~\Documents\Hassan Nipita Medina\Codigo Terminado 2.1.py

```python
#Portfolio Management - Coursework 2: Asset simulation and Liability Driven Portfolios
#1.CPPI

# Import libraries
# pandas
import pandas as pd

#numpy
import numpy as np

# matplotlib.pyplot
import matplotlib.pyplot as plt

#Load data industry returns
df_returns= pd.read_csv('C:/Users/52241/Downloads/EXCEL PYTHON/index30_returns.csv', header=0, index_col=0,parse_dates=True)/100 #convert to percentages
df_returns.index= pd.to_datetime(df_returns.index,format="%Y%m").to_period('M') #to time series in monthly periods
df_returns.columns=df_returns.columns.str.strip() #remove blank spaces in names

# E1. Load the total market index size and industry firms. Give time series format in monthly periods. Note: Do not transform to percentages.
#Load the total market index
#get earnings sizes from the companies for each industry
# Load the total market index size
df_index_size = pd.read_csv('C:/Users/52241/Downloads/EXCEL PYTHON/index30_size.csv',
                            header=0, index_col=0, parse_dates=True)

# Convert the index to a time series in monthly periods
df_index_size.index = pd.to_datetime(df_index_size.index, format="%Y%m").to_period('M')

# Remove any whitespace in column names
df_index_size.columns = df_index_size.columns.str.strip()

#get number of firms in the index of each industry
# Load the number of firms in the index for each industry
df_index_firms = pd.read_csv('C:/Users/52241/Downloads/EXCEL PYTHON/index30_nfirms.csv',
                             header=0, index_col=0, parse_dates=True)

# Convert the index to a time series in monthly periods
df_index_firms.index = pd.to_datetime(df_index_firms.index, format="%Y%m").to_period('M')

# Remove any whitespace in column names
df_index_firms.columns = df_index_firms.columns.str.strip()

#E2. Calculate Market Capitalization and Weighted Market Return
#market capitalization
# Calculate industry market capitalization
ind_mktcap = df_index_firms * df_index_size

# Calculate total market capitalization (sum across industries for each period)
total_mktcap = ind_mktcap.sum(axis="columns")
```

```
50
51  #weighted capitalization
52  ind_capweight = ind_mktcap.divide(total_mktcap, axis="rows")
53
54  #returns from market indez and industry returns
55  total_market_return = (ind_capweight * df_returns).sum(axis="columns")
56
57  #For the following, consider just the returns for Steel, Finance, and Beer industries
    from 2000 onwards. These industries will form our risky assets.
58  #Select specific industries as risky assets and create a safe asset with fixed monthly
    returns.
59  #Consider returns from 2000 for steel, fin, and beer
60  # Select Steel, Finance, and Beer industries from 2000 onwards
61  df_risky = df_returns.loc["2000":, ["Steel", "Fin", "Beer"]]
62
63  #We also need safe assets. For this, we create a dataframe with the same number of
    returns and assume a monthly fixed safe return of 0.02 annualized. This is, we have a
    risk-free asset that pays 0.02 per year.
64  df_safe = pd.DataFrame().reindex_like(df_risky)
65  df_safe[:] = 0.02 / 12  # Monthly fixed safe return (2% annualized)
66
67  #As we have risky and risk-free assets,
68  # let's implement CPPI and explore how it works.
69  # First, assume we invest 1000 USD, that the floor
70  # (the minimum value below which the portfolio should not
71  # fall) is 80% of the initial value, and that the multiplier
72  # (the level of exposure to risky assets based on the cushion,
73  #i.e. the aggressiveness of risky asset allocations) is 3.
74
75  #Set up initial account parameters:
76  start = 1000  # Starting account value
77  floor = 0.70  # Floor as a percentage of starting account, this is just a percentage
78  account_value = start #starting value of the investment
79  floor_value = start * floor #floor applied to the account value
80  m = 4  # CPPI multiplier
81
82  #CPPI works through out time, thus we need to define an entity to save the results of our
    simulated example. For this, we use dataframes of with the same number of trading periods
    than our risky assets, which are the number of steps for the simulation. As we are
    interested in tracking the evolution
83  #of portfolio values, risky and safe allocations, and total
84  #returns over time, we are going to define 3 dataframes.
85
86  # Prepare to track the evolution of account values
87  dates = df_risky.index
88
89  n_steps = len(dates)
90
91
92  # Prepare trackers for the the history of account values, cushion, and risky weights
93  account_history = pd.DataFrame().reindex_like(df_risky)
94  cushion_history = pd.DataFrame().reindex_like(df_risky)
95  risky_w_history = pd.DataFrame().reindex_like(df_risky)
96
```

```
#E3. Implement CCPI
#This loop performs the CPPI calculations by updating allocations based on the cushion. Use your defined strating parameters
for step in range(n_steps):
    # Calculate the current cushion
    cushion = (account_value - floor_value) / account_value

    # Calculate weights for allocation to risky assets based on the multiplier
    risky_w = m * cushion

    # Ensure allocation to risky does not exceed 100% and there is no short selling
    risky_w = np.minimum(risky_w, 1)  # Cap at 100%
    risky_w = np.maximum(risky_w, 0)  # No short selling (min 0%)

    # Compute weights for the safe asset
    safe_w = 1 - risky_w

    # Allocate the money in the account to risky and safe assets
    risky_alloc = account_value * risky_w
    safe_alloc = account_value * safe_w

    # Update account value based on risky and safe returns
    account_value = (
        risky_alloc * (1 + df_risky.iloc[step]) + safe_alloc * (1 + df_safe.iloc[step])
    )

    # Store history for visualization or tracking
    cushion_history.iloc[step] = cushion
    account_history.iloc[step] = account_value
    risky_w_history.iloc[step] = risky_w

#First values of our CCPI
account_history.head()

#Before seeing the effects of CPPI strategy, what would have happened if we had put all the money in the risky assets and not using the CPPI? Well, this is basically the cumulative returns of the risky assets:
#Plot the account history for one asset, comparing CPPI-managed wealth with a fully risky allocation strategy.
risky_wealth = start * (1 + df_risky).cumprod()
risky_wealth.plot()
plt.show()
#But, what is the investment allocation recommended using CPPI? Well, we can know this by plotting our simulated weights.
risky_w_history.plot()
plt.show()

#This is the evolution of the allocation to risky assets. Notice the increment in investment on beer. Let's compare then the CPPI vs Full Risky Allocation to beer:
# Plot CPPI-managed wealth vs. full-risky strategy for comparison
ax = account_history["Beer"].plot(figsize=(12, 6), title="CPPI vs Full Risky Allocation")
risky_wealth["Beer"].plot(ax=ax, style="k:", label="Full Risky Allocation (Beer)")
plt.axhline(y=floor_value, color='r', linestyle="--", label="Floor Value")  # Plot the floor line
```

```
plt.legend()
plt.show()

#E4. Compare CPPI vs Risky Allocation for Finance and Steel
#Finance

start = 1000  # Starting account value
floor = 0.70  # Floor as a percentage of starting account
account_value = start  # Starting value of the investment
floor_value = start * floor  # Floor applied to the account value
m = 4  # CPPI multiplier

# Prepare to track the evolution of account values
dates = df_risky["Fin"].index
n_steps = len(dates)

# Initialize DataFrames to track results
account_history_fin = pd.DataFrame(index=dates, columns=["Fin"])
cushion_history_fin = pd.DataFrame(index=dates, columns=["Fin"])
risky_w_history_fin = pd.DataFrame(index=dates, columns=["Fin"])

# Implement CPPI for Finance (Fin)
for step in range(n_steps):
    cushion = (account_value - floor_value) / account_value
    risky_w = np.clip(m * cushion, 0, 1)  # Risky weight between 0 and 1
    safe_w = 1 - risky_w

    # Allocate money to risky and safe assets
    risky_alloc = account_value * risky_w
    safe_alloc = account_value * safe_w

    # Update account value
    account_value = (
        risky_alloc * (1 + df_risky["Fin"].iloc[step]) +
        safe_alloc * (1 + df_safe["Fin"].iloc[step])
    )

    # Track history
    cushion_history_fin.iloc[step, 0] = cushion
    account_history_fin.iloc[step, 0] = account_value
    risky_w_history_fin.iloc[step, 0] = risky_w

# First values of CPPI for Finance
account_history_fin.head()

# Plot CPPI vs Full Risky Allocation for Finance
fig, ax = plt.subplots(figsize=(12, 6))
risky_wealth_fin = start * (1 + df_risky["Fin"]).cumprod()
account_history_fin["Fin"].plot(ax=ax, label="CPPI (Fin)")
risky_wealth_fin.plot(ax=ax, style="k--", label="Full Risky (Fin)")
plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")
plt.title("CPPI vs Full Risky Allocation for Finance (Fin)")
plt.legend()
plt.show()
```

```
198
199 # Plot Risky Asset Allocation Over Time for Finance
200 risky_w_history_fin.plot(figsize=(12, 6), title="Risky Asset Allocation Over Time
    (Finance)")
201 plt.show()
202 #Here the effect of CPPI is more cleared, as in 2009 the allocation was really good, we
    had no violation and we had protection when the market crashed, the defect was that when
    the market rose we didn't enjoy all the upside benefit.
203 #Steel
204
205 start = 1000  # Starting account value
206 floor = 0.70  # Floor as a percentage of starting account
207 account_value = start  # Starting value of the investment
208 floor_value = start * floor  # Floor applied to the account value
209 m = 4  # CPPI multiplier
210
211 # Prepare to track the evolution of account values
212 dates = df_risky["Steel"].index
213 n_steps = len(dates)
214
215 # Initialize DataFrames to track results
216 account_history_steel = pd.DataFrame(index=dates, columns=["Steel"])
217 cushion_history_steel = pd.DataFrame(index=dates, columns=["Steel"])
218 risky_w_history_steel = pd.DataFrame(index=dates, columns=["Steel"])
219
220 # Implement CPPI for Steel
221 for step in range(n_steps):
222     cushion = (account_value - floor_value) / account_value
223     risky_w = np.clip(m * cushion, 0, 1)  # Risky weight between 0 and 1
224     safe_w = 1 - risky_w
225
226     # Allocate money to risky and safe assets
227     risky_alloc = account_value * risky_w
228     safe_alloc = account_value * safe_w
229
230     # Update account value
231     account_value = (
232         risky_alloc * (1 + df_risky["Steel"].iloc[step]) +
233         safe_alloc * (1 + df_safe["Steel"].iloc[step])
234     )
235
236     # Track history
237     cushion_history_steel.iloc[step, 0] = cushion
238     account_history_steel.iloc[step, 0] = account_value
239     risky_w_history_steel.iloc[step, 0] = risky_w
240
241 # First values of CPPI for Steel
242 account_history_steel.head()
243
244 # Plot CPPI vs Full Risky Allocation for Steel
245 fig, ax = plt.subplots(figsize=(12, 6))
246 risky_wealth_steel = start * (1 + df_risky["Steel"]).cumprod()
247 account_history_steel["Steel"].plot(ax=ax, label="CPPI (Steel)")
248 risky_wealth_steel.plot(ax=ax, style="r--", label="Full Risky (Steel)")
```

```
plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")
plt.title("CPPI vs Full Risky Allocation for Steel")
plt.legend()
plt.show()

# Plot Risky Asset Allocation Over Time for Steel
risky_w_history_steel.plot(figsize=(12, 6), title="Risky Asset Allocation Over Time (Steel)")
plt.show()

#E5. Compute the summary statistics studied in CW1 and apply
# them to df_risky
from scipy.stats import skew, kurtosis

def summary_stats(r):
    """
    Calculate summary statistics for returns.
    Includes annualized return, volatility, skewness, kurtosis, VaR, CVaR, Sharpe ratio, and max drawdown.
    """
    # Calculate annualized return using compounded growth
    compounded_growth = (1 + r).prod()
    n_periods = r.shape[0]
    ann_r = (compounded_growth) ** (12 / n_periods) - 1  # Annualized return assuming 12 periods per year (monthly returns)

    # Annualized volatility (standard deviation)
    ann_vol = r.std() * np.sqrt(12)  # Multiply by sqrt(12) to annualize

    # Skewness and kurtosis (without bias correction)
    skewness = skew(r, bias=False)
    kurt = kurtosis(r, bias=False)

    # Value at Risk (5%) using Cornish-Fisher expansion
    z = 1.645  # Z-score for 5% VaR (assuming normal distribution for simplicity)
    cf_var5 = -(ann_r - 0.5 * ann_vol**2) + z * ann_vol
    cf_var5 += ((z*2 - 1) * skewness / 6) + (((z**3 - 3*z) * (kurt - 3)) / 24) - (((2*z**3 - 5*z) * (skewness*2)) / 36)

    # Historical VaR (5%)
    hist_var5 = -r.quantile(0.05)

    # Conditional Value at Risk (CVaR) - Average of returns worse than the VaR
    cvar5 = -r[r <= hist_var5].mean()

    # Sharpe Ratio assuming a risk-free rate of 0 for simplicity
    sharpe_ratio = ann_r / ann_vol if ann_vol != 0 else np.nan  # Avoid division by zero

    # Maximum drawdown
    cumulative = (1 + r).cumprod()  # Cumulative returns
    peak = cumulative.cummax()  # Track the peak of the cumulative returns
    max_dd = ((cumulative / peak) - 1).min()  # Max drawdown as the largest loss from peak

```

```
    # Compile results into a DataFrame
    return pd.Series({
        "Annualized Return": ann_r,
        "Annualized Vol": ann_vol,
        "Skewness": skewness,
        "Kurtosis": kurt,
        "Cornish-Fisher VaR(5%)": cf_var5,
        "Historic VaR(5%)": hist_var5,
        "CVar(5%)": cvar5,
        "Sharpe Ratio": sharpe_ratio,
        "Max Drawdown": max_dd
    })

# Apply the summary_stats function to each column of df_risky
summary_stats_df = df_risky.apply(summary_stats)

# Display the results
print(summary_stats_df)


# E6. Implement CPPI as a function

def run_cppi1(risky_r, safe_r=None, m=3, start=1000, floor=0.8, riskfree_rate=0.03):
    """
    Runs a CPPI strategy given a set of returns for the risky asset.
