

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

```
1 #Portfolio Management - Coursework 2: Asset simulation and Liability Driven Portfolios
2 #1.CPPI
3
4 # Import libraries
5 # pandas
6 import pandas as pd
7
8 #numpy
9 import numpy as np
10
11 # matplotlib.pyplot
12 import matplotlib.pyplot as plt
13
14 #Load data industry returns
15 df_returns= pd.read_csv('C:/Users/52241/Downloads/EXCEL PYTHON/index30_returns.csv',
16 header=0, index_col=0,parse_dates=True)/100 #convert to percentages
16 df_returns.index= pd.to_datetime(df_returns.index,format="%Y%m").to_period('M') #to time
series in monthly periods
17 df_returns.columns=df_returns.columns.str.strip() #remove blank spaces in names
18
19 # E1. Load the total market index size and industry firms. Give time series format in
monthly periods. Note: Do not transform to percentages.
20 #Load the total market index
21 #get earnings sizes from the companies for each industry
22 # Load the total market index size
23 df_index_size = pd.read_csv('C:/Users/52241/Downloads/EXCEL PYTHON/index30_size.csv',
24 header=0, index_col=0, parse_dates=True)
25
26 # Convert the index to a time series in monthly periods
27 df_index_size.index = pd.to_datetime(df_index_size.index, format="%Y%m").to_period('M')
28
29 # Remove any whitespace in column names
30 df_index_size.columns = df_index_size.columns.str.strip()
31
32 #get number of firms in the index of each industry
33 # Load the number of firms in the index for each industry
34 df_index_firms = pd.read_csv('C:/Users/52241/Downloads/EXCEL PYTHON/index30_nfirms.csv',
35 header=0, index_col=0, parse_dates=True)
36
37 # Convert the index to a time series in monthly periods
38 df_index_firms.index = pd.to_datetime(df_index_firms.index, format="%Y%m").to_period('M')
39
40 # Remove any whitespace in column names
41 df_index_firms.columns = df_index_firms.columns.str.strip()
42
43 #E2. Calculate Market Capitalization and Weighted Market Return
44 #market capitalization
45 # Calculate industry market capitalization
46 ind_mktcap = df_index_firms * df_index_size
47
48 # Calculate total market capitalization (sum across industries for each period)
49 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
58 #from 2000 onwards. These industries will form our risky assets.
59 #Select specific industries as risky assets and create a safe asset with fixed monthly
60 #returns.
61 #Consider returns from 2000 for steel, fin, and beer
62 # Select Steel, Finance, and Beer industries from 2000 onwards
63 df_risky = df_returns.loc["2000":, ["Steel", "Fin", "Beer"]]
64
65 #We also need safe assets. For this, we create a dataframe with the same number of
66 #returns and assume a monthly fixed safe return of 0.02 annualized. This is, we have a
67 #risk-free asset that pays 0.02 per year.
68 df_safe = pd.DataFrame().reindex_like(df_risky)
69 df_safe[:] = 0.02 / 12 # Monthly fixed safe return (2% annualized)
70
71 #As we have risky and risk-free assets,
72 # let's implement CPPI and explore how it works.
73 # First, assume we invest 1000 USD, that the floor
74 # (the minimum value below which the portfolio should not
75 # fall) is 80% of the initial value, and that the multiplier
76 # (the level of exposure to risky assets based on the cushion,
77 # i.e. the aggressiveness of risky asset allocations) is 3.
78
79 #Set up initial account parameters:
80 start = 1000 # Starting account value
81 floor = 0.70 # Floor as a percentage of starting account, this is just a percentage
82 account_value = start #starting value of the investment
83 floor_value = start * floor #floor applied to the account value
84 m = 4 # CPPI multiplier
85
86 #CPPI works through out time, thus we need to define an entity to save the results of our
87 #simulated example. For this, we use dataframes of with the same number of trading periods
88 #than our risky assets, which are the number of steps for the simulation. As we are
89 #interested in tracking the evolution
90 #of portfolio values, risky and safe allocations, and total
91 #returns over time, we are going to define 3 dataframes.
92
93 # Prepare to track the evolution of account values
94 dates = df_risky.index
95
96 n_steps = len(dates)
97
98 # Prepare trackers for the the history of account values, cushion, and risky weights
99 account_history = pd.DataFrame().reindex_like(df_risky)
100 cushion_history = pd.DataFrame().reindex_like(df_risky)
101 risky_w_history = pd.DataFrame().reindex_like(df_risky)

```

```

97 #E3. Implement CCPI
98 #This loop performs the CPPI calculations by updating allocations based on the cushion.
99 Use your defined strating parameters
100 for step in range(n_steps):
101     # Calculate the current cushion
102     cushion = (account_value - floor_value) / account_value
103
104     # Calculate weights for allocation to risky assets based on the multiplier
105     risky_w = m * cushion
106
107     # Ensure allocation to risky does not exceed 100% and there is no short selling
108     risky_w = np.minimum(risky_w, 1) # Cap at 100%
109     risky_w = np.maximum(risky_w, 0) # No short selling (min 0%)
110
111     # Compute weights for the safe asset
112     safe_w = 1 - risky_w
113
114     # Allocate the money in the account to risky and safe assets
115     risky_alloc = account_value * risky_w
116     safe_alloc = account_value * safe_w
117
118     # Update account value based on risky and safe returns
119     account_value = (
120         risky_alloc * (1 + df_risky.iloc[step]) + safe_alloc * (1 + df_safe.iloc[step])
121     )
122
123     # Store history for visualization or tracking
124     cushion_history.iloc[step] = cushion
125     account_history.iloc[step] = account_value
126     risky_w_history.iloc[step] = risky_w
127
128 #First values of our CCPI
129 account_history.head()
130
131 #Before seeing the effects of CPPI strategy, what would have happened if we had put all
132 #the money in the risky assets and not using the CPPI? Well, this is basically the
133 #cumulative returns of the risky assets:
134 #Plot the account history for one asset, comparing CPPI-managed wealth with a fully risky
135 #allocation strategy.
136 risky_wealth = start * (1 + df_risky).cumprod()
137 risky_wealth.plot()
138 plt.show()
139 #But, what is the investment allocation recommended using CPPI? Well, we can know this by
140 #plotting our simulated weights.
141 risky_w_history.plot()
142 plt.show()
143
144 #This is the evolution of the allocation to risky assets. Notice the increment in
145 #investment on beer. Let's compare then the CPPI vs Full Risky Allocation to beer:
146 # Plot CPPI-managed wealth vs. full-risky strategy for comparison
147 ax = account_history["Beer"].plot(figsize=(12, 6), title="CPPI vs Full Risky Allocation")
148 risky_wealth["Beer"].plot(ax=ax, style="k:", label="Full Risky Allocation (Beer)")
149 plt.axhline(y=floor_value, color='r', linestyle="--", label="Floor Value") # Plot the
150 floor line

