Implementation of the LIBOR market model

05.11.2018

Abstract

In this essay we implement the LIBOR market model using a Monte Carlo method and use this implementation to price two interest rate products, a cap and a ratchet floater. The implementation is done using object-oriented design in the programming language Python 3 with the package NumPy. Although the focus of this essay is on the implementation, we start by introducing the theoretical background of the LIBOR market model.

Contents

1. Theoretical Background

In this section we introduce the theoretical framework of the LIBOR market model. We will start with the Black-76 model for pricing caplets, which not only provides a motivation for the formulation of the LIBOR market model but is also important when calibrating the model to market data. Afterwards we introduce the actual LIBOR market model, which we will implement in the next section through a Monte Carlo method. In preparation for this, this section ends with a discussion about the choice of the correlation and volatility structure used in the LIBOR market model.

Notation: Before we begin, let us fix some notation for the rest of this essay. A forward LIBOR rate at time t for the accrual period $[S,T]$ with $t \leq S < T$ is denoted by $L(t;S,T)$. If $t = S$, $L(S;S,T)$ is a spot rate and will also be denoted by $L(S,T)$. A zero-coupon bond of maturity T (also called T-bond) is denoted by $P(t,T)$. We will also use the notation $(\,\cdot\,)^+ = \max(\,\cdot\,,0)$. As always we implicitly assume that we are working on a filtered probability space $(\Omega, \mathcal{F}, \mathbb{P}, \{\mathcal{F}_t\})$.

1.1. Black-76 model

This subsection is based on [Bjö09, Chapter 27]. We consider an accrual period $[T_0,T_1]$ and remember that a caplet with strike K, notional N and maturity T_1 is an European call option on the spot rate $L(T_0;T_0,T_1)$ with payoff

$$\textbf{Caplet}(T_1) = \tau N(L(T_0;T_0,T_1) - K)^+, \quad \text{where} \;\; \tau = T_1 - T_0. \tag{1}$$

We note that although the payoff happens at time T_1, the LIBOR rate is already fixed at time T_0, i. e. after the reset date T_0 the value of the caplet becomes deterministic.

To obtain the value for time $t \leq T_0$, we use the risk-neutral pricing formula with the T_1-bond as numeraire, which gives us the T_1-forward measure $\mathbb{Q}_{T_1}$. Using this, we can write

$$\textbf{Caplet}(t) = P(t,T_1)\mathbb{E}^{Q_{T_1}}\left[\tau N(L(T_0;T_0,T_1) - K)^+ \mid \mathcal{F}_t\right]. \tag{2}$$

To evaluate this expression, we need to make an assumption about the dynamics of the underlying. The Black-76 model assumes a log-normal process, i. e.

$$\mathrm{d}L(t;T_0,T_1) = \sigma L(t;T_0,T_1)\mathrm{d}W^{\mathbb{Q}_{T_1}}(t), \tag{3}$$

where σ, a constant, denotes the volatility, the derivative is taken with respect to t and $W^{\mathbb{Q}_{T_1}}(t)$ denotes a Brownian motion under the measure $\mathbb{Q}_{T_1}$.

It can now be shown that in this case we get

$$\textbf{Caplet}^{Black76}(t) = P(t,T_1)\tau N\left(L(t;T_0,T_1)\mathcal{N}(d_1) - K\mathcal{N}(d_2)\right), \tag{4}$$

where $\mathcal{N}(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{t} \mathrm{e}^{-\frac{1}{2}x^2} \mathrm{d}x$, i. e. the cumulative distribution function of the standard normal distribution, and d_1 and d_2 are

$$d_1 = \frac{\log\left(\frac{L(t;T_0,T_1)}{K}\right) + \frac{1}{2}\sigma^2(T_0 - t)}{\sigma\sqrt{T_0 - t}} \quad \text{and} \quad d_- = d_2 - \sigma\sqrt{T_0 - t}. \tag{5}$$

1.2. LIBOR market model

This subsection is again based on [Bjö09, Chapter 27]. The basic idea behind a market model is to model interest rates directly observable in the market,[1] which are the forward LIBOR rates in the case of the LIBOR market model.

We consider a set of dates $0 \leq T_0 < T_1 < \cdots < T_n$. For each accrual period $[T_{i-1}, T_i]$, $i = 1, \ldots, n$, we want to model the forward LIBOR rate $L(t; T_{i-1}, T_i)$. For convenience, we denote $L(t; T_{i-1}, T_i)$ as $L_i(t)$. Similar to the Black-76 model the LIBOR market model assumes a log-normal process for each $L_i(t)$ under the T_i-forward measure $\mathbb{Q}_{T_i}$, i. e.

$$\mathrm{d}L_i(t) = \sigma_i(t) L_i(t) \mathrm{d}W_i^{\mathbb{Q}_{T_i}}(t), \quad i = 1, \ldots, n, \tag{6}$$

where $\sigma_i(t)$, a deterministic function, is the *instantaneous volatility* and $W_i^{\mathbb{Q}_{T_i}}(t)$ a Brownian motion under $\mathbb{Q}_{T_i}$. Furthermore, we assume that the Brownian motions are correlated via

$$\mathrm{d}\left\langle W_i^{\mathbb{Q}_{T_i}}(t), W_j^{\mathbb{Q}_{T_j}}(t) \right\rangle = \rho_{ij} \mathrm{d}t, \tag{7}$$

where $\rho = (\rho_{ij})$, a constant matrix, is the *instantaneous correlation*.

We see that the difference between the dynamics of a forward rate in the Black-76 model and the LIBOR market model is the time-dependence of the volatility. We will come back to this in Subsec. 1.3.1. In this sense, the LIBOR market model can be viewed as a set of correlated Black-76 like models.

