# Q1

# %%

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from datetime import datetime

# -------------------

# bond characteristics

# -------------------

bond\_characteristics = {

'Issuer': 'BNP Paribas',

'Bond Name': 'Variable Rate Bond 2033',

'ISIN': 'XS2392609181',

'Currency': 'EUR',

'Nominal Value': 1000,

'Issue Price': 1000,

'Issue Date': pd.Timestamp('2022-07-29'),

'Maturity Date': pd.Timestamp('2027-07-29'),

'Trade Date': pd.Timestamp('2024-11-24'),

'Coupon Rate': '3M Euribor',

'Floor': 0.016,

'Cap': 0.037,

'Coupon Frequency': 4,

'Interest Payment Dates': ['29/01', '29/04', '29/07', '29/10'],

'Settlement Lag': 2,

'Business Day Convention (Interest Payment)': 'Modified Following',

'Business Day Convention (Maturity Date)': 'Modified Following',

'Day Count Convention (Days in Month)': 30,

'Day Count Convention (Days in Year)': 360,

'Redemption Type': 'At Par',

'Credit Rating': 'AA-',

'Listing': 'EuroTLX Listing'

}

def get\_next\_payment\_date(trade\_date, payment\_dates):

"""

Given a trade date (datetime) and a list of interest payment dates (as 'DD/MM' strings),

return the next interest payment date as a datetime object.

"""

year = trade\_date.year

candidates = []

for date\_str in payment\_dates:

day, month = map(int, date\_str.strip().split('/'))

candidate = datetime(year, month, day)

candidates.append(candidate)

future\_dates = [d for d in candidates if d > trade\_date]

if future\_dates:

return min(future\_dates)

else:

day, month = map(int, payment\_dates[0].strip().split('/'))

return datetime(year + 1, month, day)

def coupon\_payoff(r, floor, cap):

return np.minimum(np.maximum(r, floor), cap)

def main():

# -------------------

# key dates

# -------------------

trade\_date = bond\_characteristics['Trade Date']

spot\_lag = 2

value\_date = trade\_date + pd.Timedelta(days=spot\_lag)

# date of first interest payment

first\_interest\_payment\_date = get\_next\_payment\_date(trade\_date, bond\_characteristics['Interest Payment Dates'])

key\_dates = {

"Trade Date": trade\_date,

"Spot Lag (days)": spot\_lag,

"Value Date": value\_date,

"First Interest Payment Date": first\_interest\_payment\_date

}

df\_key\_dates = pd.DataFrame(list(key\_dates.items()), columns=["Key Date", "Value"])

print(df\_key\_dates)

# create dataframe of bond char.

df\_bond = pd.DataFrame(list(bond\_characteristics.items()), columns=["Characteristic", "Value"])

df\_bond.set\_index("Characteristic", inplace=True)

# -------------------

# cap and floor the ref. rate

# -----------------

floor = bond\_characteristics['Floor']

cap = bond\_characteristics['Cap']

euribor\_range = np.arange(0, 0.0501, 0.001)

coupon\_values = coupon\_payoff(euribor\_range, floor, cap)

df\_coupon\_payoff = pd.DataFrame({

"Euribor Rate (%)": euribor\_range \* 100,

"Coupon Payoff (%)": coupon\_values \* 100

})

print(df\_coupon\_payoff)

# -------------------

# plot coupon payoff vs 3M euribor rate

# ----------------

sns.set(style="whitegrid", context="notebook")

plt.figure(figsize=(10, 6))

plt.axhline(y=cap \* 100, color='red', linestyle='--', label='Cap (3.70%)')

plt.axhline(y=floor \* 100, color='green', linestyle='--', label='Floor (1.60%)')

sns.lineplot(x="Euribor Rate (%)", y="Coupon Payoff (%)", data=df\_coupon\_payoff, label="Coupon Payoff", linewidth=2)

plt.xlabel("3M Euribor Rate (%)")

plt.ylabel("Coupon Payoff (%)")

plt.title("Coupon Payoff vs 3M Euribor Rate (With Floor & Cap)")

plt.legend()

plt.tight\_layout()

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

main()

# %%

# Q2

# %%

import os

from datetime import datetime

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

import QuantLib as ql

from q1 import bond\_characteristics # we need cap, floor, notional, issue date

# ----------------------------

# 0. file paths and bond details

# ----------------------------

ROOT\_PATH = os.path.dirname(\_\_file\_\_)

euribor\_rates\_file\_path = os.path.join(ROOT\_PATH, 'datasets', 'HistoricalEuribor.csv')

holidays\_file\_path = os.path.join(ROOT\_PATH, 'datasets', 'Holidays.csv')

# ----------------------------

# 1. load 3MEUR data and holidays

# ----------------------------

def load\_euribor(filepath):

df = pd.read\_csv(filepath)

df.columns = df.columns.str.strip()

df = df[["Date", "3M"]]

df["Date"] = pd.to\_datetime(df["Date"])

df.set\_index("Date", inplace=True)

return df

def load\_holidays(filepath):

df = pd.read\_csv(filepath, parse\_dates=["Date"])

return [d.date() for d in df["Date"]]

# ----------------------------

# 2. helper function that converts QuantLib date into datetime

# ----------------------------

def ql\_to\_datetime(qldate):

return datetime(qldate.year(), qldate.month(), qldate.dayOfMonth()).date()

# ----------------------------

# 3. build historical coupon rates and coupons

# ----------------------------

def build\_historical\_coupons(euribor\_df, holidays):

# bond characteristics

calendar = ql.TARGET()

business\_convention = ql.ModifiedFollowing

fixing\_days = bond\_characteristics["Settlement Lag"]

frequency = ql.Quarterly

notional = bond\_characteristics["Nominal Value"]

cap = bond\_characteristics["Cap"]

floor = bond\_characteristics["Floor"]

start\_date = ql.Date(bond\_characteristics["Issue Date"].day,

bond\_characteristics["Issue Date"].month,

bond\_characteristics["Issue Date"].year)

end\_date = ql.Date(bond\_characteristics["Trade Date"].day,

bond\_characteristics["Trade Date"].month,

bond\_characteristics["Trade Date"].year)

# ql.Date.todaysDate()

# end\_date = ql.Date.todaysDate()

day\_counter = ql.Actual360()

# build schedule of coupon payments

schedule = ql.Schedule(

start\_date, end\_date,

ql.Period(frequency),

calendar,

business\_convention, business\_convention,

ql.DateGeneration.Forward, False

)

# loop over periods

coupon\_data = []

for i in range(len(schedule) - 1):

start = schedule[i]

end = schedule[i + 1]

reset = calendar.advance(start, -fixing\_days, ql.Days)

reset\_dt = ql\_to\_datetime(reset)

# collect EUR rate as reference rate

rate = euribor\_df.loc[euribor\_df.index == pd.to\_datetime(reset\_dt), "3M"]

if rate.empty:

rate = np.nan

else:

rate = rate.iloc[0] / 100

# cap/floor adjustment

if not np.isnan(rate):

capped\_rate = max(min(rate, cap), floor)

else:

capped\_rate = np.nan

# year fraction and coupon calc.

yf = day\_counter.yearFraction(start, end)

coupon = notional \* capped\_rate \* yf if not np.isnan(capped\_rate) else np.nan

coupon\_data.append({

"Reset Date": reset\_dt,

"Start Date": ql\_to\_datetime(start),

"End Date": ql\_to\_datetime(end),

"Reference Rate (3M)": rate \* 100 if rate else np.nan,

"Coupon Rate (%)": capped\_rate \* 100 if capped\_rate else np.nan,

"Coupon Amount": coupon

})

return pd.DataFrame(coupon\_data)

# ----------------------------

# 4. run and plot

# ----------------------------

def main():

euribor\_df = load\_euribor(euribor\_rates\_file\_path)

holidays = load\_holidays(holidays\_file\_path)

df\_coupons = build\_historical\_coupons(euribor\_df, holidays)

print(df\_coupons)

# plot historical capped/floored coupon rates

plt.figure(figsize=(10, 5))

plt.plot(df\_coupons["Start Date"], df\_coupons["Coupon Rate (%)"], marker="o")

plt.title("Historical Capped/Floored Coupon Rates")

plt.xlabel("Start Date")

plt.ylabel("Coupon Rate (%)")

plt.grid(True)

plt.xticks(rotation=45)

plt.tight\_layout()

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

main()

# %%

# Q3

# %%

import os

import QuantLib as ql

import pandas as pd

import matplotlib.pyplot as plt

import numpy as np

import re

# ----------------------------

# 0. setup evaluation date and day count convention

# ----------------------------

eval\_date = ql.Date(18, 11, 2024) # this is because we value the curve on 18th Nov 2024 using Excel MarketData.xlsx to consider whether to invest in a 5-year bond

ql.Settings.instance().evaluationDate = eval\_date

calendar = ql.TARGET()

fixing\_days = 2

deposit\_day\_counter = ql.Actual360()

curve\_day\_counter = ql.Actual360()

# ----------------------------

# 1. load deposit and swap rates and create helper functions

# ----------------------------

def load\_rate\_helpers(file\_path, fixing\_days=2):

"""

Load rate helpers for deposits and interest rate swaps (IRS) from an Excel file.