    Returns a dictionary containing:
    - Asset Value History
    - Risk Budget History
    - Risky Weight History
    """
    # Set up the parameters
    dates = risky_r.index
    n_steps = len(dates)
    account_value = start  # Initial portfolio value
    floor_value = start * floor  # Minimum guaranteed portfolio value

    # Ensure the input is a DataFrame
    if isinstance(risky_r, pd.Series):
        risky_r = pd.DataFrame(risky_r, columns=["R"])

    # If no safe asset is provided, use a fixed risk-free return
    if safe_r is None:
        safe_r = pd.DataFrame().reindex_like(risky_r)
        safe_r[:] = riskfree_rate / 12  # Convert annualized rate to monthly

    # Initialize tracking dataframes
    account_history = pd.DataFrame().reindex_like(risky_r)
    cushion_history = pd.DataFrame().reindex_like(risky_r)
    risky_w_history = pd.DataFrame().reindex_like(risky_r)  # Risky asset weights

    # CPPI implementation
    for step in range(n_steps):
        # Compute cushion
        cushion = (account_value - floor_value) / account_value
```

```
352
353         # Compute allocation weights
354         risky_w = m * cushion  # Risky asset allocation
355         risky_w = np.minimum(risky_w, 1)  # Cap at 100%
356         risky_w = np.maximum(risky_w, 0)  # No short selling
357         safe_w = 1 - risky_w  # Safe asset allocation
358
359         # Compute allocations
360         risky_alloc = account_value * risky_w
361         safe_alloc = account_value * safe_w
362
363         # Update account value
364         account_value = (
365             risky_alloc * (1 + risky_r.iloc[step]) + safe_alloc * (1 + safe_r.iloc[step])
366         )
367
368         # Store results
369         cushion_history.iloc[step] = cushion
370         account_history.iloc[step] = account_value
371         risky_w_history.iloc[step] = risky_w
372
373     # Compute risky asset cumulative wealth (for comparison)
374     risky_wealth = start * (1 + risky_r).cumprod()
375
376     # Package results into a dictionary
377     backtest_result = {
378         "Wealth": account_history,
379         "Risky Wealth": risky_wealth,
380         "Risk Budget": cushion_history,
381         "Risky Allocation": risky_w_history,
382         "m": m,
383         "start": start,
384         "floor": floor,
385         "risky_r": risky_r,
386         "safe_r": safe_r,
387     }
388
389     return backtest_result
390
391 #Apply CCPI to our df_risky and compute the summary statistics.
392 # Run CPPI
393 btr = run_cppi1(df_risky)
394
395 #As CPPI returns account value, we need to compute returns:
396 for col in btr["Wealth"].columns:
397     print(f"Summary for {col}:")
398     print(summary_stats(btr["Wealth"].pct_change().dropna()[col]))
399
400 #E7. Update CPPI to introduce the drawdown constrain
401 def run_cppi(risky_r, safe_r=None, m=3, start=1000, floor=0.8, riskfree_rate=0.03,
    drawdown=None):
402     """
403     Returns a basket of the CPPI strategy, given a set of returns for the risky asset.
```

```
    Returns a dictionary containing: Asset Value History, Risk Budget History, Risky weight history.
    """
    # Set up the parameters
    dates = risky_r.index
    n_steps = len(dates)
    account_value = start  # Initial portfolio value
    peak = start  # At the start, the peak is the initial account value
    floor_value = start * floor  # Initial floor value

    # Ensure risky_r is a DataFrame
    if isinstance(risky_r, pd.Series):
        risky_r = pd.DataFrame(risky_r, columns=["R"])

    # If no safe asset is provided, assume a fixed risk-free return
    if safe_r is None:
        safe_r = pd.DataFrame(riskfree_rate / 12, index=dates, columns=risky_r.columns)

    # Initialize tracking DataFrames
    account_history = pd.DataFrame(index=dates, columns=risky_r.columns)
    cushion_history = pd.DataFrame(index=dates, columns=risky_r.columns)
    risky_w_history = pd.DataFrame(index=dates, columns=risky_r.columns)

    # CPPI implementation
    for step in range(n_steps):
        date = dates[step]

        # If there is a drawdown constraint, adjust the floor dynamically
        if drawdown is not None:
            # Update the peak value only if account_value exceeds the current peak
            peak = np.maximum(peak, account_value)
            floor_value = peak * (1 - drawdown)  # Recalculate floor dynamically based on drawdown

        # Compute cushion based on the difference between account value and floor value
        cushion = (account_value - floor_value) / account_value

        # Compute risky asset allocation based on the cushion
        risky_w = np.clip(m * cushion, 0, 1)  # Ensure allocation is between 0 and 1
        safe_w = 1 - risky_w  # Remaining allocation to safe asset

        # Allocate funds to risky and safe assets
        risky_alloc = account_value * risky_w
        safe_alloc = account_value * safe_w

        # Update account value based on returns
        account_value = (
            risky_alloc * (1 + risky_r.iloc[step]) +
            safe_alloc * (1 + safe_r.iloc[step])
        )

        # Debug: print the account value and peak
        print(f"Step {step} - Account Value: {account_value}, Peak: {peak}, Floor Value: {floor_value}, Drawdown: {(peak - account_value) / peak}")
```

```
455
456         # Store results
457         cushion_history.iloc[step] = cushion
458         account_history.iloc[step] = account_value
459         risky_w_history.iloc[step] = risky_w
460
461     # Fill missing values (if any)
462     account_history.fillna(method="ffill", inplace=True)
463
464     # Compute risky asset cumulative wealth for comparison
465     risky_wealth = start * (1 + risky_r).cumprod()
466
467     # Pack all backtest info into a dictionary
468     backtest_result = {
469         "Wealth": account_history,
470         "Risky Wealth": risky_wealth,
471         "Risk Budget": cushion_history,
472         "Risky Allocation": risky_w_history,
473         "m": m,
474         "start": start,
475         "floor": floor,
476         "risky_r": risky_r,
477         "safe_r": safe_r
478     }
479
480     return backtest_result
481
482 # Load data from the provided file
483 df_returns = pd.read_csv('C:/Users/52241/Downloads/EXCEL PYTHON/index30_returns.csv',
    header=0, index_col=0, parse_dates=True) / 100  # Convert to percentages
484 df_returns.index = pd.to_datetime(df_returns.index, format="%Y%m").to_period('M')
485 df_returns.columns = df_returns.columns.str.strip()
486
487 # Run CPPI on selected assets
488 btr = run_cppi(df_returns.loc["2007":, ["Steel", "Fin", "Beer"]], drawdown=0.25)
489
490 # Plot results
491 ax = btr["Wealth"].plot(figsize=(12, 6))
492 btr["Risky Wealth"].plot(ax=ax, style="--", label="Total Market Risky Wealth")
493 plt.legend()
494 plt.show()
495
496 # Apply summary stats function to CPPI results
497 summary_stats_df = btr["Risky Wealth"].pct_change().dropna().apply(summary_stats)
498
499 # Display results
500 print(summary_stats_df)
501
502 #As you can see there's an important difference in the Drawdowns. Now, this was because
    we were updating the
503 #floor every month, in practice you would want to save trading costs, so there are other
    tools that can be added to
504 #this.
505 #Another question is, what happens when we vary the drawdown constrain level?
```

```
# R1. Choose three industries and perform a risky allocation strategy. Analyze the risk-return statistics of this strategy. Apply CPPI without drawdown constrain. Analyze the evolution of the weights and cushion and explain what you see. Analyze the protection of CPPI
# to downside risk. Is there any relevant protection or oportunity cost identified? Analyze the risk-return. Apply CPPI with 10%, 20%, and 30% of drawdown constrain. Calculate summary statistics for each of these different drawdown constraints to see how they influence the CPPI strategy's risk-return profile. Analyze the results fo the strategy, explain how they influence the CPPI
# strategy's risk-return profile. Finally, explain the results and compare the five portfolio allocation strategies you have created. Which one would you recommend for each industry and why? Explain any interesting observation, the effect of CPPI, its pros and cons and the effect of the drawdown constrains.
# Selecciona las industrias específicas desde el mes de enero de 2000 (sin importar el día exacto)
# Cargar los datos de retornos de las industrias
# Games(No Drawdown)
# Load industry returns data
df_returns = pd.read_csv('C:/Users/52241/Downloads/EXCEL PYTHON/index30_returns.csv', header=0, index_col=0, parse_dates=True) / 100
df_returns.index = pd.to_datetime(df_returns.index, format="%Y%m")

# Select returns for Games from 2000 onwards
df_risky_games = df_returns.loc["2000":, ["Games"]]

# Run CPPI without drawdown
btr_1 = run_cppi1(df_risky_games)

# 📈 Plot "Total Market Risky Wealth"
fig, ax = plt.subplots(figsize=(12, 6))
btr_1["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
plt.title("Total Market Risky Wealth - Games (No Drawdown)")
plt.legend()
plt.show()

# Compute full risky wealth
risky_wealth_games = start * (1 + df_risky_games).cumprod()

# 📈 Plot "CPPI vs Full Risky Allocation" with correct labels
fig, ax = plt.subplots(figsize=(12, 6))
btr_1["Wealth"]["Games"].plot(ax=ax, label="CPPI")  # Correct label for CPPI
risky_wealth_games["Games"].plot(ax=ax, style="k--", label="Full Risky Allocation")  # Correct label for Full Risky Allocation
plt.axhline(y=start * 0.7, color='gray', linestyle="--", label="Floor Value")  # Floor Value
plt.title("CPPI vs Full Risky Allocation for Games (No Drawdown)")
plt.legend()
plt.show()

# 📊 Compute and display summary statistics
summary_stats_df = btr_1["Wealth"].pct_change().dropna().apply(summary_stats)
print(summary_stats_df)

#Games Drawdown 10%

```

```
btr_10 = run_cppi(df_risky_games, drawdown=0.10)

# 📈 Plot "Total Market Risky Wealth"
fig, ax = plt.subplots(figsize=(12, 6))
btr_10["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
plt.title("Total Market Risky Wealth - Drawdown 10%")
plt.legend()
plt.show()

# Compute full risky wealth
risky_wealth_games = start * (1 + df_risky_games).cumprod()

# 📈 Plot "CPPI vs Full Risky Allocation" (correct format)
fig, ax = plt.subplots(figsize=(12, 6))
btr_10["Wealth"]["Games"].plot(ax=ax, label="CPPI")
risky_wealth_games["Games"].plot(ax=ax, style="k--", label="Full Risky Allocation")
plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")
plt.title("CPPI vs Full Risky Allocation for Games (Drawdown 10%)")
plt.legend()
plt.show()

# 📊 Compute and display summary statistics
summary_stats_df = btr_10["Wealth"].pct_change().dropna().apply(summary_stats)
print(summary_stats_df)

# Run CPPI with a 30% drawdown constraint
btr_30 = run_cppi(df_risky_games, drawdown=0.30)

# 📈 Plot "Total Market Risky Wealth"
fig, ax = plt.subplots(figsize=(12, 6))
btr_30["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
plt.title("Total Market Risky Wealth - Games (Drawdown 30%)")
plt.legend()
plt.show()

# Compute full risky wealth
risky_wealth_games = start * (1 + df_risky_games).cumprod()

# 📈 Plot "CPPI vs Full Risky Allocation" with correct labels
fig, ax = plt.subplots(figsize=(12, 6))
btr_30["Wealth"]["Games"].plot(ax=ax, label="CPPI")  # CPPI Wealth
risky_wealth_games["Games"].plot(ax=ax, style="k--", label="Full Risky Allocation")  # Full Risky Wealth
plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")  # Correct Floor value
plt.title("CPPI vs Full Risky Allocation for Games (Drawdown 30%)")
plt.legend()
plt.show()

# 📊 Compute and display summary statistics
summary_stats_df = btr_30["Wealth"].pct_change().dropna().apply(summary_stats)
print(summary_stats_df)

# Run CPPI with a 20% drawdown constraint
```

```
btr_20 = run_cppi(df_risky_games, drawdown=0.20)

# 📈 Plot "Total Market Risky Wealth"
fig, ax = plt.subplots(figsize=(12, 6))
btr_20["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
plt.title("Total Market Risky Wealth - Games (Drawdown 20%)")
plt.legend()
plt.show()

# Compute full risky wealth
risky_wealth_games = start * (1 + df_risky_games).cumprod()

# 📈 Plot "CPPI vs Full Risky Allocation" with correct labels
fig, ax = plt.subplots(figsize=(12, 6))
btr_20["Wealth"]["Games"].plot(ax=ax, label="CPPI")  # CPPI Wealth
risky_wealth_games["Games"].plot(ax=ax, style="k--", label="Full Risky Allocation")  # Full Risky Wealth
plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")  # Correct Floor value
plt.title("CPPI vs Full Risky Allocation for Games (Drawdown 20%)")
plt.legend()
plt.show()

# 📊 Compute and display summary statistics
summary_stats_df = btr_20["Wealth"].pct_change().dropna().apply(summary_stats)
print(summary_stats_df)

# Select returns for Autos from 2000 onwards
df_risky_autos = df_returns.loc["2000":, ["Autos"]]

# Run CPPI without drawdown
btr_1_autos = run_cppi1(df_risky_autos)

# 📈 Plot "Total Market Risky Wealth" for Autos
fig, ax = plt.subplots(figsize=(12, 6))
btr_1_autos["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
plt.title("Total Market Risky Wealth - Autos (No Drawdown)")
plt.legend()
plt.show()

# Compute full risky wealth for Autos
risky_wealth_autos = start * (1 + df_risky_autos).cumprod()

# 📈 Plot "CPPI vs Full Risky Allocation" for Autos (No Drawdown)
fig, ax = plt.subplots(figsize=(12, 6))
btr_1_autos["Wealth"]["Autos"].plot(ax=ax, label="CPPI")  # Correct label for CPPI
risky_wealth_autos["Autos"].plot(ax=ax, style="k--", label="Full Risky Allocation")  # Correct label for Full Risky Allocation
plt.axhline(y=start * 0.7, color='gray', linestyle="--", label="Floor Value")  # Floor Value
plt.title("CPPI vs Full Risky Allocation for Autos (No Drawdown)")
plt.legend()
plt.show()

```

```python
# 📊 Compute and display summary statistics for Autos
summary_stats_df_autos = btr_1_autos["Wealth"].pct_change().dropna().apply(summary_stats)
print(summary_stats_df_autos)

# Autos Drawdown 10%
btr_10_autos = run_cppi(df_risky_autos, drawdown=0.10)