In general, the payoff of interest rate products can depend on all forward rates $L_i(t)$, $i = 1, \ldots, n$. When pricing such products via the LIBOR market model using a Monte Carlo method, we have to evaluate the expectation of such a payoff. This requires to express the dynamics in Eq. (6) of all forward rates under a common measure (the measure used in the expectation). A popular choice is the measure $\mathbb{Q}_{T_n}$, which in this context is also called the *terminal measure*. Using the change of numeraire technique, it can be shown that under $\mathbb{Q}_{T_n}$ Eq. (6) becomes

$$\mathrm{d}L_i(t) = L_i(t)\mu_i(t)\mathrm{d}t + \sigma_i(t) L_i(t) \mathrm{d}W_i^{\mathbb{Q}_{T_n}}(t), \quad i = 1, \ldots, n, \tag{8}$$

[1] As opposed to instantaneous short rates in short rate models or instantaneous forward rates in the HJM framework, which are mathematical abstractions.

with

$$\mu_i(t) = -\sigma_i(t) \sum_{m=i+1}^{n} \frac{\sigma_m(t)\tau_m L_m(t)}{1+\tau_m L_m(t)} \rho_{im}, \quad \text{where } \tau_m = T_m - T_{m-1}. \tag{9}$$

We note that under the new measure, the (constant) correlation matrix ρ between the Brownian motions stays the same, i. e. Eq (7) is still valid under $\mathbb{Q}_{T_n}$.

1.3. Choice of volatility and correlation

In the previous section we assumed the instantaneous volatilities $\sigma_i(t)$ and the instantaneous correlation ρ as given. Before using or implementing the LIBOR market model, we need to specify both. There are multiple choices for each, each having different advantages and disadvantages, and in this sense the LIBOR market model can be regarded as a framework rather than a model.

Depending on the choice of $\sigma_i(t)$ and ρ, there will be a different number of free parameters which can be used to calibrate the model to market data. In the following we introduce and discuss our choices for $\sigma_i(t)$ and ρ.

1.3.1. Instantaneous volatility

This subsection is based on [MZ15, Sections 5.7, 6.4]. Our choice for the instantaneous volatilities $\sigma_i(t)$ is a time-homogeneous, piecewise constant form, i. e. over each period $[T_{i-1}, T_i]$ the volatility is constant. Time-homogeneous means that $\sigma_i(t)$ only depends on the number of reset dates left to maturity. This is plausible from a financial perspective, as we would expect the term structure of volatilities to have a stationary progression. Put into a formula, we get

$$\sigma_i(t) = \widehat{\sigma}_{i-\alpha(t)}, \quad \text{where } \alpha(t) = \min\{j \mid t \leq T_j,\, j = 0, \ldots, n\}. \tag{10}$$

Looking at (10) we see that we have n constant parameters $\widehat{\sigma}_1, \ldots, \widehat{\sigma}_n$ (cf. Table 1) that we can use to calibrate to market data, in our case to caplets. Calibration means that given market caplet prices, we want to set the parameters $\widehat{\sigma}_i$ in such a way that the LIBOR market model reproduces those prices as close as possible. We can then use this calibrated model to price more exotic products.

It is market practice to quote caplet prices by their implied volatilities via the Black-76 model, i. e. if we would like to know the price of a caplet for the accrual period $[T_{i-1}, T_i]$ and look it up on a market data provider, we would not get the actual price but rather a volatility $\hat{\sigma}_i^{market}$ which we first have to insert into Eq. (4) to get the actual market price.[2]

[2] Actually, we would most likely not get caplet prices (or their implied volatilities) from a market data provider, but rather implied volatilities for caps, which are just baskets of caplets for different accrual periods. We would then first need to bootstrap the implied caplet volatilities from the implied cap volatilities. In this essay we assume that we are directly given implied caplet volatilities.

	$[0, T_0]$	$(T_0, T_1]$	$(T_1, T_2]$	$\dots$	$(T_{n-2}, T_{n-1}]$	$(T_{n-1}, T_n]$
$L_1(t)$	$\widehat{\sigma}_1$	expired	expired	$\dots$	expired	expired
$L_2(t)$	$\widehat{\sigma}_2$	$\widehat{\sigma}_1$	expired	$\dots$	expired	expired
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
$L_n(t)$	$\widehat{\sigma}_n$	$\widehat{\sigma}_{n-1}$	$\widehat{\sigma}_{n-2}$	$\dots$	$\widehat{\sigma}_1$	expired

Table 1: The constant parameters $\widehat{\sigma}_i$ as used in the forward rates $L_i(t)$. Since $L_i(t)$ resets at T_{i-1}, the forward rate $L_i(t)$ is expired for $t > T_{i-1}$.

In order to use the quoted volatilities $\hat{\sigma}_i^{market}$ to determine the parameters $\widehat{\sigma}_i$, we need to know how caplets are priced using the LIBOR market model. We already saw in Subsec. 1.2 that the only difference between the LIBOR market model and Black-76 model in that regard is the time-dependence of the volatility. Indeed, it can be shown that using the LIBOR market model we get the exact same caplet pricing formula as Eq. (4) if we replace $\sigma\sqrt{T_0 - t}$ with $\sqrt{\int_t^{T_0} \sigma_i(s)^2 \mathrm{d}s}$ in Eq. (5). By equating the implied Black-76 volatility and the LIBOR market model volatility we get

$$\left(\hat{\sigma}_i^{market}\right)^2 = \frac{1}{T_{i-1} - t} \int_t^{T_{i-1}} \sigma_i(s)^2 \mathrm{d}s. \tag{11}$$

Inserting Eq. (10) into Eq. (11) then yields

$$\begin{aligned} \left(\hat{\sigma}_i^{market}\right)^2 =& \frac{1}{T_{i-1} - t} \sum_{j=1}^{i-1} \widehat{\sigma}_{i-j}^2 (T_j - T_{j-1}) \\ =& \frac{1}{T_{i-1} - t} \left(\widehat{\sigma}_{i-1}^2 T_1 + \sum_{j=2}^{i-1} \widehat{\sigma}_{i-j}^2 (T_j - T_{j-1}) \right), \end{aligned} \tag{12}$$

which we can rearrange into

$$\widehat{\sigma}_{i-1}^2 = \frac{1}{T_1 - T_0} \left(\left(\hat{\sigma}_i^{market}\right)^2 (T_{i-1} - t) - \sum_{j=2}^{i-1} \widehat{\sigma}_{i-j}^2 (T_j - T_{j-1}) \right). \tag{13}$$

We thus obtained a bootstrapping formula, i. e. given $\widehat{\sigma}_1, \dots, \widehat{\sigma}_{i-2}$ we can determine $\widehat{\sigma}_{i-1}$ using Eq. (13). Since $\widehat{\sigma}_1 = \hat{\sigma}_2^{market}$ using Eq. (13), given a fixed time $t \leq T_0$ and quoted volatilities $\hat{\sigma}_i^{market}$ for $i = 2, \dots, n+1$ (i. e. for accrual periods $[T_{i-1}, T_i]$) we can determine $\widehat{\sigma}_1, \dots, \widehat{\sigma}_n$.