```

```

144 plt.legend()
145 plt.show()
146
147 #E4. Compare CPPI vs Risky Allocation for Finance and Steel
148 #Finance
149
150 start = 1000 # Starting account value
151 floor = 0.70 # Floor as a percentage of starting account
152 account_value = start # Starting value of the investment
153 floor_value = start * floor # Floor applied to the account value
154 m = 4 # CPPI multiplier
155
156 # Prepare to track the evolution of account values
157 dates = df_risky["Fin"].index
158 n_steps = len(dates)
159
160 # Initialize DataFrames to track results
161 account_history_fin = pd.DataFrame(index=dates, columns=["Fin"])
162 cushion_history_fin = pd.DataFrame(index=dates, columns=["Fin"])
163 risky_w_history_fin = pd.DataFrame(index=dates, columns=["Fin"])
164
165 # Implement CPPI for Finance (Fin)
166 for step in range(n_steps):
167     cushion = (account_value - floor_value) / account_value
168     risky_w = np.clip(m * cushion, 0, 1) # Risky weight between 0 and 1
169     safe_w = 1 - risky_w
170
171     # Allocate money to risky and safe assets
172     risky_alloc = account_value * risky_w
173     safe_alloc = account_value * safe_w
174
175     # Update account value
176     account_value = (
177         risky_alloc * (1 + df_risky["Fin"].iloc[step]) +
178         safe_alloc * (1 + df_safe["Fin"].iloc[step])
179     )
180
181     # Track history
182     cushion_history_fin.iloc[step, 0] = cushion
183     account_history_fin.iloc[step, 0] = account_value
184     risky_w_history_fin.iloc[step, 0] = risky_w
185
186 # First values of CPPI for Finance
187 account_history_fin.head()
188
189 # Plot CPPI vs Full Risky Allocation for Finance
190 fig, ax = plt.subplots(figsize=(12, 6))
191 risky_wealth_fin = start * (1 + df_risky["Fin"]).cumprod()
192 account_history_fin["Fin"].plot(ax=ax, label="CPPI (Fin)")
193 risky_wealth_fin.plot(ax=ax, style="k--", label="Full Risky (Fin)")
194 plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")
195 plt.title("CPPI vs Full Risky Allocation for Finance (Fin)")
196 plt.legend()
197 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_steeel = pd.DataFrame(index=dates, columns=["Steel"])
217 cushion_history_steeel = pd.DataFrame(index=dates, columns=["Steel"])
218 risky_w_history_steeel = 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_steeel.iloc[step, 0] = cushion
238     account_history_steeel.iloc[step, 0] = account_value
239     risky_w_history_steeel.iloc[step, 0] = risky_w
240
241 # First values of CPPI for Steel
242 account_history_steeel.head()
243
244 # Plot CPPI vs Full Risky Allocation for Steel
245 fig, ax = plt.subplots(figsize=(12, 6))
246 risky_wealth_steeel = start * (1 + df_risky["Steel"]).cumprod()
247 account_history_steeel["Steel"].plot(ax=ax, label="CPPI (Steel)")
248 risky_wealth_steeel.plot(ax=ax, style="r--", label="Full Risky (Steel)")