# 📈 Plot "Total Market Risky Wealth" for Autos with Drawdown 10%
fig, ax = plt.subplots(figsize=(12, 6))
btr_10_autos["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
plt.title("Total Market Risky Wealth - Autos (Drawdown 10%)")
plt.legend()
plt.show()

# Compute full risky wealth for Autos
risky_wealth_autos = start * (1 + df_risky_autos).cumprod()

# 📈 Plot "CPPI vs Full Risky Allocation" for Autos (Drawdown 10%)
fig, ax = plt.subplots(figsize=(12, 6))
btr_10_autos["Wealth"]["Autos"].plot(ax=ax, label="CPPI")
risky_wealth_autos["Autos"].plot(ax=ax, style="k--", label="Full Risky Allocation")
plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")
plt.title("CPPI vs Full Risky Allocation for Autos (Drawdown 10%)")
plt.legend()
plt.show()

# 📊 Compute and display summary statistics for Autos (Drawdown 10%)
summary_stats_df_autos =
btr_10_autos["Wealth"].pct_change().dropna().apply(summary_stats)
print(summary_stats_df_autos)

# Run CPPI with a 30% drawdown constraint for Autos
btr_30_autos = run_cppi(df_risky_autos, drawdown=0.30)

# 📈 Plot "Total Market Risky Wealth" for Autos with Drawdown 30%
fig, ax = plt.subplots(figsize=(12, 6))
btr_30_autos["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
plt.title("Total Market Risky Wealth - Autos (Drawdown 30%)")
plt.legend()
plt.show()

# Compute full risky wealth for Autos
risky_wealth_autos = start * (1 + df_risky_autos).cumprod()

# 📈 Plot "CPPI vs Full Risky Allocation" for Autos (Drawdown 30%)
fig, ax = plt.subplots(figsize=(12, 6))
btr_30_autos["Wealth"]["Autos"].plot(ax=ax, label="CPPI")
risky_wealth_autos["Autos"].plot(ax=ax, style="k--", label="Full Risky Allocation")
plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")
plt.title("CPPI vs Full Risky Allocation for Autos (Drawdown 30%)")
plt.legend()
plt.show()

# 📊 Compute and display summary statistics for Autos (Drawdown 30%)
```

```
summary_stats_df_autos = btr_30_autos["Wealth"].pct_change().dropna().apply(summary_stats)
print(summary_stats_df_autos)

# Run CPPI with a 20% drawdown constraint for Autos
btr_20_autos = run_cppi(df_risky_autos, drawdown=0.20)

# 📈 Plot "Total Market Risky Wealth" for Autos with Drawdown 20%
fig, ax = plt.subplots(figsize=(12, 6))
btr_20_autos["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
plt.title("Total Market Risky Wealth - Autos (Drawdown 20%)")
plt.legend()
plt.show()

# Compute full risky wealth for Autos
risky_wealth_autos = start * (1 + df_risky_autos).cumprod()

# 📈 Plot "CPPI vs Full Risky Allocation" for Autos (Drawdown 20%)
fig, ax = plt.subplots(figsize=(12, 6))
btr_20_autos["Wealth"]["Autos"].plot(ax=ax, label="CPPI")
risky_wealth_autos["Autos"].plot(ax=ax, style="k--", label="Full Risky Allocation")
plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")
plt.title("CPPI vs Full Risky Allocation for Autos (Drawdown 20%)")
plt.legend()
plt.show()

# 📊 Compute and display summary statistics for Autos (Drawdown 20%)
summary_stats_df_autos = btr_20_autos["Wealth"].pct_change().dropna().apply(summary_stats)
print(summary_stats_df_autos)

# Select returns for Telecom (Telcm) from 2000 onwards
df_risky_telcm = df_returns.loc["2000":, ["Telcm"]]

# Run CPPI without drawdown
btr_1_telcm = run_cppi1(df_risky_telcm)

# 📈 Plot "Total Market Risky Wealth" for Telecom (Telcm)
fig, ax = plt.subplots(figsize=(12, 6))
btr_1_telcm["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
plt.title("Total Market Risky Wealth - Telecom (Telcm) (No Drawdown)")
plt.legend()
plt.show()

# Compute full risky wealth for Telecom
risky_wealth_telcm = start * (1 + df_risky_telcm).cumprod()

# 📈 Plot "CPPI vs Full Risky Allocation" for Telecom (Telcm) (No Drawdown)
fig, ax = plt.subplots(figsize=(12, 6))
btr_1_telcm["Wealth"]["Telcm"].plot(ax=ax, label="CPPI")  # Correct label for CPPI
risky_wealth_telcm["Telcm"].plot(ax=ax, style="k--", label="Full Risky Allocation")  # Correct label for Full Risky Allocation
plt.axhline(y=start * 0.7, color='gray', linestyle="--", label="Floor Value")  # Floor Value
```

```
plt.title("CPPI vs Full Risky Allocation for Telecom (Telcm) (No Drawdown)")
plt.legend()
plt.show()

# 📊 Compute and display summary statistics for Telecom (Telcm)
summary_stats_df_telcm = btr_1_telcm["Wealth"].pct_change().dropna().apply(summary_stats)
print(summary_stats_df_telcm)

# Telecom (Telcm) Drawdown 10%
btr_10_telcm = run_cppi(df_risky_telcm, drawdown=0.10)

# 📈 Plot "Total Market Risky Wealth" for Telecom (Telcm) with Drawdown 10%
fig, ax = plt.subplots(figsize=(12, 6))
btr_10_telcm["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
plt.title("Total Market Risky Wealth - Telecom (Telcm) (Drawdown 10%)")
plt.legend()
plt.show()

# Compute full risky wealth for Telecom
risky_wealth_telcm = start * (1 + df_risky_telcm).cumprod()

# 📈 Plot "CPPI vs Full Risky Allocation" for Telecom (Telcm) (Drawdown 10%)
fig, ax = plt.subplots(figsize=(12, 6))
btr_10_telcm["Wealth"]["Telcm"].plot(ax=ax, label="CPPI")
risky_wealth_telcm["Telcm"].plot(ax=ax, style="k--", label="Full Risky Allocation")
plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")
plt.title("CPPI vs Full Risky Allocation for Telecom (Telcm) (Drawdown 10%)")
plt.legend()
plt.show()

# 📊 Compute and display summary statistics for Telecom (Telcm) (Drawdown 10%)
summary_stats_df_telcm =
btr_10_telcm["Wealth"].pct_change().dropna().apply(summary_stats)
print(summary_stats_df_telcm)

# Run CPPI with a 30% drawdown constraint for Telecom (Telcm)
btr_30_telcm = run_cppi(df_risky_telcm, drawdown=0.30)

# 📈 Plot "Total Market Risky Wealth" for Telecom (Telcm) with Drawdown 30%
fig, ax = plt.subplots(figsize=(12, 6))
btr_30_telcm["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
plt.title("Total Market Risky Wealth - Telecom (Telcm) (Drawdown 30%)")
plt.legend()
plt.show()

# Compute full risky wealth for Telecom
risky_wealth_telcm = start * (1 + df_risky_telcm).cumprod()

# 📈 Plot "CPPI vs Full Risky Allocation" for Telecom (Telcm) (Drawdown 30%)
fig, ax = plt.subplots(figsize=(12, 6))
btr_30_telcm["Wealth"]["Telcm"].plot(ax=ax, label="CPPI")
risky_wealth_telcm["Telcm"].plot(ax=ax, style="k--", label="Full Risky Allocation")
plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")
plt.title("CPPI vs Full Risky Allocation for Telecom (Telcm) (Drawdown 30%)")
```

```
plt.legend()
plt.show()

# 📊 Compute and display summary statistics for Telecom (Telcm) (Drawdown 30%)
summary_stats_df_telcm = btr_30_telcm["Wealth"].pct_change().dropna().apply(summary_stats)
print(summary_stats_df_telcm)

# Run CPPI with a 20% drawdown constraint for Telecom (Telcm)
btr_20_telcm = run_cppi(df_risky_telcm, drawdown=0.20)

# 📈 Plot "Total Market Risky Wealth" for Telecom (Telcm) with Drawdown 20%
fig, ax = plt.subplots(figsize=(12, 6))
btr_20_telcm["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
plt.title("Total Market Risky Wealth - Telecom (Telcm) (Drawdown 20%)")
plt.legend()
plt.show()

# Compute full risky wealth for Telecom
risky_wealth_telcm = start * (1 + df_risky_telcm).cumprod()

# 📈 Plot "CPPI vs Full Risky Allocation" for Telecom (Telcm) (Drawdown 20%)
fig, ax = plt.subplots(figsize=(12, 6))
btr_20_telcm["Wealth"]["Telcm"].plot(ax=ax, label="CPPI")
risky_wealth_telcm["Telcm"].plot(ax=ax, style="k--", label="Full Risky Allocation")
plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")
plt.title("CPPI vs Full Risky Allocation for Telecom (Telcm) (Drawdown 20%)")
plt.legend()
plt.show()

# 📊 Compute and display summary statistics for Telecom (Telcm) (Drawdown 20%)
summary_stats_df_telcm = btr_20_telcm["Wealth"].pct_change().dropna().apply(summary_stats)
print(summary_stats_df_telcm)

## 2 Random Walks and Asset Simulation
# E8. Implement GBM

def gbm0(n_years=10, n_scenarios=1000, mu=0.07, sigma=0.15, steps_per_year=12, s_0=100.0):
    """
    Evolution of a Stock Price using GBM (Geometric Brownian Motion)
    """
    dt = 1 / steps_per_year  # Step size
    n_steps = int(n_years * steps_per_year)  # Total number of steps
    xi = np.random.normal(size=(n_steps, n_scenarios))  # Generate random returns

    # Apply the GBM to calculate returns for each step
    rets = mu * dt + sigma * np.sqrt(dt) * xi  # GBM returns formula

    # Calculate prices by cumulative product of (1 + returns)
    prices = s_0 * (1 + rets).cumprod(axis=0)  # Cumulative product of returns

    # Convert numpy array to DataFrame for better handling
```

```
    prices_df = pd.DataFrame(prices)

    return prices_df

#Now, generate 3 scenarios for a stock price for 10 years.
# Generate sample data with gbm0
p = gbm0(n_years=10, n_scenarios=3)
print(p.head())

# Plot the results
p.plot(figsize=(12, 6),legend=False)
plt.title("Geometric Brownian Motion (Basic Implementation)")
plt.show()

# E9. Simulate 100 scenarios for 10 years.
p = gbm0(n_years=10, n_scenarios=100)
print(p.head())

# Plot the results
p.plot(figsize=(12, 6),legend=False)
plt.title("Geometric Brownian Motion (Basic Implementation)")
plt.show()

#The next function, gbm, is an optimized version of gbm0, which uses vectorization for faster computation. Instead of calculating returns and adding 1 to each element in a loop, it directly generates the adjusted return values.

def gbm(n_years=10, n_scenarios=1000, mu=0.07, sigma=0.15, steps_per_year=12, s_0=100.0):
    """
    Optimized Evolution of a Stock Price using GBM (Geometric Brownian Motion)
    """
    dt = 1 / steps_per_year  # Time step
    n_steps = int(n_years * steps_per_year)  # Total number of time steps

    # Generate the adjusted returns (1 + r_i) using vectorization
    rets_plus_1 = np.random.normal(loc=1 + mu * dt, scale=sigma * np.sqrt(dt), size=(n_steps, n_scenarios))

    # Calculate prices by taking the cumulative product of returns for each scenario
    prices = s_0 * pd.DataFrame(rets_plus_1).cumprod(axis=0)  # Optimized cumulative product calculation

    return prices

# Apply the optimized GBM function
p_optimized = gbm(n_years=10, n_scenarios=100)

# Plot the optimized results
p_optimized.plot(figsize=(12, 6), legend=False)
plt.title("Optimized Geometric Brownian Motion")
plt.show()

#If the improvement is not clear, run the following cell to compute the time it takes to run each GBM implementation.
```

```python
# Compare performance
import time
# Timing the execution of gbm0 (non-optimized)
start = time.time()
gbm0(n_years=5, n_scenarios=1000)
end = time.time()
print(f"Time taken by gbm0: {end - start:.4f} seconds")

# Timing the execution of gbm (optimized)
start = time.time()
gbm(n_years=5, n_scenarios=1000)
end = time.time()
print(f"Time taken by gbm: {end - start:.4f} seconds")

#The last refinement ensures that the simulated stock prices start exactly at the initial value s_0 for all scenarios by setting the first row to 1 in rets_plus_1.
def gbm(n_years=10, n_scenarios=1000, mu=0.07, sigma=0.15, steps_per_year=12, s_0=100.0):
    """
    Final Evolution of a Stock Price using GBM, ensuring initial value starts at s_0.
    """
    dt = 1 / steps_per_year
    n_steps = int(n_years * steps_per_year)
    rets_plus_1 = np.random.normal(loc=1 + mu * dt, scale=sigma * np.sqrt(dt), size=(n_steps, n_scenarios))
    rets_plus_1[0] = 1  # Start all scenarios at s_0
    prices = s_0 * pd.DataFrame(rets_plus_1).cumprod()
    return prices

gbm(n_years=10,n_scenarios=1000).plot(legend=False)

#Now, we can use GBM to simulate assets and then apply CPPI.
#E10. Define a function show_cppi to run a Monte Carlo simulation of the CPPI strategy. This function uses simulated risky asset returns generated by the gbm function assuming an initial investment of 100. For the simulation,
#  use n_scenarios=50, mu=0.07, sigma=0.15, m=3, floor=0.0, riskfree_rate=0.03

def show_cppi(n_scenarios=50, mu=0.07, sigma=0.15, m=3, floor=0.0, riskfree_rate=0.03, y_max=100):
    """
    Plots the result of a Monte Carlo Simulation of CPPI, including a histogram of terminal wealth.
    """
    start = 100  # Initial investment
    sim_rets = gbm(n_years=10, n_scenarios=n_scenarios, mu=mu, sigma=sigma)  # Simulate returns using GBM function
    risky_r = pd.DataFrame(sim_rets.pct_change().dropna())  # Convert returns to percentage change

    # Run CPPI back-test
    btr = run_cppi(risky_r=risky_r, riskfree_rate=riskfree_rate, m=m, start=start, floor=floor)
    wealth = btr["Wealth"]

    # Calculate terminal wealth stats
    y_max = wealth.values.max() * y_max / 100
```

```
    terminal_wealth = wealth.iloc[-1]

    # Plot wealth evolution and terminal wealth histogram
    fig, (wealth_ax, hist_ax) = plt.subplots(nrows=1, ncols=2, sharey=True, gridspec_kw={'width_ratios': [3, 2]}, figsize=(24, 9))
    plt.subplots_adjust(wspace=0.0)

    # Plot wealth evolution
    wealth.plot(ax=wealth_ax, legend=False, alpha=0.3, color="indianred")  # Keep the specified line for wealth evolution
    wealth_ax.axhline(y=start, ls=":", color="black")  # Line for initial wealth
    wealth_ax.axhline(y=start * floor, ls="--", color="red")  # Floor line
    wealth_ax.set_ylim(top=y_max)  # Set the y-axis limit to maximum wealth

    # Plot terminal wealth histogram
    terminal_wealth.plot.hist(ax=hist_ax, bins=50, ec='w', fc='indianred', orientation='horizontal')
    hist_ax.axhline(y=start, ls=":", color="black")  # Line for initial wealth
    plt.title("Monte Carlo Simulation of CPPI Strategy with Terminal Wealth Distribution")
    plt.show()

# Now, run the simulation with your desired parameters
show_cppi(n_scenarios=50, mu=0.07, sigma=0.15, m=3, floor=0.0, riskfree_rate=0.03, y_max=100)

# E11. Enhance show_cppi by including a histogram of
# terminal wealth at the end of the simulation.
#  This histogram shows the distribution of outcomes
#  across different scenarios. Also, include  additional
# statistics, such as mean, median, the
# probability of falling below the floor, and
# expected shortfall if the floor is violated.

def show_cppi(n_scenarios=50, mu=0.07, sigma=0.15, m=3, floor=0.0, riskfree_rate=0.03, y_max=100):
    """
    Plots the result of a Monte Carlo Simulation of CPPI, including terminal wealth stats.
