Fixed Income: Project

Emanuele Fogliati

January 2023

1 Assignment

That project is developed in order to solve the following assignments:

- Pricing fixed-income portfolios under the single curve approach.
- Vasicek model: calibration and Montecarlo simulation.
- Comparison between the classical and the multicurve approach to price fixed-income instruments.

2 Importing data

The first assignment involves the implementation of some Python codes in order to price a set of fixed-income instruments under the single curve approach; the instruments that are going to be priced are a fixed-rate bond, a floating rate note and two structured notes. All the portfolios are priced under the classical approach commonly used until 2008, which employs the OIS curve both for discounting and for forwarding the cash flows. Therefore, to construct both the discount and the spot rates curves under continuous compounding, it is firstly necessary to download the available data for the Euro ESTR OIS zero yield curve, in that case from Refinitiv searching for 0#EURESTOISZ=R, and to read them with Python, after having imported all the relevant packages. Once the term structure data have been imported and properly adjusted into a dataframe, the trade date is chosen to be the 30th January 2023 for all the instruments and the required calendar is set as the standard European one, defined as TARGET. All the dates are then converted into QuantLib dates, and the fraction of year (`Time in year` in the code) between the trade date and each maturity is derived using the function `year.fraction()`, according to the Actual/360 convention. The continuously compounded sport rates (`R`), required to bootstrap the market spot rate curve, are derived as:

$$R(t,T) = -\frac{log(P(t,T))}{\alpha(t,T)}$$

where P(t,T) is the discount factor and $\alpha(t,T)$ is the time in years between the dates. The discount curve and the market spot rates curve, in Figure 1, display on the abscissa the time to maturity and

on the ordered respectively the discount factors and the spot rates. This OIS curve will be used also for all the other assignments of the project.

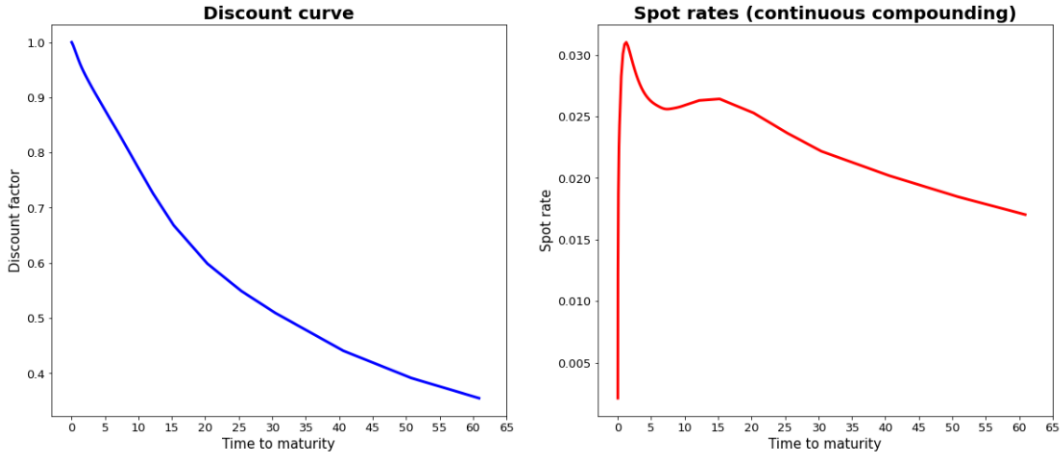


Figure 1: Discount curve and Spot rate curve

Except for the input data, for all the fixed income instruments the procedure of computing the schedule of payments is the same and exploits the QuantLib library and the function `ql.Schedule` as follows. The inputs needed to compute the schedule are the effective date, the maturity date, the frequency of payments, the convention chosen to account for non-business days, the rule of generation, in that case backward, and the convention on how to consider the last day of the month. Since it is necessary to account for the period between the date in which the previous coupon should be paid and the effective date, another schedule, `scheduleShift`, is computed considering the time length of the contract.

2.1 Fixed-Rate Bond

Data:

- Maturity: 12-Dec-2026
- Notional: Euro 100
- Coupon: 3%
- Frequency: semiannual payments
- Day rolling: Modified following
- Day count convention (basis): act/act

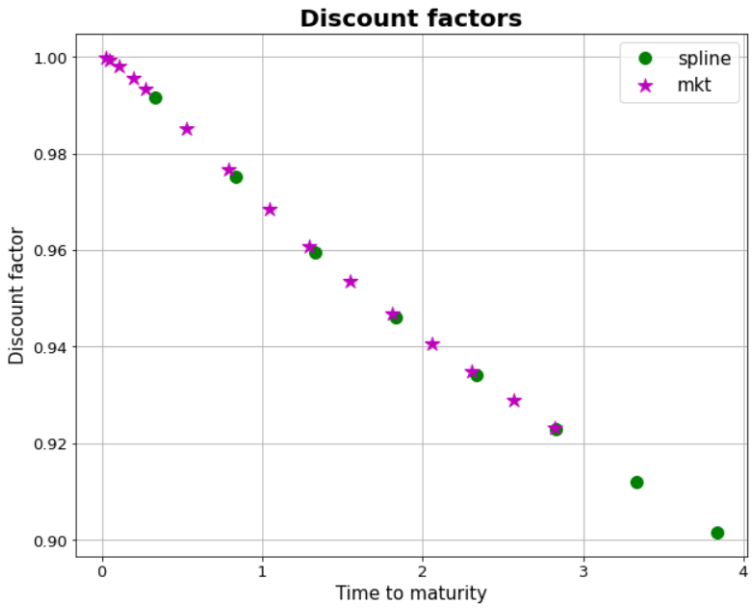


Figure 2: Discount factors: market data vs cubic interpolation

The first instrument to price is a fixed-rate bond maturing on 12th December 2026, with notional equal to of 100 Euro, 3% coupon rate, Actual/Actual day count convention and semi-annual payment frequency. The dates in which each coupon period starts are set as `StartDates` and the dates in which the coupons and, eventually, the notional are paid as `EndDates`. The accrual factor, `AF`, $\alpha(T_i, T_{i+1})$, based on the Actual/Actual convention, is the effective time in year between the coupon payments, whereas the time is years between the settlement date and each coupon payment date is defined as `CFlowTimes`, $\alpha(0, T_i)$, similarly, according to the Actual/Actual convention. In both cases the function used is the aforementioned `year.fraction()`, combined with the stated day count convention `ql.ActualActual()`. The discount factor associated to each future payment date is derived by the cubic interpolation of the discount curve through the Scipy function `interpolate.CubicSpline()`; by doing that, it is possible to extract the specific discount factor associated to each needed maturity.
The figure above [figure 2] displays both the market discount factors and the discount factors deriving from the interpolation of the discount curve for the required maturities.

Since the coupon bond to price is a fixed-rate bond, its price is just the sum of all the discounted cash flows, which include each coupon payment computed as the notional multiplying the coupon rate and the coupon period length, and the face value at maturity.
Summarizing, each coupon payment is computed as:

$$CF(i) = Notional \times CpnRate \times \alpha(T_{i-1}, T_i), \;\; i = 1, ..., N$$

$$CF(N) = Notional \times CpnRate \times \alpha(T_{N-1}, T_N) + Notional$$

and the discounted cash flow payments as:

$$DCF(i) = CF(i) \times P(0, T_i), \;\; i = 1, ..., N$$

Therefore, the price of the bond is given by:

$$CB(0) = \sum_{i=1}^{n} DCF(i)$$

which, with respect to the given inputs, is 101.471.

	StartDates	EndDates	AF	CFLowTimes	CpnRates	CFlowAmounts	DiscountedCFlowAmounts
0	December 12th, 2022	June 12th, 2023	0.5	0.333333	0.03	1.5	1.487390
1	June 12th, 2023	December 12th, 2023	0.5	0.833333	0.03	1.5	1.462645
2	December 12th, 2023	June 12th, 2024	0.5	1.333333	0.03	1.5	1.439286
3	June 12th, 2024	December 12th, 2024	0.5	1.833333	0.03	1.5	1.419310
4	December 12th, 2024	June 12th, 2025	0.5	2.333333	0.03	1.5	1.401321
5	June 12th, 2025	December 12th, 2025	0.5	2.833333	0.03	1.5	1.384428
6	December 12th, 2025	June 12th, 2026	0.5	3.333333	0.03	1.5	1.368206
7	June 12th, 2026	December 14th, 2026	0.5	3.833333	0.03	101.5	91.508391

Figure 3: Schedule of payments for the fixed-rate coupon bond

2.2 Floating Rate Note

Data:

- Maturity: 18-Jun-2024
- Notional: Euro 100
- Coupon: 3mEuribor + 200bp
- Frequency: quarterly payments
- Day rolling: Modified following
- Day count convention (basis): act/360

The second instrument to price is a floating rate note having a notional of one hundred euros, maturity on the 18th June 2024, quarterly payments and a coupon rate equal to the three-months Euribor plus a 2% spread. The three-months Euribor at time being amounts to 2.482%. The method of computing the schedule and the payment dates is the same used before, as also the interpolation of the interest rate curve to find the discount factors needed.

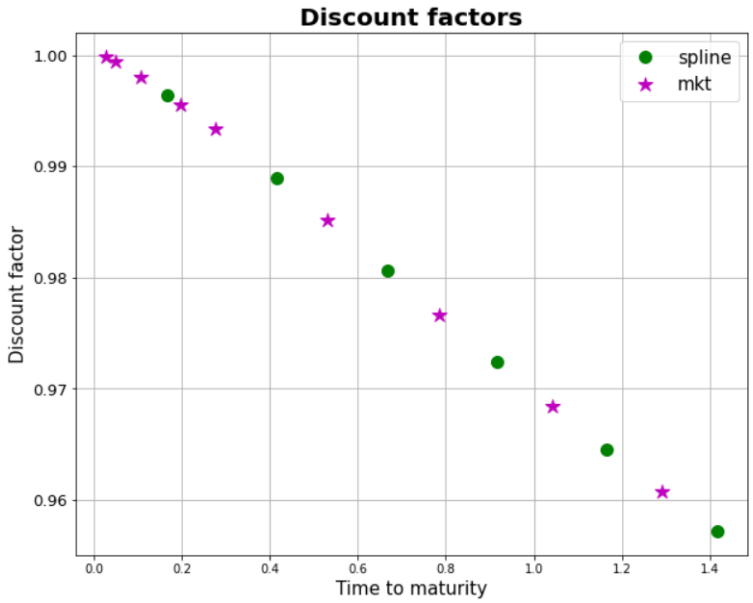


Figure 4: Discount factors: market data vs cubic interpolation

To price the floating rate note it is necessary to find the forward rates (`FwdRates`) considering that the first forward rate is the known coupon rate and the forward rates for the following periods are given by:

$$F(0,T_i,T_{i+1}) = \frac{1}{\alpha(T_i,T_{i+1})}\left[\frac{P(0,T_i)}{P(0,T_{i+1})} - 1\right] \quad i = 1,...,N$$

In the Python code the forward rates are computed in a vectorized form. After adding the spread to each forward rate, the cash flows and the discounted cash flows are computed as before, with the only difference that the coupon rate in that case is not fixed but is given by the forward rates plus the spread:

$$CF(i) = Notional \times F(0,T_i,T_{i+1}) \times \alpha(T_i,T_{i+1}), \quad i = 1,...,N$$

$$CF(N) = Notional \times F(0,T_{N-1},T_N) \times \alpha(T_{N-1},T_N) + Notional$$

and the discounted cash flow payments as:

$$DCF(i) = CF(i) \times P(0,T_i), \quad i = 1,...,N$$

Therefore, the price of the bond is:

$$FRN(0) = \sum_{i=1}^{n} P(0,T_i) \cdot CF(i)$$

which, for the given inputs, amounts to 103.242.

2.3 Structured Note 1

Data:

- Maturity: 10-Jan-2028
- Notional: Euro 100
- Coupon: 6mEuribor + 100bp
- Min Coupon: 0.4%; Max Coupon: 2%
- Frequency: semiannual payments
- Day rolling: Modified following
- Day count convention (basis): act/360

The first structured note to price is characterized by a notional of one hundred euros, maturity on the 10th January 2028, Actual/360 day count convention and semi-annual payments. The coupon rate is the six-month Euribor, 3.029%, plus a 1% spread; the minimum and the maximum coupon rates are respectively fixed at 0.4% and 2%. In the following, two different replicating strategies will be shown. For both strategies, the schedule of payments and the forward rates are computed with the procedure explained above.

Given that, considering X as the Euribor value:

$$\begin{cases} CPN = 0.4\% \iff X \leq -1\% + 0.4\% \leq -0.6\% \\ CPN = 2\% \iff X \geq -1\% + 2\% \geq 1\% \\ CPN = X + 1\% \ otherwise \end{cases}$$

The first replicating portfolio combines a short vanilla floor with 1% strike, a long vanilla floor with -0.6% strike and a fixed-rate bond with coupon rate equal to the maximum coupon rate given. To handle the negative prices, the floorlets are priced through the Bachelier pricing formula for put options, which is is given by:

$$P_0(K) = (K - F_0)\Phi(-d) + \sigma\sqrt{T}\phi(-d)$$

with

$$d = \frac{F_0 - K}{\sigma\sqrt{T}}$$

The inputs needed are: K, which is the strike price, F_0, the forward price following a normal distribution, σ, that is the flat volatility, and the maturity T. $\Phi(\cdot)$ and $\phi(\cdot)$ are, respectively, the cumulative distribution function and the probability density funciton of a normal standard variable. Translating the formula into a Python code, the floor price is derived by the function `Floor_Bachelier(K, df, FwdRate, sigma, T, tenor, Notional)`, which computes both the floorlet prices and the floor price, seen as the sum of all floorlets multiplied by the notional. The i-th floorlet is defined as `F[i]`.

Specifically, `K` is the strike price, `df` is the set of discount factors, one for each floorlet, `FwdRate` collects all the forward rates computed for each option, `T` is the maturity of each floorlet, and the `tenor` identifies the set of accrual factors; σ is the flat volatility, which is quoted and can be found through Refinitiv searching for 'CAPF'. Specifically, the flat volatility of a floor with 1% strike amounts to 1.0375% while the one corresponding to a floor with -0.6% strike is 1.084%. The fixed rate bond with coupon rate equal to the maximum coupon rate (2%) is priced as in paragraph 2.1. Therefore, by implementing the first strategy we get:

$$ReplicatingPortfolio1 = Floor(-0.6\%) - Floor(1\%) + Fixed(2\%)$$

the price of the structured note is 96.471. The same price can be derived through other replicating portfolios; one additional way to compute the price of this structured note is combining two vanilla caps and a fixed-rate bond with coupon rate equal to the minimum coupon rate.
Similarly as before, the Bachelier pricing formula for call options is involved to price the caplets needed :

$$C_0(K) = (F_0 - K)\Phi(d) + \sigma\sqrt{T}\phi(d)$$

with

$$d = \frac{F_0 - K}{\sigma\sqrt{T}}$$

In the Python code the function computing each caplet and, finally, the cap, is defined as `Cap_Bachelier(K, df, FwdRate, sigma, T, tenor, Notional)`, where the inputs are the same as the ones used in the function for pricing the floors.
Given all the inputs, the function computes the price of each caplet, `C[i]`, and the cap, which corresponds to the sum of all the caplets multiplied by the notional. The same flat volatilities as the ones taken to price the floors are used and the replicating strategy is the following:

$$ReplicatingPortfolio1bis = -Cap(1\%) + Cap(-0.6\%) + Fixed(0.4\%)$$

which leads to the same price found before 96.471.

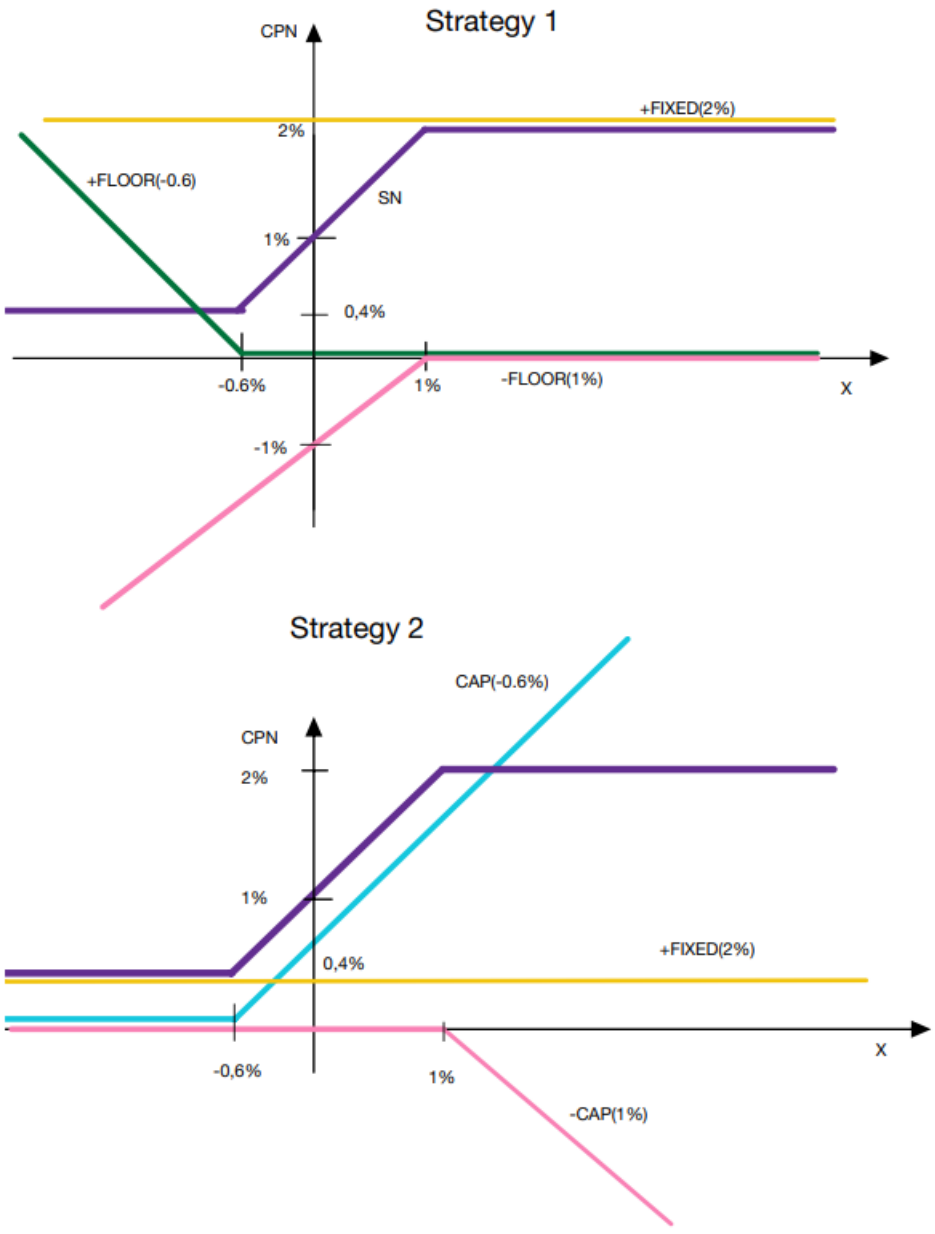


Figure 5: Replicating strategies

2.4 Structured Note 2

Data:

- Maturity: 15-Mar-2027
- Notional: Euro 100
- Coupon: 3mEuribor + 150bp if 3mEuribor $\leq$ 0,5%; 0,9% otherwise.
- Frequency: semiannual payments
- Day rolling: Modified following
- Day count convention (basis): act/360

The last fixed-income instrument to price is a structured note maturing on the 15th March 2027, notional of 100 euros, quarterly payments and actual/360 day count convention. The coupon rate is set equal to the three-months Euribor (2.482%) plus a 1.5% spread, if the Euribor is lower or equal to 0.5%, and 0.9% elsewhere.

