Part 3:

Given:

- Reserves: x_t (BTC), y_t (USDC), constant product $k = x_t y_t$
- Pool mid price $P_t = \frac{y_t}{x_t}$ (USDC per BTC)
- Fee rate $\gamma \in (0,1)$
- If external price S_{t+1} exits the band, arbitragers trade until the post-trade boundary just contains S_{t+1}.

(a) Derive swap sizes $\Delta x, \Delta y$ and one-step revenue $R(S_{t+1})$

Case 1: $S_{t+1} > \frac{P_t}{1-\gamma}$ (BTC cheaper in pool)

Arbitragers swap USDC $\rightarrow$ BTC, so BTC leaves the pool and USDC enters the pool.

Updates (given):

$$x_{t+1} = x_t - \Delta x, \qquad y_{t+1} = y_t + (1-\gamma)\Delta y, \qquad \Delta x, \Delta y > 0.$$

Constant product must still hold:

$$x_{t+1} y_{t+1} = k$$

Boundary condition (given):

$$\frac{P_{t+1}}{1-\gamma} = S_{t+1}$$

But $P_{t+1} = \frac{y_{t+1}}{x_{t+1}}$, so:

$$\frac{1}{1-\gamma} \cdot \frac{y_{t+1}}{x_{t+1}} = S_{t+1} \longrightarrow \frac{y_{t+1}}{x_{t+1}} = S_{t+1}(1-\gamma)$$

Now solve for (x_{t+1}, y_{t+1}) using:

$$y_{t+1} = S_{t+1}(1-\gamma)x_{t+1}, \qquad x_{t+1}y_{t+1} = k.$$

Plugging in:

$$x_{t+1} \cdot S_{t+1}(1-\gamma)x_{t+1} = k \longrightarrow x_{t+1}^2 = \frac{k}{S_{t+1}(1-\gamma)} \longrightarrow x_{t+1} = \sqrt{\frac{k}{S_{t+1}(1-\gamma)}}$$

Then:

$$y_{t+1} = S_{t+1}(1-\gamma)x_{t+1} = \sqrt{kS_{t+1}(1-\gamma)}$$

Convert to swap sizes:

$$\Delta x = x_t - x_{t+1} = x_t - \sqrt{\frac{k}{S_{t+1}(1-\gamma)}}$$

And from $y_{t+1} = y_t + (1-\gamma)\Delta y$:

$$\Delta y = \frac{y_{t+1} - y_t}{1-\gamma} = \frac{\sqrt{kS_{t+1}(1-\gamma)} - y_t}{1-\gamma}$$

Fee revenue is from input asset y: $\boxed{\text{fee} = \gamma \Delta y}$

Case 2: $S_{t+1} < P_t(1-\gamma)$ (BTC cheaper outside)

Arbitragers swap BTC $\longrightarrow$ USDC, so BTC enters the pool and USDC leaves the pool.

Updates (given):

$$x_{t+1} = x_t + (1-\gamma)\Delta x, \qquad y_{t+1} = y_t - \Delta y, \qquad \Delta x \, \Delta y > 0$$

Constant product:

$$x_{t+1} y_{t+1} = k$$

Boundary condition (given):

$$P_{t+1}(1-\gamma) = S_{t+1} \longrightarrow \frac{y_{t+1}}{x_{t+1}} = \frac{S_{t+1}}{1-\gamma}$$

So:

$$y_{t+1} = \frac{S_{t+1}}{1-\gamma} x_{t+1}$$

Plug into constant product:

$$x_{t+1} \cdot \frac{S_{t+1}}{1-\gamma} x_{t+1} = k \longrightarrow x_{t+1} = \frac{k(1-\gamma)}{S_{t+1}} \longrightarrow x_{t+1} = \sqrt{\frac{k(1-\gamma)}{S_{t+1}}}$$

Then:

$$y_{t+1} = \sqrt{\frac{kS_{t+1}}{1-\gamma}}$$

Swap sizes:

From $x_{t+1} = x_t + (1-\gamma)\Delta x$:

$$\Delta x = \frac{x_{t+1} - x_t}{1-\gamma} = \frac{\sqrt{\frac{k(1-\gamma)}{S_{t+1}}} - x_t}{1-\gamma}$$