1.3.2. Instantaneous correlation

This section is based on [MZ15, Section 5.8]. Our choice for the instantaneous correlation matrix is a parametric form. Since a correlation matrix is necessarily symmetric, there

are in general $\frac{n(n+1)}{2}$ independent parameters, where n is the dimension of the matrix. If we would directly calibrate such a correlation matrix to market data, this would generally include a lot of "noise" and lead to an unstable calibration. To avoid this, we want our parametric form to have as few parameters as reasonably possible.

Consider a set of dates $0 \leq T_0 < T_1 < \cdots < T_n$. A popular choice (and ours as well) is

$$\rho_{ij} = \exp(-\beta \, |T_{i-1} - T_{j-1}|), \quad i, j = 1, \ldots, n, \tag{14}$$

where $\beta > 0$ is the only parameter. We see that this form produces a symmetric and positive definite correlation matrix. We also see that all correlations are positive and monotonically decreasing, i. e. ρ_{ij} is decreasing for a fixed i and variable $j > i$ (in other words, the correlation between forward rates decreases the further apart their reset date is), both of which we would expect to hold based on empirical observations.

A possible way to calibrate to market data would be to first obtain a sample correlation matrix $\hat{\rho}$ based on historical fixings of LIBOR rates. Having obtained such a $\hat{\rho}$, we would then use e. g. a least-square optimization to try to fit the matrix produced by Eq. (14) to $\hat{\rho}$ as best as possible by determining β accordingly. Since this will not be done in this essay, we won't go into any more details.

2. Implementation

In this section we discuss the implementation of the LIBOR market model using a Monte Carlo method. The implementation is done in Python 3 using NumPy, a Python package providing a powerful array object. The reason for choosing Python is to have a fully-fledged programming language on the one hand but still an easy-to-understand syntax on the other hand. Appx. A contains the entire code of the implementation.

Before we begin, let us fix the general set-up of the implementation: Consider a set of dates $0 = T_0 < T_1 < \cdots < T_n$ with corresponding accrual periods $\tau_i = T_i - T_{i-1}$. Starting at $t = 0$, we want to simulate the forward rates $L_i(t)$ for $i = 1, \ldots, n$, which reset at time T_{i-1} and mature at time T_i. Since $T_0 = 0$, the forward rate $L_1(t)$ is already expired at $t = 0$, hence we only need to simulate $L_i(t)$ for $i = 2, \ldots, n$. This also implies that we only need $n - 1$ implied caplet volatilities for the calibration, since we only need to determine the parameters $\widehat{\sigma}_1, \ldots, \widehat{\sigma}_{n-1}$ (cf. Table 1). This set-up is the basis for our implementation and will be assumed implicitly throughout the rest of this essay.

2.1. General overview

We first present a general overview of the structure of the implementation. A more detailed discussion can be found in the following subsections.

We can put all classes (cf. Fig. 1) into one of three categories:

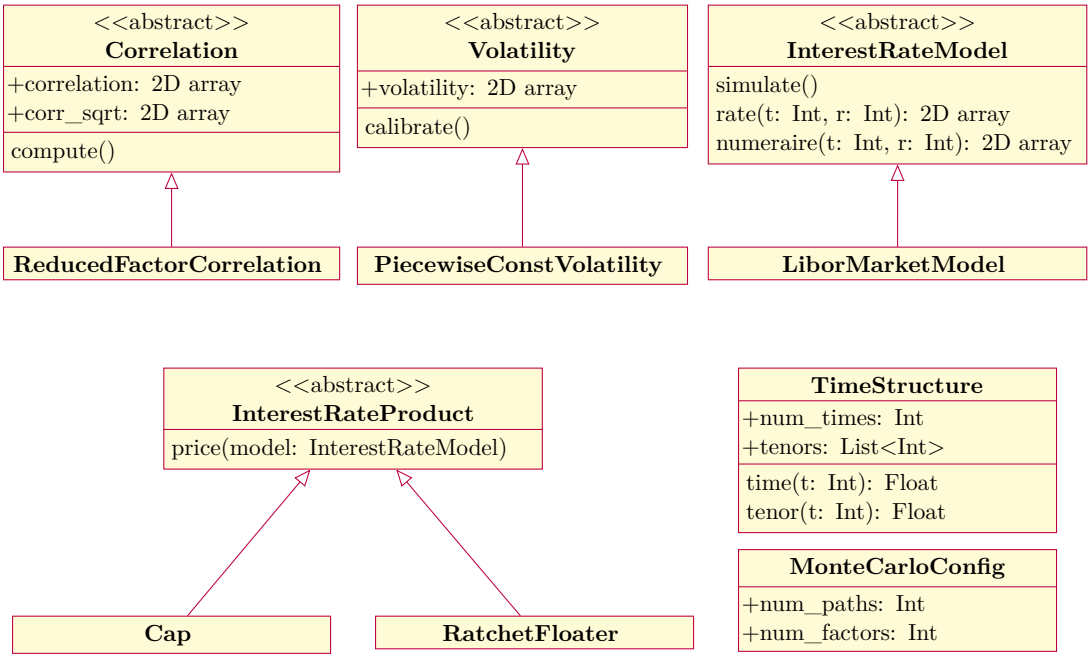

Figure 1: Simplified UML diagram representing the structure of the implementation.

Model: Classes `ReducedFactorCorrelation`, `PiecewiseConstVolatility`, `LiborMarketModel` with interfaces `Correlation`, `Volatility`, `InterestRateModel`.

Product: Classes `Cap` and `RatchetFloater` with interface `InterestRateProduct`.