As done before, two different replicating strategies are implemented: the first one combines a floor with a fixed-rate bond and a cash-or-nothing digital call, whereas the second combines a floating rate note with a cap and the same cash-or-nothing digital call (CNDC). To price the fixed-rate bond, the floating rate note, the cap and the floor the procedure is the same exploited in the previous paragraphs. In order to price the cash or nothing digital call the Bachelier formula is revised as follows:

$$CNDC = \Phi(\frac{F_0 - K}{\sigma\sqrt{T}})$$

The formula is implemented in the code as the function `Cash_Nothing_Digital_Call(K, df, sigma, FwdRates, T, tenor, Notional)`, with the same inputs considered before. The first replicating portfolio is made of a short floor and a cash-or-nothing digital call, both with 0.5% strike, and a fixed-rate bond with coupon rate equal to 2%. Indeed:

$$\begin{cases} CPN = X + 1.5\% \iff X \leq 0.5\% \\ CPN = 0.9\% \ otherwise \end{cases}$$

Thus, as $X = 0.5\%$ the $CPN = 2\%$, which is the coupon rate used to price the fixed-rate bond. Moreover, since cash-or-nothing digital needs to account for the "jump" in the coupon, given by $2\% - 0.9\% = 1.1\%$, the strategies are:

$$ReplicatingPortfolio2 = -Floor(0,5\%) + Fixed(2\%) - 1.1\%CNDC(0.5\%)$$

and

$$ReplicatingPortfolio2bis = Floating\_rate\_note - Cap(0.5\%) - 1.1\%CNDC(0.5\%)$$

The flat volatility for a 0.5% strike option is 1.063%, set the other input data given. The price obtained by the implementation of both strategies is equal to 93.128.

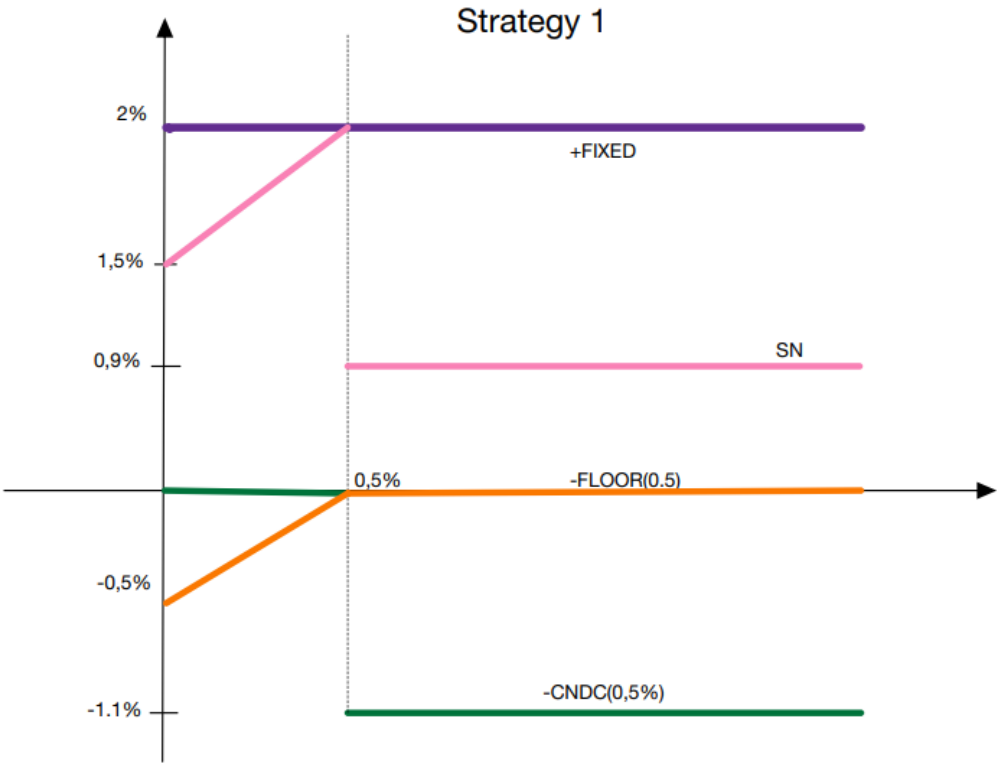


Figure 6: Replicating strategies

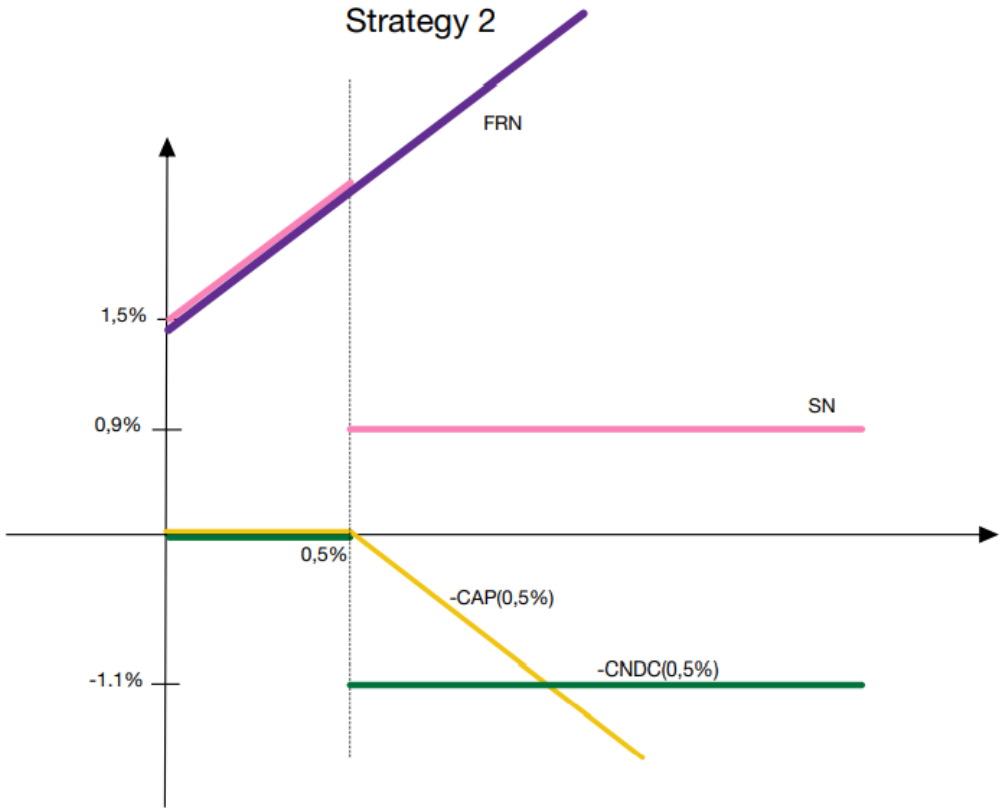


Figure 7: Replicating strategies

3 Montecarlo simulation in the Vasicek model

3.1 The Vasicek Model

The Vasicek model is a mathematical framework utilized in finance to describe the temporal evolution of interest rates. It posits that interest rates are governed by a mean-reverting Ornstein-Uhlenbeck process, where the interest rate at any given moment is influenced by both the long-term average interest rate and the deviation from this average. This model can be represented mathematically as a stochastic differential equation, as seen in equation (1).

$$dr(t) = k(\theta - r(t))dt + \sigma dW(t), \;\; r(0) = r_0 \tag{1}$$

, where k, θ, σ, r_0 are positive constants.

In this equation, $r(t)$ represents the interest rate at time t, k denotes the reversion speed, θ signifies the long-term average interest rate, σ represents the volatility of interest rates, and $dW(t)$

represents a Brownian motion. The reversion speed, k, determines the speed at which the interest rate returns to the long-term average. If k is large, this indicates a rapid return to the average, while a small value of k indicates a slow return. The long-term average interest rate, θ, reflects the average level of interest rates over an extended period, and the volatility of interest rates, σ, determines the degree of variability in interest rates over time.

3.2 Calibration of the Vasicek model to the interest rate curve

The process of calibrating the Vasicek model by using the data of interest rates was initiated by obtaining a set of data from the Refinitiv platform as explained in the introductory chapter. Unfortunately, it is not possible to download enough data to calibrate the Vasicek model over a time horizon of one year, and thus, it was necessary to interpolate the data to a larger set of data points. This was achieved through dividing the year into days and using the `CubicSpline` method from `scipy.interpolate` to perform a daily interpolation.

The result of this process was a larger set of data that was used to calibrate the Vasicek model, as shown in the accompanying table.

	R	Time	df	ql_dates	ttm
0	0.003937	2023-02-09	0.999891	February 9th, 2023	0.005556
1	0.005565	2023-02-10	0.999830	February 10th, 2023	0.008333
2	0.007012	2023-02-11	0.999767	February 11th, 2023	0.011111
3	0.008298	2023-02-12	0.999701	February 12th, 2023	0.013889
4	0.009441	2023-02-13	0.999633	February 13th, 2023	0.016667
...	...	...	...	...	...
360	0.030821	2024-02-04	0.968818	February 4th, 2024	1.005556
361	0.030826	2024-02-05	0.968730	February 5th, 2024	1.008333
362	0.030832	2024-02-06	0.968642	February 6th, 2024	1.011111
363	0.030837	2024-02-07	0.968554	February 7th, 2024	1.013889
364	0.030842	2024-02-08	0.968466	February 8th, 2024	1.016667

365 rows × 5 columns

The objective of the calibration process was to determine the values of the parameters k, θ, σ, and r_0 such that the Mean Squared Error (MSE) between the market spot rates, R_{market}, and the model spot rates, R_{model}, was minimized. The MSE was calculated as follows:

$$\sum_{i=1}^{n}(R_{market}(t,T_i) - R_{model}(t,T_i))^2 \tag{2}$$

To calculate R_{model}, the discount factor was firstly computed using the formula:

$$P(t,T) = A(t,T)e^{-B(t,T)r(t)} \tag{3}$$

where

$$A(t,T) = e^{-\frac{\sigma^2}{2k^2}(B(t,T)-T+t)-\frac{\sigma^2 B(t,T)^2}{4k}} \tag{4}$$

$$B(t,T) = \frac{1}{k}\left(1 - e^{-k(T-t)}\right) \tag{5}$$

Then, using the relationship in equation (2), R was calculated by means of a function called `ComputeR(ttm, k, theta, sigma, r0)`. A function called `mse(par, ttm, MRKdata)` was written to calculate the MSE between R_{model} and R_{market}, and a function called `calibration(param, ttm, MRKdata)` was developed to find the optimal parameters given a starting point.

However, as the Mean Squared Error to be minimized was not a convex function, multiple local minima exist, thereby rendering it challenging to determine the global minimum. To address this issue, a grid-based approach was implemented. The function `CreateGrid()` was utilized to generate a set of starting points for the parameter optimization process. The function `FindBestParams(ttm, MRKdata)` was then employed to iterate the optimization search, varying the starting point among the points from the grid and resulting in multiple sets of parameters and corresponding Mean Squared Error values. The optimal parameters were ultimately determined as the set yielding the minimum Mean Squared Error.

After a prolonged search for the minimum Mean Squared Error and, thus, the optimal parameters (with a runtime of approximately 23 hours), the following values were obtained:

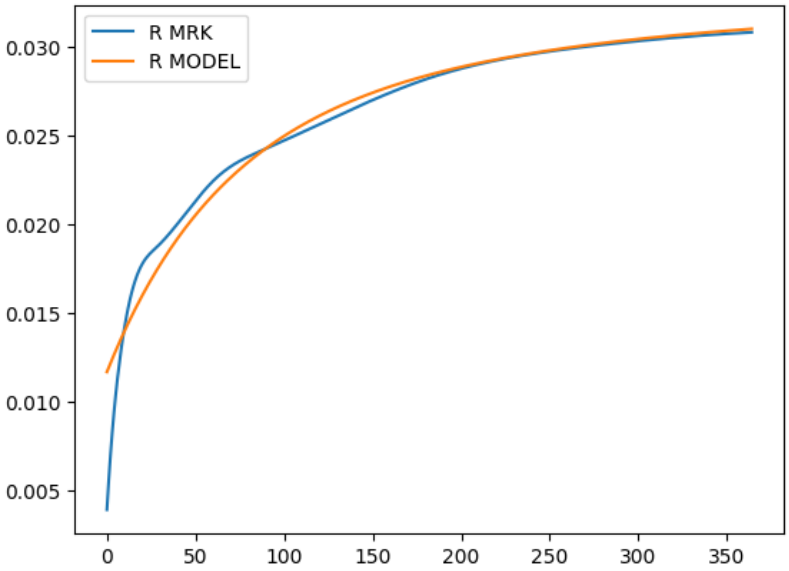
$r_0 = 0.015263159819135117$

$k = 3.600008006536917$

$\theta = 0.0382381139289971$

$\sigma = 0.019036071875234508$

The comparison between the market spot rates, R_{market}, and the computed model spot rates, R_{model}, is depicted in the following figure:



For all the following simulations of the MMA, the 6-months Euribor and the ZCB T refers to daily data.

3.3 Simulate the risk-neutral dynamics of interest rates on 1 year time horizon in terms of: MMA and 6m Euribor rates

Now that the optimal parameters of the Vasicek model have been found, we can simulate the risk-neutral dynamics of interest rates. First, the dynamics of interest rates were simulated in terms of a Money Market Account (MMA). We define $B(t)$ to be the value of a MMA at time $t \geq 0$. We assume $B(0) = 1$ and that the bank account evolves according to the following differential equation:

$$dB(t) = r_t B(t)dt, B(0) = 1 \tag{6}$$

where $r(t)$ is a positive function of time. As a result,

$$B(t) = \exp\left(\int_0^t r(s)ds\right) \tag{7}$$

The above definition tells us that investing a unit amount at time 0 yields at time t the value in (8), and $r(t)$ is the instantaneous rate at which the bank account accrues. In our case, $r(t)$ is simulated with the Vasicek model.