And:

$$\Delta y = y_t - y_{t+1} = y_t - \sqrt{\frac{kS_{t+1}}{1-\gamma}}$$

Fee revenue is in BTC: $\gamma \Delta x$. Converting to USDC at S_{t+1}:

$$\boxed{fee = \gamma \Delta x S_{t+1}}$$

Final one-step revenue function

Let $P_t = y_t / x_t$. Then:

$$\boxed{R(S_{t+1}) = 1\{S_{t+1} > \frac{P_t}{1-\gamma}\} \gamma \Delta y(S_{t+1}) + 1\{S_{t+1} < P_t(1-\gamma)\} \gamma \Delta x(S_{t+1}) S_{t+1}}$$

Where $\Delta y(\cdot)$ uses Case 1 formula and $\Delta x(\cdot)$ uses Case 2 formula above.

(b) $\mathbb{E}[R] = \int_{1/(1-\gamma)}^{\infty} \gamma\, \Delta y(s) f(s)\, ds + \int_{0}^{1-\gamma} \gamma\, \Delta x(s)\, s f(s)\, ds$

- $f(s)$ is the lognormal density with $\ln S_{t+1} \sim N(m, v)$
- $m = -\frac{1}{2}\sigma^2 \Delta t$
- $v = \sigma^2 \Delta t$

Since the integral does not have a closed form, I approximate it numerically using the trapezoidal rule. On a grid $S_0 < S_1 < \ldots < S_n$,

$$\int_a^b g(s)\, ds \approx \sum_{i=0}^{n-1} \frac{g(S_i) + g(S_{i+1})}{2} (S_{i+1} - S_i)$$

I truncate the upper limit at a large value capturing essentially all probality mass (e.g. ± 8 standard deviations in log-space.)

With $\Delta t = 1/365$, $\sigma = 0.2$, $\gamma = 0.003$, and a fine grid (40,000 points per region), the numerical result is:

$$\boxed{\mathbb{E}[R] \approx 0.008522036 \text{ USDC per one-step}}$$

(c) Using the trapezoidal approximation from part (b), I computed $\mathbb{E}[R]$ for each combination of:

$\sigma \in \{0.2, 0.6, 1.0\}, \quad \gamma \in \{0.001, 0.003, 0.01\}$

```
Part (c) table: E[R] (USDC per one-step)
----------------------------------------------------------------
 sigma |   gamma=0.001          gamma=0.003          gamma=0.01    best
----------------------------------------------------------------
   0.2 |   0.003685220      0.008522036      0.009430398   0.01
   0.6 |   0.011923375      0.032983290      0.081082357   0.01
   1.0 |   0.020060721      0.057383758      0.160689894   0.01
----------------------------------------------------------------
```