Utilities: Classes `TimeStructure` and `MonteCarloConfig`.

Category **Model** contains all classes related to the implementation of the actual LIBOR market model, category **Product** contains classes representing the products we want to price using the LIBOR market model and lastly category **Utilities** contains self-contained helper classes used in the other two categories.

Since Monte Carlo methods require the handling and storage of large amount of high-dimensional data, we use NumPy's array object extensively throughout the implementation. Since e. g. slicing such an array creates a new array object through a view (in which the data is shared with the old array) instead of creating a copy of the data from the old array, NumPy arrays are performant and therefore a good fit for our implementation.

Another thing to note is that we use abstract classes for defining interfaces. In Python this is done by inheriting from the class `ABC` of the `abc` module (which stands for *Abstract Base Classes*), which is included in the standard library.

2.2. Design aspects

In the implementation, we focus on an object-oriented design. We especially will pay attention to two things:

1. Defining interfaces for the main classes.
2. Separation of product and model classes.

The first point helps to avoid strong coupling between classes as long interfaces are used instead of implementations of those interfaces. This avoids that a class knows too much (ideally nothing) about the implementation of another class, which can then be swapped with a different implementation without changing much existing code, if any.

We already saw the second point in the general overview, where we put classes into different categories **Model** and **Product**. Since a product can in general be priced using different models, classes from **Product** should not know too much about (and hence depend on) classes from **Model** and vice versa. The only link between these two categories is the passing of a model as parameter of the method *price* of the class `InterestRateProduct` (cf. Fig. 1).

To further substantiate the first point, lets take the class `Volatility` as an example, which provides an interface for volatility functions. Since `LiborMarketModel` only knows this interface, it only uses the methods and properties provided by that interface to simulate forward rates. Via the constructor of `LiborMarketModel` we can pass different implementations of that interface, e. g. the existing class `PiecewiseConstVolatility` or a new class implementing a completely different form of volatility functions. Whatever we pass, we wouldn't need to change code inside `LiborMarketModel` as long as the passed class implements the interface `Volatility` correctly.

2.3. Discussion of implementation

We now discuss the actual implementation of the LIBOR market model. For this, we won't go through each line of code and discuss it, but rather pick out the most important bits and focus on them.

In our implementation most logic is contained in the methods *compute()*, *calibrate()* and *simulate()* of the classes `ReducedFactorCorrelation`, `PiecewiseConstVolatility` and `LiborMarketModel` (cf. Fig. 1). Therefore, we will focus on discussing these methods (or parts of these methods) in this subsection. For the rest of the code, we refer to Appx. A, where the entire code can be found documented.

We start with the *compute()* method. For a given correlation matrix of dimension n, this method computes a rank F (where $F \leq n$) approximation of the correlation matrix and the corresponding (n, F)-dimensional pseudo-square root of the approximation. This approximation is also called a *factor reduction*. It is important because of numerical

efficiency, since it reduces the computational complexity in the time-stepping procedure when simulating forward rates, especially if F is way smaller than n.

The approximation algorithm is taken from [Jos11, Section 23.5]. The important part of code implementing this algorithm is

```
1   eigvals, eigvecs = np.linalg.eig(self._correlation)
2
3   sorted_indices = eigvals.argsort()[::-1][:self._num_factors]
4   eigvals = eigvals[sorted_indices]
5   eigvecs = eigvecs.T[sorted_indices]
6
7   sqrt = np.column_stack([np.sqrt(l)*v for l, v in zip(eigvals, eigvecs)])
8   covariance = sqrt @ sqrt.T
9
10  normalize = lambda i, j: sqrt[i, j]/np.sqrt(covariance[i, i])
11  self._corr_sqrt = np.fromfunction(normalize, self._corr_sqrt.shape, dtype=int)
12  self._corr_reduced = self._corr_sqrt @ self._corr_sqrt.T
```

We explain the algorithm by explaining the code above:

- First we compute the eigenvalues λ_i and eigenvectors v_i of the correlation matrix. This is done in line 1.
- In the second step we order the eigenvalues and correspondingly eigenvectors descendingly (line 3) and take the first F of those (line 4 and 5).
- The next step is to create the pseudo-square root with F columns, where the column vectors are $\sqrt{\lambda_i} v_i$ (line 7). We then compute the covariance matrix by squaring the pseudo-square root (line 8).
- Since in general the squaring will only create a covariance matrix, we need to normalize it to obtain a correlation matrix. We do this by diving through the standard deviations, i. e. square roots of the diagonal elements of the covariance matrix. This is done in lines $10 - 12$.

We now discuss the *calibrate()* method of the class `PiecewiseConstVolatility`. This method performs a bootstrapping algorithm to calibrate the volatility functions of the LIBOR market model to implied caplet volatilities. The theoretical background of this algorithm is discussed in Subsec. 1.3.1. The algorithm itself can be derived from the formula Eq. (13). The important bit of code implementing the algorithm is

```
T_1 = self._timestruct.time(1)
bs_volas = []

for i, capletvola in enumerate(self._capletvolas, 2):
    bs_vola = capletvola**2 * self._timestruct.time(i-1)

    for j, vola in enumerate(bs_volas, 2):
        bs_vola -= vola**2 * self._timestruct.tenor(j-1)
```

```
9
10          bs_volas.append(np.sqrt(bs_vola/T_1))
```

As we can see, the implementation is rather simple. We start by storing the value T_1 into a variable (line 1) and initializing a list (line 2), which will eventually contain the bootstrapped volatilities. The first iteration of the outer loop (line 4) will produce the value $\widehat{\sigma}_1$, since the inner loop (line 7) won't execute because *bs_volas* is empty at that point in time. In subsequent iterations the inner loop will then execute and subtract already bootstrapped volatilities weighted by the period length from the current volatility being bootstrapped. This inner loop corresponds to the sum in Eq. (13). At the end of each iteration of the outer loop, the bootstrapped volatility is then appended to the list *bs_volas* (line 10).