```

```

249 plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")
250 plt.title("CPPI vs Full Risky Allocation for Steel")
251 plt.legend()
252 plt.show()
253
254 # Plot Risky Asset Allocation Over Time for Steel
255 risky_w_history_steeel.plot(figsize=(12, 6), title="Risky Asset Allocation Over Time
256 (Steel)")
257 plt.show()
258
259 #E5. Compute the summary statistics studied in CW1 and apply
260 # them to df_risky
261 from scipy.stats import skew, kurtosis
262
263 def summary_stats(r):
264     """
265         Calculate summary statistics for returns.
266         Includes annualized return, volatility, skewness, kurtosis, VaR, CVaR, Sharpe ratio,
267         and max drawdown.
268     """
269
270     # Calculate annualized return using compounded growth
271     compounded_growth = (1 + r).prod()
272     n_periods = r.shape[0]
273     ann_r = (compounded_growth) ** (12 / n_periods) - 1 # Annualized return assuming 12
274     periods per year (monthly returns)
275
276     # Annualized volatility (standard deviation)
277     ann_vol = r.std() * np.sqrt(12) # Multiply by sqrt(12) to annualize
278
279     # Skewness and kurtosis (without bias correction)
280     skewness = skew(r, bias=False)
281     kurt = kurtosis(r, bias=False)
282
283     # Value at Risk (5%) using Cornish-Fisher expansion
284     z = 1.645 # Z-score for 5% VaR (assuming normal distribution for simplicity)
285     cf_var5 = -(ann_r - 0.5 * ann_vol**2) + z * ann_vol
286     cf_var5 += ((z**2 - 1) * skewness / 6) + (((z**3 - 3*z) * (kurt - 3)) / 24) -
287     (((2*z**3 - 5*z) * (skewness**2)) / 36)
288
289     # Historical VaR (5%)
290     hist_var5 = -r.quantile(0.05)
291
292     # Conditional Value at Risk (CVaR) - Average of returns worse than the VaR
293     cvar5 = -r[r <= hist_var5].mean()
294
295     # Sharpe Ratio assuming a risk-free rate of 0 for simplicity
296     sharpe_ratio = ann_r / ann_vol if ann_vol != 0 else np.nan # Avoid division by zero
297
298     # Maximum drawdown
299     cumulative = (1 + r).cumprod() # Cumulative returns
300     peak = cumulative.cummax() # Track the peak of the cumulative returns
301     max_dd = ((cumulative / peak) - 1).min() # Max drawdown as the largest loss from
302     peak
303
304 
```

```

298     # Compile results into a DataFrame
299     return pd.Series({
300         "Annualized Return": ann_r,
301         "Annualized Vol": ann_vol,
302         "Skewness": skewness,
303         "Kurtosis": kurt,
304         "Cornish-Fisher VaR(5)": cf_var5,
305         "Historic VaR(5)": hist_var5,
306         "CVar(5)": cvar5,
307         "Sharpe Ratio": sharpe_ratio,
308         "Max Drawdown": max_dd
309     })
310
311 # Apply the summary_stats function to each column of df_risky
312 summary_stats_df = df_risky.apply(summary_stats)
313
314 # Display the results
315 print(summary_stats_df)
316
317
318 # E6. Implement CPPI as a function
319
320 def run_cppi1(risky_r, safe_r=None, m=3, start=1000, floor=0.8, riskfree_rate=0.03):
321     """
322         Runs a CPPI strategy given a set of returns for the risky asset.
323         Returns a dictionary containing:
324             - Asset Value History
325             - Risk Budget History
326             - Risky Weight History
327     """
328     # Set up the parameters
329     dates = risky_r.index
330     n_steps = len(dates)
331     account_value = start # Initial portfolio value
332     floor_value = start * floor # Minimum guaranteed portfolio value
333
334     # Ensure the input is a DataFrame
335     if isinstance(risky_r, pd.Series):
336         risky_r = pd.DataFrame(risky_r, columns=["R"])
337
338     # If no safe asset is provided, use a fixed risk-free return
339     if safe_r is None:
340         safe_r = pd.DataFrame().reindex_like(risky_r)
341         safe_r[:] = riskfree_rate / 12 # Convert annualized rate to monthly
342
343     # Initialize tracking dataframes
344     account_history = pd.DataFrame().reindex_like(risky_r)
345     cushion_history = pd.DataFrame().reindex_like(risky_r)
346     risky_w_history = pd.DataFrame().reindex_like(risky_r) # Risky asset weights
347
348     # CPPI implementation
349     for step in range(n_steps):
350         # Compute cushion
351         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_cppi(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.

```

```

404     Returns a dictionary containing: Asset Value History, Risk Budget History, Risky
405     weight history.
406     """
407
408     # Set up the parameters
409     dates = risky_r.index
410     n_steps = len(dates)
411     account_value = start # Initial portfolio value
412     peak = start # At the start, the peak is the initial account value
413     floor_value = start * floor # Initial floor value
414
415     # Ensure risky_r is a DataFrame
416     if isinstance(risky_r, pd.Series):
417         risky_r = pd.DataFrame(risky_r, columns=["R"])
418
419     # If no safe asset is provided, assume a fixed risk-free return
420     if safe_r is None:
421         safe_r = pd.DataFrame(riskfree_rate / 12, index=dates, columns=risky_r.columns)
422
423     # Initialize tracking DataFrames
424     account_history = pd.DataFrame(index=dates, columns=risky_r.columns)
425     cushion_history = pd.DataFrame(index=dates, columns=risky_r.columns)
426     risky_w_history = pd.DataFrame(index=dates, columns=risky_r.columns)
427
428     # CPPI implementation
429     for step in range(n_steps):
430         date = dates[step]
431
432         # If there is a drawdown constraint, adjust the floor dynamically
433         if drawdown is not None:
434             # Update the peak value only if account_value exceeds the current peak
435             peak = np.maximum(peak, account_value)
436             floor_value = peak * (1 - drawdown) # Recalculate floor dynamically based on
437             drawdown
438
439             # Compute cushion based on the difference between account value and floor value
440             cushion = (account_value - floor_value) / account_value
441
442             # Compute risky asset allocation based on the cushion
443             risky_w = np.clip(m * cushion, 0, 1) # Ensure allocation is between 0 and 1
444             safe_w = 1 - risky_w # Remaining allocation to safe asset
445
446             # Allocate funds to risky and safe assets
447             risky_alloc = account_value * risky_w
448             safe_alloc = account_value * safe_w
449
450             # Update account value based on returns
451             account_value = (
452                 risky_alloc * (1 + risky_r.iloc[step]) +
453                 safe_alloc * (1 + safe_r.iloc[step])
454             )
455
456             # Debug: print the account value and peak
457             print(f"Step {step} - Account Value: {account_value}, Peak: {peak}, Floor Value:
458 {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',
484 header=0, index_col=0, parse_dates=True) / 100 # Convert to percentages
485 df_returns.index = pd.to_datetime(df_returns.index, format="%Y%m").to_period('M')
486 df_returns.columns = df_returns.columns.str.strip()
487
488 # Run CPPI on selected assets
489 btr = run_cppi(df_returns.loc["2007":, ["Steel", "Fin", "Beer"]], drawdown=0.25)
490
491 # Plot results
492 ax = btr["Wealth"].plot(figsize=(12, 6))
493 btr["Risky Wealth"].plot(ax=ax, style="--", label="Total Market Risky Wealth")
494 plt.legend()
495 plt.show()
496
497 # Apply summary stats function to CPPI results
498 summary_stats_df = btr["Risky Wealth"].pct_change().dropna().apply(summary_stats)
499
500 # Display results
501 print(summary_stats_df)
502
503 #As you can see there's an important difference in the Drawdowns. Now, this was because
504 # we were updating the
505 # floor every month, in practice you would want to save trading costs, so there are other
506 # tools that can be added to
507 # this.
508 #Another question is, what happens when we vary the drawdown constrain level?

