Interest Rate Swap analysis – individual project

Fixed Income Securities, 2023/2024

Luís Filipe Ribeiro, nº 20231536

Contents

Task A – Interpolating yield curve

For this task, it is required to interpolate a yield curve given Bloomberg's reference EUR yield curve on the valuation date of 14/04/2019. Firstly, it was loaded onto a Panda's dataframe the following rates (using tabula to read the pdf automatically) and the "theta" (maturity in years from valuation date to reference date) using QuantLib's "*yearFraction*" function on a Money Market (Actual/360) basis:

```
import QuantLib as ql
import numpy as np
import pandas as pd

def count_days(d1: ql.Date, d2: ql.Date, basis = ql.Actual360()):
    return basis.dayCount(d1, d2)

def count_years(d1: ql.Date, d2: ql.Date, basis = ql.Actual360()):
    return basis.yearFraction(d1, d2)

def get_coupon_dates(issue_date:ql.Date, maturity_date: ql.Date, frequency: int):
    schedule = ql.Schedule(
        issue_date,
        maturity_date,
        ql.Period(frequency),
        ql.TARGET(),
        ql.Following,
        ql.Following,
        ql.DateGeneration.Backward,
        False
    )
    return [dt for dt in schedule]
```

```
from tabula.io import read_pdf

yield_curve_df = read_pdf("project.pdf", pages='5')[0]

# rename columns to actually be useful later on...
yield_curve_df.rename(columns = {"Maturity Date":"maturity", "Market Rate (%)": "rate"}, inplace=True)

#Store valuation date as variable
valuation_date = ql.Date(14,4,2019)

# apply some operations for changhing index and rate (divide by 100 to be used as-is)
yield_curve_df['maturity'] = yield_curve_df['maturity'] \
    .apply(lambda d: pd.to_datetime(d, infer_datetime_format=True)) \
    .apply(lambda d: ql.Date(d.day, d.month, d.year))

yield_curve_df['rate'] = yield_curve_df['rate'].apply(lambda r: r/100)


yield_curve_df['theta'] = yield_curve_df['maturity'] \
    .apply(lambda maturity: count_years(valuation_date, maturity))


yield_curve_df
```

Python

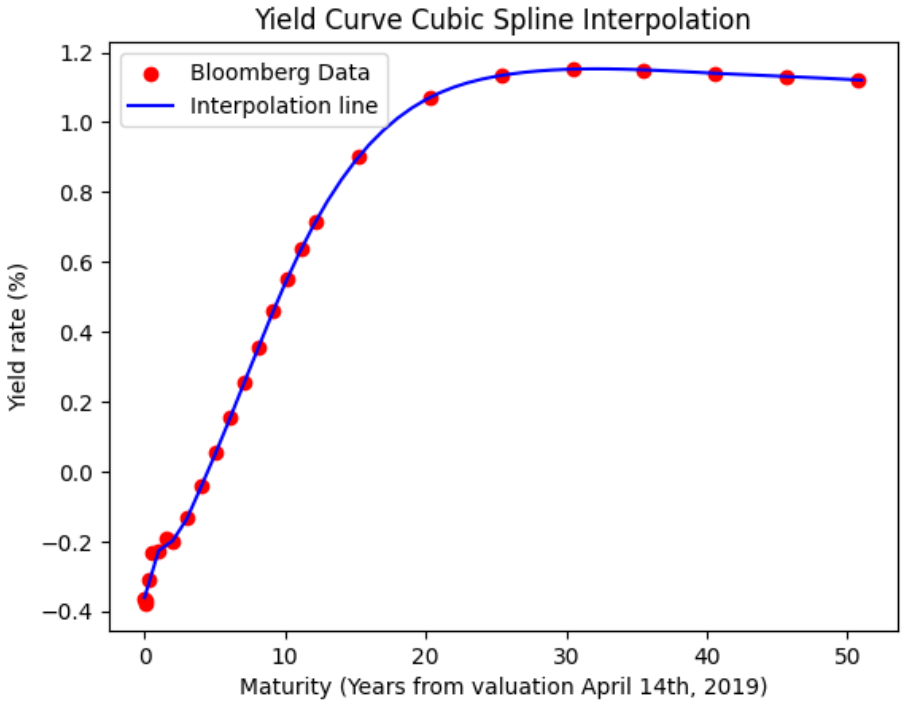
The above snippets of code will render us the table with the respective rates and maturity fractions, which will be the “y” and “x” axis respectively of our yield curve to interpolate:

	maturity	rate	theta
0	April 15th, 2019	-0.003640	0.002778
1	April 23rd, 2019	-0.003780	0.025000
2	May 16th, 2019	-0.003670	0.088889
3	July 16th, 2019	-0.003100	0.258333
4	October 16th, 2019	-0.002320	0.513889
5	April 16th, 2020	-0.002270	1.022222
6	October 16th, 2020	-0.001910	1.530556
7	April 16th, 2021	-0.001992	2.036111
8	April 19th, 2022	-0.001305	3.058333
9	April 17th, 2023	-0.000398	4.066667
10	April 16th, 2024	0.000553	5.080556
11	April 16th, 2025	0.001543	6.094444
12	April 16th, 2026	0.002565	7.108333
13	April 16th, 2027	0.003573	8.122222
14	April 18th, 2028	0.004582	9.144444
15	April 16th, 2029	0.005524	10.152778
16	April 16th, 2030	0.006385	11.166667
17	April 16th, 2031	0.007165	12.180556
18	April 17th, 2034	0.009010	15.227778
19	April 18th, 2039	0.010710	20.302778
20	April 19th, 2044	0.011340	25.380556
21	April 20th, 2049	0.011520	30.455556
22	April 16th, 2054	0.011500	35.516667
23	April 16th, 2059	0.011402	40.588889
24	April 16th, 2064	0.011310	45.663889
25	April 16th, 2069	0.011210	50.736111

Finally, we can interpolate it using a Cubic Spline which will render the following curve:

```python
from scipy.interpolate import CubicSpline
import matplotlib.pyplot as plt

#Cubic spline the yield curve
yield_curve = CubicSpline(yield_curve_df.theta, yield_curve_df.rate)
```

Python



Task B – Compute accrued interest

On this task, it is necessary to compute the accrued interest for both legs. Before proceeding, it is necessary to note that both legs have different coupon frequencies, rates, and day count, but an equal face value of 10 million EUR. Also, an important assumption is that the settlement date will be "T+0", the same as the trading date of 15/04/2019.

Fixed Leg

The fixed leg accrued interest is calculated on a 30U/360 basis (Bond basis) with an annual coupon of 0.05982%. To determine the accrual period, we need to know the cash flows of this leg.

Using a predefined function in our code that itself uses QuantLib's "*Schedule*" class, we can easily generate the coupon payment dates:

```
fixed_coupon_dates = [
    date for date in get_coupon_dates(issue_date, maturity_date, 1)
    if date.year() >= settlement_date.year()
]

pd.Series(fixed_coupon_dates)
```

[28] ✓ 0.0s

```
0     January 21st, 2019
1     January 20th, 2020
2     January 19th, 2021
3     January 19th, 2022
4     January 19th, 2023
5     January 19th, 2024
6     January 20th, 2025
7     January 19th, 2026
8     January 19th, 2027
9     January 19th, 2028
10    January 19th, 2029
11    January 21st, 2030
12    January 20th, 2031
13    January 19th, 2032
dtype: object
```

Since the coupon will be paid on business days, we can see that relative to our trade date (15/04/2019), the previous coupon was paid on the 21st of January 2019 and the next coupon will be paid 20th of January 2020. **Using QuantLib's "*dayCount*" function on a Bond basis we compute it to an accrual period of 84 days.**

Alternatively, we could use the 30/360 formula which will render the same result:

$$\text{Bond Basis} = \frac{360(Y_2 - Y_1) + 30(M_2 - M_1) + (D_2 - D_1)}{360}$$

In our case:

$$\text{Bond Basis} = \frac{360(2019 - 2019) + 30(4 - 1) + (15 - 21)}{360} = \frac{84}{360}$$

$$AI = \text{face value} \times \frac{\text{annual coupon rate } (\%)}{\text{coupon frequency}} \times \frac{u}{w}$$

u = N. of days from (and including) the last coupon (or from the date when interest begins to accrue)
w = N. days between the last and the next coupon

```python
fixed_coupon_dates = [
    date for date in get_coupon_dates(issue_date, maturity_date, 1)
    if date.year() >= settlement_date.year()
]

print("Coupon dates:\n", pd.Series(fixed_coupon_dates))

#30U/360 basis
fixed_leg_basis = ql.Thirty360(ql.Thirty360.BondBasis)

#Frequency of coupons
fixed_frequency = 1

#Accrued days since last coupon
fixed_ai_u = count_days(fixed_coupon_dates[0], settlement_date, fixed_leg_basis)
fixed_ai_w = 360
print(f"Accrued days since last coupon: {fixed_ai_u}")

#Compute accrued interest (AI)
fixed_ai = fixed_rate/fixed_frequency * fixed_ai_u/fixed_ai_w
print(f"Fixed Leg Accrued Interest: {round(fixed_ai*100, 4)}% of face value or {fixed_ai * face_value} EUR")
```