We now come to the *simulate()* method of class `LiborMarketModel`, which can be considered the "heart" of the implementation. This is the place where the actual simulation of the forward rates happen. The simulation is done via the Euler-Maruyama time-stepping scheme applied to Eq. (8). Instead of directly using the scheme on Eq. (8), we use the scheme on the logarithm of Eq. (8), which leads to a better convergence. Stepping from time T_i to T_{i+1} then yields (cf. [MZ15, Section 6.2])

$$\log L_i(T_{i+1}) = \log L_i(T_i) + \left(\mu_i(T_i) - \frac{1}{2}\sigma_i(T_i)^2\right)\tau_{i+1} + \sigma_i(T_i)\sqrt{\tau_{i+1}}\sum_{k=1}^{F}\eta_{ik}Z_k, \tag{15}$$

where η_{ik} is the pseudo-square root discussed earlier and Z_k a sample of a standard normal random variable. By exponentiating, we then get the scheme

$$L_i(T_{i+1}) = L_i(T_i)\exp\left(\left(\mu_i(T_i) - \frac{1}{2}\sigma_i(T_i)^2\right)\tau_{j+1} + \sigma_i(T_i)\sqrt{\tau_{i+1}}\sum_{k=1}^{F}\eta_{ik}Z_k\right). \tag{16}$$

This is the time-stepping scheme used in the *simulate()* method. The important part of the code is

```
1   for k in range(self._mc_config.num_paths):
2       L = self._rates[:, :, k]
3       P = self._numeraire[:, :, k]
4
5       L[:, 0] = self._forwardcurve
6       P[:, 0] = np.cumprod(1/(1 + tau*L[:, 0]))
7
8       for j in range(1, self._timestruct.num_times):
9           drift = np.zeros(num_rates)
10          for m in range(j+1, num_rates):
11              drift[m] = ((rho[m, j]*tau[m-1]*L[m, j-1]*vols[m, j])/
12                          (1 + tau[m-1]*L[m, j-1]))
13
14          for i in range(j, num_rates):
15              dW = self._correlation.corr_sqrt[i, :] @ increments[j, k]
```

```
16
17              L[i, j] = L[i, j-1] * np.exp((-vols[i, j] * np.sum(drift[i+1:])
18                                           - 0.5 * vols[i, j]**2) * tau[j]
19                                           + vols[i, j] * sqrt_tau[j] * dW)
20              P[i, j] = (1 if i==j else P[i-1, j])/(1 + tau[j]*L[i, j])
```

We see that we have three intertwined loops (line 1, 8 and 14). The loops represent looping through the number of paths (line 1), stepping forward in time (line 8) and looping through the number of forward rates (line 14). The variables L and P are the NumPy arrays containing the values of the simulated forward rates and the corresponding numeraire values. The three loops are basically there to fill those two arrays with simulated values. Before stepping through time we set the initial forward rates and corresponding numeraire values (line 5 and 6) for each simulated path, since they do not change. In the time-stepping loop (line 8) we start by calculating the drift under the terminal measure. The loop in line 10 is just the sum from Eq. (9). Since we have to do the time-stepping for every forward rate, we then loop through every forward rate (line 14) and apply the Euler-Maruyama scheme from Eq. (16). The lines 17-19 are just the Eq. (16). Before applying the scheme, we of course have to compute the random increment (line 15). Afterwards we compute the corresponding numeraire values from the simulated forward rates (line 20).

3. Results

In this section we are going to use our implementation of the LIBOR market model to price two interest rate products, a cap and a ratchet floater. We will simulate the forward rates for semi-annual accrual periods over the span of five years, i. e. we consider the dates $0 = T_0 < 0.5 < 1.0 < \cdots < 4.5 < 5.0 = T_{10}$ with periods $\tau_i = T_i - T_{i-1} = 0.5$. For both products we will use the same set of hypothetical market data presented in Table 2.[3] The simulations will be performed with 100000 sample paths and 4 factors each.

We start with the cap. A cap is simply a basket of caplets, where all caplets have the same strike, called the cap rate. The caplets in our case will be for semi-annual accrual periods with payoffs at times $T_1, T_2, \ldots, T_n$. The price of a cap is then just the sum of the prices of caplets in the basket. Since we calibrate our LIBOR market model to implied caplet volatilities, we expect the LIBOR market model to match the price of a cap quite closely. For caplets we also have the Black-76 formula from Eq. (4), therefore we can compute the price of a cap analytically as well. This serves as a good "sanity check" for our implementation.

For a cap with a notional of 10000000 and a cap rate of 1.1% the results can be found in Table 2. As expected, the LIBOR market model does indeed match the price quite closely. The "sanity check" was therefore successful.

[3] The correlation matrix used for the simulations was generated using the parametric form in Eq. (14) with $\beta = 0.2$. Due to space restrictions, we didn't include it in the table.

i	T_i	$L_i(0)$	$\hat{\sigma}_i^{market}$
1	0.5	0.0112	expired
2	1.0	0.0118	0.2366
3	1.5	0.0123	0.2487
4	2.0	0.0127	0.2573
5	2.5	0.0132	0.2564
6	3.0	0.0137	0.2476
7	3.5	0.0145	0.2376
8	4.0	0.0154	0.2252
9	4.5	0.0163	0.2246
10	5.0	0.0174	0.2223

T_i	Black-76	LMM	Difference
0.5	6058.88	6039.00	-0.33%
1.0	9415.56	9371.50	-0.47%
1.5	12124.80	12118.71	-0.05%
2.0	14807.67	14872.78	0.44%
2.5	17123.77	17196.78	0.43%
3.0	20420.86	20553.50	0.65%
3.5	23975.40	24130.03	0.64%
4.0	27876.56	28027.23	0.54%
4.5	32492.46	32540.41	0.15%
Sum	164295.96	164849.94	0.34%

Table 2: *Left*: Hypothetical market data used for simulations. *Right*: Simulation results for cap with notional of 10000000 and cap rate of 1.1%.

We now come to the ratcher floater, which is a path dependent interest rate product. The ratchet floater is a good example for the use of the LIBOR market model, since no analytic formula exists and we have to resort to numerical methods.