At this point, it is clear that we must simulate $r(t)$. Integrating equation (1), we obtain, for each $s \leq t$,

$$r(t) = r(s)e^{-k(t-s)} + \theta(1 - e^{-k(t-s)}) + \sigma \int_s^t e^{-k(t-u)}dW(u) \tag{8}$$

such that $r(t)$ conditional on $\mathcal{F}_s$ is normally distributed with mean and variance given respectively by

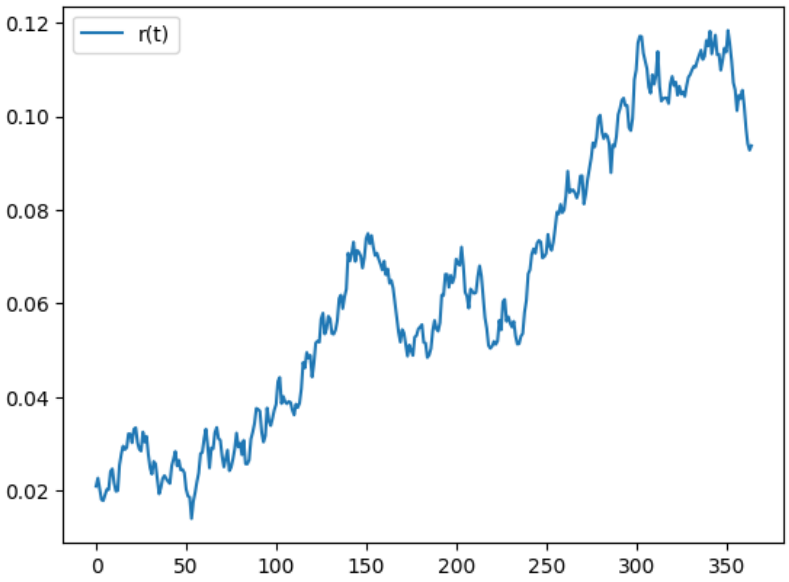
$$E[r(t)|\mathcal{F}_s] = r(s)e^{-k(t-s)} + \theta\left(1 - e^{-k(t-s)}\right) \tag{9}$$

$$\text{Var}[r(t)|\mathcal{F}_s] = \frac{\sigma^2}{2k}\left(1 - e^{-2k(t-s)}\right) \tag{10}$$

The function $rt(k, \theta, \sigma, T, r_0)$ is used to simulate a path of $r(t)$. In particular, to calculate the integral, the trapezoidal rule is used. In detail, the formula used is

$$r(t) = r(t-1)e^{-k(t-s)} + \theta(1 - e^{-k(t-s)}) + \epsilon_t\sqrt{\frac{\sigma^2}{2k}\left(1 - e^{-2k(t-s)}\right)} \tag{11}$$

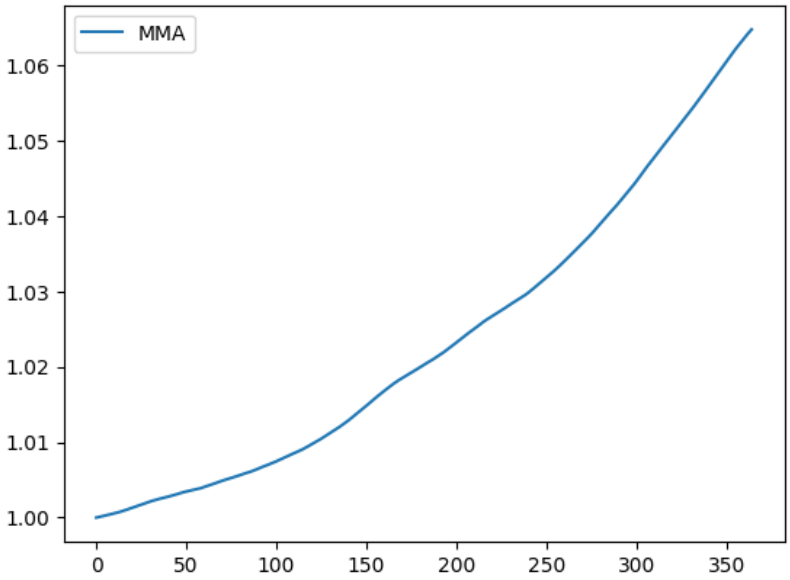
where $\epsilon_t \sim \mathcal{N}(0,1)$. Below it is illustrated a path of $r(t)$.



At this point, we are able to simulate the dynamics of a MMA, in particular through the function `MMA(T,r)`, which uses $r(t)$ previously simulated. In particular, to calculate the MMA, the following function is used:

$$MMA(t) = MMA(t-1)\exp\left(\int_{t-1}^{t} r(s)ds\right), MMA(0) = 1 \tag{12}$$

Below it is illustrated the path of the MMA generated by the previous $r(t)$.



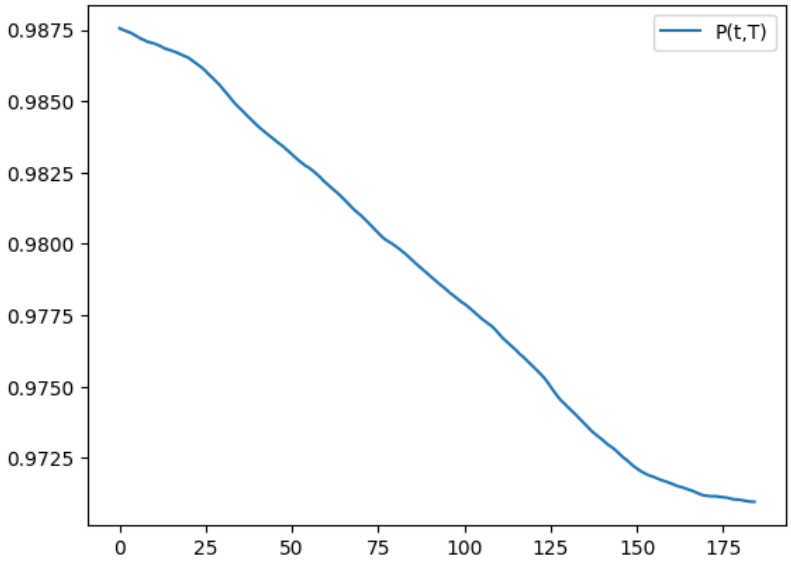
Now, the dynamics of interest rates in terms of the Euribor 6-month rate will be simulated. To do this, we first need to simulate the path of the discount factor, $P(t,T)$. This will be done by using the function `PtT(k, theta, sigma,r)`, which calculates $P(t,t_i+1)$ using the following formula:

$$P(t-1,t) = A(t-1,t)\exp\left(-B(t-1,t)r(t-1)\right) \tag{13}$$

where A and B are given by equations (5) and (6), respectively. Using this function, we will calculate the first spot rate, $P(0,1)$, and the subsequent forward rates, $P(0,1,2),\ldots,P(0,t-1,t)$. To calculate $P(t,T)$, we will use the function `computePiiplus6month(p)`, which uses a rolling window to calculate

$$P(0,6m) = \prod_{i=1}^{6m} P(i-1,i), \quad P(1,6m+1) = \prod_{i=2}^{6m+1} P(i-1,i),\ldots \tag{14}$$

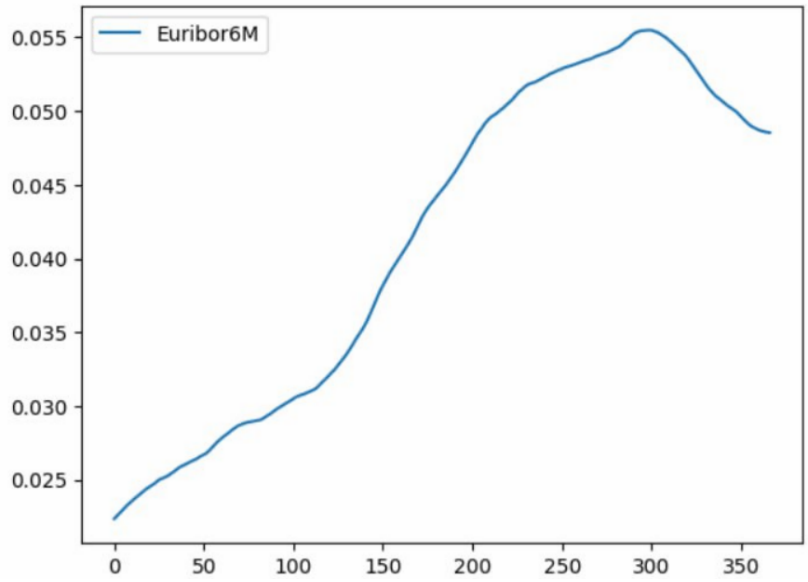
The results that combines all the $P(t,t+6m)$ into $P(t,T)$ can be seen in the image below.



Having now computed $P(t,T)$, we can then move on to calculating the Euribor 6-month rate using the function `Euribor(piiplus6m)`. This function uses the following formula to calculate the Euribor 6-month rate:

$$Euribor(t-1,t) = \frac{1}{\alpha(t-1,t)}\left[\frac{1}{P(t-1,t)} - 1\right] \tag{15}$$

The result obtained can be seen in the image below.



3.4 Compute the price of 1YR-zcb by Monte Carlo

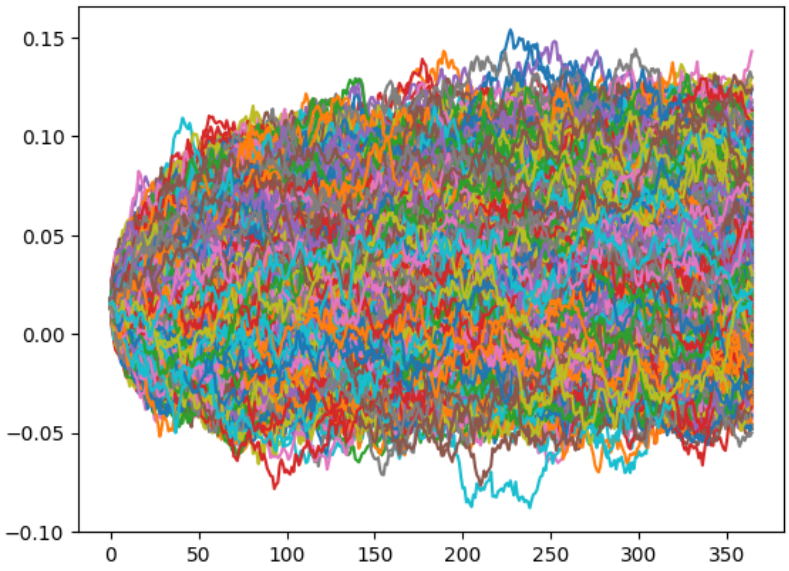
In order to calculate the price of a one year zero-coupon bond (ZCB), it is necessary to use the formula

$$P(0,1) = \exp\left(-\int_0^1 r(s)ds\right) \tag{16}$$

This can be achieved through the function `POT(nsim, T)`.

However, the price of the ZCB will only be given by the one path generated. For this reason, it is more accurate to perform this process with many different path of $r(t)$, calculate the prices of all ZCBs derived from these simulations, and take an average. Therefore, through the `montecarlo(nsim)` function, it is possible to set the number of simulations to be carried out. In the

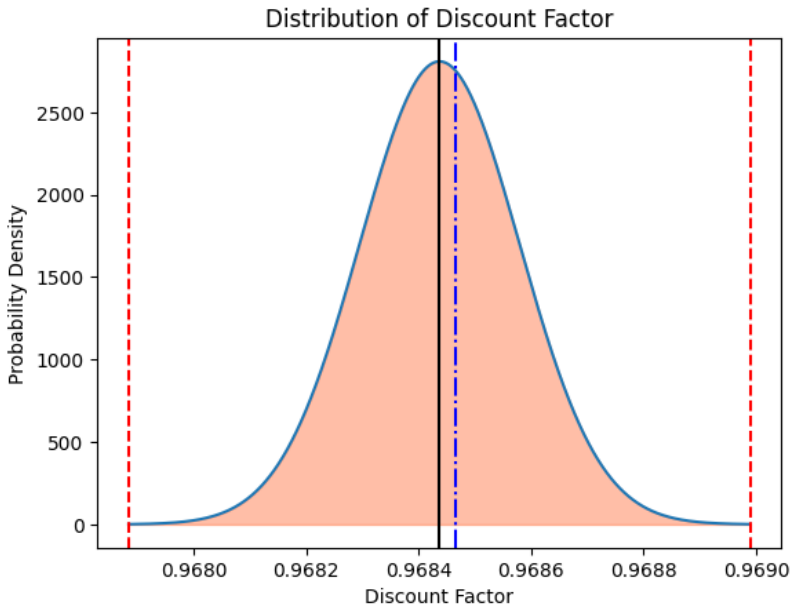
results shown below, 10000 simulations were carried out .



The Monte Carlo simulation was performed to estimate the price of a Zero Coupon Bond (ZCB). The estimated price obtained from the simulation was 0.96843, and it was compared with the true price, which was obtained from Refinitiv's data and was 0.96846.

To further validate the accuracy of the simulation results, a confidence interval was constructed at a level of 99.99 percent on the distribution generated by the simulations. The visualization of the confidence interval indicated that the true price of the ZCB was within the bounds of the interval,

providing evidence that the simulation accurately estimated the price of the ZCB.



4 Classical vs multicurve approach in pricing fixed-income instruments

4.1 Introduction to multicurve and construction of the interest rate curve

The valuation of fixed income instruments involves two main components, namely discount factors and forward rates. So far, the so called single curve classical approach has been employed. This approach is characterized by the utilization of the same interest rate curve for both discounting and the calculation of forward rates of all tenors. This implies that the same interest rate curve is employed both in the determination of the present value of future cash flows and in the computation of forward rates for all periods of the considered fixed income instrument. The above code for pricing the portfolio of fixed income instruments uses the Overnight Index Swap (OIS) curve. However, after the financial crisis of 2008, the industry has moved away from this practice and adopted a new paradigm: the multicurve approach. The multicurve modern approach employs two distinct curves for valuation purposes. Discounting, which involves the calculation of the present value of future cash flows, uses the OIS curve. The forward curves, on the other hand, are utilized for the calculation of forward rates. These curves must be constructed according to the tenor of the underlying rate of the contract (usually 1 month, 3 months, 6 months, or 12 months). In this code, forward rates are computed using the Euro Interbank Offered Rate (Euribor) curve. This approach is considered to be more sophisticated than the single curve approach and allows for a more precise representation of the fixed income market.

The first step is to construct the interest rate curve and this is done in a similar way to paragraph one. Data of the 3-month Euribor are downloaded from Refinitiv by searching for the ticker 0#EURABSETS=R. Initially, the data related to Euribor was imported from the Excel file named "euribor.xlsx" and the content was stored in a data frame named `data_euribor`. Then the code removes the "Instrument" column from the data frame and renames the remaining columns. It converts the `MaturityDate` column to a datetime format and removes any missing values. After that, it defines the trade date of January 30th, 2023, and sets the calendar to the TARGET calendar, which is the calendar used by the Eurozone. Then it calculates the time to each maturity date in the "Euribor" DataFrame in terms of years, using the Actual360 day count convention. The resulting `Time in year` values are added as a new column to the "Euribor" Dataframe. Finally, the code filters the Dataframe to only include maturities that have not yet passed the trade date. Subsequently, the code creates a new data frame called `irc_m` which contains the columns "Time in year" and "Discount". The code then calculates the spot rate from the discount factors and stores the result in the "R" column of the `irc_m` data frame. Finally, the code uses Seaborn's lineplot function to plot two graphs, one for the discount curve and another for the spot rates.

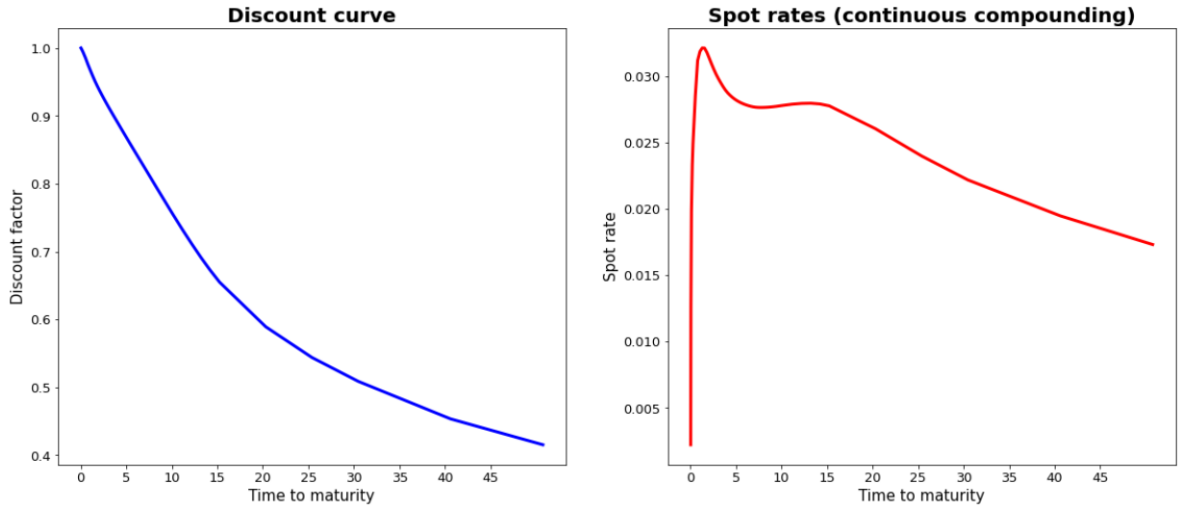


Figure 8: Discount curve and Spot rate curve in the multicurve approach

4.2 Valuation under the multiple curve approach

1. The valuation of the fixed-rate bond under the multicurve approach is the same as under the single curve approach. This is due to the fact that the pricing formula for fixed-rate bond does not involve forward rates. For this reason it is not necessary to price it again, the code would be exactly the same as the one presented in the section "Pricing Fixed-income Portfolios".

2. On the contrary, the price of the floating rate note slightly differs in the two different approaches as the pricing formula uses forward rates, now computed starting from the Euribor and no

longer from OIS curve.
Initially, the code defines the characteristics of the instrument (in the same way as in paragraph 2.2):

- Maturity: 18-Jun-2024
- Notional: Euro 100
- Coupon: 3mEuribor + 200 BP
- Frequency: quarterly payments
- Day rolling: Modified following
- Day count convention (basis): act/360

Then, the schedule of payments is computed in the usual way, as well as the Accrual Factor `(AF)` and the cash flow times `(CFLowTimes)`. AF array contains year fractions between start dates and end dates of coupon payment periods. `CFLowTimes` array contains the year fractions between the effective date and end dates of coupon payment periods. After that, the code calculates the discount factors `(DiscountFactors)` for each coupon payment period using the `CubicSpline` interpolation method from the SciPy library and the discount factor data stored in `irc`, i.e. using the OIS interest rate curve. The discount factors are used for computing the present value of future cash flows `(DiscountedCFlowAmounts)`. Then, forward rates `(DiscountFactorsF)` are calculated by using again the `CubicSpline` interpolation method and the discount factor data stored in `irc_m`, i.e. using now the Euribor interest rate curve. The forward rates are used for computing the coupon rates `(CpnRates)`. The coupon rates are calculated as the sum of forward rates and spread. Finally, the cash flow amounts are calculated by multiplying coupon rates with year fractions and notional. These cash flow amounts are then discounted and summed up to get the price of the floating rate note, which is now equal to 103.484. As expected, the price of the FRN computed under the multicurve approach is higher than the price of the one computed under the single curve approach; the reason is that the 3m-Euribor curve is riskier than the OIS curve, since it also incorporates the credit risk component. Therefore, as in the multicure framework the coupon payments are computed with discount factors deriving from the 3m-Euribor curve, they are higher than the payments computed using just the OIS curve. However, since the discount factors used to discount all the cash flows is in both cases the same, the price of the FRN computed under the multicurve approach is higher than the price of the same FRN priced under the single curve approach. Then the code constructs the pricing table that summarizes all the necessary information.