    """
    start = 100  # Starting account value
    sim_rets = gbm(n_years=40, n_scenarios=n_scenarios, mu=mu, sigma=sigma)  # Simulate returns using GBM
    risky_r = sim_rets.pct_change().dropna()  # Calculate the returns from the simulated prices

    # Run CPPI back-test
    btr = run_cppi(risky_r=risky_r, riskfree_rate=riskfree_rate, m=m, start=start, floor=floor)
    wealth = btr["Wealth"]
    terminal_wealth = wealth.iloc[-1]  # Terminal wealth for each scenario

    # Calculate terminal wealth stats
    y_max = wealth.values.max() * y_max / 100  # Set y_max based on the maximum wealth value
```

```
    tw_mean = terminal_wealth.mean()  # Compute mean terminal wealth
    tw_median = terminal_wealth.median()  # Compute median terminal wealth
    failure_mask = terminal_wealth < start * floor  # Mask for floor violations
    n_failures = failure_mask.sum()  # Sum the number of violations
    p_fail = n_failures / len(terminal_wealth)  # Probability of violating the floor
    e_shortfall = (terminal_wealth - start * floor)[failure_mask].mean() if n_failures > 0 else 0.0  # Expected shortfall

    # Plot wealth evolution and terminal wealth histogram
    fig, (wealth_ax, hist_ax) = plt.subplots(nrows=1, ncols=2, sharey=True, gridspec_kw={'width_ratios': [3, 2]}, figsize=(24, 9))
    plt.subplots_adjust(wspace=0.0)

    # Plot wealth evolution
    wealth.plot(ax=wealth_ax, legend=False, alpha=0.3, color="indianred")  # Wealth evolution plot
    wealth_ax.axhline(y=start, ls=":", color="black")  # Initial wealth line
    wealth_ax.axhline(y=start * floor, ls="--", color="red")  # Floor line
    wealth_ax.set_title("CPPI Wealth Evolution")
    wealth_ax.set_ylim(top=y_max)

    # Plot terminal wealth histogram
    terminal_wealth.plot.hist(ax=hist_ax, bins=50, alpha=0.7, color="indianred", edgecolor="black", orientation='horizontal')
    hist_ax.axhline(y=start, ls=":", color="black")  # Initial wealth line
    hist_ax.axhline(y=start * floor, ls="--", color="red", linewidth=3)  # Floor line

    # Annotate statistics
    hist_ax.annotate(f"Mean: ${int(tw_mean)}", xy=(0.7, 0.9), xycoords='axes fraction', fontsize=24)
    hist_ax.annotate(f"Median: ${int(tw_median)}", xy=(0.7, 0.85), xycoords='axes fraction', fontsize=24)

    # If there are violations, display them, otherwise show 0 violations
    if n_failures == 0:
        hist_ax.annotate(f"Violations: 0 (0.00%)", xy=(0.7, 0.7), xycoords="axes fraction", fontsize=24)
        hist_ax.annotate(f"Expected Shortfall: $0.00", xy=(0.7, 0.6), xycoords="axes fraction", fontsize=24)  # Expected shortfall 0 if no violation
    else:
        hist_ax.annotate(f"Violations: {n_failures} ({p_fail * 100:.2f}%)\nE(shortfall)=${e_shortfall:.2f}",
                         xy=(0.7, 0.7), xycoords="axes fraction", fontsize=24)

    plt.title("Monte Carlo Simulation of CPPI Strategy with Terminal Wealth Distribution")
    plt.show()

    print(f"Configuration: μ={mu}, σ={sigma}")
    print(f"  Mean Terminal Wealth: ${tw_mean:.2f}")
    print(f"  Median Terminal Wealth: ${tw_median:.2f}")
    print(f"  Probability of Violating the Floor: {p_fail * 100:.2f}%")
    print(f"  Expected Shortfall (if violated): ${e_shortfall:.2f}")
    print("-" * 50)

```

```
# Now, run the simulation with your desired parameters
show_cppi(n_scenarios=50, mu=0.07, sigma=0.15, m=3, floor=0.0, riskfree_rate=0.03, y_max=100)

# R2. You are going to simulate returns and apply CPPI. First, explain how GBM is used and computed, and how it can be used into CPPI. Then, perform 5 different simulations of returns using GBM with
# different configurations for 1000 scenarios for periods of 40 years. Apply CPPI to the simulated returns. Plot a histogram of the terminal wealth at the end of the simulation. Also, include additional statistics,
# such as mean, median, the probability of falling below the floor (which is computed from the times the floor was violated during the simulations), and the expected shortfall if the floor is violated.
# Analyze the results and give some key conclusions about GBM usage for Portfolio Management and CPPI

# 1. Simulation with mu=0.12, sigma=0.25
mu_1 = 0.12
sigma_1 = 0.25
show_cppi(n_scenarios=1000, mu=mu_1, sigma=sigma_1, m=8, floor=0.9, riskfree_rate=0.03, y_max=100)

# 2. Simulation with mu=0.07, sigma=0.18
mu_2 = 0.07
sigma_2 = 0.18
show_cppi(n_scenarios=1000, mu=mu_2, sigma=sigma_2, m=6, floor=0.8, riskfree_rate=0.03, y_max=100)

# 3. Simulation with mu=0.03, sigma=0.10
mu_3 = 0.03
sigma_3 = 0.10
show_cppi(n_scenarios=1000, mu=mu_3, sigma=sigma_3, m=4, floor=0.7, riskfree_rate=0.03, y_max=100)

# 4. Simulation with mu=0.05, sigma=0.15
mu_4 = 0.05
sigma_4 = 0.15
show_cppi(n_scenarios=1000, mu=mu_4, sigma=sigma_4, m=7, floor=0.8, riskfree_rate=0.03, y_max=100)

# 5. Simulation with mu=0.02, sigma=0.05
mu_5 = 0.02
sigma_5 = 0.05
show_cppi(n_scenarios=1000, mu=mu_5, sigma=sigma_5, m=10, floor=0.9, riskfree_rate=0.03, y_max=100)

# 3 Present Value of Liabilities and Funding Ratio
# E12. Implement fucntions to compute the the price of a
# pure discount bond, the PV of liabilities, and the
# funding ratio

def discount(t, r):
    """
    Compute the price of a pure discount bond that pays $1 at time t,
    given an interest rate r.
```

```
1082
1083        t: time in years when the bond pays $1.
1084        r: annual interest rate.
1085
1086        Returns the price of the bond at time 0.
1087        """
1088        return 1 / (1 + r)**t
1089
1090    #To check the discount factor for a payment due in 10 years at a 3% interest rate
1091    discount(10, 0.03) #returns 0.7440939148967249
1092
1093    def pv(liabilities, r):
1094        """
1095        Computes the present value of a set of liabilities.
1096        `liabilities` is indexed by the time, and values are the amounts.
1097        Returns the present value of the set.
1098
1099        liabilities: a dictionary where keys are times (in years) and values are the
    liability amounts at each time.
1100        r: the annual interest rate.
1101
1102        Returns the present value of the liabilities.
1103        """
1104        total_pv = 0
1105        for t, amount in liabilities.items():
1106            total_pv += amount / (1 + r)**t  # Discount the amount at time t
1107        return total_pv
1108
1109    #Define a set of liabilities and calculate their present value with a 3% discount rate.
1110    liabilities = pd.Series(data=[1, 1.5, 2, 2.5], index=[3, 3.5, 4, 4.5])
1111    pv(liabilities, 0.03)  # Returns 6.233320315080045
1112
1113    def funding_ratio(assets, liabilities, r):
1114        """
1115        Computes the funding ratio given assets, liabilities, and interest rate.
1116
1117        assets: amount of available money to cover the liabilities.
1118        liabilities: a dictionary where keys are times (in years) and values are the
    liability amounts at each time.
1119        r: annual interest rate.
1120
1121        Returns the funding ratio (assets / present value of liabilities).
1122        """
1123        pv_liabilities = pv(liabilities, r)  # Calculate the present value of the liabilities
1124        return assets / pv_liabilities
1125
1126    #To calculate the funding ratio with $5 in assets and a
1127    #  3% interest rate
1128    funding_ratio(5, liabilities, 0.03)  # Returns 0.8021407126958777
1129
1130    #To observe the effect of interest rate changes on the funding ratio, we can recalculate
    it with different rates. A drop in the interest rate generally lowers the funding ratio
    because liabilities' present value increases when discounted at a lower rate.
1131    # Calculate funding ratio with interest rate of 2%
```

```
1132 funding_ratio(5, liabilities, 0.02)  # Returns 0.7720304366941648
1133
1134 #E13. Vary the interest rate and number of assets to see the effect
1135
1136 # Example liabilities (indexed by time) and their amounts
1137 liabilities = pd.Series(data=[1, 1.5, 2, 2.5], index=[3, 3.5, 4, 4.5])
1138
1139 # Example 1: Calculate present value of liabilities at 3% interest rate
1140 print("Present Value of Liabilities at 3% interest rate:", pv(liabilities, 0.03))
1141
1142 # Example 2: Calculate funding ratio with $5 in assets and a 3% interest rate
1143 print("Funding Ratio with 5 assets and 3% interest rate:", funding_ratio(5, liabilities,
     0.03))
1144
1145 # Example 3: Calculate funding ratio with 2% interest rate
1146 print("Funding Ratio with 5 assets and 2% interest rate:", funding_ratio(5, liabilities,
     0.02))
1147
1148 # Example 4: Vary the number of assets and calculate funding ratio
1149 print("Funding Ratio with 7 assets and 3% interest rate:", funding_ratio(7, liabilities,
     0.03))
1150 print("Funding Ratio with 10 assets and 3% interest rate:", funding_ratio(10,
     liabilities, 0.03))
1151
1152 # Example 5: Calculate funding ratio with a higher interest rate (4%)
1153 print("Funding Ratio with 5 assets and 4% interest rate:", funding_ratio(5, liabilities,
     0.04))
1154
1155 # 4 CIR Model for Interest Rates
1156 #E14. Implement the conversions from and to force of interest and the CIR model to
     simulate interest rates movements.
1157
1158 def force_to_ann(r):
1159     """
1160     Converts force of interest to an annualized rate.
1161     """
1162     return np.exp(r) - 1
1163
1164 def ann_to_force(r):
1165     """
1166     Converts an annualized rate to the force of interest.
1167     """
1168     return np.log(1 + r)
1169
1170 #The cir function implements the Cox-Ingersoll-Ross (CIR) model, which simulates the
     evolution of interest rates over time.
1171
1172
1173 def cir(n_years=10, n_scenarios=1, a=0.05, b=0.03, sigma=0.05, steps_per_year=12,
     r_0=None):
1174     """
1175     Implements the CIR model for simulating interest rate evolution over time.
1176     """
1177     if r_0 is None:  # Don't change this line
1178         r_0 = b
```

```
1179
1180      # Convert r_0 to the force of interest
1181      r_0 = np.log(1 + r_0)
1182
1183      # Step size in years
1184      dt = 1 / steps_per_year
1185
1186      # Number of time steps
1187      num_steps = int(n_years * steps_per_year) + 1
1188
1189      # Simulate normal variables (Wiener process increments)
1190      shock = np.random.normal(0, scale=np.sqrt(dt), size=(num_steps, n_scenarios))
1191
1192      # Array to hold simulated rates
1193      rates = np.empty_like(shock)
1194
1195      # Initial rate
1196      rates[0] = r_0
1197
1198      # Simulate the rates movement using the CIR model
1199      for step in range(1, num_steps):
1200          r_t = rates[step - 1]
1201          d_r_t = a * (b - r_t) * dt + sigma * np.sqrt(r_t) * shock[step]  # CIR model
    equation
1202          rates[step] = abs(r_t + d_r_t)  # Ensure the rate is non-negative
1203
1204      # Convert the rates from force of interest to annualized rates
1205      annualized_rates = np.exp(rates) - 1
1206
1207      # Return as DataFrame
1208      return pd.DataFrame(data=annualized_rates, index=range(num_steps))
1209
1210  #Apply the CIR model
1211  cir(n_scenarios=10).plot(figsize=(12, 5), title="CIR Model Simulation of Interest Rates")
1212  plt.show()
1213
1214  #E15. Extend the CIR model to valuate ZC bonds.
1215
1216  import numpy as np
1217  import pandas as pd
1218  import math
1219
1220  def cir(n_years=10, n_scenarios=1, a=0.05, b=0.03, sigma=0.05, steps_per_year=12,
    r_0=None):
1221      if r_0 is None:  # Don't change this line
1222          r_0 = b
1223
1224      # Convert r_0 to the force of interest
1225      r_0 = np.log(1 + r_0)
1226
1227      # Step size in years
1228      dt = 1 / steps_per_year
1229
1230      # Number of time steps
```

```
    num_steps = int(n_years * steps_per_year) + 1

    # Simulate normal variables (Wiener process increments)
    shock = np.random.normal(0, scale=np.sqrt(dt), size=(num_steps, n_scenarios))

    # Array to hold simulated rates and prices
    rates = np.empty_like(shock)
    prices = np.empty_like(shock)

    # Initial rate
    rates[0] = r_0

    # CIR model dynamics simulation
    h = np.sqrt(a ** 2 + 2 * sigma ** 2)

    # Function to calculate the price of a zero-coupon bond
    def price(ttm, r):  # ttm: time to maturity, r: current interest rate
        # Compute A(t, T)
        _A = ((2 * h * np.exp((h + a) * ttm / 2)) / (2 * h + (h + a) * (np.exp(h * ttm) - 1))) ** (2 * a * b / sigma ** 2)

        # Compute B(t, T)
        _B = (2 * (np.exp(h * ttm) - 1)) / (2 * h + (h + a) * (np.exp(h * ttm) - 1))

        # Zero-coupon bond price
        return _A * np.exp(-_B * r)

    # Calculate bond prices at the initial time
    prices[0] = price(n_years, rates[0])

    # Simulate the rates and prices over time
    for step in range(1, num_steps):
        r_t = rates[step - 1]
        # Simulate the changes in interest rates (Cox-Ingersoll-Ross)
        d_r_t = a * (b - r_t) * dt + sigma * np.sqrt(r_t) * shock[step]
        rates[step] = abs(r_t + d_r_t)

        # Calculate bond prices
        prices[step] = price(n_years - step * dt, rates[step])  # Compute the price based on time to maturity

    # Convert rates from force of interest to annualized rates
    rates = pd.DataFrame(data=np.exp(rates) - 1, index=range(num_steps))

    # Convert prices to a DataFrame
    prices = pd.DataFrame(data=prices, index=range(num_steps))

    return rates, prices

#Apply the cir model
cir(r_0=0.03,a=0.5,b=0.03,sigma=0.05,n_scenarios=5)[1].plot(legend=False)
plt.show()

# R3. Explain the CIR model and its applications.
```

```
#  Use it to simulate 5 interest rates with 10
# scenarios each and different configurations that allow you to
#  see how it behaves under different scenarios. Be wise when
# choosing your configurations. Analyze the results and provide
#  insights about interest rate modelling.

import matplotlib.ticker as ticker

# Simulating interest rate paths for different configurations

# Configuration 1: High mean reversion speed, low volatility
rates_config1, _ = cir(a=1.0, b=0.03, sigma=0.02, n_scenarios=10)

# Configuration 2: Low mean reversion speed, high volatility
rates_config2, _ = cir(a=0.1, b=0.03, sigma=0.05, n_scenarios=10)

# Configuration 3: Moderate mean reversion speed, high volatility
rates_config3, _ = cir(a=0.5, b=0.03, sigma=0.1, n_scenarios=10)

# Configuration 4: Very high volatility
rates_config4, _ = cir(a=0.5, b=0.03, sigma=0.2, n_scenarios=10)

# Configuration 5: Low mean reversion speed, low volatility
rates_config5, _ = cir(a=0.1, b=0.03, sigma=0.02, n_scenarios=10)

# Plotting the results
fig, axes = plt.subplots(5, 1, figsize=(12, 12), sharex=True)

configs = [
    (rates_config1, "Configuration 1: High Mean Reversion & Low Volatility"),
    (rates_config2, "Configuration 2: Low Mean Reversion & High Volatility"),
    (rates_config3, "Configuration 3: Moderate Mean Reversion & High Volatility"),
    (rates_config4, "Configuration 4: Very High Volatility"),
    (rates_config5, "Configuration 5: Low Mean Reversion & Low Volatility")
]

for ax, (rates, title) in zip(axes, configs):
    rates.plot(ax=ax, legend=False)
    ax.set_title(title, fontsize=10)
    ax.set_ylabel("Interest Rate", fontsize=8)
    ax.yaxis.set_major_locator(ticker.MaxNLocator(5))  # Reduce number of Y ticks
    ax.xaxis.set_major_locator(ticker.MaxNLocator(10))  # Reduce number of X ticks
    ax.tick_params(axis="both", which="major", labelsize=8)  # Adjust font size

axes[-1].set_xlabel("Time (Months)", fontsize=10)
plt.xticks(rotation=45)  # Rotate X labels for better readability

plt.tight_layout()
plt.show()

#In class, we also talked about the risk perspective of using cash vs bonds to fund
liabilities.