We briefly describe a ratchet floater, see [MZ15, Section 7.4] for more details. At each time T_i, $i > 0$ the ratchet floater pays $\tau_i N(L(T_{i-1}, T_i) + X)$, where X is a constant spread and N is the notional, and requires the payment of a coupon c_i. For the first period the coupon c_1 is defined as $\tau_i N(L(T_0, T_1) + Y)$, where Y is a constant spread as well. For subsequent periods c_i, $i > 1$, the coupon is defined as

$$c_i = c_{i-1} + \min((\tau_i N(L(T_{i-1}, T_i) + Y) - c_{i-1})^+ , \ N\alpha), \tag{17}$$

where $\alpha > 0$ is a fixed cap. In other words, the coupon amount c_i is at least as much as the previous coupon amount, but no more than the previous amount plus a fixed amount $N\alpha$. At each T_i, $i > 0$ the cashflow $\tau_i N(L(T_{i-1}, T_i) + Y) - c_i$ is produced. The value of the ratchet floater is then just the sum of all discounted cashflows.

For ratchet floaters with a notional of 10000000, spreads $X = 0.15\%$, $Y = 0.15\%$ and various fixed caps $\alpha = 0.01\%, 0.05\%, 0.10\%, 0.20\%$, the results can be found in Table 3. Since we don't have an analytical formula to compare to, we can't validate the results easily. Nonetheless, the results seem at least plausible: For once, we have that for $T_i = 0.5$ all values are 0. This is indeed correct and due to the fact that the only difference between the first payment and the first coupon of a ratchet floater are the constant spreads X and Y. If they are equal, which they are in our case, they should cancel each other out and produce a cashflow of value 0. Furthermore, we see that for small α the produced

T_i	$\alpha = 0.01\%$	$\alpha = 0.05\%$	$\alpha = 0.10\%$	$\alpha = 0.20\%$
0.5	0.0	0.0	0.0	0.0
1.0	1942.18	168.63	-1246.30	-2501.23
1.5	3348.26	39.98	-2581.20	-4740.16
2.0	5177.02	422.74	-3356.23	-6401.34
2.5	6981.63	798.37	-4022.85	-7927.09
3.0	10195.74	2366.28	-3518.36	-8432.24
3.5	13841.96	4315.78	-2704.23	-8666.65
4.0	17333.35	6172.06	-1894.36	-9038.23
4.5	21691.95	8758.74	-347.96	-8656.31
Sum	80512.09	23042.58	-19671,49	-56363,25

Table 3: Simulation results for ratchet floater with spreads $X = Y = 0.15\%$, notional of 10000000 and $\alpha = 0.01\%$, 0.05%, 0.10%, 0.20%.

cashflows are positive and gets larger with every further cashflow, while for bigger α the same thing happens, just in the other direction. This can be explained because the coupon that has to be paid is capped by α. For small α we see that the rates grow and the holder of the ratchet floater gets successively larger payments at each T_i, while the coupon that needs to be paid for the ratchet floater is capped above by α and can't grow just as much. If α gets bigger, the coupons can now grow larger, even larger than the payments made by the ratchet floater, and the produced cashflows become negative. Thus, even if we can't analytically verify the prices of the ratchet floaters obtained through the simulations, we have at least some assurance that they can't be too far off.

References

[Bjö09] Tomas Björk. *Arbitrage theory in continuous time.* Oxford University Press, 2009.

[Jos11] Mark S Joshi. *More mathematical finance.* Pilot Whale Press Melbourne, 2011.

[MZ15] Daragh McInerney and Tomasz Zastawniak. *Stochastic Interest Rates.* Cambridge University Press, 2015.

Appendices

Appendix A Code of implementation

This section of the appendix contains the entire code for the implementation of the LIBOR market model. The caption above each listing denotes the filename, which in Python is the same as the module name (excluding the file extension). The starting point for executing the implementation is the file *main.py*.

The code was written in Python 3.6.5 and uses the additional package NumPy 1.14.3.

Listing 1: main.py

```
import numpy as np
from utilities import TimeStructure, MonteCarloConfig
from volatility import PiecewiseConstVolatility
from correlation import ReducedFactorCorrelation
from interestratemodel import LiborMarketModel
from product import Cap, RatchetFloater


# Set-up hypothetical market data
termstruct = TimeStructure([0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0])
forwardcurve = np.array([0.0112, 0.0118, 0.0123, 0.0127, 0.0132, 0.0137, 0.0145,
                         0.0154, 0.0163, 0.0174])
capletvolas = np.array([0.2366, 0.2487, 0.2573, 0.2564, 0.2476, 0.2376, 0.2252,
                        0.2246, 0.2223])

# Correlation matrix generated by using parametric form with beta = 0.2
# Full correlation matrix initialization removed due to space restrictions
# - CODE WILL NOT WORK WITHOUT FULL CORRELATION MATRIX -
correlation_matrix = np.array([[1, 0.904837, ...], ..., [..., 0.904837, 1]])

# Simulate forward rates with 100000 sample paths and 4 factors
config = MonteCarloConfig(10, 4)

# Initialize LIBOR market model
volatility = PiecewiseConstVolatility(termstruct, capletvolas)
correlation = ReducedFactorCorrelation(correlation_matrix, config.num_factors)
libor_marketmodel = LiborMarketModel(config, termstruct, volatility, correlation,
                                     forwardcurve)

volatility.calibrate()
correlation.compute()
libor_marketmodel.simulate()

# Price cap with cap rate 1.1% and notional of 10000000
cap = Cap(0.0110, 10000000, termstruct)
cap_price = cap.price(libor_marketmodel)

```

```
# Price ratchet floater with spreads both 0.15%, fixed rate 0.01%
# and notional of 10000000
ratchetfloater = RatchetFloater(0.0015, 0.0015, 0.0001, 10000000, termstruct)
ratchetfloater_price = ratchetfloater.price(libor_marketmodel)
```