	StartDates	EndDates	AF	CFLowTimes	DiscountFactors	FwdRates	CpnRates	CFlowAmounts	DiscountedCF lowAmounts
0	December 19th, 2022	March 20th, 2023	0.252778	0.136111	0.997243	NaN	0.044820	1.132950	1.129826
1	March 20th, 2023	June 19th, 2023	0.252778	0.388889	0.989831	0.029586	0.049586	1.253433	1.240687
2	June 19th, 2023	September 18th, 2023	0.252778	0.641667	0.981424	0.035076	0.055076	1.392211	1.366349
3	September 18th, 2023	December 18th, 2023	0.252778	0.894444	0.973103	0.036141	0.056141	1.419126	1.380956
4	December 18th, 2023	March 18th, 2024	0.252778	1.147222	0.965079	0.033458	0.053458	1.351309	1.304120
5	March 18th, 2024	June 18th, 2024	0.255556	1.402778	0.957551	0.033426	0.053426	101.365319	97.062467

Figure 9: Schedule of payments for the floating rate note under the multicurve approach

Lastly, the code plots a graph of the discount factors calculated from a spline (in green) and the market discount factors (in magenta).

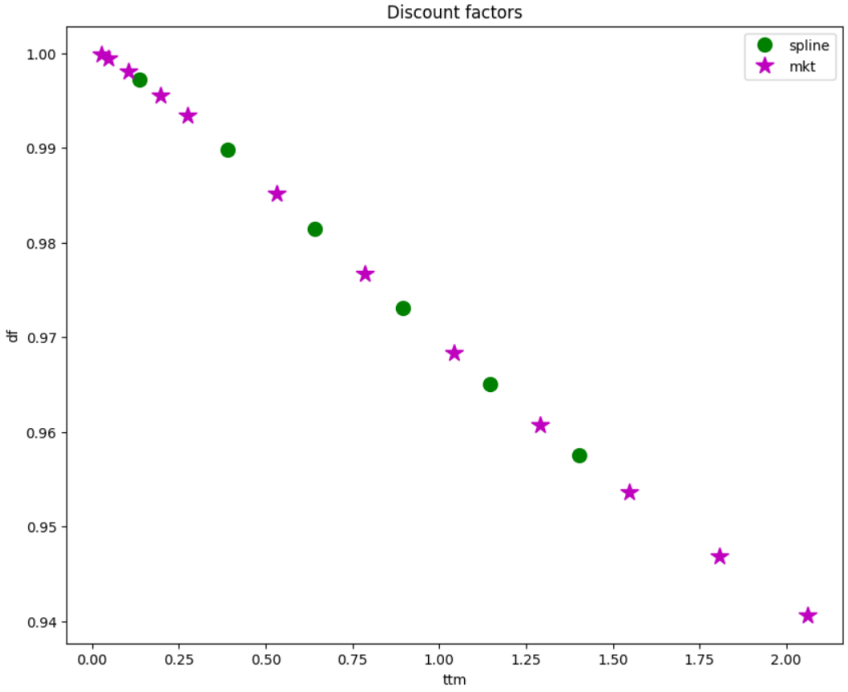


Figure 10: Discount factors: market data vs cubic interpolation

4.3 Appendix: Code and Flat Volatility

In [1]:
```
import pandas as pd
import numpy as np
import QuantLib as ql
from scipy import interpolate
import seaborn as sns
import matplotlib.pyplot as plt
import itertools
from datetime import timedelta, date
from scipy.interpolate import CubicSpline
from scipy.stats import t
from scipy.optimize import minimize
import random
from scipy.stats import norm
import warnings
warnings.filterwarnings("ignore")
```

In [2]:
```
df = pd.read_excel("ois.xlsx")
df.head()
```

Out[2]:

	Instrument	CF_NAME	CF_LAST	SEC_ACT_1	MATUR_DATE
0	EURESTOISONZ=R	ON ZERO YIELD	1.946045	0.999947	2023-02-08
1	EURESTOISTNZ=R	TN ZERO YIELD	2.016097	0.999891	2023-02-09
2	EURESTOIS1WZ=R	1W ZERO YIELD	2.362926	0.999424	2023-02-16
3	EURESTOIS1MZ=R	1M ZERO YIELD	2.433661	0.998026	2023-03-09
4	EURESTOIS2MZ=R	2M ZERO YIELD	2.612254	0.995559	2023-04-11

In [3]:
```
df = df.drop("Instrument",axis = 1)
df = df.rename(columns = {"CF_LAST":"Yield","CF_NAME":"EUR","SEC_ACT_1":"Discount","MATUR_DATE":"MaturityDate"})
df["MaturityDate"] = pd.to_datetime(df["MaturityDate"])
df = df.dropna()
df["ql_dates"] = df.MaturityDate.apply(ql.Date().from_date)
df = df.reset_index(drop = True)
df.head()
```

Out[3]:

	EUR	Yield	Discount	MaturityDate	ql_dates
0	ON ZERO YIELD	1.946045	0.999947	2023-02-08	February 8th, 2023
1	TN ZERO YIELD	2.016097	0.999891	2023-02-09	February 9th, 2023
2	1W ZERO YIELD	2.362926	0.999424	2023-02-16	February 16th, 2023
3	1M ZERO YIELD	2.433661	0.998026	2023-03-09	March 9th, 2023
4	2M ZERO YIELD	2.612254	0.995559	2023-04-11	April 11th, 2023

In [4]:
```
trade_date = ql.Date(30, 1, 2023)
calender_EUR = ql.TARGET()
times = []
for i in range(len(df["MaturityDate"])):
    times.append(ql.Actual360().yearFraction(trade_date,df['ql_dates'][i]))
df["Time in year"] = times
df = df[df["Time in year"]>= 0]
df.head()
```

Out[4]:

	EUR	Yield	Discount	MaturityDate	ql_dates	Time in year
0	ON ZERO YIELD	1.946045	0.999947	2023-02-08	February 8th, 2023	0.025000
1	TN ZERO YIELD	2.016097	0.999891	2023-02-09	February 9th, 2023	0.027778
2	1W ZERO YIELD	2.362926	0.999424	2023-02-16	February 16th, 2023	0.047222
3	1M ZERO YIELD	2.433661	0.998026	2023-03-09	March 9th, 2023	0.105556
4	2M ZERO YIELD	2.612254	0.995559	2023-04-11	April 11th, 2023	0.197222

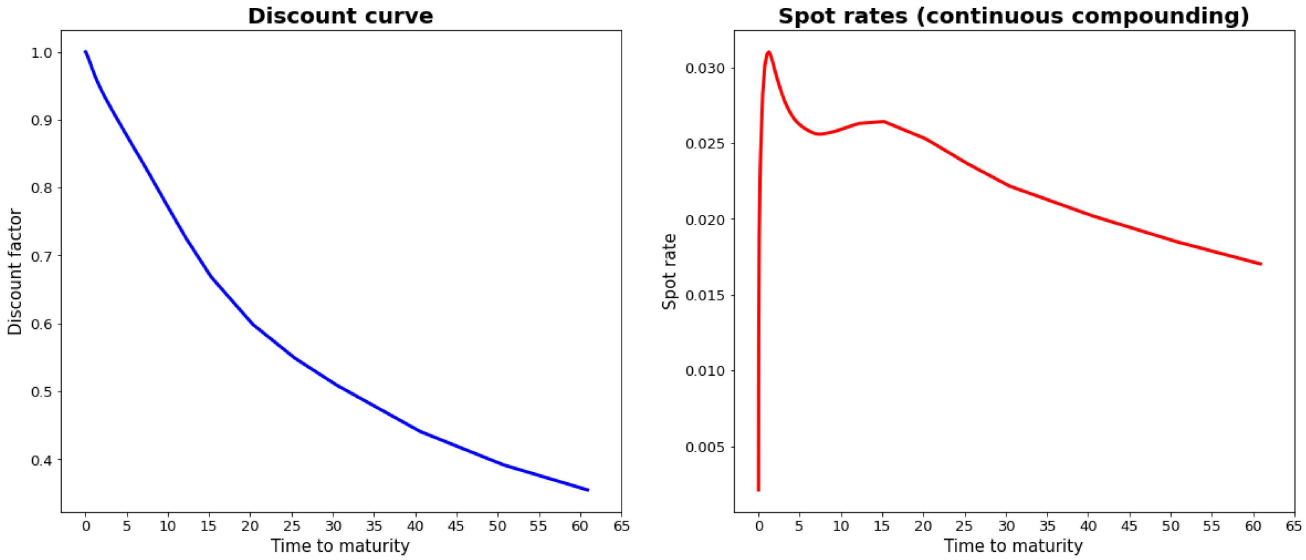
In [5]:
```
irc = df[["Time in year","Discount"]]
irc = irc.rename(columns = {"Time in year": "ttm","Discount": "df" })
irc["R"]= - (np.log(irc["df"]))/irc["ttm"]
irc.head()
```

Out[5]:

	ttm	df	R
0	0.025000	0.999947	0.002112
1	0.027778	0.999891	0.003937
2	0.047222	0.999424	0.012195
3	0.105556	0.998026	0.018723
4	0.197222	0.995559	0.022568

In [6]:
```
fig, axes = plt.subplots(1,2, figsize=(20,8))

sns.lineplot(data=irc, x="ttm", y="df", ax=axes[0], color='blue', lw=3)
axes[0].set_title('Discount curve', fontsize=20, fontweight='bold')
axes[0].set_xlabel('Time to maturity', fontsize=15)
axes[0].set_ylabel('Discount factor', fontsize=15)
axes[0].set_xticks(np.arange(0, 66, 5))
axes[0].tick_params(axis='both', which='major', labelsize=13)

sns.lineplot(data=irc, x="ttm", y="R", ax=axes[1], color='red', lw=3)
axes[1].set_title('Spot rates (continuous compounding)', fontsize=20, fontweight='bold')
axes[1].set_xlabel('Time to maturity', fontsize=15)
```

```
axes[1].set_ylabel('Spot rate', fontsize=15)
axes[1].set_xticks(np.arange(0, 66, 5))
axes[1].tick_params(axis='both', which='major', labelsize=13)
```



Excercise 1: PRICING FIXED-INCOME PORTFOLIOS

1. PRICING PORTFOLIO INSTRUMENTS

Price the portfolio of fixed-income instruments, whose features are described below:

(a) FIXED-RATE BOND

- Maturity: 12-Dec-2026
- Notional: Euro 100
- Coupon = %
- Frequency: semi-annual payments
- Day count convention (basis): act/act

In [7]:

```
# Input data for the fixed-rate bond
notional = 100
cpnRate = 0.03
trade_date = ql.Date(30, 1, 2023)

effectiveDate = trade_date
terminationDate = ql.Date(12,12,2026)

frequency = ql.Period('6M')

convention = ql.ModifiedFollowing
terminationDateConvention = ql.ModifiedFollowing
rule = ql.DateGeneration.Backward
calendar = ql.TARGET()
endOfMonth = False

# Compute the basic schedule
schedule = ql.Schedule(effectiveDate,
terminationDate,
frequency,
calendar,
convention,
terminationDateConvention,
rule,
endOfMonth)

print('Schedule:')
for i, d in enumerate(schedule):
    print (i+1, d)
```

```
Schedule:
1 January 30th, 2023
2 June 12th, 2023
3 December 12th, 2023
4 June 12th, 2024
5 December 12th, 2024
6 June 12th, 2025
7 December 12th, 2025
8 June 12th, 2026
9 December 14th, 2026
```

In [8]:

```
# Compute the PreviousCpnDate
scheduleShift = ql.Schedule(effectiveDate-frequency,
terminationDate,
frequency,
calendar,
convention,
terminationDateConvention,
rule,
endOfMonth)

print('ScheduleShift: ')

for i, d in enumerate(scheduleShift) :
```

```
    print(i+1, d)

previousCpnDate = scheduleShift[1]
print('PreviousCpnDate: ')
print(previousCpnDate)
```

```
ScheduleShift:
1 July 29th, 2022
2 December 12th, 2022
3 June 12th, 2023
4 December 12th, 2023
5 June 12th, 2024
6 December 12th, 2024
7 June 12th, 2025
8 December 12th, 2025
9 June 12th, 2026
10 December 14th, 2026
PreviousCpnDate:
December 12th, 2022
```

In [9]:

```
# Pricing the fixed-rate bond
nDates = len(schedule)-1
StartDates = [schedule[i] for i in range(nDates)]
StartDates[0] = previousCpnDate
EndDates = [schedule[i+1] for i in range(nDates) ]
AF = np.array([ql.ActualActual(ql.ActualActual.Bond).yearFraction(StartDates[i],EndDates[i]) for i in range(nDates)]).reshape(nDates,1)
CFLowTimes = np.array([ql.ActualActual(ql.ActualActual.Bond).yearFraction(effectiveDate,EndDates[i]) for i in range(nDates) ]).reshape(nDates,1)
DiscountFactors = interpolate.CubicSpline(irc.loc[:,"ttm"],irc.loc[:,"df"],bc_type='natural')(CFLowTimes)
CpnRates = np.full((nDates,1), cpnRate)
CFlowAmounts = notional * CpnRates * AF
CFlowAmounts[-1] = CFlowAmounts[-1] + notional
DiscountedCFlowAmounts = CFlowAmounts * DiscountFactors
price_CB = sum(DiscountedCFlowAmounts )

print('price_CB:', price_CB)
```

```
price_CB: [101.47097664]
```

In [10]:

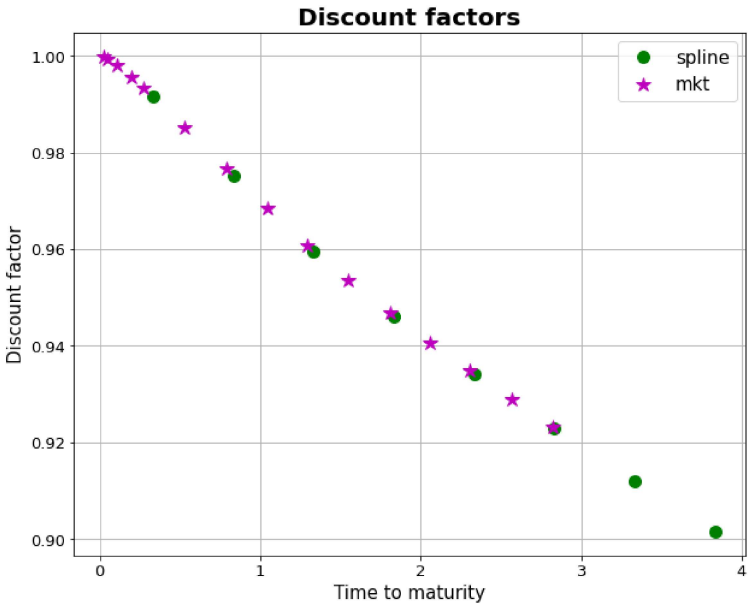
```
# Pricing table
data_frame = pd.DataFrame()
data_frame["StartDates"] = StartDates
data_frame["EndDates"] = EndDates
data_frame["AF"] = AF
data_frame["CFLowTimes"] = CFLowTimes
data_frame["CpnRates"] = CpnRates
data_frame["CFlowAmounts"] = CFlowAmounts
data_frame["DiscountedCFlowAmounts" ] = DiscountedCFlowAmounts

data_frame
```

Out[10]:

	StartDates	EndDates	AF	CFLowTimes	CpnRates	CFlowAmounts	DiscountedCFlowAmounts
0	December 12th, 2022	June 12th, 2023	0.5	0.333333	0.03	1.5	1.487390
1	June 12th, 2023	December 12th, 2023	0.5	0.833333	0.03	1.5	1.462645
2	December 12th, 2023	June 12th, 2024	0.5	1.333333	0.03	1.5	1.439286
3	June 12th, 2024	December 12th, 2024	0.5	1.833333	0.03	1.5	1.419310
4	December 12th, 2024	June 12th, 2025	0.5	2.333333	0.03	1.5	1.401321
5	June 12th, 2025	December 12th, 2025	0.5	2.833333	0.03	1.5	1.384428
6	December 12th, 2025	June 12th, 2026	0.5	3.333333	0.03	1.5	1.368206
7	June 12th, 2026	December 14th, 2026	0.5	3.833333	0.03	101.5	91.508391

In [11]:

```
# Discount Factors plot
plt.figure(figsize=(10,8))
plt.plot(CFLowTimes ,DiscountFactors, "go", label='spline', markersize=10)
plt.plot(irc.loc[1:15,"ttm"],irc.loc[1:15,"df"], "m*", label='mkt', markersize=13)
plt.title("Discount factors", fontsize=20, fontweight='bold')
plt.xlabel("Time to maturity", fontsize=15)
plt.ylabel("Discount factor", fontsize=15)
plt.xticks(np.arange(0, 5, 1), fontsize=13)
plt.yticks(fontsize=13)
plt.legend(fontsize=15)
plt.grid(True)
plt.show()
```



(b) Floating rate note

- Maturity: 18-Jun-2024
- Notional: Euro 100
- Coupon: 3mEuribor + 200 BP
- Frequency: quaterly payments
- Day rolling: Modified following
- Day count convention (basis): act/360