Python

```
Accrued days since last coupon: 83
Fixed Leg Accrued Interest: 0.0138% of face value or 1379.1833333333336 EUR
```

Finally using the accrued interest formula, we obtain Fixed Leg Accrued Interest of 0.0138% of face value or 1379.18 EUR.

Float Leg

Applying the same methodology but for the floating leg, **which has a semiannual coupon (meaning the coupon will be halved) with a null reset margin and spread, given a last Euribor 6-month index of -0.236% with an ACT/360 day-count basis we obtain a Float Leg Accrued Interest of 0.05441% of face value or 5441.11 EUR.**

```python
float_coupon_dates = [
    date for date in get_coupon_dates(issue_date, maturity_date, 2)
    if date.year() >= settlement_date.year()
]

print("Coupon dates:\n", pd.Series(float_coupon_dates))

#30U/360 basis
float_leg_basis = ql.Actual360()

#Frequency of coupons
float_frequency = 2

#Accrued days since last coupon
float_ai_u = count_days(float_coupon_dates[0], settlement_date, float_leg_basis)
float_ai_w = 360/float_frequency
print(f"Accrued days since last coupon: {float_ai_u}")

#Compute accrued interest (AI)
float_ai = np.abs(float_rate)/float_frequency * float_ai_u/float_ai_w
print(f"Float Leg Accrued Interest: {round(float_ai*100, 5)}% of face value or {float_ai * face_value} EUR")
```

Python

```
Accrued days since last coupon: 83
Float Leg Accrued Interest: 0.05441% of face value or 5441.1111111111095 EUR
```

Notice in this case the accrued interest is more interesting because the accrued interest reference day-count will be halved. This is because despite it being an actual day count over 360 days, our floating leg pays semi-annually. So, the reference day count will be 180 days.

Task C – Clean & Dirty Market Values

To calculate the clean and dirty market values of the swap contract, we will need to calculate the PV (Present Value) of each leg. Since we are dealing with a plain vanilla IRS, fixed receiving, we are swapping our floating leg coupons for fixed ones.

Since we have an initial Euribor index, which is the initial floating rate, and a yield curve, we can use the Forward Projection Market Approach. Therefore, we will use the equation:

$$V^{swap}(T_i, c, T_m) = N \times \left(\sum_{i=1}^{n} c \left(\frac{T_{ki} - T_{k(i-1)}}{360} \right) B(t, T_{ki}) - \sum_{j=1}^{m} F(t, L_{j-1}) \left(\frac{T_j - T_{j-1}}{360} \right) B(t, T_j) \right)$$

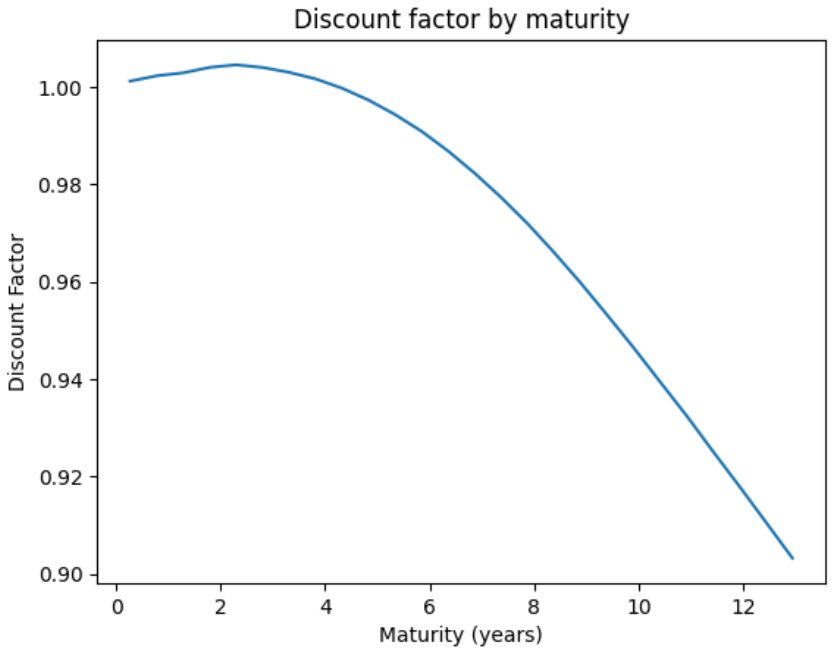
For plain vanilla swaps, in this approach, the future floating rates of the floating leg are equal to the forward rates. We can then use the initial value as a starting point:

$$F(t, L_{j-1}) = \left(\frac{B(t, T_{j-1})}{B(t, T_j)} - 1 \right) \cdot \frac{360}{T_j - T_{j-1}}$$

In any case, we will start by decomposing the above swap value formula into two separate parts yielding two separate steps.

The first step will be computing the discount factors and plotting them for validation:

```python
all_coupon_dates = float_coupon_dates[1:]

def get_discount_factors(yield_curve = yield_curve, payment_freq = 1):
    step = 1/float_frequency #since floating has twice as many cashflows
    n_cashflows = len(all_coupon_dates) * step

    #time array / cashflow number
    t = np.arange(step, n_cashflows + step, step)
    r = yield_curve(t)/payment_freq
    return 1/((r + 1) ** t)

def plot_discount_factors(discount_factors):
    plt.figure();
    plt.plot(
        [count_years(settlement_date, d) for d in all_coupon_dates],
        discount_factors
    );
    plt.title('Discount factor by maturity');
    plt.xlabel('Maturity (years)');
    plt.ylabel('Discount Factor');

plot_discount_factors(get_discount_factors(yield_curve))
```



We can see that it is fine since they are going down by maturity, and therefore can use them commonly for both legs.

Fixed leg market value

Directly adapting the fixed leg part of the value equation above, we can directly adapt into code:

```
def get_fixed_df(rate = fixed_rate, yield_curve = yield_curve):
    coupon_dates = all_coupon_dates[1::2]
    discount_factors = get_discount_factors(yield_curve)[1::2]
    cashflows = discount_factors * rate * face_value

    #specific cases
    cashflows[0] *= fixed_short_discount
    cashflows[-1] += (face_value * discount_factors[-1])

    return pd.DataFrame(data = {
        'discount_factors' : discount_factors,
        'discounted_cashflows': cashflows
    }, index = coupon_dates)


fixed_df = get_fixed_df()
fixed_df
```

With these calculations, we can obtain the cashflows and discount factors per coupon date:

	discount_factors	discounted_cashflows
January 20th, 2020	1.002275	4.563325e+03
January 19th, 2021	1.003995	6.005899e+03
January 19th, 2022	1.003967	6.005733e+03
January 19th, 2023	1.001622	5.991701e+03
January 19th, 2024	0.997269	5.965664e+03
January 20th, 2025	0.990828	5.927134e+03
January 19th, 2026	0.982267	5.875919e+03
January 19th, 2027	0.971917	5.814010e+03
January 19th, 2028	0.959781	5.741410e+03
January 19th, 2029	0.946449	5.661659e+03
January 21st, 2030	0.932429	5.577792e+03
January 20th, 2031	0.917939	5.491110e+03
January 19th, 2032	0.903264	9.038046e+06

The result for the PV of the fixed leg:

```
fixed_pv = fixed_df.discounted_cashflows.sum()
fixed_npv = fixed_pv + fixed_ai*face_value

print(f"Fixed Leg PV (Clean): {fixed_pv} EUR or {fixed_pv/face_value * 100}")
print(f"Fixed Leg NPV (Dirty): {fixed_npv} EUR or {fixed_npv/face_value * 100}")
```

```
Fixed Leg PV (Clean): 9106667.141241258 EUR or 91.06667141241257
Fixed Leg NPV (Dirty): 9108046.324574592 EUR or 91.08046324574592
```

The dirty price is simply the clean price plus the accrued interest. Another important note is that the first cashflow is less because it is a short coupon since some days have accrued from the previous coupon to the settlement date. **Therefore, it can be computed that the coupon will only have 76.11% of its value:**

```
fixed_short_discount = count_days(settlement_date, fixed_coupon_dates[1], fixed_leg_basis)/360
print(f"We will received {round(fixed_short_discount * 100, 3)}% of the first fixed leg coupon")
```

```
We will received 76.111% of the first fixed leg coupon
```

Floating leg market value

To compute the floating leg market value, we will need to forward Euribor 6-month rates. We can therefore use the future floating rate equations and the initial Euribor index given by the problem sheet.