Listing 2: utilities.py

```
import numpy as np


class TimeStructure:
    """Class representing a discrete set of points in time T_0, T_1, ..., T_n."""

    """Args:

    times: List of floats representing points in time with unit [1/year].
    """
    def __init__(self, times):
        self._times = np.array(times)

    @property
    def num_times(self):
        """Number of points in time."""
        return self._times.size

    @property
    def tenors(self):
        """Returns 1D array [T_1 - T_0, T_2 - T_1, ..., T_n - T_n-1]."""
        return np.diff(self._times)

    def time(self, timeindex):
        """Returns time value for index i"""
        return self._times[timeindex]

    def tenor(self, timeindex):
        """Returns length of the time period [T_i, T_i+1] for index i."""
        return self._times[timeindex + 1] - self._times[timeindex]


class MonteCarloConfig:
    """Container class bundling configuration for a Monte Carlo simulation."""

    """Args:

    num_paths: Number of paths in Monte Carlo simulation.
    num_factors: Number of factors for correlation matrix in Monte Carlo simulation.
    """
    def __init__(self, num_paths, num_factors):
        self._num_paths = num_paths
        self._num_factors = num_factors

```

```
45      @property
46      def num_paths(self):
47          return self._num_paths
48
49      @property
50      def num_factors(self):
51          return self._num_factors
```

Listing 3: correlation.py

```
1   from abc import ABC, abstractmethod
2   import numpy as np
3
4
5   class Correlation(ABC):
6       """Abstract class providing interface for a correlation matrix."""
7
8       @abstractmethod
9       def compute(self):
10          """Computes correlation matrix. Needs to be called before other methods."""
11          pass
12
13      @abstractmethod
14      def correlation(self):
15          """Returns correlation matrix as 2D array."""
16          pass
17
18      @abstractmethod
19      def corr_sqrt(self):
20          """Returns pseudo-square root of correlation matrix as 2D array."""
21          pass
22
23
24  class ReducedFactorCorrelation(Correlation):
25      """Implements n-dimensional reduced-factor correlation matrix.
26      Correlation matrix will be of rank F, where F is the given number of factors.
27      """
28
29      """Args:
30
31      correlation: Correlation matrix as 2D array to approximate.
32      num_factors: Number of factors to use for approximation.
33      """
34      def __init__(self, correlation, num_factors):
35          self._correlation = correlation
36          self._num_factors = num_factors
37
38          corr_dim = correlation.shape[0]
39          self._corr_sqrt = np.zeros((corr_dim, num_factors))
40          self._corr_reduced = np.zeros(correlation.shape)
41
42      @property
```

```
    def correlation(self):
        """Returns approximation to correlation matrix with rank F."""
        return self._corr_reduced

    @property
    def corr_sqrt(self):
        """Returns (n, F)-dimensional pseudo-square root of correlation matrix."""
        return self._corr_sqrt

    def compute(self):
        """Computes a rank F approximation of given correlation matrix."""

        # Compute eigenvalues and vectors from correlation matrix
        eigvals, eigvecs = np.linalg.eig(self._correlation)

        # Sort eigenvalues and corresponding eigenvectors descendingly
        sorted_indices = eigvals.argsort()[::-1][:self._num_factors]
        eigvals = eigvals[sorted_indices]
        eigvecs = eigvecs.T[sorted_indices]

        # Create covariance matrix with reduced rank
        sqrt = np.column_stack([np.sqrt(l)*v for l, v in zip(eigvals, eigvecs)])
        covariance = sqrt @ sqrt.T

        # Create correlation matrix with reduced rank by normalizing covariance matrix
        normalize = lambda i, j: sqrt[i, j]/np.sqrt(covariance[i, i])
        self._corr_sqrt = np.fromfunction(normalize, self._corr_sqrt.shape, dtype=int)
        self._corr_reduced = self._corr_sqrt @ self._corr_sqrt.T
```

Listing 4: volatility.py

```
from abc import ABC, abstractmethod
import numpy as np


class Volatility(ABC):
    """Abstract class providing interface for volatility functions."""

    @abstractmethod
    def volatility(self):
        """Returns 2D array representing volatility functions.
        First index is forward rate, second index is time.
        """
        pass

    @abstractmethod
    def calibrate(self):
        """Calibrates to market data. Needs to be called before other methods."""
        pass


```

```
class PiecewiseConstVolatility(Volatility):
    """Implements time-homogeneous, piecewise constant volatility functions.
    The volatility functions will be calibrated to implied caplet volatilities.
    """

    """Args:

    timestruct: Instance of TimeStructure class containing times
                at which volatility functions should be computed.
    capletvolas: 1D array containing caplet volatilities for calibration.
    """
    def __init__(self, timestruct, capletvolas):
        self._timestruct = timestruct
        self._capletvolas = capletvolas

        num_rates = timestruct.num_times - 1
        self._volas = np.zeros((num_rates, timestruct.num_times))

    @property
    def volatility(self):
        """Returns 2D array of piecewise constant volatilities."""
        return self._volas

    def calibrate(self):
        """Calibrates volatility functions to given caplet volatilities."""

        # Bootstrap caplet volatilities
        T_1 = self._timestruct.time(1)
        bs_volas = []

        for i, capletvola in enumerate(self._capletvolas, 2):
            bs_vola = capletvola**2 * self._timestruct.time(i-1)

            for j, vola in enumerate(bs_volas, 2):
                bs_vola -= vola**2 * self._timestruct.tenor(j-1)

            bs_volas.append(np.sqrt(bs_vola/T_1))

        # Insert bootstrapped volatilities into 2D array
        self._volas[1:, 1] = bs_volas
        for i in range(2, self._timestruct.num_times):
            self._volas[i:, i] = bs_volas[:-i+1]
```