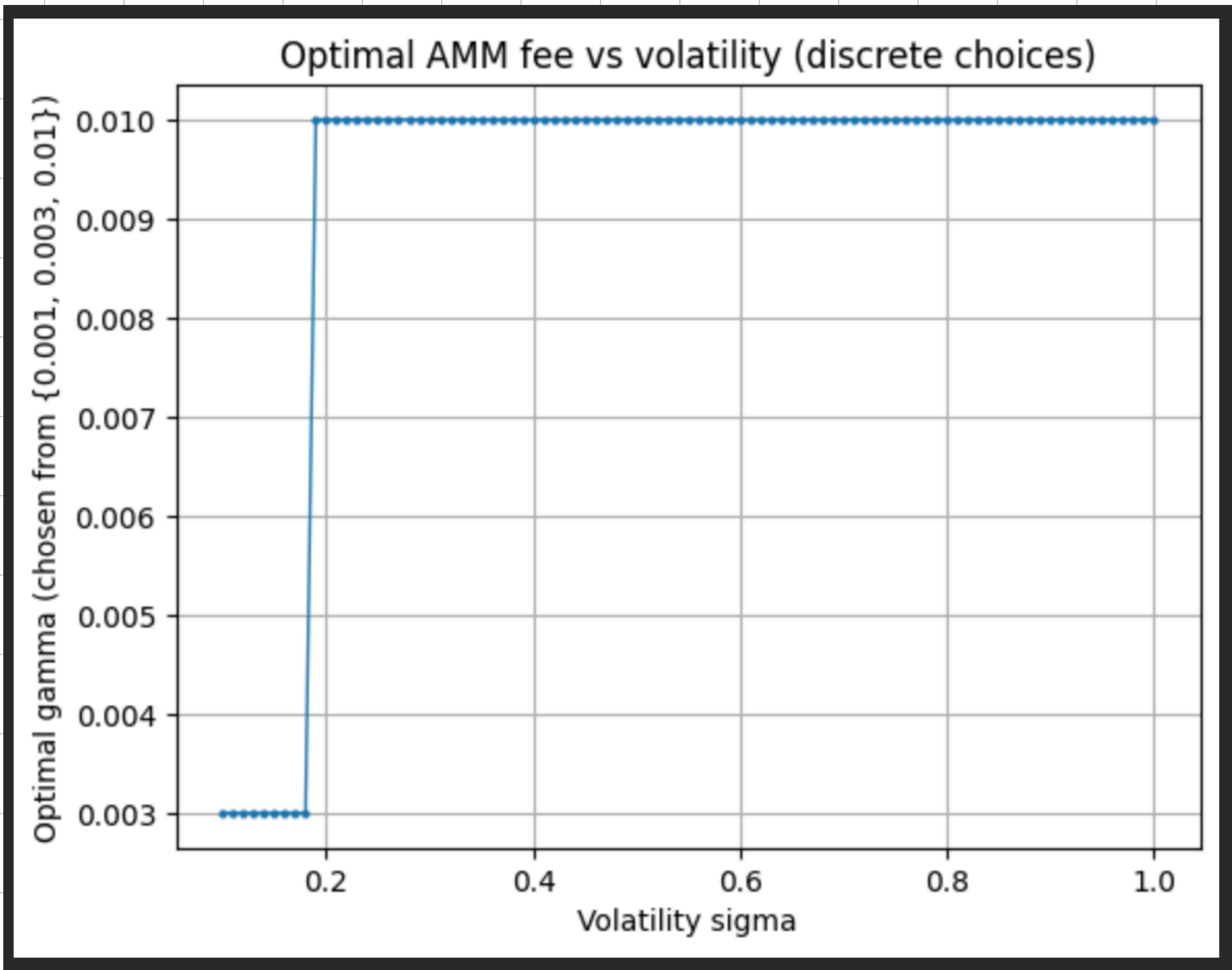
Therefore,

$$\gamma^*(\sigma) = \arg\max \, E[R(S_{t+1})]$$

yields:

$$\gamma^*(0.2) = 0.01, \quad \gamma^*(0.6) = 0.01, \quad \gamma^*(1.0) = 0.01$$

The highest fee (1%) generates the largest expected revenue at these volatility levels.



For very low volatility, the intermediate fee (0.003) can occasionally be optimal. However, once volatility increases beyond a modest level, the highest available fee (0.01) becomes optimal and remains so across the rest of the range.

As volatility increases, the external market price exits the no-arbitrage band more frequently and by larger magnitudes. This generates more arbitrage trading volume through the pool. Since arbitrage trades pay fees, higher volatility increases total fee-generating flow.