#The following compares zc bonds to cash. Assume that liabilities are the bond
prices,i.e. we are using bond prices to model liabilities as they are almost the same
```

```
from a computation and mathematical finance perspective.

import pandas as pd

# Initial cash on hand (in millions)
a_0 = 0.75

# Simulate interest rates and bond prices using the CIR model
rates, bond_prices = cir(r_0=0.03, b=0.03, n_scenarios=10)

# Assume that liabilities are the bond prices
liabilities = bond_prices

# Present value of a Zero-Coupon Bond (ZCB) maturing in 10 years
zcbond_10 = pd.Series(data=[1], index=[10])  # $1 received in year 10

# Compute the present value of the ZC bond today with a 3% interest rate
zc_0 = pv(zcbond_10, 0.03)  # The present value of $1 in 10 years at 3% discount rate

# How many bonds can we buy?
n_bonds = a_0 / zc_0

# Asset value assuming we invest in Zero-Coupon Bonds
av_zc_bonds = n_bonds * bond_prices

# Asset value assuming we invest in cash
av_cash = a_0 * (rates / 12 + 1).cumprod()

av_cash.plot(legend=False)
plt.show()

#Notice we have good and not so good situations. So if you we're at a pension fund and you put all the money in the safe option, and you don't make the million (liability) then your return to the persons in the pension wouldn't be enough, you wouldn't have money to pay.
av_zc_bonds.plot(legend=False)
plt.show()

#Notice that the return at the end of the period was one million dolars. You could have two looks at this. You could say the bonds were not safe, or that they gave you a safe reward.
(av_cash/liabilities).pct_change().plot(title='Returns of Funding Ratio with Cash (10 scenarios)',legend=False,figsize=(12,5) )
plt.show()

#As we said in class it's very risky.
(av_zc_bonds/liabilities).pct_change().plot(title='Returns of Funding Ratio with Bonds (10 scenarios)',legend=False,figsize=(12,5))
plt.show()

# E15. Compute the final funding ratio using the CIR model

# Initial cash on hand (in millions)
a_0 = 0.75
```

```
1381
1382 # Simulate interest rates and bond prices using the CIR model (10,000 scenarios)
1383 rates, bond_prices = cir(r_0=0.03, b=0.03, n_scenarios=10000)
1384
1385 # Assume liabilities are the bond prices
1386 liabilities = bond_prices
1387
1388 # Define a zero-coupon bond that pays $1 in 10 years
1389 zcbond_10 = pd.Series(data=[1], index=[10])
1390
1391 # Compute the present value of the zero-coupon bond at a 3% interest rate
1392 zc_0 = pv(zcbond_10, 0.03)
1393
1394 # Determine how many bonds can be purchased
1395 n_bonds = a_0 / zc_0
1396
1397 # Asset value assuming we buy bonds
1398 av_zc_bonds = n_bonds * bond_prices
1399
1400 # Asset value assuming we invest in cash
1401 av_cash = a_0 * (rates / 12 + 1).cumprod()
1402
1403 #at the last point in time
1404 tfr_cash=av_cash.iloc[-1]/liabilities.iloc[-1]
1405 tfr_zc_bonds=av_zc_bonds.iloc[-1]/liabilities.iloc[-1]
1406 ax=tfr_cash.plot.hist(label="Cash", figsize=(15,6), bins=100, legend=True)
1407 tfr_zc_bonds.plot.hist(ax=ax,label="ZC Bonds", bins=100, legend=True, secondary_y=True)
1408 plt.show()
1409
1410 #The unique "convinent" assumption is that we have 0.75 million and 10 years to get the
     million. Repeat this time stating with 0.5
1411
1412 # Initial cash on hand (0.5 million)
1413 a_0 = 0.5
1414
1415 # Simulate interest rates and bond prices using the CIR model
1416 rates, bond_prices = cir(r_0=0.03, b=0.03, n_scenarios=10000)
1417
1418 # Assume liabilities are the bond prices
1419 liabilities = bond_prices
1420
1421 # Define a zero-coupon bond that pays $1 in 3 years
1422 zcbond_10 = pd.Series(data=[1], index=[3])
1423
1424
1425 # Compute the present value of the zero-coupon bond at a 3% interest rate
1426 zc_0 = pv(zcbond_10, 0.03)
1427
1428 # Determine how many bonds can be purchased
1429 n_bonds = a_0 / zc_0
1430
1431 # Asset value assuming we buy bonds
1432 av_zc_bonds = n_bonds * bond_prices
1433
```

```
# Asset value assuming we invest in cash
av_cash = a_0 * (rates / 12 + 1).cumprod()

# Compute the terminal funding ratio (TFR) for cash and bonds
tfr_cash = av_cash.iloc[-1] / liabilities.iloc[-1]
tfr_zc_bonds = av_zc_bonds.iloc[-1] / liabilities.iloc[-1]

# Plot the histogram of the terminal funding ratios
ax = tfr_cash.plot.hist(label="Cash", figsize=(15,6), bins=100, legend=True)
tfr_zc_bonds.plot.hist(ax=ax, label="ZC Bonds", bins=100, legend=True, secondary_y=True)
plt.show()

#R4. Assume that your liability is in Cash. Use the previous 5 scenarios you defined in R3 for the CIR modelling, and compute the funding ratio. Provide an analysis on how to use CIR to model liabilities and the funding ratio
#Configuration 1

# Initial cash on hand (0.75 million)
a_0 = 0.75

# Simulate interest rates and bond prices using the CIR model
rates_1, bond_prices_1 = cir(a=0.7, b=0.03, sigma=0.02, n_scenarios=10000)

# Assume liabilities are the bond prices
liabilities_1 = bond_prices_1

# Define a zero-coupon bond that pays $1 in 10 years
zcbond_10 = pd.Series(data=[1], index=[10])

# Compute the present value of the zero-coupon bond at a 3% interest rate
zc_0 = pv(zcbond_10, 0.03)

# Determine how many bonds can be purchased
n_bonds = a_0 / zc_0

# Asset value assuming we buy bonds
av_zc_bonds_1 = n_bonds * bond_prices_1

# Asset value assuming we invest in cash
av_cash_1 = a_0 * (rates_1 / 12 + 1).cumprod()

# Compute the terminal funding ratio (TFR) for cash and bonds
tfr_cash_1 = av_cash_1.iloc[-1] / liabilities_1.iloc[-1]
tfr_zc_bonds_1 = av_zc_bonds_1.iloc[-1] / liabilities_1.iloc[-1]

# Plot the histogram of the terminal funding ratios
ax = tfr_cash_1.plot.hist(label="Cash", figsize=(15,6), bins=100, legend=True)
tfr_zc_bonds_1.plot.hist(ax=ax, label="ZC Bonds", bins=100, legend=True, secondary_y=True)
plt.title("Funding Ratio Distribution - High Mean Reversion & Low Volatility")
plt.xlabel("Funding Ratio")
plt.show()

#Configuration 2
```

```
1485
1486 # Initial cash on hand (0.75 million)
1487 a_0 = 0.75
1488
1489 # Simulate interest rates and bond prices using the CIR model
1490 rates_2, bond_prices_2 = cir(a=0.1, b=0.03, sigma=0.05, n_scenarios=10000)
1491
1492 # Assume liabilities are the bond prices
1493 liabilities_2 = bond_prices_2
1494
1495 # Define a zero-coupon bond that pays $1 in 10 years
1496 zcbond_10 = pd.Series(data=[1], index=[10])
1497
1498 # Compute the present value of the zero-coupon bond at a 3% interest rate
1499 zc_0 = pv(zcbond_10, 0.03)
1500
1501 # Determine how many bonds can be purchased
1502 n_bonds = a_0 / zc_0
1503
1504 # Asset value assuming we buy bonds
1505 av_zc_bonds_2 = n_bonds * bond_prices_2
1506
1507 # Asset value assuming we invest in cash
1508 av_cash_2 = a_0 * (rates_2 / 12 + 1).cumprod()
1509
1510 # Compute the terminal funding ratio (TFR) for cash and bonds
1511 tfr_cash_2 = av_cash_2.iloc[-1] / liabilities_2.iloc[-1]
1512 tfr_zc_bonds_2 = av_zc_bonds_2.iloc[-1] / liabilities_2.iloc[-1]
1513
1514 # Plot the histogram of the terminal funding ratios
1515 ax = tfr_cash_2.plot.hist(label="Cash", figsize=(15,6), bins=100, legend=True)
1516 tfr_zc_bonds_2.plot.hist(ax=ax, label="ZC Bonds", bins=100, legend=True,
    secondary_y=True)
1517 plt.title("Funding Ratio Distribution - Low Mean Reversion & High Volatility")
1518 plt.xlabel("Funding Ratio")
1519 plt.show()
1520
1521 #Configuration 3
1522
1523 # Initial cash on hand (0.75 million)
1524 a_0 = 0.75
1525
1526 # Simulate interest rates and bond prices using the CIR model
1527 rates_3, bond_prices_3 = cir(a=0.5, b=0.03, sigma=0.1, n_scenarios=10000)
1528
1529 # Assume liabilities are the bond prices
1530 liabilities_3 = bond_prices_3
1531
1532 # Define a zero-coupon bond that pays $1 in 10 years
1533 zcbond_10 = pd.Series(data=[1], index=[10])
1534
1535 # Compute the present value of the zero-coupon bond at a 3% interest rate
1536 zc_0 = pv(zcbond_10, 0.03)
1537
```

```
# Determine how many bonds can be purchased
n_bonds = a_0 / zc_0

# Asset value assuming we buy bonds
av_zc_bonds_3 = n_bonds * bond_prices_3

# Asset value assuming we invest in cash
av_cash_3 = a_0 * (rates_3 / 12 + 1).cumprod()

# Compute the terminal funding ratio (TFR) for cash and bonds
tfr_cash_3 = av_cash_3.iloc[-1] / liabilities_3.iloc[-1]
tfr_zc_bonds_3 = av_zc_bonds_3.iloc[-1] / liabilities_3.iloc[-1]

# Plot the histogram of the terminal funding ratios
ax = tfr_cash_3.plot.hist(label="Cash", figsize=(15,6), bins=100, legend=True)
tfr_zc_bonds_3.plot.hist(ax=ax, label="ZC Bonds", bins=100, legend=True,
secondary_y=True)
plt.title("Funding Ratio Distribution - Moderate Mean Reversion & High Volatility")
plt.xlabel("Funding Ratio")
plt.show()

# Configuration 4

# Initial cash on hand (0.75 million)
a_0 = 0.75

# Simulate interest rates and bond prices using the CIR model
rates_4, bond_prices_4 = cir(a=0.5, b=0.03, sigma=0.2, n_scenarios=10000)

# Assume liabilities are the bond prices
liabilities_4 = bond_prices_4

# Define a zero-coupon bond that pays $1 in 10 years
zcbond_10 = pd.Series(data=[1], index=[10])

# Compute the present value of the zero-coupon bond at a 3% interest rate
zc_0 = pv(zcbond_10, 0.03)

# Determine how many bonds can be purchased
n_bonds = a_0 / zc_0

# Asset value assuming we buy bonds
av_zc_bonds_4 = n_bonds * bond_prices_4

# Asset value assuming we invest in cash
av_cash_4 = a_0 * (rates_4 / 12 + 1).cumprod()

# Compute the terminal funding ratio (TFR) for cash and bonds
tfr_cash_4 = av_cash_4.iloc[-1] / liabilities_4.iloc[-1]
tfr_zc_bonds_4 = av_zc_bonds_4.iloc[-1] / liabilities_4.iloc[-1]

# Plot the histogram of the terminal funding ratios
ax = tfr_cash_4.plot.hist(label="Cash", figsize=(15,6), bins=100, legend=True)
```

```
tfr_zc_bonds_4.plot.hist(ax=ax, label="ZC Bonds", bins=100, legend=True, secondary_y=True)
plt.title("Funding Ratio Distribution - Very High Volatility")
plt.xlabel("Funding Ratio")
plt.show()

#Configuration 5

# Initial cash on hand (0.75 million)
a_0 = 0.75

# Simulate interest rates and bond prices using the CIR model
rates_5, bond_prices_5 = cir(a=0.1, b=0.03, sigma=0.02, n_scenarios=10000)

# Assume liabilities are the bond prices
liabilities_5 = bond_prices_5

# Define a zero-coupon bond that pays $1 in 10 years
zcbond_10 = pd.Series(data=[1], index=[10])

# Compute the present value of the zero-coupon bond at a 3% interest rate
zc_0 = pv(zcbond_10, 0.03)

# Determine how many bonds can be purchased
n_bonds = a_0 / zc_0

# Asset value assuming we buy bonds
av_zc_bonds_5 = n_bonds * bond_prices_5

# Asset value assuming we invest in cash
av_cash_5 = a_0 * (rates_5 / 12 + 1).cumprod()

# Compute the terminal funding ratio (TFR) for cash and bonds
tfr_cash_5 = av_cash_5.iloc[-1] / liabilities_5.iloc[-1]
tfr_zc_bonds_5 = av_zc_bonds_5.iloc[-1] / liabilities_5.iloc[-1]

# Plot the histogram of the terminal funding ratios
ax = tfr_cash_5.plot.hist(label="Cash", figsize=(15,6), bins=100, legend=True)
tfr_zc_bonds_5.plot.hist(ax=ax, label="ZC Bonds", bins=100, legend=True, secondary_y=True)
plt.title("Funding Ratio Distribution - Low Mean Reversion & Low Volatility")
plt.xlabel("Funding Ratio")
plt.show()

# 5 GHP and Duration Matching

#E16. Implement a function to compute the returns of a bond. Then implement a fucntion to use these cashflows to compute the price of the bond

def bond_cash_flows(maturity, principal=100, coupon_rate=0.03, coupons_per_year=12):
    """
    Returns a series of cash flows generated by a bond, indexed by coupon number.
    """
    # Calculate number of coupons
```

```
    n_coupons = round(maturity * coupons_per_year)  # Total number of coupon payments

    # Calculate the coupon amount
    coupon_amt = principal * coupon_rate / coupons_per_year  # Amount paid at each coupon date

    # Generate the times of coupon payments
    coupon_times = np.arange(1, n_coupons + 1)

    # Create the cash flows (all coupon payments)
    cash_flows = pd.Series(data=coupon_amt, index=coupon_times)

    # Add the principal amount to the final cash flow (maturity)
    cash_flows.iloc[-1] += principal  # The final payment includes the principal

    return cash_flows

#To get cash flows for a 3-year bond with a 3% coupon rate paid semiannually
bond_cash_flows(3, 100, 0.03, 2)

#The bond_price function calculates the bond's price based on its cash flows and the discount rate. The price is the present value of the cash flows.