Listing 5: interestratemodel.py

```
from abc import ABC, abstractmethod
import numpy as np


class InterestRateModel(ABC):
    """Abstract class providing interface for a interest rate model."""
```

```
7
8       @abstractmethod
9       def rate(self, rateindex, timeindex):
10          """Returns 1D array containing all realizations of simulation for given
11          forward rate and time.
12          """
13          pass
14
15      @abstractmethod
16      def numeraire(self, rateindex, timeindex):
17          """Returns 1D array containing numeraires corresponding to realizations
18          of simulation for given forward rate and time.
19          """
20          pass
21
22      @abstractmethod
23      def simulate(self):
24          """Simulates forward rates. Needs to be called before other methods."""
25          pass
26
27
28  class LiborMarketModel(InterestRateModel):
29      """Implements LIBOR market model using an Euler-Maruyama scheme."""
30
31      """Args:
32
33      mc_config: Instance of MonteCarloConfig class.
34      timestruct: Instance of TimeStructure class containing times
35                  at which forward rates should be simulated.
36      volatility: Instance implementing Volatility interface.
37      correlation: Instance implementing Correlation interface.
38      forwardcurve: Initial forward curve as 1D array of floats.
39      """
40      def __init__(self, mc_config, timestruct, volatility, correlation, forwardcurve):
41          self._timestruct = timestruct
42          self._volatility = volatility
43          self._correlation = correlation
44          self._forwardcurve = forwardcurve
45          self._mc_config = mc_config
46
47          shape = (timestruct.num_times - 1, timestruct.num_times, mc_config.num_paths)
48          self._rates = np.zeros(shape)
49          self._numeraire = np.zeros(shape)
50
51      def rate(self, rateindex, timeindex):
52          """Returns realizations for given forward rate and time."""
53          return self._rates[rateindex, timeindex]
54
55      def numeraire(self, rateindex, timeindex):
56          """Returns numeraire for given forward rate and time."""
57          return self._numeraire[rateindex, timeindex]
58
59      def simulate(self):
```

```
        """Simulates forward rates using Euler-Maruyama time-stepping procedure."""

        # Pre-compute standard normal increments for all time steps and rates
        mean = np.zeros(self._mc_config.num_factors)
        cov = np.identity(self._mc_config.num_factors)
        increments = np.random.multivariate_normal(
            mean, cov, (self._timestruct.num_times, self._mc_config.num_paths))

        # Declare variables for convenience
        num_rates = self._timestruct.num_times - 1
        rho = self._correlation.correlation
        vols = self._volatility.volatility
        tau = self._timestruct.tenors
        sqrt_tau = np.sqrt(tau)

        # Simulate forward rates
        for k in range(self._mc_config.num_paths):
            L = self._rates[:, :, k]
            P = self._numeraire[:, :, k]

            L[:, 0] = self._forwardcurve
            P[:, 0] = np.cumprod(1/(1 + tau*L[:, 0]))

            for j in range(1, self._timestruct.num_times):
                # Compute drift
                drift = np.zeros(num_rates)
                for m in range(j+1, num_rates):
                    drift[m] = ((rho[m, j]*tau[m-1]*L[m, j-1]*vols[m, j])/
                                (1 + tau[m-1]*L[m, j-1]))

                # Compute forward rates and numeraire
                for i in range(j, num_rates):
                    dW = self._correlation.corr_sqrt[i, :] @ increments[j, k]

                    L[i, j] = L[i, j-1] * np.exp((-vols[i, j] * np.sum(drift[i+1:])
                                                  - 0.5 * vols[i, j]**2) * tau[j]
                                                  + vols[i, j] * sqrt_tau[j] * dW)
                    P[i, j] = (1 if i==j else P[i-1, j])/(1 + tau[j]*L[i, j])


```

Listing 6: product.py

```
from abc import ABC, abstractmethod
import numpy as np

class InterestRateProduct(ABC):
    """Abstract class providing interface interest rate product."""

    @abstractmethod
    def price(self, ratemodel):
        """Returns price(s) using the given interest rate model."""
```

```
        pass

class Cap(InterestRateProduct):
    """Implementation of a cap."""

    """Args:

    caprate: Cap rate.
    notional: Notional of cap.
    timestruct: Instance of TimeStructure class containing cap maturities.
    """
    def __init__(self, caprate, notional, timestruct):
        self._caprate = caprate
        self._notional = notional
        self._timestruct = timestruct

    def price(self, ratemodel):
        """Returns price of cap"""
        N = self._notional
        K = self._caprate

        caplet_prices = []
        num_rates = self._timestruct.num_times - 1
        for i in range(1, num_rates):
            tau = self._timestruct.tenor(i-1)
            L = ratemodel.rate(i, i)
            P = ratemodel.numeraire(num_rates-1, i+1)
            D = ratemodel.numeraire(num_rates-1, 0)

            payoff = tau * N * (L - K)
            payoff[payoff < 0] = 0.0  # Maximum with 0

            # Compute expectation under numeraire measure
            caplet_price = np.mean(D*payoff/(1 if i==num_rates-1 else P),
                                   dtype=np.float64)

            caplet_prices.append(caplet_price)

        return np.sum(caplet_prices)


class RatchetFloater(InterestRateProduct):
    """Implementation of a ratchet floater"""

    """Args:

    X: Constant spread for forward rate.
    Y: Constant spread for coupons.
    alpha: Fixed cap.
    notional: Notional of ratchet floater.
    timestruct: Instance of TimeStructure class containing forward rate maturities.
    """
    def __init__(self, X, Y, alpha, notional, timestruct):
```

```
63            self._X = X
64            self._Y = Y
65            self._alpha = alpha
66            self._notional = notional
67            self._timestruct = timestruct
68
69        def price(self, ratemodel):
70            """Returns price of ratchet floater"""
71            N = self._notional
72            X = self._X
73            Y = self._Y
74            alpha = self._alpha
75
76            num_rates = self._timestruct.num_times - 1
77            coupons = []
78            payoffs = []
79            for i in range(1, num_rates):
80                tau = self._timestruct.tenor(i-1)
81                L = ratemodel.rate(i, i)
82                P = ratemodel.numeraire(num_rates-1, i+1)
83                D = ratemodel.numeraire(num_rates-1, 0)
84
85                if i == 1:
86                    coupon = tau * (L + Y)
87                else:
88                    coupon = tau * (L + Y) - coupons[i-2]
89                    coupon[coupon < 0] = 0.0  # Maximum with 0
90                    coupon[coupon > alpha] = alpha  # Minimum with alpha
91                    coupon = coupons[i-2] + coupon
92
93                coupons.append(coupon)
94
95                payoff = N * (tau * (L + X) - coupon)
96                # Compute expectation under numeraire measure
97                payoffs.append(np.mean(D*payoff/(1 if i==num_rates-1 else P),
98                               dtype=np.float64))
99
100           return np.sum(payoffs)
```