This function reads deposit rates and IRS rates from an Excel file and creates

QuantLib rate helpers for use in curve construction. Deposit rate helpers are

created for short-term maturities, while swap rate helpers are created for

maturities greater than one year.

Args:

file\_path (str): The path to the Excel file containing the deposit and IRS rates.

fixing\_days (int, optional): The number of fixing days for the instruments.

Defaults to 2.

Returns:

list: A list of QuantLib rate helpers, including both deposit and swap rate helpers.

Notes:

- The Excel file should have two sheets:

1. "Deposit Rates": Contains deposit rate data with columns 'RIC' and 'Last'.

2. "IRS Rates": Contains IRS rate data with columns 'Name' and 'Last'.

- The 'RIC' column in the "Deposit Rates" sheet should match the keys in the

`deposit\_ric\_map` dictionary.

- The 'Name' column in the "IRS Rates" sheet should contain maturity information

in the format "<number>Y" (e.g., "5Y" for a 5-year maturity).

"""

calendar = ql.TARGET()

deposit\_day\_counter = ql.Actual360()

deposit\_ric\_map = {

"EURSWD": ql.Period(1, ql.Weeks),

"EUR1MD": ql.Period(1, ql.Months),

"EUR3MD": ql.Period(3, ql.Months),

"EUR6MD": ql.Period(6, ql.Months),

"EUR9MD": ql.Period(9, ql.Months)

}

deposits\_df = pd.read\_excel(file\_path, sheet\_name="Deposit Rates")

irs\_df = pd.read\_excel(file\_path, sheet\_name="IRS Rates")

depo\_helpers = [

ql.DepositRateHelper(

ql.QuoteHandle(ql.SimpleQuote(row['Last'] / 100.0)),

deposit\_ric\_map[row['RIC']],

fixing\_days,

calendar,

ql.ModifiedFollowing,

True,

deposit\_day\_counter

)

for \_, row in deposits\_df.iterrows() if row['RIC'] in deposit\_ric\_map

]

def extract\_maturity(name):

match = re.search(r'(\d+)(?=Y)', name)

return int(match.group(1)) if match else None

swap\_helpers = [

ql.SwapRateHelper(

ql.QuoteHandle(ql.SimpleQuote(row['Last'] / 100.0)),

ql.Period(maturity, ql.Years),

calendar,

ql.Annual,

ql.ModifiedFollowing,

ql.Thirty360(ql.Thirty360.BondBasis),

ql.Euribor3M()

)

for \_, row in irs\_df.iterrows()

if (maturity := extract\_maturity(row['Name'])) and maturity >= 1

]

return depo\_helpers + swap\_helpers

# ----------------------------

# 2. function to build yield curves

# ----------------------------

def build\_curves():

ROOT\_PATH = os.path.dirname(\_\_file\_\_)

market\_data\_file\_path = os.path.join(ROOT\_PATH, 'datasets', 'MarketData.xlsx')

rate\_helpers = load\_rate\_helpers(market\_data\_file\_path)

linear\_curve = ql.PiecewiseLinearZero(eval\_date, rate\_helpers, curve\_day\_counter)

flat\_curve = ql.PiecewiseFlatForward(eval\_date, rate\_helpers, curve\_day\_counter)

cubic\_curve = ql.PiecewiseCubicZero(eval\_date, rate\_helpers, curve\_day\_counter)

log\_cubic\_curve = ql.PiecewiseLogCubicDiscount(eval\_date, rate\_helpers, curve\_day\_counter)

return {

"Linear": linear\_curve,

"Flat": flat\_curve,

"Cubic": cubic\_curve,

"Log-Cubic": log\_cubic\_curve

}

# ----------------------------

# 3. evaluation grid

# ----------------------------

# Note: these dates are used for curve plotting and tabling, but are not indicative of the full range of the curve

# The curve is built for al maturities up to 60 years and can be evaluated at any date within that range

end\_date = calendar.advance(eval\_date, ql.Period(60, ql.Years))

n\_points = 100 # number of points on the curve

dates = [eval\_date + ql.Period(int(i \* (end\_date.serialNumber() - eval\_date.serialNumber()) / n\_points), ql.Days)

for i in range(n\_points + 1)] # evaluation dates

date\_strings = [d.ISO() for d in dates]

max\_forward\_date = calendar.advance(end\_date, -ql.Period(1, ql.Years))

# ----------------------------

# 4. helper function to extract curve data

# ----------------------------

def get\_curve\_data(curve, day\_counter):

discount\_factors = [curve.discount(d) for d in dates]

spot\_rates = [curve.zeroRate(d, day\_counter, ql.Continuous).rate() \* 100 for d in dates]

forward\_rates = []

for d in dates:

if d <= max\_forward\_date:

d1 = calendar.advance(d, ql.Period(1, ql.Years))

fwd = curve.forwardRate(d, d1, day\_counter, ql.Continuous).rate() \* 100

forward\_rates.append(fwd)

else:

forward\_rates.append(np.nan)

return discount\_factors, spot\_rates, forward\_rates

# ----------------------------

# 5. only run full workflow if script is executed directly

# ----------------------------

if \_\_name\_\_ == "\_\_main\_\_":

curves = build\_curves()

curve\_dataframes = {}

for name, curve in curves.items():

disc, spot, fwd = get\_curve\_data(curve, curve\_day\_counter)

df = pd.DataFrame({

'Date': date\_strings,

'Discount Factor': disc,

'Spot Rate (%)': spot,

'1Y Forward Rate (%)': fwd

}).set\_index('Date')

curve\_dataframes[name] = df

# create plots

fig, axs = plt.subplots(3, 1, figsize=(10, 15), sharex=True)

fig.suptitle(f'{name} Interpolation Yield Curve', fontsize=16, y=0.96)

axs[0].plot(date\_strings, disc, label=name)

axs[0].set\_ylabel('Discount Factor')

axs[0].set\_title('Discount Factors vs Maturity')

axs[0].grid(axis='y')

axs[1].plot(date\_strings, spot, label=name, color='orange')

axs[1].set\_ylabel('Spot Rate (%)')

axs[1].set\_title('Spot Rates vs Maturity')

axs[1].grid(axis='y')

axs[2].plot(date\_strings, fwd, label=name, color='green')

axs[2].set\_ylabel('1Y Forward Rate (%)')

axs[2].set\_title('1Y Forward Rates vs Maturity')

axs[2].set\_xlabel('Maturity Date')

axs[2].grid(axis='y')

tick\_count = 5

plt.xticks(date\_strings[::tick\_count], rotation=45)

plt.tight\_layout(rect=[0, 0, 1, 0.97])

plt.show()

print(f"{name} Interpolation Yield Curve Data (first 10 rows):")

print(df.head(10))

print("\n" + "="\*80 + "\n")

# comparison of fwd curves

plt.figure(figsize=(12, 6),dpi=250)

for name, df in curve\_dataframes.items():

plt.plot(df.index, df['1Y Forward Rate (%)'], label=name)

plt.title("1Y Forward Rate Curve Comparison")

plt.xlabel("Maturity Date")

plt.ylabel("1Y Forward Rate (%)")

plt.xticks(date\_strings[::5], rotation=45)

plt.grid(axis='y')

plt.legend()

plt.tight\_layout()

plt.show()

# %%

# Q4

# %%

import os

import pandas as pd

import matplotlib.pyplot as plt

import numpy as np

import QuantLib as ql

from q1 import bond\_characteristics

from q3 import build\_curves

# -----------------

# 0. setup paths and curve

# ------------------

ROOT\_PATH = os.path.dirname(\_\_file\_\_)

df\_curve = build\_curves()["Log-Cubic"]

# -----------------

# 1. bond settings

# ----------------

calendar = ql.TARGET()

convention = ql.ModifiedFollowing

frequency = ql.Quarterly

settlement\_lag = bond\_characteristics["Settlement Lag"]

notional = bond\_characteristics["Nominal Value"]

cap = bond\_characteristics["Cap"]

floor = bond\_characteristics["Floor"]

issue\_date = ql.Date(bond\_characteristics["Issue Date"].day,

bond\_characteristics["Issue Date"].month,

bond\_characteristics["Issue Date"].year)

maturity\_date = ql.Date(29, 7, 2027)

day\_counter = ql.Actual360()

eval\_date = ql.Settings.instance().evaluationDate

# -----------------

# 2. build schedule

# ------------------

schedule = ql.Schedule(

max(issue\_date, eval\_date), maturity\_date,

ql.Period(frequency),

calendar,

convention, convention,

ql.DateGeneration.Forward, False

)

# -----------------

# 3. helper to build cash flows

# ------------------

def generate\_cashflows(rate):

"""

Generates a DataFrame of cashflows for a given coupon rate.