```

```

506 # 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
507 # 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
508 # 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 obseration, the effect of CPPI, its pros and
      cons and the effect of the drawdown constrains.
509 # Selecciona las industrias específicas desde el mes de enero de 2000 (sin importar el
      día exacto)
510 # Cargar los datos de retornos de las industrias
511 # Games(No Drawdown)
512 # Load industry returns data
513 df_returns = pd.read_csv('C:/Users/52241/Downloads/EXCEL PYTHON/index30_returns.csv',
      header=0, index_col=0, parse_dates=True) / 100
514 df_returns.index = pd.to_datetime(df_returns.index, format="%Y%m")
515
516 # Select returns for Games from 2000 onwards
517 df_risky_games = df_returns.loc["2000":, ["Games"]]
518
519 # Run CPPI without drawdown
520 btr_1 = run_cippi1(df_risky_games)
521
522 #  Plot "Total Market Risky Wealth"
523 fig, ax = plt.subplots(figsize=(12, 6))
524 btr_1["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
525 plt.title("Total Market Risky Wealth - Games (No Drawdown)")
526 plt.legend()
527 plt.show()
528
529 # Compute full risky wealth
530 risky_wealth_games = start * (1 + df_risky_games).cumprod()
531
532 #  Plot "CPPI vs Full Risky Allocation" with correct labels
533 fig, ax = plt.subplots(figsize=(12, 6))
534 btr_1["Wealth"]["Games"].plot(ax=ax, label="CPPI") # Correct label for CPPI
535 risky_wealth_games["Games"].plot(ax=ax, style="k--", label="Full Risky Allocation") # Correct label for Full Risky Allocation
536 plt.axhline(y=start * 0.7, color='gray', linestyle="--", label="Floor Value") # Floor Value
537 plt.title("CPPI vs Full Risky Allocation for Games (No Drawdown)")
538 plt.legend()
539 plt.show()
540
541 #  Compute and display summary statistics
542 summary_stats_df = btr_1["Wealth"].pct_change().dropna().apply(summary_stats)
543 print(summary_stats_df)
544
545 #Games Drawdown 10%
546

```

```

547 btr_10 = run_cppi(df_risky_games, drawdown=0.10)
548
549 # ✅ Plot "Total Market Risky Wealth"
550 fig, ax = plt.subplots(figsize=(12, 6))
551 btr_10["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
552 plt.title("Total Market Risky Wealth - Drawdown 10%")
553 plt.legend()
554 plt.show()
555
556 # Compute full risky wealth
557 risky_wealth_games = start * (1 + df_risky_games).cumprod()
558
559 # ✅ Plot "CPPI vs Full Risky Allocation" (correct format)
560 fig, ax = plt.subplots(figsize=(12, 6))
561 btr_10["Wealth"]["Games"].plot(ax=ax, label="CPPI")
562 risky_wealth_games["Games"].plot(ax=ax, style="k--", label="Full Risky Allocation")
563 plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")
564 plt.title("CPPI vs Full Risky Allocation for Games (Drawdown 10%)")
565 plt.legend()
566 plt.show()
567
568 # 📈 Compute and display summary statistics
569 summary_stats_df = btr_10["Wealth"].pct_change().dropna().apply(summary_stats)
570 print(summary_stats_df)
571
572 # Run CPPI with a 30% drawdown constraint
573 btr_30 = run_cppi(df_risky_games, drawdown=0.30)
574
575 # ✅ Plot "Total Market Risky Wealth"
576 fig, ax = plt.subplots(figsize=(12, 6))
577 btr_30["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
578 plt.title("Total Market Risky Wealth - Games (Drawdown 30%)")
579 plt.legend()
580 plt.show()
581
582 # Compute full risky wealth
583 risky_wealth_games = start * (1 + df_risky_games).cumprod()
584
585 # ✅ Plot "CPPI vs Full Risky Allocation" with correct labels
586 fig, ax = plt.subplots(figsize=(12, 6))
587 btr_30["Wealth"]["Games"].plot(ax=ax, label="CPPI") # CPPI Wealth
588 risky_wealth_games["Games"].plot(ax=ax, style="k--", label="Full Risky Allocation") # Full Risky Wealth
589 plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value") # Correct Floor value
590 plt.title("CPPI vs Full Risky Allocation for Games (Drawdown 30%)")
591 plt.legend()
592 plt.show()
593
594 # 📈 Compute and display summary statistics
595 summary_stats_df = btr_30["Wealth"].pct_change().dropna().apply(summary_stats)
596 print(summary_stats_df)
597
598 # Run CPPI with a 20% drawdown constraint