In [12]:

```
# Input data for the floating rate note
notional = 100
spread = 0.02
currentCpnRate = 0.02482 + spread # https://www.euribor-rates.eu/en/
effectiveDate = trade_date
terminationDate = ql.Date(18,6,2024)

dayCount = ql.Actual360()

frequency = ql.Period('3M')

convention = ql.ModifiedFollowing
terminationDateConvention = ql.ModifiedFollowing
rule = ql.DateGeneration.Backward

endOfMonth = False

# Compute the basic schedule
schedule = ql.Schedule(effectiveDate,
terminationDate,
frequency,
calendar,
convention,
terminationDateConvention,
rule,
endOfMonth)

print('Schedule:')
for i, d in enumerate(schedule):
    print (i+1, d)
```

```
Schedule:
1 January 30th, 2023
2 March 20th, 2023
3 June 19th, 2023
4 September 18th, 2023
5 December 18th, 2023
6 March 18th, 2024
7 June 18th, 2024
```

In [13]:

```
# Compute the PreviousCpnDate
scheduleShift = ql.Schedule(effectiveDate-frequency,
terminationDate,
frequency,
calendar,
convention,
terminationDateConvention,
rule,
endOfMonth)

print('ScheduleShift:')

for i, d in enumerate(scheduleShift) :
    print (i+1, d)

previousCpnDate = scheduleShift[1]
print('PreviousCpnDate: ')
print(previousCpnDate)
```

```
ScheduleShift:
1 October 31st, 2022
2 December 19th, 2022
```

```
3 March 20th, 2023
4 June 19th, 2023
5 September 18th, 2023
6 December 18th, 2023
7 March 18th, 2024
8 June 18th, 2024
PreviousCpnDate:
December 19th, 2022
```

In [14]:

```
nDates = len(schedule)-1
StartDates = [schedule[i] for i in range(nDates)]
StartDates[0] = previousCpnDate
EndDates = [schedule[i+1] for i in range(nDates)]

# alfa(Tn,Tn+1)
AF = np.array([dayCount.yearFraction(StartDates[i],EndDates[i]) for i in range(nDates)]).reshape(nDates,1)
CFLowTimes = np.array([ql.ActualActual(ql.ActualActual.Bond).yearFraction(effectiveDate,EndDates[i]) for i in range(nDates)]).reshape(nDates,1)
# Discount factors
DiscountFactors = interpolate.CubicSpline(irc.loc[:,"ttm"],irc.loc[:,"df"],bc_type='natural')(CFLowTimes)

FwdRates = (DiscountFactors[ :5]/DiscountFactors[1:]-1)/AF[:1]
FwdRates = np.insert(FwdRates, 0, np.nan).reshape(nDates,1)
CpnRates = FwdRates+spread
CpnRates[0] = currentCpnRate
CFlowAmounts = notional*CpnRates*AF
CFlowAmounts[-1] = CFlowAmounts[-1] + notional
DiscountedCFlowAmounts = CFlowAmounts*DiscountFactors
price_FRN = sum(DiscountedCFlowAmounts )

print('price_FRN:', price_FRN)
```

```
price_FRN: [103.24221481]
```

In [15]:

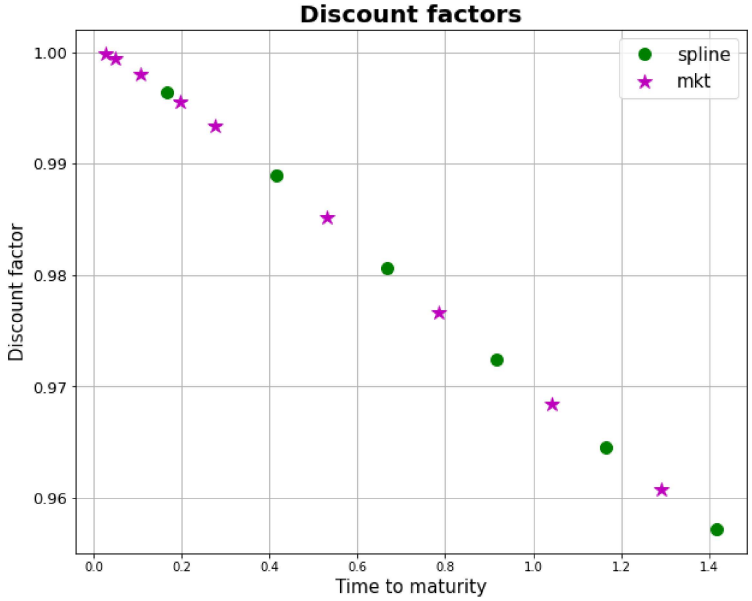
```
# Pricing table
data_frame= pd.DataFrame()
data_frame["StartDates"] = StartDates
data_frame["EndDates"] = EndDates
data_frame["AF"] = AF
data_frame["CFLowTimes"] = CFLowTimes
data_frame['DiscountFactors'] = DiscountFactors
data_frame["FwdRates"]= FwdRates
data_frame["CpnRates"] = CpnRates
data_frame["CFlowAmounts"] = CFlowAmounts
data_frame["DiscountedCF lowAmounts"] = DiscountedCFlowAmounts

data_frame
```

Out[15]:

	StartDates	EndDates	AF	CFLowTimes	DiscountFactors	FwdRates	CpnRates	CFlowAmounts	DiscountedCF lowAmounts
0	December 19th, 2022	March 20th, 2023	0.252778	0.166667	0.996412	NaN	0.044820	1.132950	1.128885
1	March 20th, 2023	June 19th, 2023	0.252778	0.416667	0.988926	0.029944	0.049944	1.262475	1.248495
2	June 19th, 2023	September 18th, 2023	0.252778	0.666667	0.980593	0.033620	0.053620	1.355392	1.329088
3	September 18th, 2023	December 18th, 2023	0.252778	0.916667	0.972382	0.033404	0.053404	1.349926	1.312644
4	December 18th, 2023	March 18th, 2024	0.252778	1.166667	0.964482	0.032406	0.052406	1.324695	1.277644
5	March 18th, 2024	June 18th, 2024	0.255556	1.416667	0.957162	0.030252	0.050252	101.284228	96.945458

In [16]:

```
# Discount Factors plot
plt.figure(figsize=(10,8))
plt.plot(CFLowTimes ,DiscountFactors, "go", label='spline', markersize=10)
plt.plot(irc.loc[1:9,"ttm"],irc.loc[1:9,"df"], "m*", label='mkt', markersize=13)
plt.title("Discount factors", fontsize=20, fontweight='bold')
plt.xlabel("Time to maturity", fontsize=15)
plt.ylabel("Discount factor", fontsize=15)

plt.yticks(fontsize=13)
plt.legend(fontsize=15)
plt.grid(True)
plt.show()
```



(c) Structured Note 1

- Maturity: January 10, 2028.
- Notional: Euro 100
- Coupon = 6m Euribor + 100BP.
- Min cpn = 0.4%, Max cpn = 2%.
- Day count convention (basis): act/360
- Frequency: semi-annual payments
- Day rolling: Modified following

In [17]:

```
# Input data for the structured note 1
trade_date = ql.Date(30,1,2023)
maturity = ql.Date(10,1,2028)
notional = 100
spread = 0.01
couponFrequency = ql.Semiannual
dayCounter = ql.Actual360()
currentcpn = 0.03029
maxCpn = 0.02
minCpn = 0.004


effectiveDate = trade_date
terminationDate = ql.Date(10,ql.January,2028)
dayCount = ql.Actual360()
frequency = ql.Period('6M')
convention = ql.ModifiedFollowing
terminationDateConvention = ql.ModifiedFollowing
rule = ql.DateGeneration.Backward
endOfMonth = False
calendar = ql.TARGET()


# Define the schedule of payments
schedule = ql.Schedule(effectiveDate,
                       terminationDate,
                       frequency,
                       calendar,
                       convention,
                       terminationDateConvention,
                       rule,
                       endOfMonth)

print('Schedule:')
for i, d in enumerate(schedule):
    print (i+1, d)
```

```
Schedule:
1 January 30th, 2023
2 July 10th, 2023
3 January 10th, 2024
4 July 10th, 2024
5 January 10th, 2025
6 July 10th, 2025
7 January 12th, 2026
8 July 10th, 2026
9 January 11th, 2027
10 July 12th, 2027
11 January 10th, 2028
```

In [18]:

```
scheduleShift = ql.Schedule(effectiveDate-frequency,
 terminationDate,
frequency,
calendar,
convention,
terminationDateConvention,
rule,
endOfMonth)
print('ScheduleShift:')
for i, d in enumerate(scheduleShift):
    print (i+1, d)

previousCpnDate = scheduleShift[1]
print('PreviousCpnDate:')
print(previousCpnDate)
```

```
ScheduleShift:
1 July 29th, 2022
2 January 10th, 2023
3 July 10th, 2023
4 January 10th, 2024
5 July 10th, 2024
6 January 10th, 2025
7 July 10th, 2025
8 January 12th, 2026
9 July 10th, 2026
10 January 11th, 2027
11 July 12th, 2027
12 January 10th, 2028
PreviousCpnDate:
January 10th, 2023
```

In [19]:

```
nDates = len(schedule)-1
StartDates = [schedule[i] for i in range(nDates)]
StartDates[0] = previousCpnDate
EndDates = [schedule[i+1] for i in range(nDates)]

AF = np.array([dayCount.yearFraction(StartDates[i],EndDates[i]) for i in range(nDates)]).reshape(nDates,1)
CashFlowTimes = np.array([ql.Actual360().yearFraction(effectiveDate, EndDates[i]) for i in range(nDates)]).reshape(nDates,1)
DiscountFactors = interpolate.CubicSpline(irc.loc[:,"ttm"], irc.loc[:,"df"], bc_type = "natural")(CashFlowTimes)
fwRates = (DiscountFactors[:-1]/DiscountFactors[1:]-1)/AF[:1]
fwRates = np.insert(fwRates, 0, np.nan).reshape(nDates,1)
```

```
fwRates[0] = currentcpn

print(fwRates)
```

```
[[0.03029   ]
 [0.03457656]
 [0.03159189]
 [0.02772566]
 [0.02524426]
 [0.02480763]
 [0.02333403]
 [0.02397448]
 [0.02374052]
 [0.02398198]]
```

In [20]:
```
# Floor with 1% strike:
K1 = 0.01
flat_vol1 = 0.010375

# Floor with -0.6% strike:
K2 = -0.006
flat_vol2 = 0.010840
```

In [21]:
```
# Cap with 1% strike:
K1 = 0.01
flat_vol1c = 0.010375

# Cap with -0.6% strike:
K2 = -0.006
flat_vol2c = 0.010840
```

In [22]:
```
def Floor_Bachelier(K, df, FwdRate, sigma, T, tenor, Notional):
    '''
    This function computes the Floor price using the Bachelier's pricing formula for floors, i.e. it computes the
    Floor price under the normal model.
    The inputs for the function are the following:
    K = Strike
    df = Discount Factor
    FwdRate = Forward Rates
    sigma = Volatility (flat)
    T = Maturity
    tenor
    N = Notional
    '''
    F = np.zeros(len(FwdRate))

    for i in range(len(FwdRate)):
        d = (FwdRate[i] - K) / (sigma * np.sqrt(T[i]))
        F[i] = df[i] * tenor[i] * ((K - FwdRate[i]) * norm.cdf(-d) + (sigma * np.sqrt(T[i]) * norm.pdf(-d)))
        floor = Notional * F.sum()
    return floor
```

In [23]:
```
def Cap_Bachelier(K, df, FwdRate, sigma, T, tenor, Notional):
    '''
    This function computes the Cap price using the Bachelier's pricing formula for caps, i.e. it computes the
    Cap price under the normal model.
    The inputs for the function are the following:
    K = Strike
    df = Discount Factor
    FwdRate = Forward Rates
    sigma = Volatility (flat)
    T = Maturity
    tenor
    N = Notional
    '''
    C = np.zeros(len(FwdRate))

    for i in range(len(FwdRate)):
        d = (FwdRate[i] - K) / (sigma * np.sqrt(T[i]))
        C[i] = df[i] * tenor[i] * ((FwdRate[i] - K) * norm.cdf(d) + (sigma * np.sqrt(T[i]) * norm.pdf(d)))
        cap = Notional * C.sum()
    return cap
```

In [24]:
```
Floor_1 = Floor_Bachelier(K1, DiscountFactors, fwRates, flat_vol1, CashFlowTimes, AF, notional)
Floor_2 = Floor_Bachelier(K2, DiscountFactors, fwRates, flat_vol2, CashFlowTimes, AF, notional)

print(Floor_1, Floor_2)
```

```
0.8252277125646874 0.20552209651465866
```

In [25]:
```
Cap_1 = Cap_Bachelier(K1, DiscountFactors, fwRates, flat_vol1c, CashFlowTimes, AF, notional)
Cap_2 = Cap_Bachelier(K2, DiscountFactors, fwRates, flat_vol2c, CashFlowTimes, AF, notional)

print(Cap_1, Cap_2)
```

```
8.850712561023881 15.761132644978023
```

In [26]:
```
FixedCpn = np.zeros(len(fwRates))
DiscountedCFFixed = np.zeros(len(fwRates))
Fixed = np.zeros(len(fwRates))
for i in range(len(fwRates)):
    FixedCpn[i] = notional * maxCpn * AF[i]
    DiscountedCFFixed[i] = FixedCpn[i] * DiscountFactors[i]
    DiscountedCFFixed[-1] = DiscountedCFFixed[-1] + (notional * DiscountFactors[-1])
    Fixed = DiscountedCFFixed.sum()
print(Fixed)
```

```
97.09086193166755
```

In [27]:
```
FixedCpn2 = np.zeros(len(fwRates))
DiscountedCFFixed2 = np.zeros(len(fwRates))
Fixed2 = np.zeros(len(fwRates))

for i in range(len(fwRates)):
    FixedCpn2[i] = notional * minCpn * AF[i]
    DiscountedCFFixed2[i] = FixedCpn2[i] * DiscountFactors[i]
    DiscountedCFFixed2[-1] = DiscountedCFFixed2[-1] + (notional * DiscountFactors[-1])
    Fixed2 = DiscountedCFFixed2.sum()
print(Fixed2)
```

```
89.56073623166338
```

In [28]:
```
# Pricing with the first strategy
SN1f = - Floor_1 + Floor_2 + Fixed
print("price_SN1f:", SN1f)
```

```
price_SN1f: 96.47115631561752
```

In [29]:
```
# Pricing with the second strategy
SN1c = - Cap_1 + Cap_2 + Fixed2
print("price_SN1c:", SN1c)
```

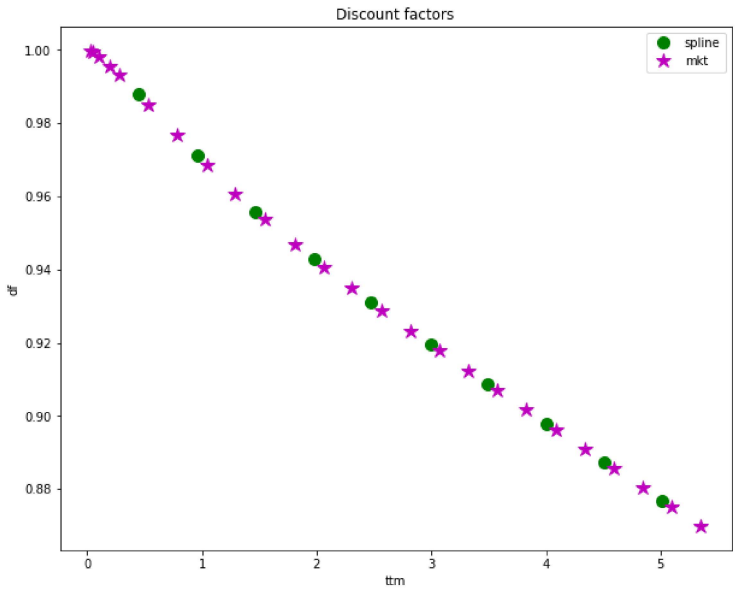
```
price_SN1c: 96.47115631561752
```

In [30]:
```
# Pricing table (1st strategy)
data_frame = pd.DataFrame()
data_frame["StartDates"] = StartDates
data_frame["EndDates" ]= EndDates
data_frame["AF"] = AF
data_frame['DiscountFactors'] = DiscountFactors
data_frame["DiscountedCFlowAmounts"] = DiscountedCFFixed

data_frame
```

Out[30]:

	StartDates	EndDates	AF	DiscountFactors	DiscountedCFlowAmounts
0	January 10th, 2023	July 10th, 2023	0.502778	0.987918	0.993407
1	July 10th, 2023	January 10th, 2024	0.511111	0.971038	0.992616
2	January 10th, 2024	July 10th, 2024	0.505556	0.955855	0.966476
3	July 10th, 2024	January 10th, 2025	0.511111	0.942714	0.963663
4	January 10th, 2025	July 10th, 2025	0.502778	0.930899	0.936070
5	July 10th, 2025	January 12th, 2026	0.516667	0.919431	0.950078
6	January 12th, 2026	July 10th, 2026	0.497222	0.908769	0.903721
7	July 10th, 2026	January 11th, 2027	0.513889	0.897946	0.922889
8	January 11th, 2027	July 12th, 2027	0.505556	0.887354	0.897213
9	July 12th, 2027	January 10th, 2028	0.505556	0.876782	88.564729

In [31]:
```
# Discount Factors plot
plt.figure(figsize=(10,8))
plt.plot(CashFlowTimes,DiscountFactors,"go",label='spline', markersize=10)
plt.plot(irc.loc[1:25,"ttm"],irc.loc[1:25,"df"],"m*",label='mkt', markersize=13)
plt.title("Discount factors")
plt.xlabel("ttm")
plt.ylabel("df")
plt.legend()
plt.show()
```



(d) Structured Note 2

- Maturity: March 15, 2027.
- Notional: Euro 100
- Frequency: quaterly payments

- Coupon = 3m Euribor + 150BP, if 3mEuribor <= 0.5%, 0.9% otherwise.
- Day count convention (basis): act/360
- Day rolling: Modified following