However, we can simplify our forward rate calculations:

$$F\left(t,L_{j-1}\right)\left(\frac{T_j-T_{j-1}}{360}\right)B\left(t,T_j\right) = \left(\frac{B\left(t,T_{j-1}\right)}{B\left(t,T_j\right)}-1\right)\cdot\frac{360}{T_j-T_{j-1}}\left(\frac{T_j-T_{j-1}}{360}\right)B\left(t,T_j\right)$$

Therefore, the float leg PV is simply equivalent to:

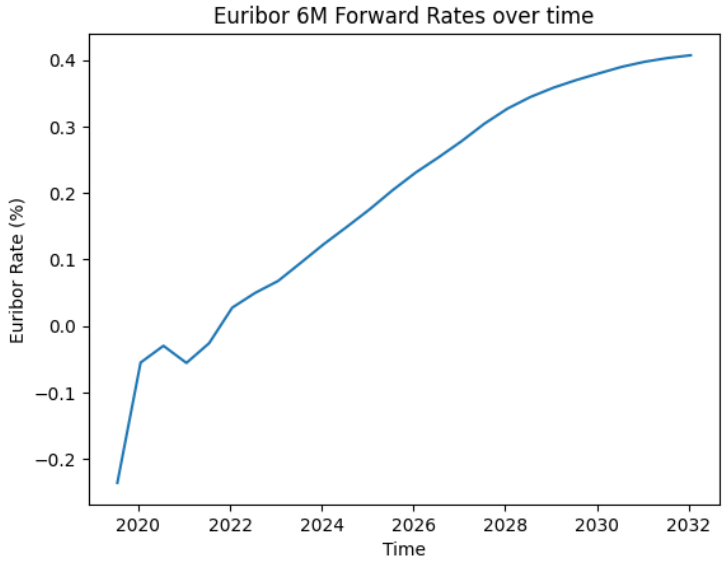
$$\left(\frac{B\left(t,T_{j-1}\right)}{B\left(t,T_j\right)}-1\right)B\left(t,T_j\right)$$

Therefore, our forward rate is simply the left-hand side of the equation above, the inverse (because discount factors increase over time) percentual difference between successive discount factors.

We can fully achieve all steps in a single snippet of code such as the one that can be seen below:

```
def get_float_df(yield_curve = yield_curve):
    coupon_dates = all_coupon_dates
    discount_factors = get_discount_factors(yield_curve, 2)

    #compute forwards
    forward_rates = np.ones( len(coupon_dates) )
    for i in range( len(coupon_dates) - 1):
        forward_rates[i+1] = discount_factors[i]/discount_factors[i+1] - 1
    forward_rates[0] = float_rate

    cashflows = discount_factors * forward_rates * face_value
    #specific cases
    cashflows[0] *= fixed_short_discount
    cashflows[-1] += (face_value * discount_factors[-1])

    return pd.DataFrame(data = {
        'discount_factors' : discount_factors,
        'forward_rates': forward_rates,
        'discounted_cashflows': cashflows
    }, index = coupon_dates)


float_df = get_float_df(yield_curve)
float_df
```

Using it, we can now plot the Euribor 6-month Forward Rates that will be the halved (semi-annual) coupons for the floating leg:



Finally, it can be computed that the floating leg clean PV is 9987861.61 EUR or 9993302.72 EUR with accrued interest.

```
float_pv = float_df.discounted_cashflows.sum()
float_npv = float_pv + float_ai*face_value

print(f"Float Leg PV (Clean): {float_pv} EUR or {float_pv/face_value * 100}")
print(f"Float Leg NPV (Dirty): {float_npv} EUR or {float_npv/face_value * 100}")
```

```
Float Leg PV (Clean): 9987861.610044172 EUR or 99.87861610044172
Float Leg NPV (Dirty): 9993302.721155284 EUR or 99.93302721155284
```