Args:

rate (float): The annual coupon rate as a decimal (e.g., 0.05 for 5%).

Returns:

pandas.DataFrame: A DataFrame containing the following columns:

- "Start Date": The start date of the cashflow period.

- "End Date": The end date of the cashflow period.

- "Year Fraction": The year fraction between the start and end dates.

- "Coupon Rate (%)": The coupon rate expressed as a percentage.

- "Coupon Amount": The coupon payment for the period.

- "Discount Factor": The discount factor for the end date.

- "PV": The present value of the coupon payment.

"""

cashflows = []

for i in range(len(schedule) - 1):

start = schedule[i]

end = schedule[i + 1]

yf = day\_counter.yearFraction(start, end)

cpn = notional \* rate \* yf

cashflows.append({

"Start Date": start,

"End Date": end,

"Year Fraction": yf,

"Coupon Rate (%)": rate \* 100,

"Coupon Amount": cpn,

"Discount Factor": df\_curve.discount(end),

"PV": cpn \* df\_curve.discount(end)

})

return pd.DataFrame(cashflows)

# ------------------

# 4. expose values for import

# ------------------

df\_best = generate\_cashflows(cap)

df\_worst = generate\_cashflows(floor)

# ------------------

# 5. main script block

# ------------------

def main():

npv\_best = df\_best["PV"].sum()

npv\_worst = df\_worst["PV"].sum()

print("best case NPV (cap rate):", round(npv\_best, 4))

print("worst case NPV (floor rate):", round(npv\_worst, 4))

# plot coupon comparison

labels = [end.to\_date() for end in df\_best["End Date"]]

x = np.arange(len(labels))

width = 0.4

plt.figure(figsize=(10, 5))

plt.bar(x - width/2, df\_best["Coupon Amount"], width=width, label="Best Case (Cap)", alpha=0.7)

plt.bar(x + width/2, df\_worst["Coupon Amount"], width=width, label="Worst Case (Floor)", alpha=0.7)

plt.xticks(x, [d.strftime('%Y-%m-%d') for d in labels], rotation=45)

plt.ylabel("Coupon Amount")

plt.xlabel("Payment Date")

plt.title("Future Coupon Comparison: Cap vs Floor Scenarios")

plt.legend()

plt.grid(True)

plt.tight\_layout()

plt.show()

# show tables

print("\nBest Case cash flows (cap rate):")

print(df\_best[["Start Date", "End Date", "Coupon Rate (%)", "Coupon Amount", "Discount Factor", "PV"]])

print("\nWorst Case cash flows (floor rate):")

print(df\_worst[["Start Date", "End Date", "Coupon Rate (%)", "Coupon Amount", "Discount Factor", "PV"]])

# ------------------

# 6. guard

# ------------------

if \_\_name\_\_ == "\_\_main\_\_":

main()

# %%

# Q5

# %%

import os

import pandas as pd

import numpy as np

import QuantLib as ql

from q1 import bond\_characteristics

from q3 import build\_curves

# ----------------------------

# 0. setup and load curve

# ----------------------------

ROOT\_PATH = os.path.dirname(\_\_file\_\_)

vol\_path = os.path.join(ROOT\_PATH, 'datasets', 'shifted\_black\_vols.csv')

log\_cubic\_curve = build\_curves()["Log-Cubic"]

eval\_date = ql.Settings.instance().evaluationDate

calendar = ql.TARGET()

cap\_rate = bond\_characteristics["Cap"]

floor\_rate = bond\_characteristics["Floor"]

notional = bond\_characteristics["Nominal Value"]

settlement\_lag = bond\_characteristics["Settlement Lag"]

issue\_date = ql.Date(bond\_characteristics["Issue Date"].day,

bond\_characteristics["Issue Date"].month,

bond\_characteristics["Issue Date"].year)

maturity\_date = ql.Date(29, 7, 2027) # from coursework

frequency = ql.Quarterly

day\_counter = ql.Actual360()

shift = 0.03

# ----------------------------

# 1. load and process vol surface

# ----------------------------

def load\_shifted\_vol\_surface(filepath):

"""

Loads a shifted volatility surface from a CSV file, processes the data, and returns it as a DataFrame.

The function reads a CSV file containing a volatility surface, removes unnecessary columns

("STK" and "ATM" if they exist), converts the "Maturity" column to float, and sets it as the

index of the DataFrame. The column names are also converted to floats, and the values are

scaled from basis points (bps) to decimals.

Args:

filepath (str): The file path to the CSV file containing the shifted volatility surface.

Returns:

pandas.DataFrame: A DataFrame containing the processed volatility surface, with maturities

as the index and scaled volatilities as the values.

"""

df = pd.read\_csv(filepath)

df = df.drop(columns=["STK", "ATM"], errors="ignore")

df["Maturity"] = df["Maturity"].astype(float)

df.set\_index("Maturity", inplace=True)

df.columns = df.columns.astype(float)

return df / 100 # convert bps to decimals

vol\_surface = load\_shifted\_vol\_surface(vol\_path)

def interpolate\_vol(maturity, strike\_percent):

"""

Interpolates the implied volatility for a given option maturity and strike percentage

using a volatility surface.

Parameters:

maturity (float): The maturity of the option in years. If the exact maturity

is not available in the volatility surface, the closest

available maturity will be used.

strike\_percent (float): The strike price as a percentage of the underlying asset's

current price.

Returns:

float: The interpolated implied volatility corresponding to the given maturity

and strike percentage.

Notes:

- The function assumes the existence of a global variable `vol\_surface`, which

is a pandas DataFrame where the rows represent maturities, the columns represent

strike percentages, and the values represent implied volatilities.

- If the exact maturity is not found in the volatility surface, the closest

maturity is selected based on the absolute difference.

- Linear interpolation is performed using numpy's `interp` function to estimate

the implied volatility for the given strike percentage.

"""

if maturity not in vol\_surface.index:

maturity = min(vol\_surface.index, key=lambda x: abs(x - maturity))

row = vol\_surface.loc[maturity]

strikes = row.index.to\_numpy()

vols = row.values

return np.interp(strike\_percent, strikes, vols)

# ----------------------------

# 2. build cap/floor schedule and leg

# ----------------------------

schedule = ql.Schedule(

max(eval\_date, issue\_date), maturity\_date,

ql.Period(frequency),

calendar,

ql.ModifiedFollowing, ql.ModifiedFollowing,

ql.DateGeneration.Forward, False

)

index = ql.Euribor3M(ql.YieldTermStructureHandle(log\_cubic\_curve))

float\_leg = ql.IborLeg([notional], schedule, index)

# ----------------------------

# 3. create cap and floor instruments

# ----------------------------

cap = ql.Cap(float\_leg, [cap\_rate])

floor = ql.Floor(float\_leg, [floor\_rate])

# determine maturity for vol interpolation

maturity = round(day\_counter.yearFraction(eval\_date, maturity\_date))

vol\_cap = interpolate\_vol(maturity, cap\_rate \* 100)

vol\_floor = interpolate\_vol(maturity, floor\_rate \* 100)

# ----------------------------

# 4. assign BlackCapFloorEngine

# ----------------------------

def make\_engine(vol):

"""

Creates a BlackCapFloorEngine with the specified volatility.

