm
from simulate import simulate
fee = 0.01
strike = 2000
initial_price = 0.8*2000
volatility = 0.5
drift = 0.5
time_steps_size = 0.0027397260274
time_horizon = 1
initial_tau = 1
total_time = 0
Pool = cfmm.CoveredCallAMM(0.5, strike, volatility, initial_tau, fee)
for i in range(100):
    t, gbm = time_series.generateGBM(time_horizon, drift, volatility, initia
    start = time.time()
    _, _, _, _ = simulate(Pool, t, gbm)
    end = time.time()
    total_time += end-start
print("Average runtime: ", total_time/100)
```

Let us do a bunch of comparisons in terminal error and runtime benchmarking.

SEED	d bisect	d illinois	d brentq	d brenth	d ridder	d toms748
15425	0.040279	0.040281	0.040280	0.040280	0.040280	0.040280
100	0.014655	0.014079	0.014079	0.014079	0.014079	0.014079
200	0.001378	0.001378	0.001378	0.001378	0.001378	0.001378
300	0.008522	0.008521	0.008522	0.008522	0.008522	0.008522

Table 1. Terminal error for different seeds and root finding algorithms.

Method	Average runtime (s)
bisect	6.3
illinois	0.95
brentq	0.564
brenth	0.577
ridder	0.325 - 0.611 (*)
toms478	0.567

Table 2. Average runtime for each method over 100 random price trajectories.

(*) average runtime over 100 price trajectories with the ridder method appears to be very variable for some reason.