Swap contract market value - solution to task C

Without further steps, we can compute the clean and dirty market values for this swap contract, which will be simply the difference between the present values of each leg:

```
def get_swap_clean_dirty_values(swap_rate = fixed_rate, yield_curve=yield_curve):
    #pvs
    fixed_pv = get_fixed_df(swap_rate, yield_curve).discounted_cashflows.sum()
    float_pv = get_float_df(yield_curve).discounted_cashflows.sum()

    #total accrued
    total_accrued = (fixed_ai + float_ai) * face_value

    #values
    swap_clean_value = fixed_pv - float_pv
    swap_dirty_value = swap_clean_value + total_accrued

    return (swap_clean_value, swap_dirty_value)

swap_clean, swap_dirty = get_swap_clean_dirty_values(fixed_rate, yield_curve)
print(f"Swap Clean Price (Principal): {swap_clean} EUR")
print(f"Swap Dirty Price (NPV): {swap_dirty} EUR")
```

```
Swap Clean Price (Principal): -881194.468802914 EUR
Swap Dirty Price (NPV): -874374.1743584696 EUR
```

To finalize the task, we can conclude that the "clean" market value of the swap contract is -881194.46 EUR.

Task D – Net Present Value of the swap contract

The Net Present Value (NPV) of the Swap is simply the dirty price, meaning principal plus accrued interest from both legs. **Using the previous method, we can compute it to be -874374.17 EUR.**

```
print(f"Swap NPV: {get_swap_clean_dirty_values(fixed_rate, yield_curve)[1]} EUR")
```

```
Swap NPV: -874374.1743584696 EUR
```

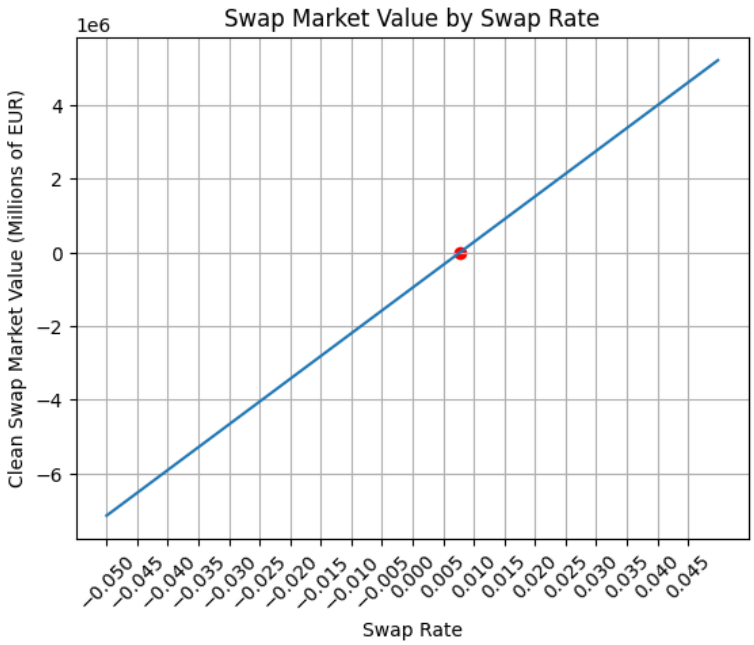
Task E – Swap Par Rate

The Swap Par Rate is the swap rate, or fixed leg coupon rate, such that the value of the swap contract is zero. The easiest way to compute the swap par rate is to use the following formula:

$$c = n \times \left(\frac{1 - B(0, T_M)}{\sum_{j=1}^{M} B(0, T_j)} \right)$$

However, since our payment frequencies between legs do not match, we will follow a different approach. Since we have previously defined a function to compute the swap contract value given a rate, we can simply either deduce it graphically or iteratively.

Graphically, if we plot the swap value by rate, we can deduce it:



Iteratively we can get a precise numerical solution. In this case, we will use an optimization library to find the root of our swap value function:

```
from scipy.optimize import fsolve

swap_par_rate = fsolve(lambda r: get_swap_clean_dirty_values(r)[0], fixed_rate)[0]

print(f"If swap par rate is {swap_par_rate*100}% then Swap value = {get_swap_clean_dirty_values(swap_par_rate)[0]}")
```

```
If swap par rate is 0.7719210345459566% then Swap value = 0.0
```