Parameters:

vol (float): The volatility to use in the BlackCapFloorEngine.

Returns:

ql.BlackCapFloorEngine: The BlackCapFloorEngine object with the specified

volatility and other parameters set.

"""

return ql.BlackCapFloorEngine(

ql.YieldTermStructureHandle(log\_cubic\_curve),

ql.QuoteHandle(ql.SimpleQuote(vol)),

day\_counter,

shift

)

fixing\_date = eval\_date # since our data is our Eval Date

index.addFixing(fixing\_date, cap\_rate)

cap.setPricingEngine(make\_engine(vol\_cap))

floor.setPricingEngine(make\_engine(vol\_floor))

npv\_cap = cap.NPV()

npv\_floor = floor.NPV()

# ----------------------------

# 5. results

# ----------------------------

print(f"cap leg NPV (short): {-npv\_cap:.4f}")

print(f"floor leg NPV (long): {npv\_floor:.4f}")

print(f"net option value (floor - cap): {npv\_floor - npv\_cap:.4f}")

# %%

# Q6

# %%

import QuantLib as ql

import numpy as np

import pandas as pd

from q1 import bond\_characteristics

from q3 import build\_curves

# ----------------------------

# 0. setup evaluation date and day count convention

# ----------------------------

log\_cubic\_curve = build\_curves()["Log-Cubic"]

eval\_date = ql.Date(18, 11, 2024)

ql.Settings.instance().evaluationDate = eval\_date

calendar = ql.TARGET()

# bond characteristics

notional = bond\_characteristics["Nominal Value"]

cap = bond\_characteristics["Cap"]

floor = bond\_characteristics["Floor"]

settlement\_lag = bond\_characteristics["Settlement Lag"]

frequency = ql.Quarterly

convention = ql.ModifiedFollowing

issue\_date = ql.Date(bond\_characteristics["Issue Date"].day,

bond\_characteristics["Issue Date"].month,

bond\_characteristics["Issue Date"].year)

maturity\_date = ql.Date(29, 7, 2027)

# day count conventions

day\_counter\_curve = ql.Actual360()

day\_counter\_coupon = ql.Thirty360(ql.Thirty360.BondBasis)

# BNP CDS data (as of 18 Nov 2024 - EVALUATION)

base\_cds\_spread = 0.004579 # 45.79 bps

base\_recovery\_rate = 0.40 # 40%

# ----------------------------

# 1. Build coupon schedule

# ----------------------------

schedule = ql.Schedule(

issue\_date, maturity\_date,

ql.Period(frequency),

calendar,

convention, convention,

ql.DateGeneration.Forward, False

)

# ----------------------------

# 2. Build variable exposure cashflows

# ----------------------------

def get\_forward\_rate\_safe(start, end):

safe\_start = max(start, eval\_date)

if safe\_start >= end:

return 0.0

return log\_cubic\_curve.forwardRate(safe\_start, end, day\_counter\_curve, ql.Simple).rate()

exposure\_rows = []

for i in range(len(schedule) - 1):

start = schedule[i]

end = schedule[i + 1]

if end <= eval\_date:

continue

if start < eval\_date:

reset\_date = calendar.advance(start, -settlement\_lag, ql.Days)

start\_for\_fwd = reset\_date if eval\_date >= reset\_date else eval\_date

else:

start\_for\_fwd = start

fwd\_rate = get\_forward\_rate\_safe(start\_for\_fwd, end)

effective\_rate = min(max(fwd\_rate, floor), cap)

yf = day\_counter\_coupon.yearFraction(start, end)

coupon = notional \* effective\_rate \* yf

df = log\_cubic\_curve.discount(end)

pv = coupon \* df

exposure\_rows.append({

"Payment Date": end,

"Forward Rate (%)": fwd\_rate \* 100,

"Effective Rate (%)": effective\_rate \* 100,

"Coupon Amount": coupon,

"Discount Factor": df,

"Present Value": pv

})

# Add redemption at maturity (only once)

df\_redemption = log\_cubic\_curve.discount(maturity\_date)

pv\_redemption = notional \* df\_redemption

exposure\_rows.append({

"Payment Date": maturity\_date,

"Forward Rate (%)": None,

"Effective Rate (%)": None,

"Coupon Amount": notional,

"Discount Factor": df\_redemption,

"Present Value": pv\_redemption

})

df\_variable\_exposure = pd.DataFrame(exposure\_rows)

# ----------------------------

# 3. Survival probability + CVA

# ----------------------------

def survival\_prob(t, cds, R):

return (np.exp(-cds \* t) - R) / (1 - R)

def compute\_cva(cds, R, df\_exposure):

cva = 0.0

rows = []

for \_, row in df\_exposure.iterrows():

payment\_date = row["Payment Date"]

t = day\_counter\_curve.yearFraction(eval\_date, payment\_date)

if t <= 0:

continue

exposure = row["Coupon Amount"]

df = row["Discount Factor"]

surv = survival\_prob(t, cds, R)

default\_prob = 1 - surv

marginal = exposure \* default\_prob \* df

cva += marginal

rows.append({

"Payment Date": payment\_date,

"Year Fraction": round(t, 4),

"Exposure": exposure,

"Discount Factor": df,

"Survival Prob": surv,

"Default Prob": default\_prob,

"Marginal CVA": marginal

})

return cva, rows

# Compute CVA using variable coupon exposures

base\_cva, base\_cva\_rows = compute\_cva(base\_cds\_spread, base\_recovery\_rate, df\_variable\_exposure)

# Compute risk-free and adjusted value

risk\_free\_npv = df\_variable\_exposure["Present Value"].sum()

adjusted\_npv = risk\_free\_npv - base\_cva

def main():

print("\n=== Base Case ===")

print(f"Risk-free Value: {risk\_free\_npv:.4f}")

print(f"Total CVA: {base\_cva:.4f}")

print(f"Credit-adjusted Value:{adjusted\_npv:.4f}")

# ----------------------------

# 4. Sensitivity Analysis

# ----------------------------

# CDS spread sensitivity

cds\_values = np.linspace(0.002, 0.007, 6)

sensitivity\_cds = []

for cds in cds\_values:

cva\_val, \_ = compute\_cva(cds, base\_recovery\_rate, df\_variable\_exposure)

adjusted\_val = risk\_free\_npv - cva\_val

sensitivity\_cds.append({

"CDS Spread": cds,

"CVA": cva\_val,

"Adjusted NPV": adjusted\_val

})

df\_sens\_cds = pd.DataFrame(sensitivity\_cds)

print("\nSensitivity Analysis - CDS Spread:")

print(df\_sens\_cds)

# Recovery rate sensitivity

recov\_values = np.linspace(0.30, 0.50, 5)

sensitivity\_recov = []

for recov in recov\_values:

cva\_val, \_ = compute\_cva(base\_cds\_spread, recov, df\_variable\_exposure)

adjusted\_val = risk\_free\_npv - cva\_val

sensitivity\_recov.append({

"Recovery Rate": recov,

"CVA": cva\_val,

"Adjusted NPV": adjusted\_val

})

df\_sens\_recov = pd.DataFrame(sensitivity\_recov)

print("\nSensitivity Analysis - Recovery Rate:")

print(df\_sens\_recov)

# ----------------------------

# 5. CVA Breakdown Table

# ----------------------------

df\_cva = pd.DataFrame(base\_cva\_rows)

print("\nDetailed CVA Breakdown:")

print(df\_cva[["Payment Date", "Exposure", "Discount Factor", "Survival Prob", "Default Prob", "Marginal CVA"]])

if \_\_name\_\_ == "\_\_main\_\_":

main()

# %%

# Q8

#%%

import QuantLib as ql

import pandas as pd

from q1 import bond\_characteristics

from q3 import build\_curves

# --- Setup ---

log\_cubic\_curve = build\_curves()["Log-Cubic"]

eval\_date = ql.Date(18, 11, 2024) # same as eval\_date

ql.Settings.instance().evaluationDate = eval\_date

calendar = ql.TARGET()

convention = ql.ModifiedFollowing

frequency = ql.Quarterly

settlement\_lag = bond\_characteristics["Settlement Lag"]

notional = bond\_characteristics["Nominal Value"]

cap = bond\_characteristics["Cap"]

floor = bond\_characteristics["Floor"]

issue\_date = ql.Date(bond\_characteristics["Issue Date"].day,

bond\_characteristics["Issue Date"].month,

bond\_characteristics["Issue Date"].year)

maturity\_date = ql.Date(29, 7, 2027)

schedule = ql.Schedule(

issue\_date, maturity\_date,

ql.Period(frequency),

calendar,

convention, convention,

ql.DateGeneration.Forward, False

)

day\_counter\_curve = ql.Actual360()

day\_counter\_coupon = ql.Thirty360(ql.Thirty360.BondBasis)