def bond_price(maturity, principal=100, coupon_rate=0.03, coupons_per_year=12, discount_rate=0.03):
    """
    Price a bond based on bond parameters and discount rate.
    """
    # Obtain the bond's cash flows
    cash_flows = bond_cash_flows(maturity, principal, coupon_rate, coupons_per_year)

    # Discount rate per period
    period_rate = discount_rate / coupons_per_year  # Monthly discount rate

    # Present value of the cash flows
    pv_cash_flows = cash_flows / (1 + period_rate) ** cash_flows.index

    # Sum the present values of all cash flows to get the bond price
    bond_price = pv_cash_flows.sum()

    return bond_price

#Calculate the price of a 20-year bond with a 5% coupon rate and a 4% discount rate
bond_price(20, 1000, 0.05, 2, 0.04)

#Now, assume we have the following rates and compute the bonds with these rates. How can you interpret the resulting plot?

rates=np.linspace(0.01,.1,num=20)
rates

prices= [bond_price(10,1000,0.05,2,rate) for rate in rates]
pd.DataFrame(data=prices,index=rates).plot(title="Prices 10y Bond with different Interest Rate", legend=False)
```

```

#The problem with these bonds is the intermediate cash flows. Remember that a bond with coupons are multiple
# zero coupon bonds together, and the problem is that some of those have a short time term.

cf=bond_cash_flows(3,1000,.06,2)
cf

# We have cashflows, but it's better to get 30 today than 3 years in the future, right? Thus, how long are we waiting until we get the cashflows? We could compute the weighted average time.

discounts=discount(cf.index,0.06/2)
discounts

#These are the discount factors. Now, we discounted values:
dcf= cf*discounts #Multiply cashflows by the discount factors
dcf

#This is the discounted values of the present values for the cashflows. Now, we can get the weights.

# Calculate the sum of the discounted cash flows
total_dcf = dcf.sum()  # Total present value of all cash flows

# Calculate the weights for each cash flow
weights = dcf / total_dcf  # Weight is the proportion of each discounted cash flow to the total discounted value
print("Weights of the cash flows:")
print(weights)

# Finally, we make a weighted average.
(cf.index*weights).sum()

#E17. Implement a function that computes the Macaulay Duration

import numpy as np

def macaulay_duration(flows, discount_rate):
    """
    Computes the Macaulay Duration of a bond.

    Arguments:
    - flows: pandas Series containing the bond's cash flows (index = time)
    - discount_rate: the periodic discount rate (decimal), NOT annual!

    Returns:
    - Macaulay Duration: weighted average time until cash flows are received.
    """
    # **Paso 1**: Calcular factores de descuento
    discounts = (1 + discount_rate) ** (-flows.index)  # Descuento aplicado correctamente

    # **Paso 2**: Calcular los flujos descontados (DCF)
```

```
    dcf = flows * discounts  # Multiplicamos cash flows por factores de descuento

    # **Paso 3**: Obtener la suma total de los flujos descontados
    total_dcf = dcf.sum()  # Total de los flujos descontados

    # **Paso 4**: Calcular los pesos (proporción de cada flujo descontado)
    weights = dcf / total_dcf  # Normalizamos los flujos descontados

    # **Paso 5**: Calcular la duración de Macaulay como media ponderada
    duration = np.dot(flows.index, weights)  # Media ponderada usando producto punto

    return duration

macaulay_duration(bond_cash_flows(3,1000,0.06,2), 0.06/2)

#If a zero-coupon bond does not match the maturity of our liabilities, we can use two bonds with different maturities to achieve the desired duration through weighted allocation.
#define liabilities
liabilities = pd.Series(data=[100000, 100000], index=[10,12])

#Now, we define bonds with different maturities and calculate their Macaulay durations.
md_10 = macaulay_duration(bond_cash_flows(10, 1000, 0.05, 1), 0.04)
md_20 = macaulay_duration(bond_cash_flows(20, 1000, 0.05, 1), 0.04)
md_10, md_20

#E18.Implement the match duration function to determine the weights of each bond required to match the liability duration

def match_duration(cf_t, cf_s, cf_l, discount_rate):
    """
    Returns the weight W in cf_s to match duration for target cash flows cf_t.

    Arguments:
    - cf_t: Target cash flows (liabilities)
    - cf_s: Cash flows of the short-duration bond
    - cf_l: Cash flows of the long-duration bond
    - discount_rate: Discount rate for duration calculation

    Returns:
    - W: Weight of the short-duration bond needed to match liability duration
    """
    d_t = macaulay_duration(cf_t, discount_rate)  # Duration of target liabilities
    d_s = macaulay_duration(cf_s, discount_rate)  # Duration of short bond
    d_l = macaulay_duration(cf_l, discount_rate)  # Duration of long bond

    # Compute the weight W for the short-duration bond
    W = (d_l - d_t) / (d_l - d_s)

    return W  # Return the computed weight

# Define the liabilities (payments at year 10 and 12)
liabilities = pd.Series(data=[100000, 100000], index=[10, 11])  # Payments of 100,000 at years 10 and 12
```

```
1788
1789 # Define the short and long bonds
1790 short_bond = bond_cash_flows(10, 1000, 0.05, 1)  # Short bond (10 years, 5% coupon,
     annual)
1791 long_bond = bond_cash_flows(20, 1000, 0.05, 1)   # Long bond (20 years, 5% coupon,
     annual)
1792
1793 # Define the discount rate
1794 discount_rate = 0.05  # 5% discount rate
1795
1796 # Compute the weight for the short bond using match_duration
1797 w_s = match_duration(liabilities, short_bond, long_bond, discount_rate)  # Weight for
     short bond
1798 w_l = 1 - w_s  # Weight for the long bond
1799
1800 # Print weights
1801 print(f"Weight of Short Bond: {w_s:.4f}")
1802 print(f"Weight of Long Bond: {w_l:.4f}")
1803
1804
1805
1806
1807
1808 price_short = bond_price(10, 1000, 0.05, 1, 0.04)
1809 price_long = bond_price(20, 1000, 0.05, 1, 0.04)
1810 a_0 = 130000  # Initial assets value, assume 130000
1811 portfolio_flows = pd.concat([a_0 * w_s * short_bond / price_short, a_0 * w_l * long_bond
     / price_long])
1812 macaulay_duration(portfolio_flows,0.04)
1813
1814
1815 #Now, let's compute the funding ratio
1816 def funding_ratio (assets,liabilities,r_a,r_l):
1817   """
1818     Computes the funding ratio of some given liabiities and interest rate.
1819     """
1820   return pv(assets,r_a)/pv(liabilities,r_l)
1821
1822 funding_ratio(portfolio_flows,liabilities,0.04,0.04)
1823
1824
1825
1826 #R5. Explain GHP and duration matching in liability driven investing. To assess the
     sensitivity of the funding ratio to changes in interest rates, calculate the funding
     ratios for a range of rates. Consider 20 rates between 0 and 0.1 using linspace(),
     consider your previous long and short bonds and prices. Then, compute the funding ratio
     for the long bond using the 20 rates.
1827 #Compute the funding ratio for the short bond with the rates. Finally, use the previous
     portfolio_flows and liabilities to compute the funding ratio with the rates. Plot these 3
     series and explain what you see. Assess the sensitivity of the funding ratio to changes
     in interest rates.
1828
1829 # Define the rates from 1% to 10%
1830 rates = np.linspace(0.01, 0.1, num=20)
1831
```

```
# Compute funding ratio for long bond
funding_ratios_long = [funding_ratio(long_bond, liabilities, rate, rate) for rate in rates]

# Compute funding ratio for short bond
funding_ratios_short = [funding_ratio(short_bond, liabilities, rate, rate) for rate in rates]

# Compute funding ratio for the matched portfolio (using portfolio_flows and liabilities)
funding_ratios_portfolio = [funding_ratio(portfolio_flows, liabilities, rate, rate) for rate in rates]

# Plot Funding Ratio for Long Bond
plt.figure(figsize=(8, 5))
plt.plot(rates, funding_ratios_long, label="Funding Ratio (Long Bond)", linestyle="--", color="blue")
plt.xlabel("Interest Rate")
plt.ylabel("Funding Ratio")
plt.title("Funding Ratio Sensitivity (Long Bond)")
plt.legend()
plt.grid(True)
plt.show()

# Plot Funding Ratio for Short Bond
plt.figure(figsize=(8, 5))
plt.plot(rates, funding_ratios_short, label="Funding Ratio (Short Bond)", linestyle="-.", color="red")
plt.xlabel("Interest Rate")
plt.ylabel("Funding Ratio")
plt.title("Funding Ratio Sensitivity (Short Bond)")
plt.legend()
plt.grid(True)
plt.show()

# Plot Funding Ratio for Matched Portfolio (with portfolio_flows)
plt.figure(figsize=(8, 5))
plt.plot(rates, funding_ratios_portfolio, label="Funding Ratio (Matched Portfolio)", linewidth=2, color="green")
plt.xlabel("Interest Rate")
plt.ylabel("Funding Ratio")
plt.title("Funding Ratio Sensitivity (Matched Portfolio)")
plt.legend()
plt.grid(True)
plt.show()


# 6 Simulation of Prices of Coupon Bonds using CIR

def discount (t,r):
  """
    Compute the price of a pure discounte bond that pays a dollar at time t, g
  iven an interest rate r. Returns a |t|x|r| series or data frame r can be a flo
  at, series or data frame
    returns a dataframe indexed by t
```

```
    """
  discounts=pd.DataFrame([(r+1)**-i for i in t])
  discounts.index=t
  return (discounts)

def pv(flows,r):
  """
    Computes PV of a set of liabilities
    flows is indexed by the time and amounts
    r can be a scalar, a series, or a dataframe with the number of rows matchi
  ng the num of rows in flows
    """
  dates= flows.index
  discounts= discount(dates,r)
  return discounts.multiply(flows, axis='rows').sum()

bond_price(5,100,0.05,12,0.03)

#simulate rates and zc prices using CIR
rates, zc_prices=cir(10,500,b=0.03,r_0=0.03)
#bond prices
selected_rates = rates.iloc[0, [1, 2, 3]]
for rate in selected_rates:
    price = bond_price(5, 100, 0.05, 12, rate)  # Calcular precio del bono para cada tasa
    print(f"Bond price with rate {rate}: {price}")

selected_rates = rates.iloc[1, [1, 2, 3]]
for rate in selected_rates:
    price = bond_price(5, 100, 0.05, 12, rate)  # Calcular precio del bono para cada tasa
    print(f"Bond price with rate {rate}: {price}")

rates[[1,2,3]].head()

#The bond_price function calculates the price of a bond that pays coupons until maturity,
with the principal returned at maturity. The function can handle a discount_rate input as
a DataFrame to simulate varying rates over time

def bond_price(maturity, principal=100, coupon_rate=0.03, coupons_per_year=12,
discount_rate=0.03):
    """
    Computes the price of a bond that pays coupons until maturity.
    """
    if isinstance(discount_rate, pd.DataFrame):
        pricing_dates = discount_rate.index
        prices = pd.DataFrame(index=pricing_dates, columns=discount_rate.columns)
        for t in pricing_dates:
            prices.loc[t] = bond_price(maturity - t / coupons_per_year, principal,
coupon_rate, coupons_per_year, discount_rate.loc[t])
        return prices
    else:
        if maturity <= 0:
            return principal + principal * coupon_rate / coupons_per_year
        cash_flows = bond_cash_flows(maturity, principal, coupon_rate, coupons_per_year)
        return pv(cash_flows, discount_rate / coupons_per_year)
```

```
1930
1931 #Using the simulated interest rates, we calculate bond prices over time with the
     bond_price function. The price evolution shows how interest rate fluctuations affect bond
     prices.
1932 bond_price(10,100,0.05,12,rates[[1,2,3,4,5]]).plot(legend=False, figsize=(12,6))
1933 plt.show()
1934
1935 #To evaluate bond performance, we calculate the annualized bond returns based on
     percentage changes in bond prices. This reveals how bond returns respond to interest rate
     changes
1936
1937 def annualize_rets(r, periods_per_year):
1938     """
1939     Annualizes a set of periodic returns.
1940
1941     Parameters:
1942     - r: pd.Series or pd.DataFrame of returns
1943     - periods_per_year: Number of compounding periods per year (e.g., 12 for monthly, 252
     for daily)
1944
1945     Returns:
1946     - Annualized return
1947     """
1948     compounded_growth = (1 + r).prod()  # Cumulative growth
1949     n_periods = len(r)  # Number of periods in the return series
1950     return compounded_growth ** (periods_per_year / n_periods) - 1  # Annualized return
     formula
1951
1952 prices=bond_price(10,100,0.05,12,rates[[1,2,3,4,5]])
1953 br = prices.pct_change().dropna()
1954 annualize_rets(br, 12)
1955
1956 #E19. Compute the bond total returns
1957
1958
1959 def bond_total_return(monthly_prices, principal, coupon_rate, coupons_per_year):
1960     """
1961     Computes the total return of a bond, including coupon payments.
1962
1963     Parameters:
1964     - monthly_prices: DataFrame of monthly bond prices (each column is a different
     scenario)
1965     - principal: Bond principal (face value)
1966     - coupon_rate: Annual coupon rate of the bond
1967     - coupons_per_year: Number of coupon payments per year
1968
1969     Returns:
1970     - DataFrame of total returns, accounting for both price changes and coupon payments.
1971     """
1972     # Initialize DataFrame for coupons, matching the structure of monthly_prices
1973     coupons = pd.DataFrame(0, index=monthly_prices.index, columns=monthly_prices.columns)
1974     t_max = monthly_prices.index.max()
1975
1976     # Define payment dates (in terms of months) for the coupons
```

```
    pay_dates = np.linspace(12 / coupons_per_year, t_max, int(coupons_per_year * t_max / 12), dtype=int)

    # Populate coupon payments at the defined payment dates
    coupons.iloc[pay_dates] = principal * coupon_rate / coupons_per_year

    # Calculate total returns by adding the coupon payments to the bond price (monthly)
    price_changes = monthly_prices.pct_change()  # Price changes (percentage return)
    total_returns = price_changes + (coupons / monthly_prices)  # Add coupon yield

    # Drop the initial row since it has no previous price for return calculation
    return total_returns.dropna()

p=bond_price(10,100,0.05,12,rates[[1,2,3,4,]])
btr=bond_total_return(p,100,0.05,12)
annualize_rets(btr,12)

# As we can see, bond prices change over time. The dynamics of interest rates is affecting us

price_10=bond_price(10,100,0.05,12,rates)
price_10[[1,2,3]].tail()

# Assume bond_price function is defined as before, and rates for 10 years have been generated

# Let's extend the rates for 30 years (as an example, we repeat the last 10 years of rates for simplicity)
extended_rates = rates.iloc[:, :10].reindex(range(30), method='ffill')  # Repeat the rates for 30 years

# Now we can calculate the price for 30-year bonds using the extended rates
price_30 = bond_price(30, 100, 0.05, 12, extended_rates)

# Display the last few rows for the scenarios 1, 2, 3 to see how the price converges
print(price_30[[1, 2, 3]].tail())

#So, bonds tend to be tought as safe,but they are not, this is clear when the maturity is long.
