Case 6: LIBOR Market Model

Asha de Meij - i6254733

December 5, 2024

Part 1: Discount bond prices

Question 1. Compute prices of discount bonds (at $t = 0$) for all maturities $T_1, \ldots, T_9$, using the flat initial term-structure at $t = 0$. (Remember to use **discrete compounding**, consistent with the definition of the forward LIBOR rates.)

To compute the prices of discount bonds at $t = 0$ for all maturities $T_1, \ldots, T_9$, I implemented the following lines of code:

```
maturities = np.arange(1, N + 1)
discount_bond_prices = [1 / ((1 + dT * f0)**t) for t in maturities]
```

here price for the next maturity differs from the previous maturity by the following factor:

$$\frac{1}{1 + \Delta T \cdot f_0}$$

This shows that the discount bond prices decrease as the maturity increases. The resulting prices are shown in the following table:

Maturity (T)	**Discount Bond Price (P(0, T))**
1	0.970874
2	0.942596
3	0.915142
4	0.888487
5	0.862609
6	0.837484
7	0.813092
8	0.789409
9	0.766417

Table 1: Discount Bond Prices for Maturities $T_1, \ldots, T_9$

Question 2. Build a Monte-Carlo simulation for a one-factor LMM using the *"final"* discount bond $D_{10}(t)$ as the numéraire and using the SDEs for the $f_i(t)$'s under the measure $\mathbb{Q}^{10}$.

```
f_t = np.full((numSim, T), f0)

# MC simulation of forward rates under measure Q^10
for step in range(1, numSteps + 1):
    dt = delta_T / numSteps
    Wt = np.random.normal(0, np.sqrt(dt), (numSim, T))

    for i in range(T - 1):
        drift = np.zeros(numSim)
        for k in range(i + 1, T):
            drift += (delta_T * sigma * f_t[:, k]) / (1 + delta_T * f_t[:, k])

        f_t[:, i] += drift * dt + sigma * f_t[:, i] * Wt[:, i]
```

Question 3. Calculate the Monte-Carlo values for all the discount bonds and confirm that the MC-values are consistent with the initial term-structure. (This is a check that all drift-corrections are correctly implemented.)

```
# discount bond prices from simulated rates
discountBondPrice = np.ones((numSim, T))
for i in range(T):
    discountBondPrice[:, i] = np.cumprod(1 / (1 + dT * f_t[:, :i + 1]), axis=1)[:, -1]

discountBondPrice_avg = np.mean(discountBondPrice, axis=0)
discountBondPrice_std = np.std(discountBondPrice, axis=0)
discountBondPrice_se = discountBondPrice_std / np.sqrt(numSim)
```

The values in the table below were computed using the code above and by setting numSim=1000000. From the table below, we can see that the standard errors are small but increase with time-to-maturity, reflecting the compounding uncertainty.

Maturity (T)	Discount Bond Price	Standard Error
1	0.973396	0.0000095
2	0.947242	0.0000130
3	0.921537	0.0000155
4	0.896277	0.0000173
5	0.871475	0.0000188
6	0.847112	0.0000200
7	0.823206	0.0000209
8	0.799745	0.0000217
9	0.776755	0.0000223
10	0.754131	0.0000216

Table 2: Monte Carlo discount bond prices and standard errors.

Furthermore, the table below compares the prices computed by Monte Carlo to the values computed in Exercise 1.

The small differences indicate that the Monte Carlo values are very close to the initial term-structure values, suggesting that the drift corrections were correctly implemented.

Maturity (T)	Initial Price	MC Price	Standard Error	Difference
1	0.970874	0.973396	0.0000095	0.002522
2	0.942596	0.947242	0.0000130	0.004646
3	0.915142	0.921537	0.0000155	0.006395
4	0.888487	0.896277	0.0000173	0.007790
5	0.862609	0.871475	0.0000188	0.008866
6	0.837484	0.847112	0.0000200	0.009628
7	0.813092	0.823206	0.0000209	0.010115
8	0.789409	0.799745	0.0000217	0.010335
9	0.766417	0.776755	0.0000223	0.010338
10	0.744094	0.754131	0.0000216	0.010037

Table 3: Comparison of Monte Carlo prices with the initial term structure.