In [32]:
```
# Input data for the structured note 2
maturity = ql.Date(15,3,2027)
notional = 100
spread = 0.015
couponFrequency = ql.Quarterly
dayCounter = ql.Actual360()
currentcpn = 0.02482


effectiveDate = trade_date
terminationDate = ql.Date(15,ql.March,2027)
dayCount = ql.Actual360()
frequency = ql.Period('3M')
convention = ql.ModifiedFollowing
terminationDateConvention = ql.ModifiedFollowing
rule = ql.DateGeneration.Backward
endOfMonth = False
calendar = ql.TARGET()


# Define the schedule of payments
schedule = ql.Schedule(effectiveDate,
                       terminationDate,
                       frequency,
                       calendar,
                       convention,
                       terminationDateConvention,
                       rule,
                       endOfMonth)
print('Schedule:')
for i, d in enumerate(schedule):
    print (i+1, d)
```

```
Schedule:
1 January 30th, 2023
2 March 15th, 2023
3 June 15th, 2023
4 September 15th, 2023
5 December 15th, 2023
6 March 15th, 2024
7 June 17th, 2024
8 September 16th, 2024
9 December 16th, 2024
10 March 17th, 2025
11 June 16th, 2025
12 September 15th, 2025
13 December 15th, 2025
14 March 16th, 2026
15 June 15th, 2026
16 September 15th, 2026
17 December 15th, 2026
18 March 15th, 2027
```

In [33]:
```
scheduleShift = ql.Schedule(effectiveDate-frequency,
                       terminationDate,
                       frequency,
                       calendar,
                       convention,
                       terminationDateConvention,
                       rule,
                       endOfMonth)
print('ScheduleShift:')
for i, d in enumerate(scheduleShift):
    print (i+1, d)

previousCpnDate = scheduleShift[1]
print('PreviousCpnDate:')
print(previousCpnDate)
```

```
ScheduleShift:
1 October 31st, 2022
2 December 15th, 2022
3 March 15th, 2023
4 June 15th, 2023
5 September 15th, 2023
6 December 15th, 2023
7 March 15th, 2024
8 June 17th, 2024
9 September 16th, 2024
10 December 16th, 2024
11 March 17th, 2025
12 June 16th, 2025
13 September 15th, 2025
14 December 15th, 2025
15 March 16th, 2026
16 June 15th, 2026
17 September 15th, 2026
18 December 15th, 2026
19 March 15th, 2027
PreviousCpnDate:
December 15th, 2022
```

In [34]:
```
def Cash_Nothing_Digital_Call (K, df, sigma, FwdRates, T, Tenor, Notional):
    '''
    This function computes the price of a Cash or Nothing Digital Call using the Bachelier formula, i.e. under the
    normal model.
    The inputs for the function are the following:
    K = strike
    df = discount factors
    sigma = volatility (flat)
```

```
    FwdRates = forward rates
    T = maturity
    tenor
    notional
    '''
    CNDC = np.zeros(len(FwdRates))
    for i in range(len(FwdRates)):
        d = (FwdRates[i]- K) / (sigma * np.sqrt(T[i]))
        CNDC[i] = Tenor[i] * df[i] * (norm.cdf(d))
        Tot_CNDC = CNDC.sum() * notional
    return Tot_CNDC
```

In [35]:
```
nDates = len(schedule)-1
StartDates = [schedule[i] for i in range(nDates)]
StartDates[0] = previousCpnDate
EndDates = [schedule[i+1] for i in range(nDates)]

AF = np.array([ql.Actual360().yearFraction(StartDates[i],EndDates[i]) for i in range(nDates)]).reshape(nDates,1)
CashFlowTimes = np.array([ql.Actual360().yearFraction(effectiveDate, EndDates[i]) for i in range(nDates)]).reshape(nDates,1)
DiscountFactors = interpolate.CubicSpline(irc.loc[:,"ttm"], irc.loc[:,"df"], bc_type = "natural")(CashFlowTimes)
fwRates = (DiscountFactors[:-1]/DiscountFactors[1:]-1)/AF[:1]
fwRates = np.insert(fwRates, 0, np.nan).reshape(nDates,1)
fwRates[0] = currentcpn

print(fwRates)
```

```
[[0.02482   ]
 [0.02996245]
 [0.03458426]
 [0.03421858]
 [0.03330523]
 [0.03219371]
 [0.02869092]
 [0.02708348]
 [0.02601314]
 [0.02527055]
 [0.02467675]
 [0.02422854]
 [0.02392789]
 [0.0237485 ]
 [0.02391246]
 [0.02363778]
 [0.02344539]]
```

In [36]:
```
K = 0.005
flat_vol_floor = 0.01063
flat_vol_cap = 0.01063
flat_vol_capdig = 0.010735
```

In [37]:
```
floor_sn2 = Floor_Bachelier(K, DiscountFactors, fwRates, flat_vol_floor, CashFlowTimes, AF, notional)
print(floor_sn2)
```

```
0.3371860811235441
```

In [38]:
```
CNDC = Cash_Nothing_Digital_Call(K, DiscountFactors, flat_vol_capdig, fwRates, CashFlowTimes, AF, notional)
print(CNDC)
```

```
370.3915234726604
```

In [39]:
```
FixedCpn3 = np.zeros(len(fwRates))
DiscountedCFFixed3 = np.zeros(len(fwRates))
Fixed3 = np.zeros(len(fwRates))
Cpn = 0.02

for i in range(len(fwRates)):
    FixedCpn3[i] = notional * Cpn * AF[i]
    DiscountedCFFixed3[i] = FixedCpn3[i] * DiscountFactors[i]
    DiscountedCFFixed3[-1] = DiscountedCFFixed3[-1] + (notional * DiscountFactors[-1])
    Fixed3 = DiscountedCFFixed3.sum()
print(Fixed3)
```

```
97.54013977157379
```

In [40]:
```
# Pricing with the first strategy
SN2 = - floor_sn2 + Fixed3 - 0.011 * CNDC
print("price_SN2:", SN2)
```

```
price_SN2: 93.12864693225097
```

In [41]:
```
CpnRates = fwRates + spread
CpnRates[0]= currentcpn + spread
CFlowAmounts = notional * CpnRates * AF
CFlowAmounts[-1] = CFlowAmounts[-1] + notional
DiscountedCFlowAmounts = CFlowAmounts * DiscountFactors
FRN = sum(DiscountedCFlowAmounts)

print(FRN)
```

```
[106.62227932]
```

In [42]:
```
cap = Cap_Bachelier(K, DiscountFactors, fwRates, flat_vol_cap, CashFlowTimes, AF, notional)
print(cap)
```

```
9.419325626148627
```

In [43]:
```
CNDC = Cash_Nothing_Digital_Call(K, DiscountFactors, flat_vol_capdig, fwRates, CashFlowTimes, AF, notional)
print(CNDC)
```

```
370.3915234726604
```

In [44]:
```
# Pricing with the second strategy
Str2_price = FRN - cap - 0.011 * CNDC
print("Str2_price:", Str2_price)
```

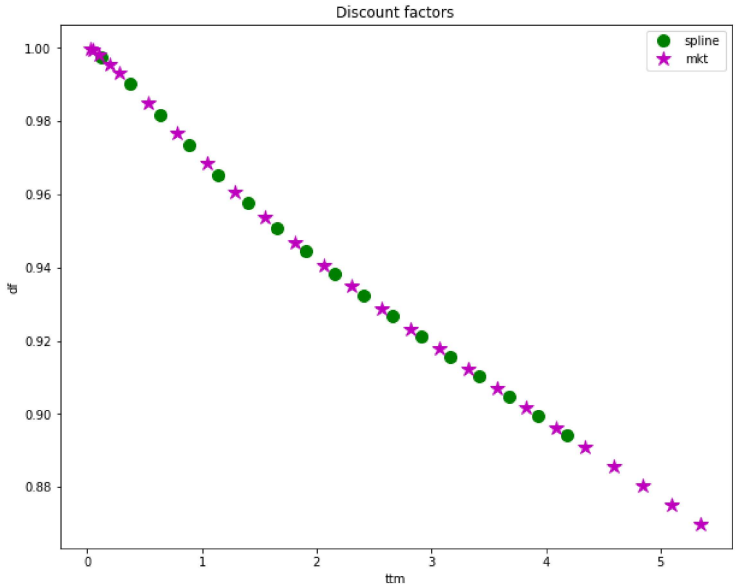
```
Str2_price: [93.12864693]
```

In [45]:
```
# Pricing table
data_frame = pd.DataFrame()
data_frame["StartDates"] = StartDates
data_frame["EndDates" ]= EndDates
data_frame["AF"] = AF
data_frame["CFLowTimes"] = CashFlowTimes
data_frame['DiscountFactors'] = DiscountFactors
data_frame["FwdRates"]= fwRates
data_frame["CpnRates"] = CpnRates
data_frame["CFlowAmounts"]= CFlowAmounts
data_frame["DiscountedCFlowAmounts"] = DiscountedCFlowAmounts

data_frame
```

Out[45]:

	StartDates	EndDates	AF	CFLowTimes	DiscountFactors	FwdRates	CpnRates	CFlowAmounts	DiscountedCFlowAmounts
0	December 15th, 2022	March 15th, 2023	0.250000	0.122222	0.997606	0.024820	0.039820	0.995500	0.993117
1	March 15th, 2023	June 15th, 2023	0.255556	0.377778	0.990189	0.029962	0.044962	1.149040	1.137767
2	June 15th, 2023	September 15th, 2023	0.255556	0.633333	0.981701	0.034584	0.049584	1.267153	1.243966
3	September 15th, 2023	December 15th, 2023	0.252778	0.886111	0.973374	0.034219	0.049219	1.244136	1.211010
4	December 15th, 2023	March 15th, 2024	0.252778	1.138889	0.965336	0.033305	0.048305	1.221049	1.178723
5	March 15th, 2024	June 17th, 2024	0.261111	1.400000	0.957629	0.032194	0.047194	1.232280	1.180067
6	June 17th, 2024	September 16th, 2024	0.252778	1.652778	0.950809	0.028691	0.043691	1.104409	1.050082
7	September 16th, 2024	December 16th, 2024	0.252778	1.905556	0.944415	0.027083	0.042083	1.063777	1.004646
8	December 16th, 2024	March 17th, 2025	0.252778	2.158333	0.938312	0.026013	0.041013	1.036721	0.972768
9	March 17th, 2025	June 16th, 2025	0.252778	2.411111	0.932422	0.025271	0.040271	1.017950	0.949159
10	June 16th, 2025	September 15th, 2025	0.252778	2.663889	0.926705	0.024677	0.039677	1.002940	0.929429
11	September 15th, 2025	December 15th, 2025	0.252778	2.916667	0.921125	0.024229	0.039229	0.991610	0.913398
12	December 15th, 2025	March 16th, 2026	0.252778	3.169444	0.915648	0.023928	0.038928	0.984011	0.901007
13	March 16th, 2026	June 15th, 2026	0.252778	3.422222	0.910244	0.023749	0.038749	0.979476	0.891562
14	June 15th, 2026	September 15th, 2026	0.255556	3.677778	0.904835	0.023912	0.038912	0.994430	0.899794
15	September 15th, 2026	December 15th, 2026	0.252778	3.930556	0.899519	0.023638	0.038638	0.976677	0.878540
16	December 15th, 2026	March 15th, 2027	0.250000	4.180556	0.894277	0.023445	0.038445	100.961135	90.287244

In [46]:
```
# Discount factors plot
plt.figure(figsize=(10,8))
plt.plot(CashFlowTimes,DiscountFactors,"go",label='spline', markersize=10)
plt.plot(irc.loc[1:25,"ttm"],irc.loc[1:25,"df"],"m*",label='mkt', markersize=13)
plt.title("Discount factors")
plt.xlabel("ttm")
plt.ylabel("df")
plt.legend()
plt.show()
```



Excercise 2: MONTECARLO SIMULATION IN THE VASICEK MODEL

Calibrate the Vasicek model to the interest rate curve on a given date

In [47]:
```
# Interpolation
sdate = date(2023,2,7)   # start date
edate = date(2024,2,9)   # end date
calendar = pd.date_range(sdate,edate-timedelta(days=1),freq='D')
```

```
interpolation = CubicSpline(df['MaturityDate'], irc['R'])
Rinterpolated = interpolation(calendar)

interpolationDF = CubicSpline(df['MaturityDate'], irc['df'])
RinterpolatedDF = interpolationDF(calendar)

df_Rinterpol={'R':Rinterpolated,'Time':calendar,'df':RinterpolatedDF}
df_Rinterpol=pd.DataFrame(df_Rinterpol)

df_Rinterpol
```

Out[47]:

	R	Time	df
0	0.000071	2023-02-07	0.999999
1	0.002112	2023-02-08	0.999947
2	0.003937	2023-02-09	0.999891
3	0.005565	2023-02-10	0.999830
4	0.007012	2023-02-11	0.999767
...	...	...	...
362	0.030821	2024-02-04	0.968818
363	0.030826	2024-02-05	0.968730
364	0.030832	2024-02-06	0.968642
365	0.030837	2024-02-07	0.968554
366	0.030842	2024-02-08	0.968466

367 rows × 3 columns

In [48]:

```
trade_date = ql.Date(7, 2, 2023)
df_Rinterpol["ql_dates"] = df_Rinterpol.Time.apply(ql.Date().from_date)
times=[]
for i in range(len(df_Rinterpol["R"])):
    times.append(ql.Actual360().yearFraction(trade_date,df_Rinterpol['ql_dates'][i]))
df_Rinterpol["ttm"] = times
df_Rinterpol = df_Rinterpol[df_Rinterpol["ttm"]>=0]
df_Rinterpol1 = df_Rinterpol.drop(df_Rinterpol.index[:2])
df_Rinterpol1 = df_Rinterpol1.reset_index(drop=True)
df_Rinterpol1
```

Out[48]:

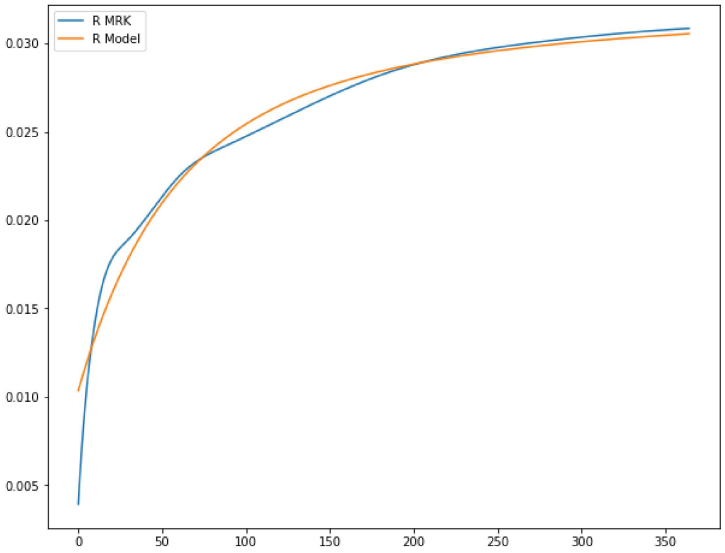
	R	Time	df	ql_dates	ttm
0	0.003937	2023-02-09	0.999891	February 9th, 2023	0.005556
1	0.005565	2023-02-10	0.999830	February 10th, 2023	0.008333
2	0.007012	2023-02-11	0.999767	February 11th, 2023	0.011111
3	0.008298	2023-02-12	0.999701	February 12th, 2023	0.013889
4	0.009441	2023-02-13	0.999633	February 13th, 2023	0.016667
...	...	...	...	...	...
360	0.030821	2024-02-04	0.968818	February 4th, 2024	1.005556
361	0.030826	2024-02-05	0.968730	February 5th, 2024	1.008333
362	0.030832	2024-02-06	0.968642	February 6th, 2024	1.011111
363	0.030837	2024-02-07	0.968554	February 7th, 2024	1.013889
364	0.030842	2024-02-08	0.968466	February 8th, 2024	1.016667

365 rows × 5 columns

In [49]:

```
# the first commented function def create_grid(), is the one that we run to try to find the global minimum,
# but it took 23 hours to run, so to run again the code we use another grid with less parameter that find
# that minimum quicker
'''def create_grid():
    r0_range = np.arange(0.01, 0.031, 0.001)
    k_range = np.arange(0.1, 10.1, 0.3)
    theta_range = np.arange(0.01, 0.1, 0.003)
    sigma_range = np.arange(0.01, 0.1, 0.003)
    return list(itertools.product(r0_range, k_range, theta_range, sigma_range))'''

def create_grid():
    r0_range = np.arange(0.01,0.1, 0.01)
    k_range = np.arange(1,6, 1)
    theta_range = np.arange(0.01,0.2, 0.02)
    sigma_range = np.arange(0.001,0.02, 0.002)
    return list(itertools.product(r0_range, k_range, theta_range, sigma_range))

def ComputeR(ttm, k, theta, sigma,r0):
    B = (1 / k) * (1 - np.exp(-k * ttm))
    A = np.exp((theta - ((sigma**2) / (2 * (k**2)))) * (B - ttm) - (sigma**2)* (B**2) / (4 * k) )
    P = A * np.exp(-B * r0)
    return -np.log(P)/ttm

def mse(par, ttm, MRKdata):
    r0, k, theta, sigma = par
    P = ComputeR(ttm, k, theta, sigma,r0)

    return np.sum((P - MRKdata)**2)

def calibration(param, ttm, MRKdata):
    sol = minimize(mse, param, args=(ttm, MRKdata), options={'maxiter':1000000},bounds=((0.001,None),(0.1,None),(0.001,None),(0.0001,None)))
    r0,k, theta, sigma = sol.x
    P = ComputeR(ttm, k, theta, sigma,r0)
```

```
    #print('r0 = ', r0)
    #print('k = ',k)
    #print('theta = ',theta)
    #print('sigma = ', sigma)
    return P

def find_best_params(ttm, MRKdata):
    grid = create_grid()
    min_mse = float('inf')
    best_params = None
    for param in grid:
        P = calibration(param, ttm, MRKdata)
        mse_value = mse(param, ttm, MRKdata)
        if mse_value < min_mse:
            min_mse = mse_value
            best_params = param
    return min_mse, best_params

ttm = df_Rinterpol1['ttm'].values
MRKdata = df_Rinterpol1['R'].values
min_mse, best_params = find_best_params(ttm, MRKdata)
#print("Minimum MSE: ", min_mse)

cal = calibration(best_params, df_Rinterpol1['ttm'].values, df_Rinterpol1['R'].values)
P = np.array(cal)

plt.figure(figsize=(10,8))
plt.plot(df_Rinterpol1['R'], label='R MRK')
plt.plot(P, label='R Model')
plt.legend()
plt.show()

T=len(df_Rinterpol1)
```