Within the discrete fee set considered, higher volatility therefore favors a higher fee rate. The optimal fee curve appears piecewise constant because γ is selected from only three discrete choices rather than being optimized continuously.

```
import numpy as np
import math
import matplotlib.pyplot as plt

# Problem 3 - AMM fee revenue using trapezoidal rule

# pool setup
x0 = 1000.0
y0 = 1000.0
k  = x0 * y0
P0 = y0 / x0

dt = 1.0 / 365.0  # one day


# lognormal pdf for S_{t+1}
# S_{t+1} = exp( -0.5*sigma^2*dt + sigma*sqrt(dt)*Z ), Z~N(0,1)
def lognormal_pdf(s, sigma):
    m = -0.5 * sigma**2 * dt
    v = sigma**2 * dt
    return (1.0 / (s * math.sqrt(2.0 * math.pi * v))) * math.exp(-
((math.log(s) - m) ** 2) / (2.0 * v))


# Case 1: S > P/(1-gamma) - BTC cheaper in pool, arbs swap USDC -> BTC
# from part (a): y' = sqrt(k * S * (1-gamma)), and y' = y + (1-gamma)*dy
def case1_dy(S, x, y, gamma, k):
    yprime = math.sqrt(k * S * (1.0 - gamma))
    dy = (yprime - y) / (1.0 - gamma)
    return dy


# Case 2: S < P*(1-gamma) - BTC cheaper outside, arbs swap BTC -> USDC
# from part (a): x' = sqrt(k*(1-gamma)/S), and x' = x + (1-gamma)*dx
def case2_dx(S, x, y, gamma, k):
    xprime = math.sqrt(k * (1.0 - gamma) / S)
    dx = (xprime - x) / (1.0 - gamma)
    return dx


# trapezoidal rule
def trapz(xs, vals):
    total = 0.0
    for i in range(len(xs) - 1):
        h = xs[i + 1] - xs[i]
        total += 0.5 * (vals[i] + vals[i + 1]) * h
```

```
    return total


# calculate E[R] using numerical integration
def expected_fee_revenue(sigma, gamma, x=x0, y=y0, n_grid=40000):
    k = x * y
    P = y / x

    # boundaries for no-arbitrage region
    s_low  = P * (1.0 - gamma)      # below this is Case 2
    s_high = P / (1.0 - gamma)      # above this is Case 1

    # integration bounds - truncate at +-8 std devs to cover most
probability
    m = -0.5 * sigma**2 * dt
    v = sigma**2 * dt
    z = 8.0

    s_min = math.exp(m - z * math.sqrt(v))
    s_max = math.exp(m + z * math.sqrt(v))

    # make sure bounds cover the thresholds
    s_min = min(s_min, 0.999 * s_low)
    s_max = max(s_max, 1.001 * s_high)

    # Left integral: integrate from s_min to s_low (Case 2)
    # fee is gamma*dx in BTC, convert to USDC by multiplying by s
    xs_left = np.linspace(s_min, s_low, n_grid)
    left_vals = np.zeros_like(xs_left)

    for i, s in enumerate(xs_left):
        f  = lognormal_pdf(s, sigma)
        dx = case2_dx(s, x, y, gamma, k)
        left_vals[i] = gamma * dx * s * f

    left_int = trapz(xs_left, left_vals)

    # Right integral: integrate from s_high to s_max (Case 1)
    # fee is gamma*dy in USDC already
    xs_right = np.linspace(s_high, s_max, n_grid)
    right_vals = np.zeros_like(xs_right)

    for i, s in enumerate(xs_right):
        f  = lognormal_pdf(s, sigma)
        dy = case1_dy(s, x, y, gamma, k)
```

```
        right_vals[i] = gamma * dy * f

    right_int = trapz(xs_right, right_vals)

    return left_int + right_int


if __name__ == "__main__":

    # Part (b)
    sigma_b = 0.2
    gamma_b = 0.003  # 30 bps

    ER_b = expected_fee_revenue(sigma_b, gamma_b, n_grid=40000)
    print("Part (b)")
    print(f"sigma = {sigma_b}, gamma = {gamma_b}")
    print(f"E[R] = {ER_b:.9f} USDC")
    print()

    # Part (c) - compute table
    sigmas = [0.2, 0.6, 1.0]
    gammas = [0.001, 0.003, 0.01]

    print("Part (c) - E[R] for different sigma and gamma")
    print("---------------------------------------------------------------
")
    print(" sigma |   gamma=0.001         gamma=0.003         gamma=0.01  
best")
    print("---------------------------------------------------------------
")

    for s in sigmas:
        ers = []
        for g in gammas:
            ers.append(expected_fee_revenue(s, g, n_grid=30000))
        best_g = gammas[int(np.argmax(ers))]
        print(f" {s:>4.1f} |   {ers[0]:>12.9f}     {ers[1]:>12.9f}
{ers[2]:>12.9f}   {best_g}")

    print("---------------------------------------------------------------
")
    print()

    # Part (c) - plot optimal gamma vs sigma
    sigma_grid = np.round(np.arange(0.10, 1.00 + 0.01, 0.01), 2)
```

```
    opt_gamma = []

    for s in sigma_grid:
        ers = [expected_fee_revenue(float(s), g, n_grid=7000) for g in gammas]
        opt_gamma.append(gammas[int(np.argmax(ers))])

    plt.figure()
    plt.plot(sigma_grid, opt_gamma, marker="o", markersize=2, linewidth=1)
    plt.xlabel("Volatility sigma")
    plt.ylabel("Optimal gamma")
    plt.title("Optimal fee rate vs volatility")
    plt.grid(True)
    plt.savefig('part3c_optimal_gamma.png', dpi=150, bbox_inches='tight')
    plt.show()

    print("Saved plot to part3c_optimal_gamma.png")
```