```

```

599 btr_20 = run_cppi(df_risky_games, drawdown=0.20)
600
601 # Plot "Total Market Risky Wealth"
602 fig, ax = plt.subplots(figsize=(12, 6))
603 btr_20["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
604 plt.title("Total Market Risky Wealth - Games (Drawdown 20%)")
605 plt.legend()
606 plt.show()
607
608 # Compute full risky wealth
609 risky_wealth_games = start * (1 + df_risky_games).cumprod()
610
611 # Plot "CPPI vs Full Risky Allocation" with correct labels
612 fig, ax = plt.subplots(figsize=(12, 6))
613 btr_20["Wealth"]["Games"].plot(ax=ax, label="CPPI") # CPPI Wealth
614 risky_wealth_games["Games"].plot(ax=ax, style="--k-", label="Full Risky Allocation") # Full Risky Wealth
615 plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value") # Correct Floor value
616 plt.title("CPPI vs Full Risky Allocation for Games (Drawdown 20%)")
617 plt.legend()
618 plt.show()
619
620 # Compute and display summary statistics
621 summary_stats_df = btr_20["Wealth"].pct_change().dropna().apply(summary_stats)
622 print(summary_stats_df)
623
624 # Select returns for Autos from 2000 onwards
625 df_risky_autos = df_returns.loc["2000":, ["Autos"]]
626
627 # Run CPPI without drawdown
628 btr_1_autos = run_cppi1(df_risky_autos)
629
630 # Plot "Total Market Risky Wealth" for Autos
631 fig, ax = plt.subplots(figsize=(12, 6))
632 btr_1_autos["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
633 plt.title("Total Market Risky Wealth - Autos (No Drawdown)")
634 plt.legend()
635 plt.show()
636
637 # Compute full risky wealth for Autos
638 risky_wealth_autos = start * (1 + df_risky_autos).cumprod()
639
640 # Plot "CPPI vs Full Risky Allocation" for Autos (No Drawdown)
641 fig, ax = plt.subplots(figsize=(12, 6))
642 btr_1_autos["Wealth"]["Autos"].plot(ax=ax, label="CPPI") # Correct label for CPPI
643 risky_wealth_autos["Autos"].plot(ax=ax, style="--k-", label="Full Risky Allocation") # Correct label for Full Risky Allocation
644 plt.axhline(y=start * 0.7, color='gray', linestyle="--", label="Floor Value") # Floor Value
645 plt.title("CPPI vs Full Risky Allocation for Autos (No Drawdown)")
646 plt.legend()
647 plt.show()
648

```

```
649 # [?] Compute and display summary statistics for Autos
650 summary_stats_df_autos = btr_1_autos["Wealth"].pct_change().dropna().apply(summary_stats)
651 print(summary_stats_df_autos)
652
653 # Autos Drawdown 10%
654 btr_10_autos = run_cppi(df_risky_autos, drawdown=0.10)
655
656 # [?] Plot "Total Market Risky Wealth" for Autos with Drawdown 10%
657 fig, ax = plt.subplots(figsize=(12, 6))
658 btr_10_autos["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
659 plt.title("Total Market Risky Wealth - Autos (Drawdown 10%)")
660 plt.legend()
661 plt.show()
662
663 # Compute full risky wealth for Autos
664 risky_wealth_autos = start * (1 + df_risky_autos).cumprod()
665
666 # [?] Plot "CPPI vs Full Risky Allocation" for Autos (Drawdown 10%)
667 fig, ax = plt.subplots(figsize=(12, 6))
668 btr_10_autos["Wealth"]["Autos"].plot(ax=ax, label="CPPI")
669 risky_wealth_autos["Autos"].plot(ax=ax, style="k--", label="Full Risky Allocation")
670 plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")
671 plt.title("CPPI vs Full Risky Allocation for Autos (Drawdown 10%)")
672 plt.legend()
673 plt.show()
674
675 # [?] Compute and display summary statistics for Autos (Drawdown 10%)
676 summary_stats_df_autos =
677 btr_10_autos["Wealth"].pct_change().dropna().apply(summary_stats)
678 print(summary_stats_df_autos)
679
680 # Run CPPI with a 30% drawdown constraint for Autos
681 btr_30_autos = run_cppi(df_risky_autos, drawdown=0.30)
682
683 # [?] Plot "Total Market Risky Wealth" for Autos with Drawdown 30%
684 fig, ax = plt.subplots(figsize=(12, 6))
685 btr_30_autos["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
686 plt.title("Total Market Risky Wealth - Autos (Drawdown 30%)")
687 plt.legend()
688 plt.show()
689
690 # Compute full risky wealth for Autos
691 risky_wealth_autos = start * (1 + df_risky_autos).cumprod()
692
693 # [?] Plot "CPPI vs Full Risky Allocation" for Autos (Drawdown 30%)
694 fig, ax = plt.subplots(figsize=(12, 6))
695 btr_30_autos["Wealth"]["Autos"].plot(ax=ax, label="CPPI")
696 risky_wealth_autos["Autos"].plot(ax=ax, style="k--", label="Full Risky Allocation")
697 plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")
698 plt.title("CPPI vs Full Risky Allocation for Autos (Drawdown 30%)")
699 plt.legend()
700 plt.show()
701 # [?] Compute and display summary statistics for Autos (Drawdown 30%)
```

```

702 summary_stats_df_autos =
703     btr_30_autos["Wealth"].pct_change().dropna().apply(summary_stats)
704
705 # Run CPPI with a 20% drawdown constraint for Autos
706 btr_20_autos = run_cppi(df_risky_autos, drawdown=0.20)
707
708 #  Plot "Total Market Risky Wealth" for Autos with Drawdown 20%
709 fig, ax = plt.subplots(figsize=(12, 6))
710 btr_20_autos["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
711 plt.title("Total Market Risky Wealth - Autos (Drawdown 20%)")
712 plt.legend()
713 plt.show()
714
715 # Compute full risky wealth for Autos
716 risky_wealth_autos = start * (1 + df_risky_autos).cumprod()
717
718 #  Plot "CPPI vs Full Risky Allocation" for Autos (Drawdown 20%)
719 fig, ax = plt.subplots(figsize=(12, 6))
720 btr_20_autos["Wealth"["Autos"]].plot(ax=ax, label="CPPI")
721 risky_wealth_autos["Autos"].plot(ax=ax, style="k--", label="Full Risky Allocation")
722 plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")
723 plt.title("CPPI vs Full Risky Allocation for Autos (Drawdown 20%)")
724 plt.legend()
725 plt.show()
726
727 #  Compute and display summary statistics for Autos (Drawdown 20%)
728 summary_stats_df_autos =
729     btr_20_autos["Wealth"].pct_change().dropna().apply(summary_stats)
730 print(summary_stats_df_autos)
731
732 # Select returns for Telecom (Telcm) from 2000 onwards
733 df_risky_telcm = df_returns.loc["2000":, ["Telcm"]]
734
735 # Run CPPI without drawdown
736 btr_1_telcm = run_cppi1(df_risky_telcm)
737
738 #  Plot "Total Market Risky Wealth" for Telecom (Telcm)
739 fig, ax = plt.subplots(figsize=(12, 6))
740 btr_1_telcm["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
741 plt.title("Total Market Risky Wealth - Telecom (Telcm) (No Drawdown)")
742 plt.legend()
743 plt.show()
744
745 # Compute full risky wealth for Telecom
746 risky_wealth_telcm = start * (1 + df_risky_telcm).cumprod()
747
748 #  Plot "CPPI vs Full Risky Allocation" for Telecom (Telcm) (No Drawdown)
749 fig, ax = plt.subplots(figsize=(12, 6))
750 btr_1_telcm["Wealth"["Telcm"]].plot(ax=ax, label="CPPI") # Correct label for CPPI
751 risky_wealth_telcm["Telcm"].plot(ax=ax, style="k--", label="Full Risky Allocation") # Correct label for Full Risky Allocation
752 plt.axhline(y=start * 0.7, color='gray', linestyle="--", label="Floor Value") # Floor Value