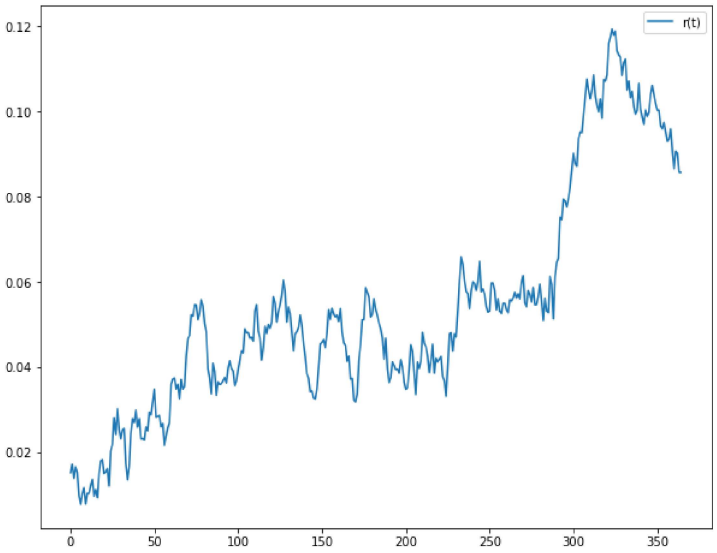
In [50]:

```
# Optimal values
r0 =  0.015263159819135117
k =  3.600008006536917
theta =  0.0382381139289971
sigma =  0.019036071875234508
```

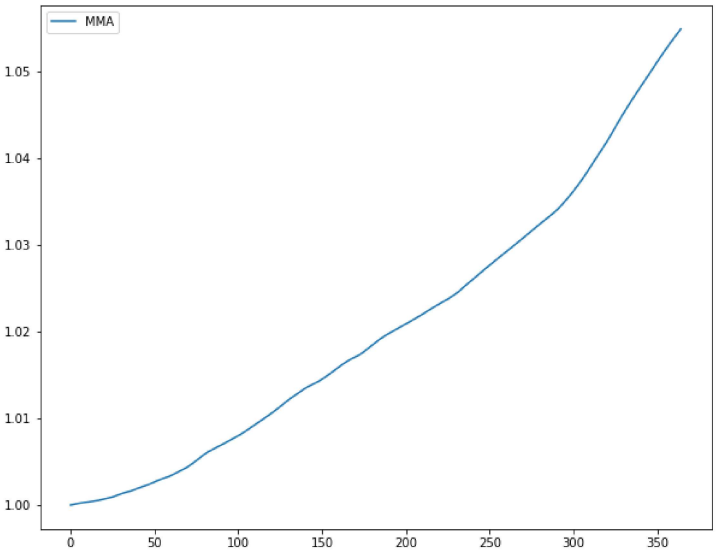
Simulate the risk-neutral dynamics of interest rates on 1 year time horizon in terms of: MMA

In [51]:

```
# Simulating r(t)-path with vasicek
def rt(k,theta,sigma,T,r0):
    dt = 1/T
    e = np.random.normal(0,1,size=(T))
    r = np.zeros((T))
    r[0] = r0
    for j in range(1, T):
        r[j] = r[j-1]*np.exp(-k*dt)+theta*(1-np.exp(-k*dt))+np.sqrt(((sigma**2)/2*k)*(1-np.exp(-2*k*dt)))*e[j]
    return r

np.random.seed(4)
r=rt(k,theta,sigma,T,r0)

plt.figure(figsize=(10,8))
plt.plot(r, label='r(t)')
plt.legend()
plt.show()
```



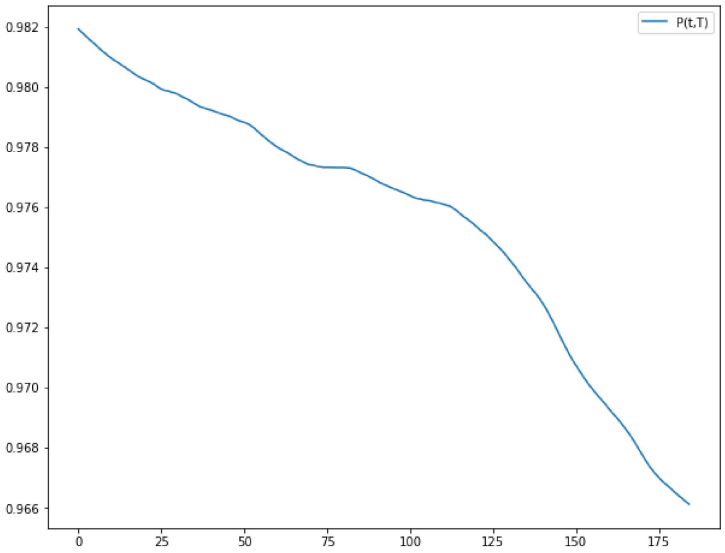
In [52]:
```
# simulating Money Market Account
def MMA(T,r):
    dt = 1/T
    MMA = np.zeros((T))
    MMA[0] = 1
    for j in range(1, T):
        #use the trapezoidal rule to compute the argument of the eponential
        MMA[j] = MMA[j-1] * np.exp((r[j-1]+r[j])*dt/2)
    return MMA

mma=MMA(T,r)

plt.figure(figsize=(10,8))
plt.plot(mma, label='MMA')
plt.legend()
plt.show()
```



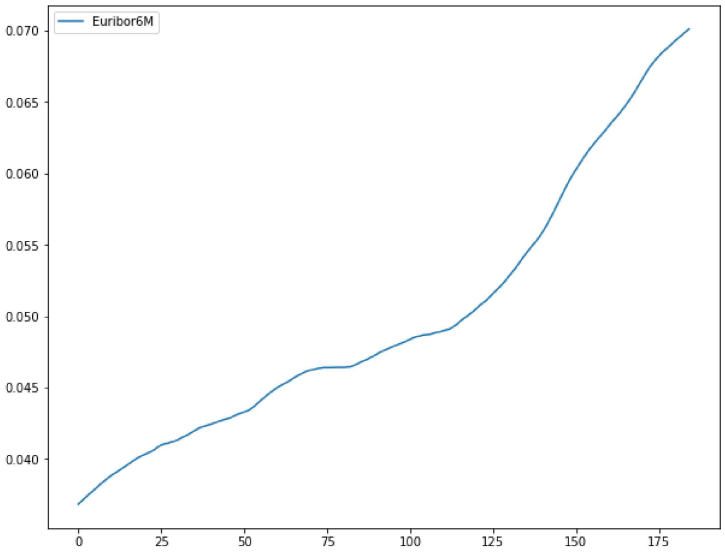
Simulate the risk-neutral dynamics of interest rates on 1 year time horizon in terms of: 6m Euribor rates

In [54]:
```
# Compute the path of P with formula
def PtT(k, theta, sigma,r):
    dt = 1/T
    B = [(1 / k) * (1 - np.exp(-k * dt)) for i in range(0, len(r))]
    A = [np.exp((theta - ((sigma**2) / (2 * (k**2)))) * (B[i] - dt) - ((sigma**2) * (B[i]**2)) / (4 * k) ) for i in range(0, len(r))]

    p = [A[i] * np.exp(-B[i] * r[i]) for i in range(0, len(r))]
    return p

p = PtT(k, theta, sigma,r)
```

In [55]:
```
def computePiiplus6month(p):
    piiplus6m=[np.prod(p[i:i+180]) for i in range(0, T-180)]
    return piiplus6m
piiplus6m=computePiiplus6month(p)

plt.figure(figsize=(10,8))
plt.plot(piiplus6m, label='P(t,T)')
plt.legend()
plt.show()
```



In [56]:
```
# simulating Euribor 6month
def Euribor(piiplus6m):
    L = [(1/0.5) * ((1/piiplus6m[i])-1) for i in range(len(piiplus6m))]
    return L

L = Euribor(piiplus6m)

plt.figure(figsize=(10,8))
plt.plot(L, label='Euribor6M')
plt.legend()
plt.show()
```

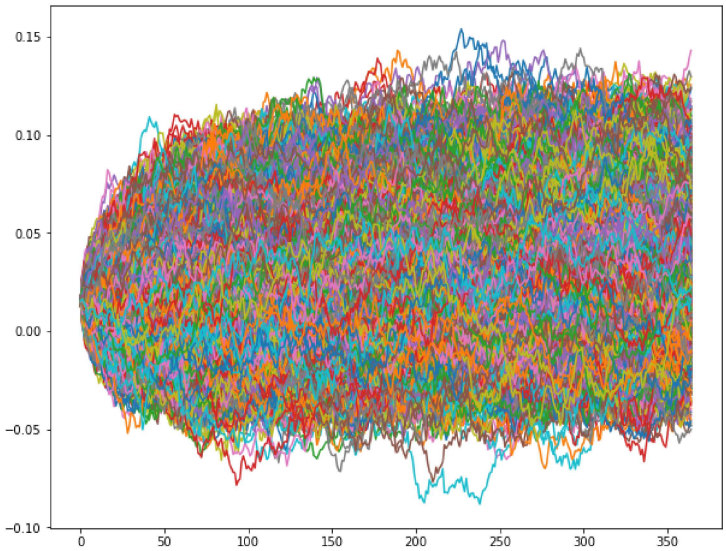


Compute the price of 1YR-zcb by MC

In [57]:
```
def montecarlo(nsim):
    simulations = []
    for i in range(nsim):
        simulations.append(rt(k,theta,sigma,T,r0))
    return simulations
```

In [58]:
```
nsim = 10000
np.random.seed(3)
sim = montecarlo(nsim)
```

In [60]:
```
def plot(nsim,sim):
    for i in range(nsim):
        plt.plot(sim[i])
    return
plt.figure(figsize=(10,8))
plot(nsim,sim)
```



In [61]:
```
def P0T(nsim,T):
    dt = 1/T
    P_0 = np.zeros((nsim,T))
    sum_integrals = np.zeros((nsim))
    for j in range(0,nsim):
        for i in range(1, T):
            P_0[j][i] = (-(sim[j][i-1]+sim[j][i])*dt/2)
        sum_integrals[j] = np.sum(P_0[j])
        exp= np.exp(sum_integrals)
    return exp
```

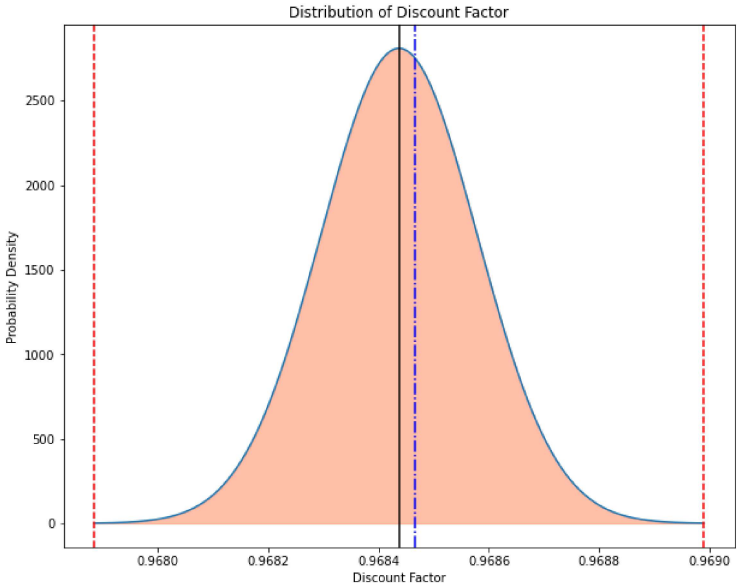
In [62]:
```
tt=P0T(nsim,T)
print('The MC estimated ZCB price is = ', np.mean(tt))
print('The true value is = ',df_Rinterpol1['df'][364])
```

```
The MC estimated ZCB price is =  0.9684369848237204
The true value is =  0.9684658369076986
```

In [63]:
```
from scipy.stats import t
mean = np.mean(tt)

# standard deviation
std = np.std(tt, ddof=1)
# number of observations
n = len(tt)
# confidence level: (1 - alpha) 99,99%
alpha = 0.0001
confidence = 1 - alpha
# confidence interval
interval = t.interval(confidence, n-1, loc=mean, scale=std/np.sqrt(n))

print("Confidence interval: ", interval)
```

```
Confidence interval:  (0.9678841605807748, 0.9689898090666661)
```

In [64]:
```
lower, upper = interval[0], interval[1]
x = np.linspace(lower, upper, n)
y = norm.pdf(x, loc=mean, scale=std / np.sqrt(n))

plt.figure(figsize=(10,8))
plt.fill_between(x, y, color='coral', alpha=0.5)
plt.plot(x, y)
plt.axvline(lower, color='red', linestyle='--')
plt.axvline(upper, color='red', linestyle='--')
plt.axvline(mean, color='black', linestyle='-')
plt.axvline(df_Rinterpol1['df'][364], color='blue', linestyle='dashdot')
plt.title('Distribution of Discount Factor')
plt.xlabel('Discount Factor')
plt.ylabel('Probability Density')
plt.show()
```



Exercise 3: CLASSICAL VS MULTICURVE APPROACH IN PRICING FIXED-INCOME INSTRUMENTS

In [65]:
```
data_euribor = pd.read_excel("euribor.xlsx")
EURIBOR = data_euribor.drop("Instrument",axis=1)
EURIBOR = EURIBOR.rename(columns={"PRIMACT_1":"Yield","GV4_TEXT":"EUR","SEC_ACT_1":"Discount","MATUR_DATE":"MaturityDate"})
EURIBOR["MaturityDate"] = pd.to_datetime(EURIBOR["MaturityDate"])
EURIBOR = EURIBOR.dropna()
EURIBOR["ql_dates"] = EURIBOR.MaturityDate.apply(ql.Date().from_date)
EURIBOR = EURIBOR.reset_index(drop=True)
EURIBOR["ql_dates"] = EURIBOR.MaturityDate.apply(ql.Date().from_date)
EURIBOR = EURIBOR.reset_index()
times = []
trade_date = ql.Date(27, 1, 2023)
for i in range(len(EURIBOR["MaturityDate"])):
    times.append(ql.Actual360().yearFraction(trade_date, EURIBOR["ql_dates"][i]))
times
EURIBOR["YearFrac"] = times
EURIBOR.head()
```

Out[65]:

	index	CF_NAME	CF_LAST	Discount	MaturityDate	ql_dates	YearFrac
0	0	ON ZERO YIELD	2.043247	0.999945	2023-02-08	February 8th, 2023	0.033333
1	1	TN ZERO YIELD	2.038074	0.999889	2023-02-09	February 9th, 2023	0.036111
2	2	1W ZERO YIELD	2.444258	0.999405	2023-02-16	February 16th, 2023	0.055556
3	3	1M ZERO YIELD	2.582936	0.997906	2023-03-09	March 9th, 2023	0.113889
4	4	2M ZERO YIELD	2.696967	0.995417	2023-04-11	April 11th, 2023	0.205556

In [66]:
```
trade_date = ql.Date(30, 1, 2023)
calender_EUR = ql.TARGET()
times=[]
for i in range(len(EURIBOR["MaturityDate"])):
    times.append(ql.Actual360().yearFraction(trade_date,EURIBOR['ql_dates'][i]))
EURIBOR["Time in year"] = times
EURIBOR = EURIBOR[EURIBOR["Time in year"]>=0]
EURIBOR.head()
```

Out[66]:

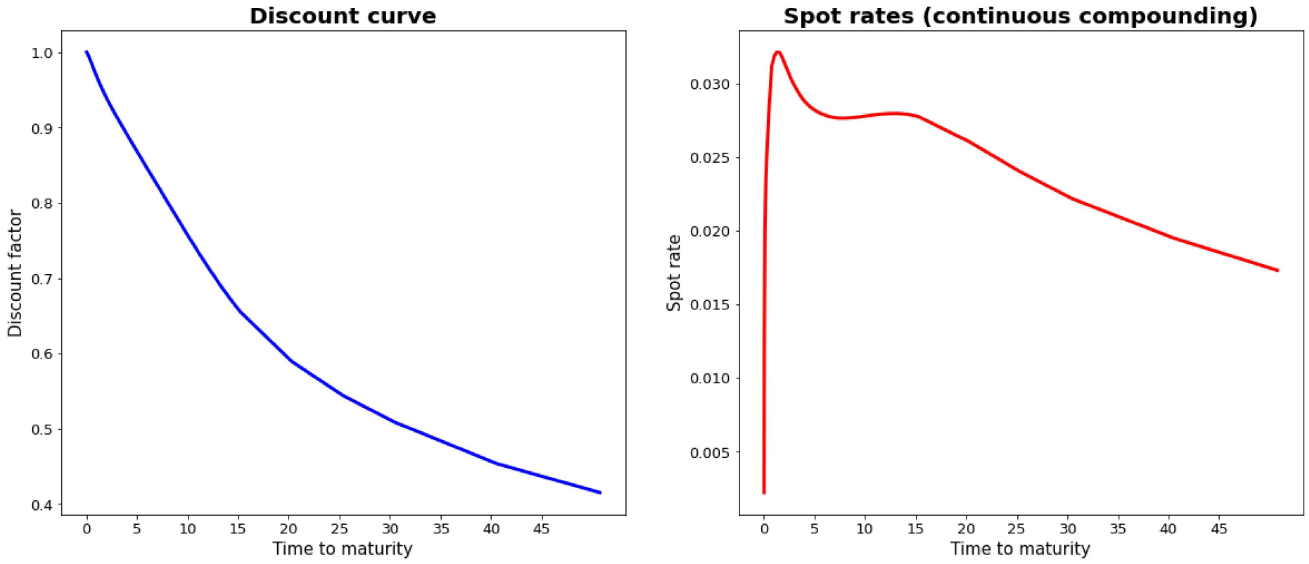
	index	CF_NAME	CF_LAST	Discount	MaturityDate	ql_dates	YearFrac	Time in year
0	0	ON ZERO YIELD	2.043247	0.999945	2023-02-08	February 8th, 2023	0.033333	0.025000
1	1	TN ZERO YIELD	2.038074	0.999889	2023-02-09	February 9th, 2023	0.036111	0.027778
2	2	1W ZERO YIELD	2.444258	0.999405	2023-02-16	February 16th, 2023	0.055556	0.047222
3	3	1M ZERO YIELD	2.582936	0.997906	2023-03-09	March 9th, 2023	0.113889	0.105556
4	4	2M ZERO YIELD	2.696967	0.995417	2023-04-11	April 11th, 2023	0.205556	0.197222