#Let's make a portfolio with the two bonds. Use a 60/40 combination using the 10 and 30 year bonds'

rets_30 = bond_total_return(price_30, 100, 0.05, 12)
rets_10 = bond_total_return(price_10, 100, 0.05, 12)

# Rebalance mensual para mantener una asignación 60/40:
rets_bonds = 0.6 * rets_10 + 0.4 * rets_30

# Calcular los rendimientos medios para cada mes
mean_rets_bonds = rets_bonds.mean(axis=1)  # Promedio para cada fila (mes)

# Asegurarnos de que los valores sean numéricos
mean_rets_bonds = pd.to_numeric(mean_rets_bonds, errors='coerce')

```

```
# Llamar a la función summary_stats con los rendimientos medios
summary_stats(mean_rets_bonds)

#R6. Explain how CIR is used for modelling coupon bonds, and give empirical evidence that bonds are not safe at all using a 60/40 composition of bonds (you can use your previous results). Then, use GBM to simulate equities for 10 years, in 500 scenarios with a mean of 0.07 and std of 0.15. Once you have the simulation of equities, convert them to returns, i.e. percentual changes (if you use pct_change() remember to use dropna() to delete the first na element). Once you have the returns of the equities combine them with the previous returns of bonds, rets_bonds, in a portfolio that has a 0.7/0.3 split (equities/bonds). Compute the mean of the 70/30 portfolio returns. Compute the risk-return statistics and analyze the results. Compare them with the bonds returns alone.¶



# Function to calculate summary statistics (mean, volatility, skewness, kurtosis, VaR, CVar, etc.)
def summary_stats(returns):
    # Calculate summary statistics
    mean_return = returns.mean()
    vol = returns.std()
    skew = returns.skew()
    kurt = returns.kurt()

    # Historical VaR (5%)
    VaR_5 = np.percentile(returns, 5)

    # CVar (5%)
    below_VaR = returns[returns <= VaR_5]
    CVar_5 = below_VaR.mean() if len(below_VaR) > 0 else np.nan

    # Sharpe ratio
    sharpe_ratio = mean_return / vol if vol != 0 else np.nan

    # Max Drawdown
    cumulative_returns = (1 + returns).cumprod()
    peak = cumulative_returns.cummax()
    drawdown = (cumulative_returns - peak) / peak
    max_drawdown = drawdown.min()

    # Summary of statistics
    stats = pd.Series({
        'Mean Return': mean_return,
        'Volatility': vol,
        'Skewness': skew,
        'Kurtosis': kurt,
        'VaR (5%)': VaR_5,
        'CVar (5%)': CVar_5,
        'Sharpe Ratio': sharpe_ratio,
        'Max Drawdown': max_drawdown
    })

    return stats

# 1. Simulate Equities using GBM
```

```
2070 np.random.seed(42)  # For reproducibility
2071
2072 # GBM parameters for equities
2073 mu = 0.07  # Mean return of stocks
2074 sigma = 0.15  # Volatility of stocks
2075 T = 10  # Time horizon in years
2076 n_scenarios = 500  # Number of simulations
2077 n_steps = T * 12  # Monthly steps for 10 years (12 months per year)
2078
2079 # Simulate equity prices
2080 dt = 1 / 12  # Monthly time step
2081 equity_prices = np.zeros((n_scenarios, n_steps))
2082 equity_prices[:, 0] = 100  # Initial stock price
2083
2084 for i in range(1, n_steps):
2085     z = np.random.normal(size=n_scenarios)  # Random normal variables
2086     equity_prices[:, i] = equity_prices[:, i-1] * np.exp((mu - 0.5 * sigma**2) * dt +
     sigma * np.sqrt(dt) * z)
2087
2088 # Convert prices to returns (percent changes)
2089 equity_returns =
     pd.DataFrame(equity_prices).pct_change(axis='columns').dropna(axis='columns')
2090
2091 # 2. Combine Bond Returns (rets_bonds) with Equities in a 70/30 Portfolio
2092 # Assuming rets_bonds is a previously calculated series of bond returns
2093 rets_bonds = pd.DataFrame(np.random.normal(0.03, 0.04, size=(500, 120)))  # Simulated
     bond returns as an example
2094
2095 # Combine the bond (30%) and equity (70%) returns
2096 portfolio_returns = 0.7 * equity_returns + 0.3 * rets_bonds
2097
2098 # 3. Calculate risk-return statistics for the 70/30 Portfolio
2099 portfolio_stats = summary_stats(portfolio_returns.mean(axis=1))
2100
2101 # 4. Calculate statistics for Bonds only (60/40 Portfolio)
2102 bond_stats = summary_stats(rets_bonds.mean(axis=1))
2103
2104 # 5. Print the full statistics
2105 print("70/30 Portfolio Statistics (Equities + Bonds):")
2106 print(portfolio_stats)
2107
2108 print("\n60/40 Portfolio Statistics (Bonds Only):")
2109 print(bond_stats)
2110
2111 # 6. Plot the results
2112 plt.figure(figsize=(10, 6))
2113 plt.hist(portfolio_returns.mean(axis=1), bins=30, alpha=0.7, label="70/30 Portfolio")
2114 plt.hist(rets_bonds.mean(axis=1), bins=30, alpha=0.7, label="60/40 Bonds")
2115 plt.legend()
2116 plt.title("Mean Return Distribution: Portfolio vs Bonds")
2117 plt.xlabel("Mean Return")
2118 plt.ylabel("Frequency")
2119 plt.show()
2120
```

```
#7 Naive Risk Budgeting Strategies between the PSP and GHP

#allocator is a free function to allocate that the user gives. **kwargs allows us to take the function and whichever variable within.
def bt_mix(r1,r2, allocator, **kwargs):
    """
    Runs a back test (simulation) of allocating between a two sets of returns
    r1 and r2 are TxN DataFrames or returns where T is the time step and N is the number of scenarios.
    allocator is a function that takes two set of returns and allocator specific parameters, and produces
    an allocation tot the first portfolio (the rest of the money is invested in the GHP) as a T x 1 DataFrame
    Returns a TxN DataFrame of the resulting N portfolio scenarios.
    """
    if not r1.shape== r2.shape:
        raise ValueError("r1 and r2 need to have the same shape")
    weights=allocator(r1,r2, **kwargs) #The allocator takes r1, r2 and a bunch of other variables
    if not weights.shape==r1.shape:
        raise ValueError("Allocator results not matching r1 shape")
    r_mix=weights*r1+(1-weights)*r2
    return r_mix

#The following function produces a time series over T steps of allocation between the PSP and the GHP across N scenarios.

def fixedmix_allocator(r1,r2, w1, **kwargs):
    """
    Produces a time series over T steps of allocation between the PSP
    and the GHP across N scenarios.
    PSP and GHP are TxN DataFrames that represent the returns of the PSP and GHP such that:
    each column is a scenario
    each row is the price for a timestep
    returns an TxN DataFrame of PSP Weights
    """
    return pd.DataFrame(data=w1, index=r1.index, columns=r1.columns)

#The following function is the same as before to compute the total bond return.

def bond_total_return(monthly_prices, principal, coupon_rate,coupons_per_year):
    """
    Computes the total return of a bond on monthly bond prices and coupon payments
    Assumes that dividends (coupons) are paid out at the end of the period (e.g. end of 3 months for quaterly div) and that dividens are reinvested in the bond
    """
    coupons=pd.DataFrame(data=0, index=monthly_prices.index, columns=monthly_prices.columns)
    t_max=monthly_prices.index.max()
    pay_date=np.linspace(12/coupons_per_year,t_max, int(coupons_per_year*t_max/12), dtype=int)
    coupons.iloc[pay_date]=principal*coupon_rate/coupons_per_year
    total_returns=(monthly_prices+coupons)/monthly_prices.shift()-1
    return total_returns.dropna()
```

```
2166
2167 #Now, we obtain the returns of bonds.
2168
2169 #Use the appropriate functions to simulate the expected results
2170 rates, zc_prices = cir(n_years=10, n_scenarios=500, b=0.03, r_0=0.03)
2171 price_10 = bond_price(10, 100, 0.05, 12, rates)
2172 price_30 = bond_price(30, 100, 0.05, 12, rates)  # Usar rates extendidos para 30 años
2173 rets_10 = bond_total_return(price_10, 100, 0.05, 12)
2174 rets_30 = bond_total_return(price_30, 100, 0.05, 12)
2175
2176 # Usar el backtester con una asignación de 0.6 para el bono a 10 años
2177 rets_bonds = bt_mix(rets_10, rets_30, allocator=fixedmix_allocator, w1=0.6)
2178
2179 mean_rets_bonds=rets_bonds.mean(axis="columns")
2180 summary_stats(mean_rets_bonds)
2181
2182 #Again, we are going to generate equity returns and bond returns and mix them. Same as
     before.
2183
2184 def gbm(n_years=10, n_scenarios=1000, mu=0.07,sigma=0.15,steps_per_year=12,
     s_0=100.0,prices=True):
2185     """
2186     Evolution of a Stock Price using GBM
2187     """
2188     dt=1/steps_per_year
2189     n_steps=int(n_years*steps_per_year)
2190
2191     if prices:
2192         rets_plus_1=np.random.normal(loc=1+mu*dt,scale=sigma*np.sqrt(dt),size=
     (n_steps+1,n_scenarios)) #loc is the mean, and scale the std
2193         rets_plus_1[0]=1
2194         prices=s_0*pd.DataFrame(rets_plus_1).cumprod()
2195         return prices
2196     else:
2197         rets=np.random.normal(loc=mu*dt,scale=sigma*np.sqrt(dt),size=
     (n_steps+1,n_scenarios))
2198         return rets
2199
2200 price_eq=gbm(n_years=10,n_scenarios=500,mu=0.07,sigma=0.15)
2201 rets_eq=price_eq.pct_change().dropna()
2202 rets_zc=zc_prices.pct_change().dropna()
2203
2204 #E21. Use the backtester mixer to mix the returns of the equities and bonds using the
     fixedmixallocator with a weight w1 of 0.7. This returns a 70/30 portfolio.
2205 # Then compute the mean of these returns using mean(axis=1). Apply the summary
     statistics. Notice that this first approach generates a time series of the average
     results and then computes the statistics. Now, take a different approach, first compute
     the statistics of the returns of the 70/30 and then take the average using mean().
     Compare the results of the two approaches. Now, repeat the same using an allocation og
     60/40.
2206 # Compare the results, which annualized return is greater? What are your insights about
     the results?
2207
2208 # 70/30 Allocation: 70% in equities and 30% in bonds
2209 rets_70_30 = bt_mix(rets_eq, rets_30, allocator=fixedmix_allocator, w1=0.7)
```

```
2210
2211 # Calculate the average returns for the 70/30 portfolio (averaged over all scenarios)
2212 mean_rets_70_30 = rets_70_30.mean(axis=1)
2213
2214 # Calculate summary statistics for each scenario in the 70/30 portfolio
2215 summary_stats_70_30_full = rets_70_30.apply(summary_stats, axis=0)
2216
2217 # Average the statistics from each scenario to get an overall view of the portfolio
2218 summary_stats_70_30_avg = summary_stats_70_30_full.mean(axis=1)
2219
2220 # 60/40 Allocation: 60% in equities and 40% in bonds
2221 rets_60_40 = bt_mix(rets_eq, rets_30, allocator=fixedmix_allocator, w1=0.6)
2222
2223 # Calculate the average returns for the 60/40 portfolio (averaged over all scenarios)
2224 mean_rets_60_40 = rets_60_40.mean(axis=1)
2225
2226 # Calculate summary statistics for each scenario in the 60/40 portfolio
2227 summary_stats_60_40_full = rets_60_40.apply(summary_stats, axis=0)
2228
2229 # Average the statistics from each scenario to get an overall view of the portfolio
2230 summary_stats_60_40_avg = summary_stats_60_40_full.mean(axis=1)
2231
2232 # Compare the annualized returns for 70/30 and 60/40, both with and without averaging the
     returns
2233 annualized_return_70_30_avg = (1 + mean_rets_70_30.mean())**12 - 1
2234 annualized_return_70_30_full = (1 + rets_70_30.mean().mean())**12 - 1
2235
2236 annualized_return_60_40_avg = (1 + mean_rets_60_40.mean())**12 - 1
2237 annualized_return_60_40_full = (1 + rets_60_40.mean().mean())**12 - 1
2238
2239 # Print the summary statistics for both portfolios
2240 print("Summary statistics for the 70/30 portfolio with averaged returns:")
2241 print(summary_stats_70_30_avg)
2242
2243 print("\nSummary statistics for the 70/30 portfolio without averaging returns:")
2244 print(summary_stats_70_30_full)
2245
2246 print("\nSummary statistics for the 60/40 portfolio with averaged returns:")
2247 print(summary_stats_60_40_avg)
2248
2249 print("\nSummary statistics for the 60/40 portfolio without averaging returns:")
2250 print(summary_stats_60_40_full)
2251
2252 # Comparison of the annualized returns
2253 print("\nAnnualized return (70/30, averaged returns):", annualized_return_70_30_avg)
2254 print("Annualized return (70/30, without averaging returns):", annualized_return_70-
     _30_full)
2255
2256 print("\nAnnualized return (60/40, averaged returns):", annualized_return_60_40_avg)
2257 print("Annualized return (60/40, without averaging returns):", annualized_return_60-
     _40_full)
2258
2259 #Now, we look at the terminal values. The following function returns the final values of
     a dollar at the end of the return period for each scenario
```

```
2260
2261 def terminal_values(rets):
2262     """
2263     Returns the final values of a dollar at the end of the return period for each
     scenario.
2264     """
2265     return (rets + 1).prod()
2266
2267 def terminal_stats(rets, floor=0.8, cap=np.inf, name="Stats"):
2268     """
2269     Produce Summary Statistics on the terminal value per invested dollar across a range
     of N scenarios.
2270     rets is a TxN DataFrame of returns, where T is the time_step (we assume rets is
     sorted by time).
2271     Returns a 1-column DataFrame of Summary Stats indexed by the stat name.
2272     """
2273     terminal_wealth = terminal_values(rets)  # Cumulative returns: (1 + returns).prod()
     across time steps
2274     breach = terminal_wealth < floor  # Check if the terminal wealth breaches the floor
2275     reach = terminal_wealth >= cap  # Check if the terminal wealth reaches or exceeds the
     cap
2276
2277     p_breach = breach.mean() if breach.sum() > 0 else np.nan  # Probability of breach:
     percentage of scenarios below the floor
2278     p_reach = reach.mean() if reach.sum() > 0 else np.nan  # Probability of reach:
     percentage of scenarios above or at the cap
2279
2280     e_short = (floor - terminal_wealth[breach]).mean() if breach.sum() > 0 else np.nan  #
     Average shortfall in scenarios below the floor
2281     e_surplus = (terminal_wealth[reach] - cap).mean() if reach.sum() > 0 else np.nan  #
     Average surplus in scenarios exceeding the cap
2282
2283     sum_stats = pd.DataFrame.from_dict({
2284         "mean": terminal_wealth.mean(),
2285         "std": terminal_wealth.std(),
2286         "p_breach": p_breach,
2287         "e_short": e_short,
2288         "p_reach": p_reach,
2289         "e_surplus": e_surplus
2290     }, orient="index", columns=[name])
2291
2292     return sum_stats
2293
2294 #E22/R7. Compute the terminal stats for the returns of the bonds, for the equities, for
     the 70/30 portfolio, and for the 60/40 portfolio. Import seaborn and use plt.figure() and
     sns.distplot(terminal values for equities,color="red",label="100% Equities"),
     distplot(terminal values for bonds,color="blue",label="100% Bonds"), etc.Analyze the
     results. Notice that you will obtain some NaNs, why? Interpret.