Part 2: Gaussian swaption formulas

Question 4. Compute prices of payer swaptions for all option maturities $T_1, \ldots, T_9$ and all strikes $K = 1\%, 2\%, \ldots, 5\%$. All swaptions have the same end-dates T_{10} for all the underlying swaps. Use the Gaussian option pricing formula for the payer swaptions, using Gaussian volatility $\sigma = 0.01$ for all the par swap-rate processes $y_{T_i T_{10}}(t)$.

To answer this question, I used the formulas listed below which were provided in slide 21.

$$V_{\text{pay\_swptn},t} = L \left(\sum_{k=1}^{N} D_{T_k,t} \right) \left[(y_{T_0 T_N,t} - K)\, N(d) + \sigma \sqrt{T_0 - t}\, \phi(d) \right]$$

with

$$y_{T_0 T_N,t} = \frac{D_{T_0,t} - D_{T_N,t}}{\sum_{k=1}^{N} D_{T_k,t}}$$

$$d = \frac{y_{T_0 T_N,t} - K}{\sigma \sqrt{T_0 - t}}$$

$$\phi(d) = \frac{e^{-\frac{1}{2}d^2}}{\sqrt{2\pi}}$$

These formulas were then implemented in Python to compute the prices of payer swaptions for all option maturities $T_1, \ldots, T_9$ and all strikes $K = 1\%, 2\%, \ldots, 5\%$, the implementation in python can be seen on the figure below.

```python
def discountBond_prices(f0, N, dT):
    maturities = np.arange(1, N + 1) * dT
    return (1 / (1 + f0)) ** maturities

def parSwapRates(D, N, dT):
    par_swap_rates = []
    annuities = []

    for i in range(1, N):
        sum_D = np.sum(D[i:])
        y_Ti_T10 = (D[i - 1] - D[-1]) / sum_D

        par_swap_rates.append(y_Ti_T10)
        annuities.append(sum_D)

    return np.array(par_swap_rates), np.array(annuities)

def gaussianSwaption_prices(par_swap_rates, annuities, strikes, sigma, N, dT):
    option_maturities = np.arange(1, N) * dT
    swaption_data = []

    for maturity_index, maturity in enumerate(option_maturities):
        annuity = annuities[maturity_index]
        par_swap_rate = par_swap_rates[maturity_index]

        for strike in strikes:
            # Calculate Gaussian parameters
            d = (par_swap_rate - strike) / (sigma * np.sqrt(maturity))

            # Gaussian swaption price
            swaption_price = annuity * (
                (par_swap_rate - strike) * norm.cdf(d)
                + sigma * np.sqrt(maturity) * norm.pdf(d)
            )

            swaption_data.append({
                'Option Maturity (Years)': maturity,
                'Strike (%)': strike * 100,
                'Gaussian Swaption Price': swaption_price
            })

    return pd.DataFrame(swaption_data)
```

The computed prices of the payer swaps (notional amount = 1) for all option maturities $T_1, \ldots, T_9$ and all strikes $K = 1\%, 2\%, \ldots, 5\%$ are shown in the table below.

Table 4: Gaussian Payer Swaption Prices

Maturity (T)	1%	2%	3%	4%	5%
1	0.1518284212	0.0818913811	0.0301573597	0.0062980906	0.0006418401
2	0.1356598717	0.0793770587	0.0373309192	0.0132097273	0.0033252089
3	0.1201069975	0.0742950175	0.0393973300	0.0172791027	0.0060751679
4	0.1042822099	0.0671713541	0.0384030172	0.0190403097	0.0080201212
5	0.0879725490	0.0584598576	0.0352408728	0.0189549011	0.0089626361
6	0.0711628744	0.0484859719	0.0304205037	0.0173558580	0.0089026466
7	0.0538984852	0.0374889284	0.0242757018	0.0144897296	0.0079000876
8	0.0362414170	0.0256511493	0.0170442875	0.0105460428	0.0060312040
9	0.0182552946	0.0131162055	0.0089055516	0.0056752664	0.0033734163