```

```

752 plt.title("CPPI vs Full Risky Allocation for Telecom (Telcm) (No Drawdown)")
753 plt.legend()
754 plt.show()
755
756 # 📈 Compute and display summary statistics for Telecom (Telcm)
757 summary_stats_df_telcm = btr_1_telcm["Wealth"].pct_change().dropna().apply(summary_stats)
758 print(summary_stats_df_telcm)
759
760 # Telecom (Telcm) Drawdown 10%
761 btr_10_telcm = run_cppi(df_risky_telcm, drawdown=0.10)
762
763 # ✎ Plot "Total Market Risky Wealth" for Telecom (Telcm) with Drawdown 10%
764 fig, ax = plt.subplots(figsize=(12, 6))
765 btr_10_telcm["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
766 plt.title("Total Market Risky Wealth - Telecom (Telcm) (Drawdown 10%)")
767 plt.legend()
768 plt.show()
769
770 # Compute full risky wealth for Telecom
771 risky_wealth_telcm = start * (1 + df_risky_telcm).cumprod()
772
773 # ✎ Plot "CPPI vs Full Risky Allocation" for Telecom (Telcm) (Drawdown 10%)
774 fig, ax = plt.subplots(figsize=(12, 6))
775 btr_10_telcm["Wealth"]["Telcm"].plot(ax=ax, label="CPPI")
776 risky_wealth_telcm["Telcm"].plot(ax=ax, style="k--", label="Full Risky Allocation")
777 plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")
778 plt.title("CPPI vs Full Risky Allocation for Telecom (Telcm) (Drawdown 10%)")
779 plt.legend()
780 plt.show()
781
782 # 📈 Compute and display summary statistics for Telecom (Telcm) (Drawdown 10%)
783 summary_stats_df_telcm =
784     btr_10_telcm["Wealth"].pct_change().dropna().apply(summary_stats)
785 print(summary_stats_df_telcm)
786
787 # Run CPPI with a 30% drawdown constraint for Telecom (Telcm)
788 btr_30_telcm = run_cppi(df_risky_telcm, drawdown=0.30)
789
790 # ✎ Plot "Total Market Risky Wealth" for Telecom (Telcm) with Drawdown 30%
791 fig, ax = plt.subplots(figsize=(12, 6))
792 btr_30_telcm["Risky Wealth"].plot(ax=ax, label="Total Market Risky Wealth")
793 plt.title("Total Market Risky Wealth - Telecom (Telcm) (Drawdown 30%)")
794 plt.legend()
795 plt.show()
796
797 # Compute full risky wealth for Telecom
798 risky_wealth_telcm = start * (1 + df_risky_telcm).cumprod()
799
800 # ✎ Plot "CPPI vs Full Risky Allocation" for Telecom (Telcm) (Drawdown 30%)
801 fig, ax = plt.subplots(figsize=(12, 6))
802 btr_30_telcm["Wealth"]["Telcm"].plot(ax=ax, label="CPPI")
803 risky_wealth_telcm["Telcm"].plot(ax=ax, style="k--", label="Full Risky Allocation")
804 plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")
805 plt.title("CPPI vs Full Risky Allocation for Telecom (Telcm) (Drawdown 30%)")