# --- Safe forward rate helper ---

def get\_forward\_rate(start, end):

# Ensure we're not asking for forward rates before the curve starts

safe\_start = max(start, eval\_date)

if safe\_start >= end:

return 0.0

return log\_cubic\_curve.forwardRate(safe\_start, end, day\_counter\_curve, ql.Simple).rate()

# --- Cash flow calculation ---

cf\_rows = []

for i in range(len(schedule) - 1):

start = schedule[i]

end = schedule[i + 1]

if end <= eval\_date:

continue

# Use reset date if we're in the current coupon period

if start < eval\_date:

reset\_date = calendar.advance(start, -settlement\_lag, ql.Days)

start\_for\_fwd = reset\_date if eval\_date >= reset\_date else eval\_date

else:

start\_for\_fwd = start

fwd\_rate = get\_forward\_rate(start\_for\_fwd, end)

effective\_rate = min(max(fwd\_rate, floor), cap)

yf = day\_counter\_coupon.yearFraction(start, end)

coupon = notional \* effective\_rate \* yf

df = log\_cubic\_curve.discount(end)

pv = coupon \* df

cf\_rows.append({

"Payment Date": end,

"Forward Rate (%)": round(fwd\_rate \* 100, 4),

"Effective Rate (%)": round(effective\_rate \* 100, 4),

"Coupon Amount": round(coupon, 4),

"Discount Factor": round(df, 6),

"Present Value": round(pv, 4)

})

# --- Redemption at Maturity ---

df\_redemption = log\_cubic\_curve.discount(maturity\_date)

pv\_redemption = notional \* df\_redemption

cf\_rows.append({

"Payment Date": maturity\_date,

"Forward Rate (%)": None,

"Effective Rate (%)": None,

"Coupon Amount": notional,

"Discount Factor": round(df\_redemption, 6),

"Present Value": round(pv\_redemption, 4)

})

coupon\_table = pd.DataFrame(cf\_rows)

# --- Accrued Interest ---

accrued = 0.0

for i in range(len(schedule) - 1):

start = schedule[i]

end = schedule[i + 1]

if start < eval\_date <= end:

reset\_date = calendar.advance(start, -settlement\_lag, ql.Days)

start\_for\_fwd = reset\_date if eval\_date >= reset\_date else eval\_date

fwd\_rate = get\_forward\_rate(start\_for\_fwd, end)

effective\_rate = min(max(fwd\_rate, floor), cap)

yf\_accrued = day\_counter\_coupon.yearFraction(start, eval\_date)

accrued = notional \* effective\_rate \* yf\_accrued

break

# --- Prices ---

model\_gross\_price = coupon\_table["Present Value"].sum()

model\_clean\_price = model\_gross\_price - accrued

def compute\_bumped\_clean\_price(bump):

"""

Compute the model clean price using a bumped yield curve. The bump is added to all zero rates of the original curve.

Returns the bumped clean price.

"""

# Create a bumped yield curve from the original log\_cubic\_curve

try:

day\_counter = log\_cubic\_curve.dayCounter()

except AttributeError:

day\_counter = ql.Actual360()

dates = log\_cubic\_curve.dates()

bumped\_rates = [log\_cubic\_curve.zeroRate(date, day\_counter\_curve, ql.Continuous).rate() + bump for date in dates]

bumped\_curve = ql.ZeroCurve(dates, bumped\_rates, day\_counter)

# Define a local function to compute the forward rate using the bumped curve

def get\_forward\_rate\_bumped(start, end):

safe\_start = max(start, eval\_date)

if safe\_start >= end:

return 0.0

return bumped\_curve.forwardRate(safe\_start, end, day\_counter\_curve, ql.Simple).rate()

# Recompute the gross price using the bumped curve

bumped\_gross\_price = 0.0

for i in range(len(schedule) - 1):

start = schedule[i]

end = schedule[i + 1]

if end <= eval\_date:

continue

if start < eval\_date:

reset\_date = calendar.advance(start, -settlement\_lag, ql.Days)

start\_for\_fwd = reset\_date if eval\_date >= reset\_date else eval\_date

else:

start\_for\_fwd = start

fwd\_rate = get\_forward\_rate\_bumped(start\_for\_fwd, end)

effective\_rate = min(max(fwd\_rate, floor), cap)

yf = day\_counter\_coupon.yearFraction(start, end)

coupon = notional \* effective\_rate \* yf

df = bumped\_curve.discount(end)

pv = coupon \* df

bumped\_gross\_price += pv

# Redemption at maturity

df\_redemption = bumped\_curve.discount(maturity\_date)

pv\_redemption = notional \* df\_redemption

bumped\_gross\_price += pv\_redemption

# Recalculate accrued interest with the bumped curve

bumped\_accrued = 0.0

for i in range(len(schedule) - 1):

start = schedule[i]

end = schedule[i + 1]

if start < eval\_date <= end:

reset\_date = calendar.advance(start, -settlement\_lag, ql.Days)

start\_for\_fwd = reset\_date if eval\_date >= reset\_date else eval\_date

fwd\_rate = get\_forward\_rate\_bumped(start\_for\_fwd, end)

effective\_rate = min(max(fwd\_rate, floor), cap)

yf\_accrued = day\_counter\_coupon.yearFraction(start, eval\_date)

bumped\_accrued = notional \* effective\_rate \* yf\_accrued

break

bumped\_clean\_price = bumped\_gross\_price - bumped\_accrued

return bumped\_clean\_price

def main():

# --- Output ---

print("\nQ8: Forward Rate-Based Coupon Table (Including Redemption)")

print(coupon\_table)

print("\nBond Pricing Summary:")

print(f"Gross Price (Dirty): {round(model\_gross\_price, 4)}")

print(f"Accrued Interest: {round(accrued, 4)}")

print(f"Clean Price: {round(model\_clean\_price, 4)}")

# --- Bumped Clean Price Example ---

bump = 0.0001 # 1 basis point bump

bumped\_price = compute\_bumped\_clean\_price(bump)

print(f"\nBumped Clean Price (bump = {bump\*10000:.2f} bps): {round(bumped\_price, 4)}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

# %%

# Q9

# TODO: The forward-implied coupon rate,

# How much credit risk reduces the expected rate,

# And the survival-adjusted perspective coupon rate.

# %%

import matplotlib.pyplot as plt

import pandas as pd

from q1 import bond\_characteristics

from q4 import df\_best, df\_worst

from q6 import df\_variable\_exposure

# Prepare coupon-only data (exclude redemption)

def filter\_coupon\_only(df):

return df[df["Coupon Amount"] < bond\_characteristics["Nominal Value"]].copy()

df\_best\_coupons = filter\_coupon\_only(df\_best)

df\_worst\_coupons = filter\_coupon\_only(df\_worst)

df\_realistic\_coupons = df\_variable\_exposure[df\_variable\_exposure["Coupon Amount"] < bond\_characteristics["Nominal Value"]].copy()

# Ensure Payment Date is datetime for sorting and plotting

df\_best\_coupons["Payment Date"] = df\_best\_coupons["End Date"].apply(lambda d: d.to\_date())

df\_worst\_coupons["Payment Date"] = df\_worst\_coupons["End Date"].apply(lambda d: d.to\_date())

df\_realistic\_coupons["Payment Date"] = df\_realistic\_coupons["Payment Date"].apply(lambda d: d.to\_date())

# Merge into single DataFrame

df\_q9 = pd.DataFrame({

"Payment Date": df\_best\_coupons["Payment Date"],

"Cap (Best Case)": df\_best\_coupons["Coupon Amount"].values,

"Floor (Worst Case)": df\_worst\_coupons["Coupon Amount"].values,

"Forward (Realistic)": df\_realistic\_coupons["Coupon Amount"].values

})