Question 5. Use your MC simulation of the LMM to calculate the value of all the swaptions in the LMM. How accurate are the Gaussian swaption-prices as an approximation to the *"true"* swaption values in the LMM?

```python
def simulatedParams(f_t_num, T_i, T):

    discountBond = np.array([
        1 / ((1 + f_t_num[:, i]) ** (i + 1)) for i in range(T)
    ]).T

    par_swap_rate = (
        discountBond[:, T_i - 1] - discountBond[:, T - 1]
    ) / np.sum(discountBond[:, T_i - 1:], axis=1)

    return discountBond, par_swap_rate

def monte_carlo_swaption_pricing(numSim, f0, strike_rates, option_maturities, T, sigma, dT):

    f_t_num = np.zeros((numSim, T))
    f_t_num[:, 0] = f0

    Wt = np.random.normal(0, np.sqrt(dT), (numSim, T))

    for i in range(1, T):
        drift_Updated = -sigma**2 * (T - i) / 2
        f_t_num[:, i] = (
            f_t_num[:, i - 1]
            + drift_Updated * dT
            + sigma * Wt[:, i]
        )
    f_t_num = np.maximum(f_t_num, 0)

    # Calculate initial discount factors for all maturities
    discountBond_init = np.array([1 / ((1 + f0) ** i) for i in range(1, T + 1)])

    # Monte Carlo pricing of swaptions
    swaptionPrices = []
    for K in strike_rates:
        pricesStrike = []

        #Simulation of all params
        for T_i in option_maturities:

            discountBond, par_swap_rate = simulatedParams(f_t_num, T_i, T)

            swaptionPayoffs = (np.maximum(par_swap_rate - K, 0) * np.sum(discountBond[:, T_i - 1:], axis=1)
                               / discountBond_init[T_i - 1])

            swaptionAVG = np.mean(swaptionPayoffs) * discountBond_init[T_i - 1]

            pricesStrike.append(swaptionAVG)
        swaptionPrices.append(pricesStrike)

    mc_swap = pd.DataFrame(
        swaptionPrices,
        index=[f"{int(K * 100)}%" for K in strike_rates],
        columns=[f"T{i}" for i in option_maturities]
    )
    mc_swap.index.name = "Strike Rate"
    mc_swap.columns.name = "Option Maturity"

    return mc_swap
```

The code gives us the following results.

Monte Carlo Swaption Prices:

Option Maturity	T1	T2	T3	T4	T5	T6	T7	T8	T9
Strike Rate									
1%	0.151076	0.137161	0.122846	0.108160	0.093055	0.077642	0.061748	0.045216	0.027184
2%	0.099795	0.093221	0.085569	0.076994	0.067574	0.057474	0.046542	0.034611	0.020871
3%	0.060456	0.059023	0.056158	0.052115	0.047011	0.041016	0.033992	0.025782	0.015628
4%	0.033027	0.034409	0.034448	0.033333	0.031179	0.028115	0.024003	0.018659	0.011394
5%	0.015953	0.018245	0.019582	0.020018	0.019630	0.018445	0.016338	0.013099	0.008074

Monte Carlo Standard Errors:

Option Maturity	T1	T2	T3	T4	T5	T6	T7	T8	T9
Strike Rate									
1%	0.000148	0.000139	0.000128	0.000116	0.000104	0.000090	0.000075	0.000059	0.000039
2%	0.000123	0.000117	0.000109	0.000100	0.000089	0.000078	0.000066	0.000052	0.000034
3%	0.000096	0.000093	0.000088	0.000082	0.000074	0.000066	0.000056	0.000044	0.000030
4%	0.000069	0.000070	0.000068	0.000065	0.000060	0.000054	0.000046	0.000037	0.000025
5%	0.000046	0.000049	0.000050	0.000049	0.000047	0.000043	0.000038	0.000031	0.000021

The Gaussian swaption prices computed in Exercise 4 again very closely align with the swap-

tion prices obtained through the Monte Carlo simulation with numSim=100000, demonstrating strong consistency between the two methods.
The Gaussian swaption pricing formula assumes constant volatility and ignores key market dynamics. In contrast, the LMM is more realistic for multiple reasons. Such as Modeling forward rates as lognormal, accounting for correlations between forward rates as well as using variable volatility that depends on time and maturity. Thus, the Gaussian model simplifies volatility and ignores term-structure effects, whereas the LMM captures these complexities, explaining the differences in swaption prices.