2295 import seaborn as sns
2296 # Calculate terminal stats for each scenario
2297 terminal_bonds = terminal_values(rets_zc)
2298 terminal_eq = terminal_values(rets_eq)
2299 terminal_70_30 = terminal_values(rets_70_30)
2300 terminal_60_40 = terminal_values(rets_60_40)
2301
```

```
# Print terminal stats for each asset and portfolio
print("Terminal Stats for Bonds:")
print(terminal_stats(rets_zc))

print("\nTerminal Stats for Equities:")
print(terminal_stats(rets_eq))

print("\nTerminal Stats for 70/30 Portfolio:")
print(terminal_stats(rets_70_30))

print("\nTerminal Stats for 60/40 Portfolio:")
print(terminal_stats(rets_60_40))

# Plot terminal values for each asset and portfolio
plt.figure(figsize=(10, 6))
sns.distplot(terminal_eq, color="red", label="100% Equities")
sns.distplot(terminal_bonds, color="blue", label="100% Bonds")
sns.distplot(terminal_70_30, color="green", label="70/30 Portfolio")
sns.distplot(terminal_60_40, color="purple", label="60/40 Portfolio")
plt.legend()
plt.title("Terminal Values Distribution")
plt.show()

# 8 Glide Paths for Allocation

def glidepath_allocator(r1,r2,start_glide=1,end_glide=0):
    """
    Simulates a Target-Date-Fund Style gradual move from r1 to r2.
    """
    n_points=r1.shape[0]
    n_col=r1.shape[1]
    path=pd.Series(data=np.linspace(start_glide,end_glide, num=n_points))
    paths= pd.concat([path]*n_col, axis=1) #we replicate our list// we concatenate n_col copies of path
    paths.index=r1.index
    paths.columns=r1.columns
    return paths

#E23/R8. Add to your previous analysis the terminal state of an allocation of 80/20 using a Glide Allocator with start_glide=.8,end_glide=.2. Interpret the results. What is the results for the probability of breach?

def glidepath_allocator(r1, r2, start_glide=1, end_glide=0):
    """
    Simulates a Target-Date-Fund Style gradual move from r1 to r2.
    """
    n_points = r1.shape[0]
    n_col = r1.shape[1]
    path = pd.Series(data=np.linspace(start_glide, end_glide, num=n_points))
    paths = pd.concat([path] * n_col, axis=1)  # replicate the glide path for each scenario
    paths.index = r1.index
    paths.columns = r1.columns
    return paths
```

```
2352
2353 # Now, let's apply the glide path allocator with an 80/20 allocation
2354 # The risky asset is equities (r1) and the less risky asset is bonds (r2)
2355 glide_path_80_20 = glidepath_allocator(rets_eq, rets_zc, start_glide=0.8, end_glide=0.2)
2356
2357 # Compute the returns of the 80/20 portfolio using the glide path allocator
2358 rets_80_20 = glide_path_80_20 * rets_eq + (1 - glide_path_80_20) * rets_zc
2359
2360 # Calculate the terminal values for the 80/20 portfolio
2361 terminal_80_20 = terminal_values(rets_80_20)
2362
2363 # Print the terminal stats for the 80/20 portfolio
2364 print("\nTerminal Stats for 80/20 Portfolio (Glide Path Allocator):")
2365 print(terminal_stats(rets_80_20))
2366
2367 # Plot the terminal values for the 80/20 portfolio alongside the others
2368 plt.figure(figsize=(10, 6))
2369 sns.distplot(terminal_eq, color="red", label="100% Equities")
2370 sns.distplot(terminal_bonds, color="blue", label="100% Bonds")
2371 sns.distplot(terminal_70_30, color="green", label="70/30 Portfolio")
2372 sns.distplot(terminal_60_40, color="purple", label="60/40 Portfolio")
2373 sns.distplot(terminal_80_20, color="orange", label="80/20 Glide Path")
2374 plt.legend()
2375 plt.title("Terminal Values Distribution with Glide Path Allocation")
2376 plt.show()
2377
2378 # 9 Dynamic Risk Budgeting
2379
2380 # Use CIR model to simulate rates and zc prices
2381 rates, zc_prices = cir(n_years=10, n_scenarios=5000, b=0.03, r_0=0.03, sigma=0.02)
2382
2383 # Use GBM to simulate equities for 10 years
2384 price_eq = gbm(n_years=10, n_scenarios=5000, mu=0.07, sigma=0.15)
2385
2386 # Compute returns for equities and zero-coupon bonds
2387 rets_eq = price_eq.pct_change().dropna()
2388 rets_zc = zc_prices.pct_change().dropna()
2389
2390 # Use the mix backtester to create a 70/30 portfolio using fixedmix allocator
2391 rets_7030b = bt_mix(rets_eq, rets_zc, allocator=fixedmix_allocator, w1=0.7)
2392
2393 # The terminal state results
2394 pd.concat([
2395     terminal_stats(rets_zc, name="ZC", floor=0.75),
2396     terminal_stats(rets_eq, name="Eq", floor=0.75),
2397     terminal_stats(rets_7030b, name="70/30", floor=0.75)
2398 ], axis=1).round(2)
2399
2400 # E23/R8. Complete the following functions and code to create an allocation strategy
    between PSP and GHP using a CPPI-style dynamic risk budgeting. To the previous
    computtation of terminal
2401 # states of ZC bonds, equities, and 70/30, add the terminal state for the floor
    allocation using a floor of 75%.Analyze the results. What happens to the probability of
    breach? Is this good or bad?
```

```
2402
2403 def floor_allocator(psp_r, ghp_r, floor, zc_prices, m=3):
2404     """
2405     Allocation between PSP and GHP with the goal to provide exposure to the upside
2406     of the PSP without going violating the floor.
2407     Uses a CPPI-style dynamic risk budgeting algorithm by investing a multiple of the
2408     cushion in the PSP.
2409     Return a DataFrame with the same shape as the psp/ghp representing the weights in the
     PSP.
2410     """
2411     if zc_prices.shape != psp_r.shape:
2412         raise ValueError("PSP and ZC Prices must have the same shape")
2413
2414     n_steps, n_scenarios = psp_r.shape
2415     account_value = np.repeat(1, n_scenarios)  # Starting with $1
2416     floor_value = np.repeat(1, n_scenarios)
2417     w_history = pd.DataFrame(index=psp_r.index, columns=psp_r.columns)
2418
2419     for step in range(n_steps):
2420         floor_value = floor * zc_prices.iloc[step]  # PV of the Floor assuming today's
     rates and flat YC
2421         cushion = account_value - floor_value  # Cushion is the amount above the floor
2422         psp_w = (m * cushion / account_value).clip(0, 1)  # Apply CPPI rule (multiplier *
     cushion)
2423         ghp_w = 1 - psp_w  # The remaining weight goes to GHP
2424
2425         # Allocation to PSP and GHP
2426         psp_alloc = psp_w * account_value  # Amount allocated to PSP
2427         ghp_alloc = ghp_w * account_value  # Amount allocated to GHP
2428
2429         # Recompute the new account value at the end of this step using the new
     allocations
2430         account_value = psp_alloc * (1 + psp_r.iloc[step]) + ghp_alloc * (1 +
     ghp_r.iloc[step])
2431
2432         # Save the weight of PSP at this step
2433         w_history.iloc[step] = psp_w
2434
2435     return w_history
2436
2437 # 1. Simulating the data for ZC prices and returns for PSP (equities) and GHP (bonds)
2438 rates, zc_prices = cir(n_years=10, n_scenarios=500, b=0.03, r_0=0.03)
2439 price_eq = gbm(n_years=10, n_scenarios=500, mu=0.07, sigma=0.15)
2440 rets_eq = price_eq.pct_change().dropna()
2441 rets_zc = zc_prices.pct_change().dropna()
2442
2443 # 2. Ensure that both the PSP (rets_eq) and ZC (zc_prices) have the same time index
2444 # Aligning the indices of zc_prices and rets_eq
2445 zc_prices = zc_prices.reindex(rets_eq.index)
2446
2447 # 3. Allocate 70/30 using the fixed mix allocator
2448 def fixedmix_allocator(r1, r2, w1, **kwargs):
2449     """
2450     Fixed mix allocator: w1 portion goes to r1 (PSP) and the rest to r2 (GHP).
```

```
    """
    return pd.DataFrame(data=w1, index=r1.index, columns=r1.columns)

# Perform the backtest with 70/30 allocation
rets_7030b = bt_mix(rets_eq, rets_zc, allocator=fixedmix_allocator, w1=0.7)

# 4. Now use the floor allocator for 75% floor
floor_value = 0.75  # The floor is 75%

# Calculate terminal states for ZC, Equities, and 70/30 using floor allocator
zc_terminal = terminal_stats(rets_zc, name="ZC", floor=floor_value)
eq_terminal = terminal_stats(rets_eq, name="Equities", floor=floor_value)
alloc_7030_terminal = terminal_stats(rets_7030b, name="70/30", floor=floor_value)

# Calculate the floor allocation using the floor_allocator function
floor_alloc = floor_allocator(rets_eq, rets_zc, floor=floor_value, zc_prices=zc_prices, m=3)
floor_alloc_terminal = terminal_stats(floor_alloc, name="Floor Allocation", floor=floor_value)

# Display the terminal statistics for each allocation
pd.concat([zc_terminal, eq_terminal, alloc_7030_terminal, floor_alloc_terminal], axis=1).round(2)

#E24/R9. And again, we can extend this to introduce a drawdown constrain. Do this extension and apply it using a max drowdawn of 0.25, then compute the terminal state and compare the output with the previous results.

# Drawdown-Based Allocator
def drawdown_allocator(psp_r, ghp_r, maxdd, m=3):
    """Implements a CPPI-style allocation with a max drawdown constraint."""
    n_steps, n_scenarios = psp_r.shape
    account_value = np.ones(n_scenarios)
    peak_value = np.ones(n_scenarios)
    w_history = pd.DataFrame(index=psp_r.index, columns=psp_r.columns)

    for step in range(n_steps):
        floor_value = (1 - maxdd) * peak_value
        cushion = (account_value - floor_value) / account_value
        psp_w = (m * cushion).clip(0, 1)
        ghp_w = 1 - psp_w

        # Update account and peak values
        account_value *= (psp_w * (1 + psp_r.iloc[step]) + ghp_w * (1 + ghp_r.iloc[step]))
        peak_value = np.maximum(peak_value, account_value)

        # Store allocation history
        w_history.iloc[step] = psp_w

    return w_history

# Define Cash Returns (since bonds can't be used for drawdown constraint)
cash_rate = 0.02
```

```
monthly_cash_return = (1 + cash_rate) ** (1 / 12) - 1
rets_cash = pd.DataFrame(data=monthly_cash_return, index=rets_eq.index,
columns=rets_eq.columns)

# Compute the Allocation with Drawdown Constraint (Max DD 25%)
w_maxdd25 = drawdown_allocator(rets_eq, rets_cash, maxdd=0.25)
rets_maxdd25 = w_maxdd25 * rets_eq + (1 - w_maxdd25) * rets_cash

# Compute the Allocation with a Floor at 75%
w_floor75 = floor_allocator(rets_eq, rets_cash, floor=0.75, zc_prices=zc_prices)
rets_floor75 = w_floor75 * rets_eq + (1 - w_floor75) * rets_cash

# Compute Terminal Values
tv_eq = terminal_values(rets_eq)
tv_zc = terminal_values(rets_zc)
tv_7030b = terminal_values(rets_7030b)
tv_floor75 = terminal_values(rets_floor75)
tv_maxdd25 = terminal_values(rets_maxdd25)

# Compute and Print Summary Stats
print("Terminal Stats for Floor at 75%:")
print(terminal_stats(rets_floor75, name="Floor 75%"))
print("\nTerminal Stats for Max Drawdown 25%:")
print(terminal_stats(rets_maxdd25, name="Max Drawdown 25%"))

#PLOT:
plt.figure(figsize=(12,6))
sns.distplot(tv_eq,color="red",label="100% Equities", bins=100)
plt.axvline(tv_eq.mean(), ls="--", color="red")
sns.distplot(tv_7030b, color="orange", label="70/30 Equities/Bonds", bins=100)
plt.axvline(tv_7030b.mean(), ls="--", color="orange")
sns.distplot(tv_floor75,color="green",label="Floor at 75%", bins=100)
plt.axvline(tv_floor75.mean(), ls="--", color="green")
sns.distplot(tv_maxdd25, color="yellow",label="MaxDD 25%", bins=100)
plt.axvline(tv_maxdd25.mean(),ls="--", color="yellow")
plt.legend();
plt.show()


#E25/R10. Finally, we are going to compare the returns produced with the 0.25 max
drawdown allocation LDI strategy agains the performance of the market. For this, we need
to compute the historical drawdowns, we can use for this porpouse the market returns we
obtained at the begining of the coursework. Compute the drawdowns of the historical
market returns, apply the LDI allocation strategy using the max drawdown constraint.
Analyze the drawdowns experienced using the historical returns produced by the mentioned
LDI strategy. Analyze the results. Give a wrap-up that explains how we went in this
coursework from CPPI and asset simulation to dynamic asset allocation strategies with
LDI. Give a general conclusion for the coursework.

# Function to compute drawdowns
def drawdown(return_series: pd.Series):
    """Computes the wealth index, previous peaks, and percentage drawdowns."""
    wealth_index = 1000 * (1 + return_series).cumprod()
    previous_peaks = wealth_index.cummax()
    drawdowns = (wealth_index - previous_peaks) / previous_peaks
```

```
    return pd.DataFrame({"Wealth": wealth_index,
                         "Previous Peak": previous_peaks,
                         "Drawdown": drawdowns})

# Compute Drawdowns of Historical Market Returns
rets_tmi = total_market_return["1990":]  # Market returns from 1990 onwards
dd_tmi = drawdown(rets_tmi)

# Ensure index is in Datetime format before plotting
dd_tmi = dd_tmi.copy()
dd_tmi.index = dd_tmi.index.to_timestamp()

# Now it should plot correctly
ax = dd_tmi["Wealth"].plot(figsize=(12, 6), ls="-", color="goldenrod", label="Market Wealth")
dd_tmi["Previous Peak"].plot(ax=ax, ls=":", color="red", label="Previous Peak")

plt.legend()
plt.title("Market Wealth and Drawdowns (1990-Present)")
plt.show()

# Apply Max Drawdown 25% LDI Allocation Strategy
cash_rate = 0.03
monthly_cash_return = (1 + cash_rate) ** (1 / 12) - 1
rets_cash = pd.DataFrame(data=monthly_cash_return, index=rets_tmi.index, columns=[0])

# Compute returns for MaxDD 25% strategy using the historical market returns
rets_maxdd25 = bt_mix(pd.DataFrame(rets_tmi), rets_cash, allocator=drawdown_allocator, maxdd=0.25, m=5)
dd_25 = drawdown(rets_maxdd25[0])

# Plot Drawdowns of Market vs MaxDD 25% Strategy
plt.figure(figsize=(12, 6))
plt.plot(dd_tmi["Drawdown"], label="Market Drawdowns", color="red")
plt.plot(dd_25["Drawdown"], label="MaxDD 25% Drawdowns", color="blue")
plt.title("Drawdowns: Market vs. MaxDD 25% Strategy")
plt.legend()
plt.show()

```