```

```

805 plt.legend()
806 plt.show()
807
808 # [?] Compute and display summary statistics for Telecom (Telcm) (Drawdown 30%)
809 summary_stats_df_telcm =
810 btr_30_telcm[["Wealth"]].pct_change().dropna().apply(summary_stats)
811 print(summary_stats_df_telcm)
812
813 # Run CPPI with a 20% drawdown constraint for Telecom (Telcm)
814 btr_20_telcm = run_cppi(df_risky_telcm, drawdown=0.20)
815
816 # [?] Plot "Total Market Risky Wealth" for Telecom (Telcm) with Drawdown 20%
817 fig, ax = plt.subplots(figsize=(12, 6))
818 btr_20_telcm[["Risky Wealth"]].plot(ax=ax, label="Total Market Risky Wealth")
819 plt.title("Total Market Risky Wealth - Telecom (Telcm) (Drawdown 20%)")
820 plt.legend()
821 plt.show()
822
823 # Compute full risky wealth for Telecom
824 risky_wealth_telcm = start * (1 + df_risky_telcm).cumprod()
825
826 # [?] Plot "CPPI vs Full Risky Allocation" for Telecom (Telcm) (Drawdown 20%)
827 fig, ax = plt.subplots(figsize=(12, 6))
828 btr_20_telcm[["Wealth"]][["Telcm"]].plot(ax=ax, label="CPPI")
829 risky_wealth_telcm[["Telcm"]].plot(ax=ax, style="k--", label="Full Risky Allocation")
830 plt.axhline(y=floor_value, color='gray', linestyle="--", label="Floor Value")
831 plt.title("CPPI vs Full Risky Allocation for Telecom (Telcm) (Drawdown 20%)")
832 plt.legend()
833 plt.show()
834
835 # [?] Compute and display summary statistics for Telecom (Telcm) (Drawdown 20%)
836 summary_stats_df_telcm =
837 btr_20_telcm[["Wealth"]].pct_change().dropna().apply(summary_stats)
838 print(summary_stats_df_telcm)
839
840 ## 2 Random Walks and Asset Simulation
841 # E8. Implement GBM
842
843 def gbm0(n_years=10, n_scenarios=1000, mu=0.07, sigma=0.15, steps_per_year=12,
844 s_0=100.0):
845     """
846         Evolution of a Stock Price using GBM (Geometric Brownian Motion)
847     """
848     dt = 1 / steps_per_year # Step size
849     n_steps = int(n_years * steps_per_year) # Total number of steps
850     xi = np.random.normal(size=(n_steps, n_scenarios)) # Generate random returns
851
852     # Apply the GBM to calculate returns for each step
853     rets = mu * dt + sigma * np.sqrt(dt) * xi # GBM returns formula
854
855     # Calculate prices by cumulative product of (1 + returns)
856     prices = s_0 * (1 + rets).cumprod(axis=0) # Cumulative product of returns
857
858     # Convert numpy array to DataFrame for better handling

```

```

856     prices_df = pd.DataFrame(prices)
857
858     return prices_df
859
860 #Now, generate 3 scenarios for a stock price for 10 years.
861 # Generate sample data with gbm0
862 p = gbm0(n_years=10, n_scenarios=3)
863 print(p.head())
864
865 # Plot the results
866 p.plot(figsize=(12, 6), legend=False)
867 plt.title("Geometric Brownian Motion (Basic Implementation)")
868 plt.show()
869
870 # E9. Simulate 100 scenarios for 10 years.
871 p = gbm0(n_years=10, n_scenarios=100)
872 print(p.head())
873
874 # Plot the results
875 p.plot(figsize=(12, 6), legend=False)
876 plt.title("Geometric Brownian Motion (Basic Implementation)")
877 plt.show()
878
879 #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.
880
881 def gbm(n_years=10, n_scenarios=1000, mu=0.07, sigma=0.15, steps_per_year=12, s_0=100.0):
882     """
883         Optimized Evolution of a Stock Price using GBM (Geometric Brownian Motion)
884     """
885     dt = 1 / steps_per_year # Time step
886     n_steps = int(n_years * steps_per_year) # Total number of time steps
887
888     # Generate the adjusted returns (1 + r_i) using vectorization
889     rets_plus_1 = np.random.normal(loc=1 + mu * dt, scale=sigma * np.sqrt(dt), size=
(n_steps, n_scenarios))
890
891     # Calculate prices by taking the cumulative product of returns for each scenario
892     prices = s_0 * pd.DataFrame(rets_plus_1).cumprod(axis=0) # Optimized cumulative
product calculation
893
894     return prices
895
896 # Apply the optimized GBM function
897 p_optimized = gbm(n_years=10, n_scenarios=100)
898
899 # Plot the optimized results
900 p_optimized.plot(figsize=(12, 6), legend=False)
901 plt.title("Optimized Geometric Brownian Motion")
902 plt.show()
903
904 #If the improvement is not clear, run the following cell to compute the time it takes to
run each GBM implementation.

```

```

905 # Compare performance
906 import time
907 # Timing the execution of gbm0 (non-optimized)
908 start = time.time()
909 gbm0(n_years=5, n_scenarios=1000)
910 end = time.time()
911 print(f"Time taken by gbm0: {end - start:.4f} seconds")
912
913 # Timing the execution of gbm (optimized)
914 start = time.time()
915 gbm(n_years=5, n_scenarios=1000)
916 end = time.time()
917 print(f"Time taken by gbm: {end - start:.4f} seconds")
918
919 #The last refinement ensures that the simulated stock prices start exactly at the initial
920 # value s_0 for all scenarios by setting the first row to 1 in rets_plus_1.
921 def gbm(n_years=10, n_scenarios=1000, mu=0.07, sigma=0.15, steps_per_year=12, s_0=100.0):
922     """
923         Final Evolution of a Stock Price using GBM, ensuring initial value starts at s_0.
924     """
925     dt = 1 / steps_per_year
926     n_steps = int(n_years * steps_per_year)
927     rets_plus_1 = np.random.normal(loc=1 + mu * dt, scale=sigma * np.sqrt(dt), size=(n_steps, n_scenarios))
928     rets_plus_1[0] = 1 # Start all scenarios at s_0
929     prices = s_0 * pd.DataFrame(rets_plus_1).cumprod()
930     return prices
931
932 gbm(n_years=10,n_scenarios=1000).plot(legend=False)
933
934 #Now, we can use GBM to simulate assets and then apply CPPI.
935 #E10. Define a function show_cppi to run a Monte Carlo simulation of the CPPI strategy.
936 # This function uses simulated risky asset returns generated by the gbm function assuming
937 # an initial investment of 100. For the simulation,
938 # use n_scenarios=50, mu=0.07, sigma=0.15, m=3, floor=0.0, riskfree_rate=0.03
939
940 def show_cppi(n_scenarios=50, mu=0.07, sigma=0.15, m=3, floor=0.0, riskfree_rate=0.03,
941 y_max=100):
942     """
943         Plots the result of a Monte Carlo Simulation of CPPI, including a histogram of
944         terminal wealth.
945     """
946     start = 100 # Initial investment
947     sim_rets = gbm(n_years=10, n_scenarios=n_scenarios, mu=mu, sigma=sigma) # Simulate
948     # returns using GBM function
949     risky_r = pd.DataFrame(sim_rets.pct_change().dropna()) # Convert returns to
950     # percentage change
951
952     # Run CPPI back-test
953     btr = run_cppi(risky_r=risky_r, riskfree_rate=riskfree_rate, m=m, start=start,
954     floor=floor)
955     wealth = btr["Wealth"]
956
957     # Calculate terminal wealth stats
958     y_max = wealth.values.max() * y_max / 100