In [67]:
```
irc_m = EURIBOR[["Time in year","Discount"]]
irc_m = irc_m.rename(columns= {"Time in year": "ttm","Discount": "df" })
irc_m["R"]= -np.log(irc_m["df"])/irc_m["ttm"]
irc_m.head()
```

Out[67]:

	ttm	df	R
0	0.025000	0.999945	0.002217
1	0.027778	0.999889	0.003980
2	0.047222	0.999405	0.012609
3	0.105556	0.997906	0.019857
4	0.197222	0.995417	0.023290

In [68]:
```
fig, axes = plt.subplots(1,2, figsize=(20,8))

sns.lineplot(data=irc_m, x="ttm", y="df", ax=axes[0], color='blue', lw=3)
axes[0].set_title('Discount curve', fontsize=20, fontweight='bold')
axes[0].set_xlabel('Time to maturity', fontsize=15)
axes[0].set_ylabel('Discount factor', fontsize=15)
axes[0].set_xticks(np.arange(0, 50, 5))
axes[0].tick_params(axis='both', which='major', labelsize=13)

sns.lineplot(data=irc_m, x="ttm", y="R", ax=axes[1], color='red', lw=3)
axes[1].set_title('Spot rates (continuous compounding)', fontsize=20, fontweight='bold')
axes[1].set_xlabel('Time to maturity', fontsize=15)
axes[1].set_ylabel('Spot rate', fontsize=15)
axes[1].set_xticks(np.arange(0, 50, 5))
axes[1].tick_params(axis='both', which='major', labelsize=13)
```



Provide the valuation of the Fixed-rate bond and the Floating rate note defined in the section "Pricing Fixedincome Portfolios", under the multiple-curve approach

(a) Fixed-rate bond: the price doesn't change with respect to single curve approach

(b) Floating rate note

- Maturity: 18-Jun-2024
- Notional: Euro 100
- Coupon: 3mEuribor + 200 BP
- Frequency: quaterly payments
- Day rolling: Modified following
- Day count convention (basis): act/360

In [69]:
```
# Input data for the floating rate note
notional = 100
spread = 0.02
currentCpnRate = 0.02482+spread # https://www.euribor-rates.eu/en/
trade_date = ql.Date(30, 1, 2023)
effectiveDate = trade_date
terminationDate = ql.Date(18,6,2024)

dayCount = ql.Actual360()

frequency = ql.Period('3M')
calendar = ql.TARGET()
convention = ql.ModifiedFollowing
terminationDateConvention = ql.ModifiedFollowing
rule = ql.DateGeneration.Backward

endOfMonth = False

# Compute the basic schedule
schedule = ql.Schedule(effectiveDate,
terminationDate,
frequency,
calendar,
convention,
terminationDateConvention,
rule,
endOfMonth)

print('Schedule:')
for i, d in enumerate(schedule):
    print (i+1, d)
```

```
Schedule:
1 January 30th, 2023
2 March 20th, 2023
3 June 19th, 2023
4 September 18th, 2023
5 December 18th, 2023
6 March 18th, 2024
7 June 18th, 2024
```

In [70]:
```
# Compute the PreviousCpnDate
scheduleShift = ql.Schedule(effectiveDate-frequency,
```

```
terminationDate,
frequency,
calendar,
convention,
terminationDateConvention,
rule,
endOfMonth)

print('ScheduleShift:')

for i, d in enumerate(scheduleShift) :
    print (i+1, d)

previousCpnDate = scheduleShift[1]
print('PreviousCpnDate: ')
print(previousCpnDate)
```

```
ScheduleShift:
1 October 31st, 2022
2 December 19th, 2022
3 March 20th, 2023
4 June 19th, 2023
5 September 18th, 2023
6 December 18th, 2023
7 March 18th, 2024
8 June 18th, 2024
PreviousCpnDate:
December 19th, 2022
```

In [71]:

```
nDates = len(schedule)-1
StartDates = [schedule[i] for i in range(nDates)]
StartDates[0] = previousCpnDate
EndDates = [schedule[i+1] for i in range(nDates)]

# alfa(Tn,Tn+1)
AF = np.array([dayCount.yearFraction(StartDates[i],EndDates[i]) for i in range(nDates)]).reshape(nDates,1)
CFLowTimes= np.array([ql.Actual360().yearFraction(effectiveDate,EndDates[i]) for i in range(nDates)]).reshape(nDates,1)
# Discount factor
DiscountFactors = interpolate.CubicSpline(irc.loc[:,"ttm"],irc.loc[:,"df"],bc_type='natural')(CFLowTimes)

# Forward rates using the irc_m curve
DiscountFactorsF = interpolate.CubicSpline(irc_m.loc[:,"ttm"],irc_m.loc[:,"df"],bc_type='natural')(CFLowTimes)

FwdRates = (DiscountFactorsF[ :5]/DiscountFactorsF[1:]-1)/AF[:1]
FwdRates = np.insert(FwdRates, 0, np.nan).reshape(nDates,1)
CpnRates = FwdRates+spread
CpnRates[0]= currentCpnRate
CFlowAmounts = notional*CpnRates*AF
CFlowAmounts[-1] = CFlowAmounts[-1] + notional
DiscountedCFlowAmounts = CFlowAmounts*DiscountFactors
price_FRN = sum(DiscountedCFlowAmounts )

print('price_FRN:', price_FRN)
```

```
price_FRN: [103.48440588]
```

In [72]:

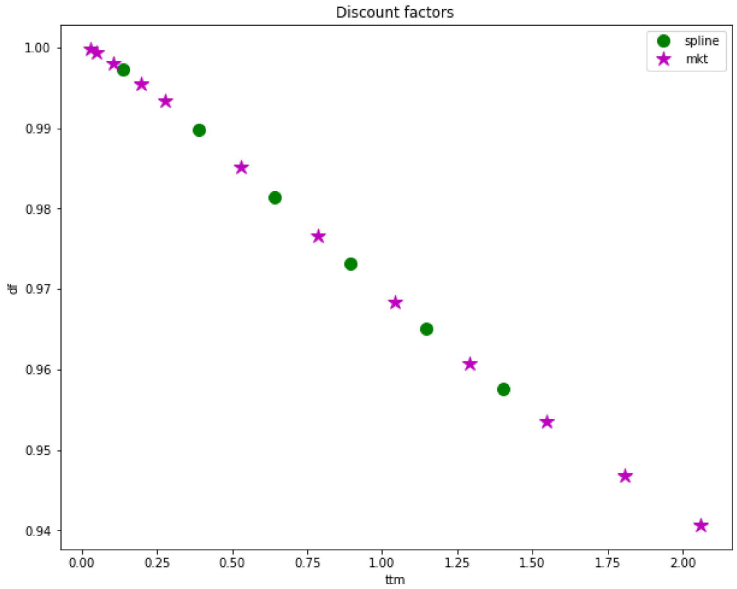
```
# Pricing table
data_frame= pd.DataFrame()
data_frame["StartDates"]= StartDates
data_frame["EndDates"]= EndDates
data_frame["AF"] = AF
data_frame["CFLowTimes"] = CFLowTimes
data_frame['DiscountFactors'] = DiscountFactors
data_frame["FwdRates"]= FwdRates
data_frame["CpnRates"] = CpnRates
data_frame["CFlowAmounts"]= CFlowAmounts
data_frame["DiscountedCF lowAmounts"]= DiscountedCFlowAmounts

data_frame
```

Out[72]:

	StartDates	EndDates	AF	CFLowTimes	DiscountFactors	FwdRates	CpnRates	CFlowAmounts	DiscountedCF lowAmounts
0	December 19th, 2022	March 20th, 2023	0.252778	0.136111	0.997243	NaN	0.044820	1.132950	1.129826
1	March 20th, 2023	June 19th, 2023	0.252778	0.388889	0.989831	0.029586	0.049586	1.253433	1.240687
2	June 19th, 2023	September 18th, 2023	0.252778	0.641667	0.981424	0.035076	0.055076	1.392211	1.366349
3	September 18th, 2023	December 18th, 2023	0.252778	0.894444	0.973103	0.036141	0.056141	1.419126	1.380956
4	December 18th, 2023	March 18th, 2024	0.252778	1.147222	0.965079	0.033458	0.053458	1.351309	1.304120
5	March 18th, 2024	June 18th, 2024	0.255556	1.402778	0.957551	0.033426	0.053426	101.365319	97.062467

In [73]:

```
# Discount factors plot
plt.figure(figsize=(10,8))
plt.plot(CFLowTimes ,DiscountFactors, "go", label='spline', markersize=10)
plt.plot(irc.loc[1:12, "ttm"],irc.loc[1:12,"df"],"m*",label='mkt', markersize=13)
plt.title("Discount factors")
plt.xlabel ("ttm")
plt.ylabel("df")
plt.legend()
plt.show()
```


Discount factors
spline
mkt
df
ttm
1.00
0.99
0.98
0.97
0.96
0.95
0.94
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

In [ ]:

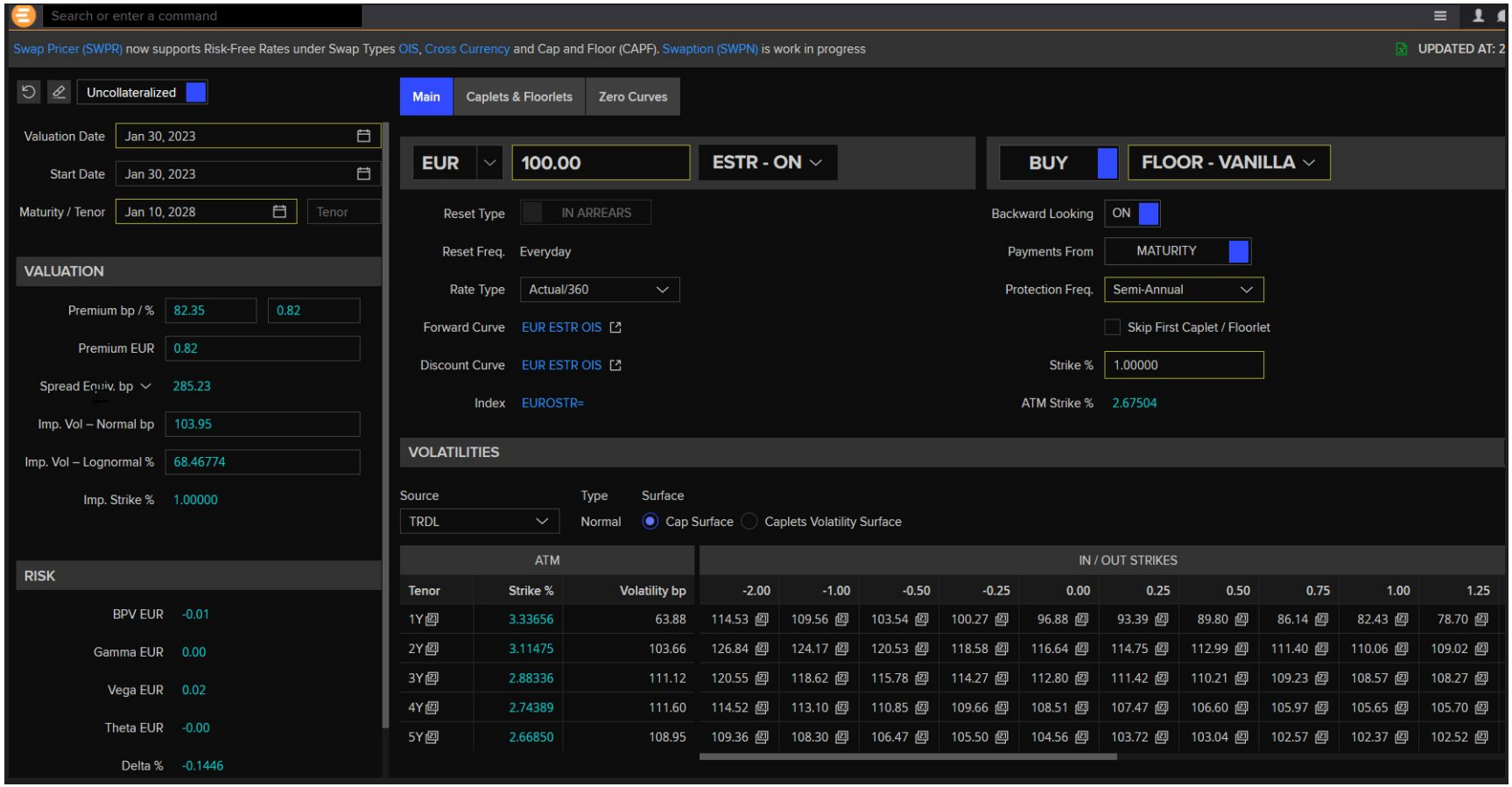


Figure 11: Flat Volatility Floor(1%)

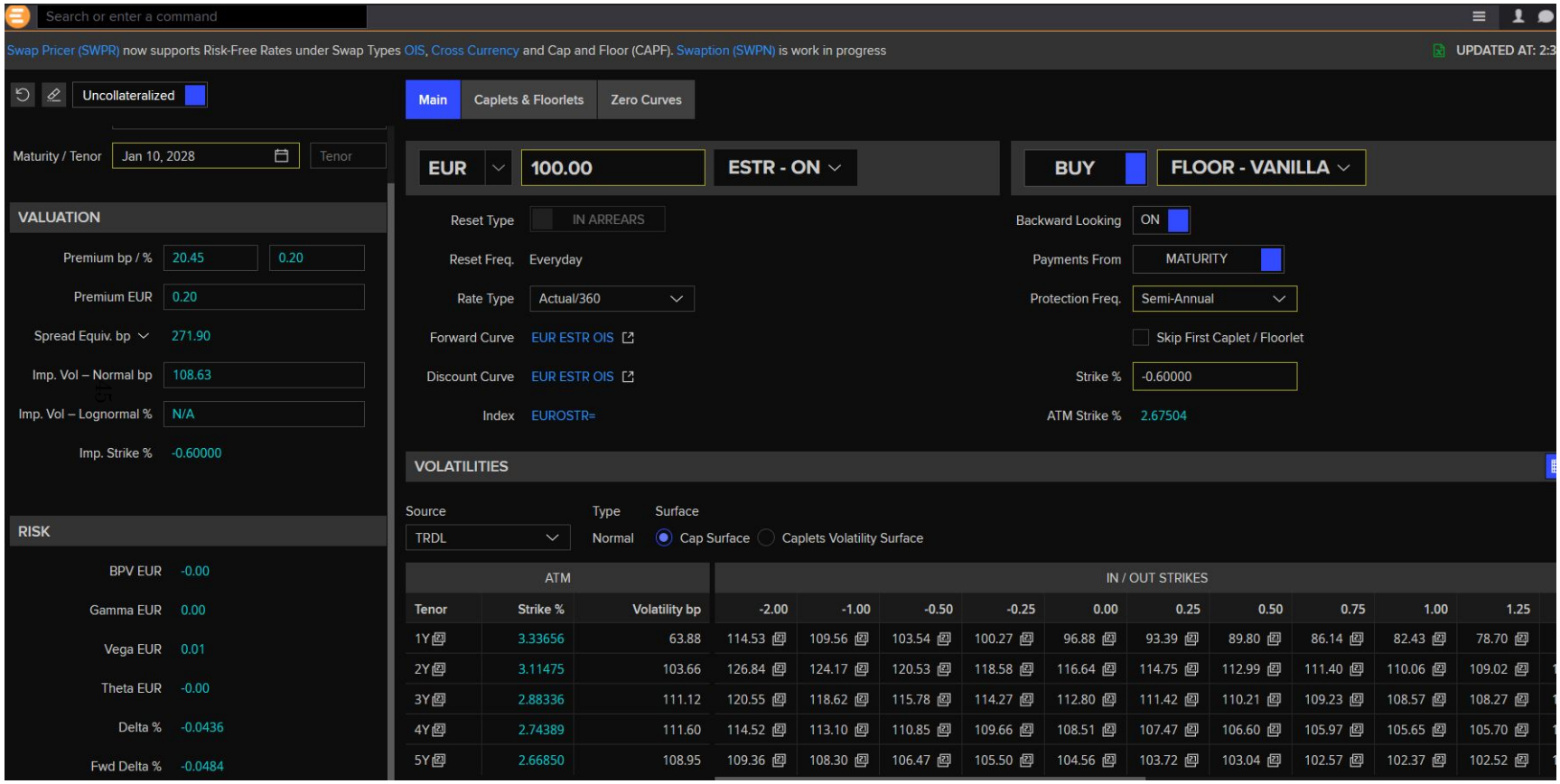


Figure 12: Flat Volatility Floor(-0.6%)