# Plot grouped bar chart

plt.figure(figsize=(12, 6))

bar\_width = 0.25

x = range(len(df\_q9))

plt.bar([p - bar\_width for p in x], df\_q9["Cap (Best Case)"], width=bar\_width, label="Cap (Best Case)")

plt.bar(x, df\_q9["Forward (Realistic)"], width=bar\_width, label="Forward (Realistic)")

plt.bar([p + bar\_width for p in x], df\_q9["Floor (Worst Case)"], width=bar\_width, label="Floor (Worst Case)")

plt.xticks(ticks=x, labels=[d.strftime("%b %Y") for d in df\_q9["Payment Date"]], rotation=45)

plt.ylabel("Coupon Amount (€)")

plt.title("Q9: Expected Coupon Cash Flows (Cap vs Floor vs Forward)")

plt.legend()

plt.grid(axis='y')

plt.tight\_layout()

plt.show()

# %%

# Q10

#%%

import numpy as np

from scipy.optimize import brentq

from q1 import bond\_characteristics

from q6 import compute\_cva, df\_variable\_exposure

from q8 import model\_clean\_price

# ----------------------------

# 1. Setup known values

# ----------------------------

market\_price\_per\_100= 98.43

notional=bond\_characteristics["Nominal Value"]

market\_clean\_price = market\_price\_per\_100 \* notional/100

recovery\_rate = 0.40 # As per coursework

# ----------------------------

# 2. Define root-finding target

# ----------------------------

def price\_difference(cds\_spread):

"""

Target: find CDS spread where model\_clean\_price - CVA(cds) ≈ market\_clean\_price

"""

cva, \_ = compute\_cva(cds\_spread, recovery\_rate, df\_variable\_exposure)

return model\_clean\_price - cva - market\_clean\_price

# ----------------------------

# 3. Run solver

# ----------------------------

cds\_solution = brentq(price\_difference, 0.001, 0.10) # Search between 10 and 1000 bps

implied\_cva, \_ = compute\_cva(cds\_solution, recovery\_rate,df\_variable\_exposure)

credit\_adjusted\_price = model\_clean\_price - implied\_cva

# ----------------------------

# 4. Output result

# ----------------------------

print("\n=== Q10: Market-Implied CDS Spread ===")

print(f"Implied CDS Spread: {cds\_solution \* 10000:.2f} bps")

print(f"Model Clean Price: {model\_clean\_price:.4f}")

print(f"Market Clean Price: {market\_clean\_price:.4f}")

print(f"Implied CVA: {implied\_cva:.4f}")

print(f"Credit-Adjusted Price: {credit\_adjusted\_price:.4f}")

print(f"Pricing Error: {credit\_adjusted\_price - market\_clean\_price:.6f}")

# %%

# Q11

#%%

import QuantLib as ql

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from q1 import bond\_characteristics

from q3 import build\_curves

from q3 import date\_strings

# --- Shift Helpers ---

def apply\_parallel\_shift(rates, shift\_bps=10):

return [r + shift\_bps / 10000 for r in rates]

def apply\_slope\_shift(rates, shift\_bps=10):

n = len(rates)

return [r + (shift\_bps / 10000 \* (2 \* i / n - 1)) for i, r in enumerate(rates)]

def apply\_curvature\_shift(rates, bump=0.001):

n = len(rates)

mid, width = n // 2, n // 4

return [r + bump \* np.exp(-((i - mid)\*\*2) / (2 \* width\*\*2)) for i, r in enumerate(rates)]

def make\_zero\_curve(dates, shifted\_rates, calendar, day\_counter):

curve = ql.ZeroCurve(dates, shifted\_rates, day\_counter, calendar)

curve.enableExtrapolation()

return curve

# --- Setup ---

spot\_date = ql.Date(26, 11, 2024)

ql.Settings.instance().evaluationDate = spot\_date

calendar = ql.TARGET()

day\_counter\_curve = ql.Actual360()

day\_counter\_coupon = ql.Thirty360(ql.Thirty360.BondBasis)

# Load base curve

base\_curve = build\_curves()["Log-Cubic"]

# dates = base\_curve.dates()

eval\_date = ql.Date(18, 11, 2024)

end\_date = calendar.advance(eval\_date, ql.Period(60, ql.Years))

n\_points = 100 # number of points on the curve

dates = [eval\_date + ql.Period(int(i \* (end\_date.serialNumber() - eval\_date.serialNumber()) / n\_points), ql.Days)

for i in range(n\_points + 1)] # evaluation dates

date\_labels = [d.ISO() for d in dates]

base\_rates = [base\_curve.zeroRate(d, day\_counter\_curve, ql.Continuous).rate() for d in dates]

base\_dfs = [base\_curve.discount(d) for d in dates]

plt.figure(figsize=(12, 6),dpi=250)

plt.plot(date\_labels, base\_rates)

plt.show()

# Build shifted curves

shifted\_curves = {

"Base": base\_curve,

"Parallel +10bps": make\_zero\_curve(dates, apply\_parallel\_shift(base\_rates, 10), calendar, day\_counter\_curve),

"Parallel -10bps": make\_zero\_curve(dates, apply\_parallel\_shift(base\_rates, -10), calendar, day\_counter\_curve),

"Slope +10bps": make\_zero\_curve(dates, apply\_slope\_shift(base\_rates, 10), calendar, day\_counter\_curve),

"Slope -10bps": make\_zero\_curve(dates, apply\_slope\_shift(base\_rates, -10), calendar, day\_counter\_curve),

"Curvature +10bps": make\_zero\_curve(dates, apply\_curvature\_shift(base\_rates, 0.001), calendar, day\_counter\_curve),

"Curvature -10bps": make\_zero\_curve(dates, apply\_curvature\_shift(base\_rates, -0.001), calendar, day\_counter\_curve),

}

# --- Bond Characteristics ---

frequency = ql.Quarterly

notional = bond\_characteristics["Nominal Value"]

cap = bond\_characteristics["Cap"]

floor = bond\_characteristics["Floor"]

settlement\_lag = bond\_characteristics["Settlement Lag"]

issue\_date = ql.Date(bond\_characteristics["Issue Date"].day,

bond\_characteristics["Issue Date"].month,

bond\_characteristics["Issue Date"].year)

maturity\_date = ql.Date(29, 7, 2027)

schedule = ql.Schedule(

issue\_date, maturity\_date,

ql.Period(frequency),

calendar,

ql.ModifiedFollowing, ql.ModifiedFollowing,

ql.DateGeneration.Forward, False

)

# --- Pricing Function ---

def get\_forward\_rate(start, end, curve):

safe\_start = max(start, spot\_date)

return curve.forwardRate(safe\_start, end, day\_counter\_curve, ql.Simple).rate() if safe\_start < end else 0.0

def compute\_gross\_price(curve):

cashflows = []

for i in range(len(schedule) - 1):

start = schedule[i]

end = schedule[i + 1]

if end <= spot\_date:

continue

if start < spot\_date:

reset = calendar.advance(start, -settlement\_lag, ql.Days)

start\_for\_fwd = reset if spot\_date >= reset else spot\_date

else:

start\_for\_fwd = start

fwd = get\_forward\_rate(start\_for\_fwd, end, curve)

rate = min(max(fwd, floor), cap)

yf = day\_counter\_coupon.yearFraction(start, end)

coupon = notional \* rate \* yf

df = curve.discount(end)

cashflows.append(coupon \* df)

# redemption

cashflows.append(notional \* curve.discount(maturity\_date))

return sum(cashflows)

# --- Accrued Interest Calculation ---

def compute\_accrued(curve):

for i in range(len(schedule) - 1):

start = schedule[i]

end = schedule[i + 1]

if start < spot\_date <= end:

reset = calendar.advance(start, -settlement\_lag, ql.Days)

start\_for\_fwd = reset if spot\_date >= reset else spot\_date

fwd = get\_forward\_rate(start\_for\_fwd, end, curve)

rate = min(max(fwd, floor), cap)

yf = day\_counter\_coupon.yearFraction(start, spot\_date)

return notional \* rate \* yf

return 0.0

# --- Pricing for Each Scenario ---

price\_summary = {}

for label, curve in shifted\_curves.items():

gross = compute\_gross\_price(curve)

accrued = compute\_accrued(curve)

clean = gross - accrued

price\_summary[label] = {

"Gross Price": round(gross, 4),

"Accrued Interest": round(accrued, 4),

"Clean Price": round(clean, 4)