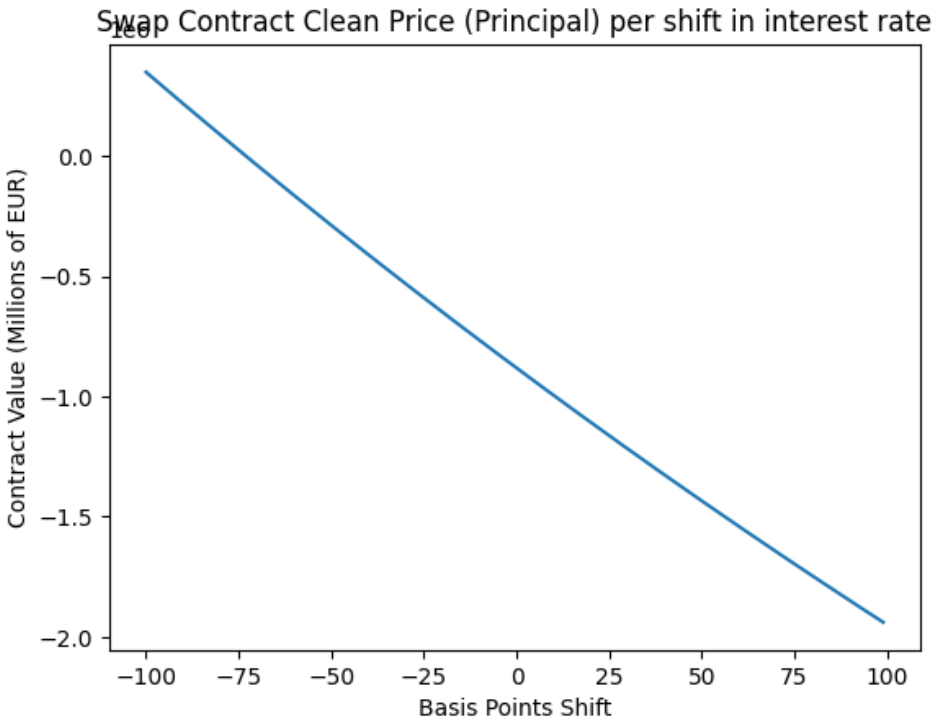
Therefore, we can conclude the Swap Par Rate is 0.772%.

Task F – IRS Greeks (PV01, DV01, and Gamma)

The IRS Greeks (PV01, DV01, and Gamma) help us understand what happens to an instrument's price whenever the interest rates change, being the primary risk metrics. Since an IRS is composed of two instruments forming a bigger instrument, for calculating Greeks we will need to also calculate them for the underlying bonds/legs.

Since we are in a fixed-receiving position (short float), we are expecting to lose value whenever interest rates rise. We can see the relationship as plotted by a small snippet of code:

```
plt.figure();
plt.plot(
    np.arange(-100, 100, 1),
    [
        get_swap_clean_dirty_values(fixed_rate, get_shifted_yield_curve(bps_shift))[0]
        for bps_shift in np.arange(-100, 100, 1)
    ]
);
plt.title("Swap Contract Clean Price (Principal) per shift in interest rate");
plt.xlabel("Basis Points Shift");
plt.ylabel("Contract Value (Millions of EUR)");
```



The PV01 is the Price Value change by Basis Point can be simply computed by hiking the interest rates by 1 basis point and computing the difference between the previous and **new price which will be the loss**:

```
def get_pv01():
    p = get_swap_clean_dirty_values(fixed_rate, yield_curve)[0]
    p_new = get_swap_clean_dirty_values(fixed_rate, get_shifted_yield_curve(1))[0]
    return p-p_new

get_pv01()
```

0.0s Python

```
11444.318733830005
```

The DV01, or dollar value change, can be computed using the below snippet of code:

```
def get_dv01():
    #similar to bloomberg calculations
    p_up = get_swap_clean_dirty_values(fixed_rate, get_shifted_yield_curve(1))[0]
    p = get_swap_clean_dirty_values(fixed_rate, yield_curve)[0]
    p_down = get_swap_clean_dirty_values(fixed_rate, get_shifted_yield_curve(-1))[0]
    return (p_down-p_up)/2

get_dv01()
```

0.0s Python

```
11452.42677406501
```

The Gamma, or convexity, can be calculated using the code below:

```
def get_gamma():
    p_up = get_swap_clean_dirty_values(fixed_rate, get_shifted_yield_curve(1))[0]
    p = get_swap_clean_dirty_values(fixed_rate, yield_curve)[0]
    p_down = get_swap_clean_dirty_values(fixed_rate, get_shifted_yield_curve(-1))[0]
    return (p_up - 2 * p + p_down)

get_gamma()
```

0.0s Python

```
16.21608047001064
```