}

df\_prices = pd.DataFrame(price\_summary).T

print("\n📊 Bond Price Sensitivity Summary:")

print(df\_prices)

plot\_groups = {

"Parallel Shift": ["Base", "Parallel +10bps", "Parallel -10bps"],

"Slope Shift": ["Base", "Slope +10bps", "Slope -10bps"],

"Curvature Shift": ["Base", "Curvature +10bps", "Curvature -10bps"]

}

for title, group\_labels in plot\_groups.items():

plt.figure(figsize=(10, 5), dpi=150)

for name in group\_labels:

curve = shifted\_curves[name]

rates = [curve.zeroRate(d, day\_counter\_curve, ql.Continuous).rate() \* 100 for d in dates]

linestyle = '-' if name == "Base" else '--' if "+10bps" in name else '-.'

plt.plot(date\_labels, rates, label=name, linestyle=linestyle)

plt.title(f"Spot Rate Curves – {title}")

plt.ylabel("Zero Rate (%)")

plt.xlabel("Maturity Date")

plt.xticks(date\_strings[::5],rotation=45)

plt.grid(axis='y')

plt.legend()

plt.tight\_layout()

plt.show()

for title, group\_labels in plot\_groups.items():

plt.figure(figsize=(10, 5), dpi=150)

for name in group\_labels:

curve = shifted\_curves[name]

rates = [curve.zeroRate(d, day\_counter\_curve, ql.Continuous).rate() \* 100 for d in dates]

linestyle = '-' if name == "Base" else '--' if "+10bps" in name else '-.'

plt.plot(date\_labels, rates, label=name, linestyle=linestyle)

plt.title(f"Spot Rate Curves – {title}")

plt.ylabel("Zero Rate (%)")

plt.xlabel("Maturity Date")

plt.xticks(date\_strings[::5],rotation=45)

plt.grid(axis='y')

plt.legend()

plt.tight\_layout()

plt.show()

df\_prices.to\_csv("bond\_sensitivity\_prices.csv") # Output the results for Q16-18

#%%

# Q12/13

import QuantLib as ql

from q1 import bond\_characteristics

from q3 import build\_curves

# =============================================================================

# 1. Set Evaluation Date Based on Trade Date

# =============================================================================

trade\_date = ql.Date(bond\_characteristics['Trade Date'].day,

bond\_characteristics['Trade Date'].month,

bond\_characteristics['Trade Date'].year)

ql.Settings.instance().evaluationDate = trade\_date

# =============================================================================

# 2. Construct Multiple Yield Curves Using Different Interpolation Methods

# =============================================================================

curves = build\_curves()

# =============================================================================

# 3. Bond Setup: Construct a Floating-Rate Bond with Cap and Floor

# =============================================================================

# Extract bond dates from bond\_characteristics

issue\_date = ql.Date(bond\_characteristics['Issue Date'].day,

bond\_characteristics['Issue Date'].month,

bond\_characteristics['Issue Date'].year)

maturity\_date\_bond = ql.Date(bond\_characteristics['Maturity Date'].day,

bond\_characteristics['Maturity Date'].month,

bond\_characteristics['Maturity Date'].year)

settlement\_days = bond\_characteristics['Settlement Lag']

# For a floating rate bond, we need a schedule.

# Since the bond pays 4 times per year, we use Quarterly frequency.

bond\_schedule = ql.Schedule(

issue\_date,

maturity\_date\_bond,

ql.Period(ql.Quarterly),

ql.TARGET(),

ql.ModifiedFollowing,

ql.ModifiedFollowing,

ql.DateGeneration.Forward,

False

)

# Parameters for the FloatingRateBond:

# - settlementDays: from bond\_characteristics

# - faceAmount: Nominal Value

# - schedule: bond\_schedule

# - index: euribor3m

# - paymentDayCounter: Using Thirty360 based on the bond's conventions (Days in Month=30, Days in Year=360)

# - paymentConvention: Modified Following

# - fixingDays: We'll assume 2 (typical for Euribor)

# - gearings: [1.0] (no gearing)

# - spreads: [0.0] (no spread)

# - caps: [bond\_characteristics['Cap']]

# - floors: [bond\_characteristics['Floor']]

# - inArrears: False

# - redemption: 100 (since the nominal is 1000 and redemption "At Par", we use 100% of par)

# - issueDate: issue\_date

# =============================================================================

# 5. Functions to Compute DV01 for a Swap and a Bond (using finite differences)

# =============================================================================

def compute\_dv01(swap, original\_curve, bump=1e-4):

"""Compute DV01 for a swap by bumping the yield curve by 1 basis point."""

original\_npv = swap.NPV()

bumped\_rates = [original\_curve.zeroRate(date, ql.Actual365Fixed(), ql.Continuous).rate() + bump

for date in original\_curve.dates()]

bumped\_curve = ql.ZeroCurve(original\_curve.dates(), bumped\_rates, ql.Actual365Fixed())

bumped\_handle = ql.YieldTermStructureHandle(bumped\_curve)

swap.setPricingEngine(ql.DiscountingSwapEngine(bumped\_handle))

bumped\_npv = swap.NPV()

dv01 = (bumped\_npv - original\_npv) / bump

return dv01

def compute\_bond\_dv01(bond, original\_curve, bump=1e-4):

"""Compute DV01 for a bond by bumping the yield curve by 1 basis point."""

yc\_handle = ql.YieldTermStructureHandle(original\_curve)

bond.setPricingEngine(ql.DiscountingBondEngine(yc\_handle))

original\_price = bond.cleanPrice()

bumped\_rates = [original\_curve.zeroRate(date, ql.Actual365Fixed(), ql.Continuous).rate() + bump

for date in original\_curve.dates()]

bumped\_curve = ql.ZeroCurve(original\_curve.dates(), bumped\_rates, ql.Actual365Fixed())

bumped\_handle = ql.YieldTermStructureHandle(bumped\_curve)

bond.setPricingEngine(ql.DiscountingBondEngine(bumped\_handle))

bumped\_price = bond.cleanPrice()

dv01 = (bumped\_price - original\_price) / bump

return dv01

# =============================================================================

# Swap Setup: Define a plain vanilla swap to be priced using each yield curve.

# =============================================================================

nominal = bond\_characteristics['Nominal Value'] # Notional amount

fixed\_rate = 0.02202 # Fixed rate payment

# Define swap start and maturity dates (5-year swap)

start\_date = issue\_date

maturity\_date = maturity\_date\_bond

# Build schedules for the fixed and floating legs.

fixed\_schedule = ql.Schedule(

start\_date, maturity\_date, ql.Period(ql.Annual),

ql.TARGET(), ql.ModifiedFollowing, ql.ModifiedFollowing,

ql.DateGeneration.Forward, False

)

floating\_schedule = ql.Schedule(

start\_date, maturity\_date, ql.Period(ql.Semiannual),

ql.TARGET(), ql.ModifiedFollowing, ql.ModifiedFollowing,

ql.DateGeneration.Forward, False

)

# =============================================================================

# 6. Loop Over Each Yield Curve: Price the Bond & Swap, Compute DV01s, and Determine Hedge Ratio

# =============================================================================

print("Comparison of Bond and Swap DV01s Using Different Yield Curves:")

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

print("{:<25} {:>12} {:>15} {:>15} {:>20}".format("Interpolation Method", "Bond Price", "Bond DV01", "Swap DV01", "Hedge Notional"))

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

for curve\_name, forward\_curve in curves.items():

# Create a handle for the forward curve

forward\_handle = ql.YieldTermStructureHandle(forward\_curve)

# -------------------------------------------------------------------------

# 1. Build a discount curve from the forward curve

# -------------------------------------------------------------------------

# Retrieve the dates from the forward curve (these should cover the maturities of interest)

dates = forward\_curve.dates()

# Get the day counter from the forward curve; if not available, you can default to Actual365Fixed

try:

day\_counter = forward\_curve.dayCounter()

except AttributeError:

day\_counter = ql.Actual365Fixed()

# Compute discount factors at each date from the forward curve

discount\_factors = [forward\_curve.discount(date) for date in dates]

# Build a discount curve using these dates and discount factors

discount\_curve = ql.DiscountCurve(dates, discount\_factors, day\_counter)

discount\_handle = ql.YieldTermStructureHandle(discount\_curve)

# -------------------------------------------------------------------------

# 2. Build the floating index using the forward curve (for projecting coupons)

# -------------------------------------------------------------------------

euribor3m = ql.Euribor3M(forward\_handle)

# Re-create the bond for this curve iteration with the new index.

# (Reinstantiating the bond is safest because the index is set at construction.)

bond = ql.FloatingRateBond(

settlementDays=settlement\_days,

faceAmount=bond\_characteristics['Nominal Value'],

schedule=bond\_schedule,

index=euribor3m, # Use forward curve for forecasting

paymentDayCounter=ql.Thirty360(ql.Thirty360.BondBasis),

paymentConvention=ql.ModifiedFollowing,

fixingDays=2,

gearings=[1.0],

spreads=[0.0],

caps=[bond\_characteristics['Cap']],

floors=[bond\_characteristics['Floor']],

inArrears=False,

redemption=100.0,

issueDate=issue\_date

)

pricer = ql.BlackIborCouponPricer()

ql.setCouponPricer(bond.cashflows(), pricer)

# Use the discount curve for pricing

bond.setPricingEngine(ql.DiscountingBondEngine(discount\_handle))

bond\_price = bond.cleanPrice()

# Compute bond DV01 using the discount curve

bond\_dv01 = compute\_bond\_dv01(bond, discount\_curve)

# -------------------------------------------------------------------------

# 3. Construct the Swap using the forward curve for the floating leg and the discount curve for discounting

# -------------------------------------------------------------------------

# Determine swap type based on bond DV01

if bond\_dv01 < 0:

swap\_type = ql.VanillaSwap.Payer

else:

swap\_type = ql.VanillaSwap.Receiver

# Rebuild the floating leg index (using the forward curve)

euribor\_index = ql.Euribor6M(forward\_handle)

swap = ql.VanillaSwap(

swap\_type,

bond\_characteristics['Nominal Value'], # Notional amount

fixed\_schedule, fixed\_rate, ql.Thirty360(ql.Thirty360.BondBasis),

floating\_schedule, euribor\_index, 0.0, ql.Actual360()

)

# Use the discount curve for discounting the swap cash flows

swap\_engine = ql.DiscountingSwapEngine(discount\_handle)

swap.setPricingEngine(swap\_engine)

swap\_npv = swap.NPV()

swap\_dv01 = compute\_dv01(swap, discount\_curve)

# Calculate hedge notional as the ratio of DV01s (absolute values)

hedge\_notional = abs(bond\_dv01) / abs(swap\_dv01) if swap\_dv01 != 0 else float('inf')

# Print the results for this curve

print("{:<25} {:>12.2f} {:>15.2f} {:>15.2f} {:>20.0f}".format(curve\_name, bond\_price, bond\_dv01, swap\_dv01, hedge\_notional))

# q16,17,18

#%%

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from scipy.stats import norm

# CDS sensitivity from q6

q6data = [

{"CDS Spread": 0.002, "Adjusted NPV": 991.779335},

{"CDS Spread": 0.003, "Adjusted NPV": 987.382883},

{"CDS Spread": 0.004, "Adjusted NPV": 982.998305},

{"CDS Spread": 0.005, "Adjusted NPV": 978.625570},

{"CDS Spread": 0.006, "Adjusted NPV": 974.264644},

{"CDS Spread": 0.007, "Adjusted NPV": 969.915497}

]

CDSsens = pd.DataFrame(q6data)

# Calculate price sensitivity (first-order forward difference)

CDSsens["Sensitivity"] = CDSsens["Adjusted NPV"].diff() / CDSsens["CDS Spread"].diff()

# Sensitivity per basis point

CDSsens["Sensitivity (per bps)"] = CDSsens["Sensitivity"] / 10000

beta\_cds = CDSsens["Sensitivity (per bps)"].mean()

# import q11 output

shiftsens = pd.read\_csv("bond\_sensitivity\_prices.csv", index\_col=0)

base\_price = shiftsens.loc["Base", "Clean Price"]

parallel\_up = shiftsens.loc["Parallel +10bps", "Clean Price"]

parallel\_dn = shiftsens.loc["Parallel -10bps", "Clean Price"]

# DV01 (symmetric approx): average price change per 1bp

dv01 = (parallel\_dn - parallel\_up) / 2 / 10

# Slope and Curvature change per 1 bp

slope\_sens = (shiftsens.loc["Slope -10bps", "Clean Price"] - shiftsens.loc["Slope +10bps", "Clean Price"]) / 2 / 10

curv\_sens = (shiftsens.loc["Curvature -10bps", "Clean Price"] - shiftsens.loc["Curvature +10bps", "Clean Price"]) / 2 / 10

print(f"Parallel DV01: {dv01:.6f}")

print(f"Slope Sensitivity: {slope\_sens:.6f} per 1bp")

print(f"Curvature Sensitivity: {curv\_sens:.6f} per 1bp")

print(f"CDS Sensitivity {beta\_cds:.6f} per 1bp")

#%%

# Given variances

var\_l = 0.022

var\_s = 0.003

var\_c = 0.001

var\_cds = 0.002

# Standard deviations

std\_l = np.sqrt(var\_l)

std\_s = np.sqrt(var\_s)

std\_c = np.sqrt(var\_c)

std\_cds = np.sqrt(var\_cds)

# Betas (from above)

beta\_l = dv01

beta\_s = slope\_sens

beta\_c = curv\_sens

beta\_cds = beta\_cds

# Monte Carlo simulation

n\_simulations = 1000000

np.random.seed(42)

delta\_l = np.random.normal(0, std\_l, n\_simulations)

delta\_s = np.random.normal(0, std\_s, n\_simulations)

delta\_c = np.random.normal(0, std\_c, n\_simulations)

delta\_cds = np.random.normal(0, std\_cds, n\_simulations)

# Change in bond price

delta\_GP = beta\_l \* delta\_l + beta\_s \* delta\_s + beta\_c \* delta\_c + beta\_cds \* delta\_cds

PnL = -delta\_GP # Loss

# Monte Carlo VaR & ES (99%)

VaR\_mc = -np.percentile(PnL, 1)

ES\_mc = PnL[PnL >= VaR\_mc].mean()

# === QUESTION 17 ===

# Analytical (exact) VaR & ES using normal distribution

# Total variance of delta\_GP

total\_var = (beta\_l\*\*2 \* var\_l +

beta\_s\*\*2 \* var\_s +

beta\_c\*\*2 \* var\_c +

beta\_cds\*\*2 \* var\_cds)

std\_total = np.sqrt(total\_var)

# Normal 99% quantile

z\_99 = norm.ppf(0.01)

VaR\_exact = -z\_99 \* std\_total

ES\_exact = std\_total \* norm.pdf(z\_99)

# === QUESTION 18 ===

# Marginal VaR is proportional to beta\_i \* std\_i / std\_total

# Component VaR = Marginal VaR \* Beta\_i

marginal\_VaRs = {

'l': beta\_l \* np.sqrt(var\_l) / std\_total,

's': beta\_s \* np.sqrt(var\_s) / std\_total,

'c': beta\_c \* np.sqrt(var\_c) / std\_total,

'cds': beta\_cds \* np.sqrt(var\_cds) / std\_total

}

component\_VaRs = {k: v \* std\_total \* z\_99 for k, v in marginal\_VaRs.items()}

print(f"Sum of component VaRs: {sum(component\_VaRs.values()):.6f}")

print(f"Exact Total VaR: {VaR\_exact:.6f}")

# === OUTPUT ===

print(f"\nMonte Carlo 99% VaR: {VaR\_mc:.4f}")

print(f"Exact (Analytical) 99% VaR: {VaR\_exact:.4f}")

print(f"Monte Carlo 99% ES: {ES\_mc:.4f}")

print(f"Exact (Analytical) 99% ES: {ES\_exact:.4f}")

print("\nComponent VaR contributions:")

for factor in component\_VaRs:

print(f" {factor}: {component\_VaRs[factor]:.4f}")

# Plot

plt.hist(PnL, bins=100, alpha=0.7, edgecolor='k')

plt.axvline(VaR\_mc, color='red', linestyle='--', linewidth=2, label=f'MC VaR = {VaR\_mc:.4f}')

plt.axvline(VaR\_exact, color='blue', linestyle='--', linewidth=2, label=f'Exact VaR = {VaR\_exact:.4f}')

plt.title('Profit and Loss Distribution')

plt.xlabel('Loss')

plt.ylabel('Frequency')

plt.legend()

plt.grid()

plt.show()

# %%