```

```

951     terminal_wealth = wealth.iloc[-1]
952
953     # Plot wealth evolution and terminal wealth histogram
954     fig, (wealth_ax, hist_ax) = plt.subplots(nrows=1, ncols=2, sharey=True, gridspec_kw=
955     {'width_ratios': [3, 2]}, figsize=(24, 9))
956     plt.subplots_adjust(wspace=0.0)
957
958     # Plot wealth evolution
959     wealth.plot(ax=wealth_ax, legend=False, alpha=0.3, color="indianred") # Keep the
960     specified line for wealth evolution
961     wealth_ax.axhline(y=start, ls=":", color="black") # Line for initial wealth
962     wealth_ax.axhline(y=start * floor, ls="--", color="red") # Floor line
963     wealth_ax.set_ylim(top=y_max) # Set the y-axis limit to maximum wealth
964
965     # Plot terminal wealth histogram
966     terminal_wealth.plot.hist(ax=hist_ax, bins=50, ec='w', fc='indianred',
967     orientation='horizontal')
968     hist_ax.axhline(y=start, ls=":", color="black") # Line for initial wealth
969     plt.title("Monte Carlo Simulation of CPPI Strategy with Terminal Wealth
970 Distribution")
971     plt.show()
972
973 # Now, run the simulation with your desired parameters
974 show_cppi(n_scenarios=50, mu=0.07, sigma=0.15, m=3, floor=0.0, riskfree_rate=0.03,
975 y_max=100)
976
977 # E11. Enhance show_cppi by including a histogram of
978 # terminal wealth at the end of the simulation.
979 # This histogram shows the distribution of outcomes
980 # across different scenarios. Also, include additional
981 # statistics, such as mean, median, the
982 # probability of falling below the floor, and
983 # expected shortfall if the floor is violated.
984
985 def show_cppi(n_scenarios=50, mu=0.07, sigma=0.15, m=3, floor=0.0, riskfree_rate=0.03,
986 y_max=100):
987     """
988         Plots the result of a Monte Carlo Simulation of CPPI, including terminal wealth
989         stats.
990     """
991     start = 100 # Starting account value
992     sim_rets = gbm(n_years=40, n_scenarios=n_scenarios, mu=mu, sigma=sigma) # Simulate
993     returns using GBM
994     risky_r = sim_rets.pct_change().dropna() # Calculate the returns from the simulated
995     prices
996
997     # Run CPPI back-test
998     btr = run_cppi(risky_r=risky_r, riskfree_rate=riskfree_rate, m=m, start=start,
999     floor=floor)
1000     wealth = btr["Wealth"]
1001     terminal_wealth = wealth.iloc[-1] # Terminal wealth for each scenario
1002
1003     # Calculate terminal wealth stats
1004     y_max = wealth.values.max() * y_max / 100 # Set y_max based on the maximum wealth
1005     value

```

```

995     tw_mean = terminal_wealth.mean() # Compute mean terminal wealth
996     tw_median = terminal_wealth.median() # Compute median terminal wealth
997     failure_mask = terminal_wealth < start * floor # Mask for floor violations
998     n_failures = failure_mask.sum() # Sum the number of violations
999     p_fail = n_failures / len(terminal_wealth) # Probability of violating the floor
1000    e_shortfall = (terminal_wealth - start * floor)[failure_mask].mean() if n_failures >
0 else 0.0 # Expected shortfall
1001
1002    # Plot wealth evolution and terminal wealth histogram
1003    fig, (wealth_ax, hist_ax) = plt.subplots(nrows=1, ncols=2, sharey=True, gridspec_kw=
{'width_ratios': [3, 2]}, figsize=(24, 9))
1004    plt.subplots_adjust(wspace=0.0)
1005
1006    # Plot wealth evolution
1007    wealth.plot(ax=wealth_ax, legend=False, alpha=0.3, color="indianred") # Wealth
evolution plot
1008    wealth_ax.axhline(y=start, ls=":", color="black") # Initial wealth line
1009    wealth_ax.axhline(y=start * floor, ls="--", color="red") # Floor line
1010    wealth_ax.set_title("CPPI Wealth Evolution")
1011    wealth_ax.set_ylim(top=y_max)
1012
1013    # Plot terminal wealth histogram
1014    terminal_wealth.plot.hist(ax=hist_ax, bins=50, alpha=0.7, color="indianred",
edgecolor="black", orientation='horizontal')
1015    hist_ax.axhline(y=start, ls=":", color="black") # Initial wealth line
1016    hist_ax.axhline(y=start * floor, ls="--", color="red", linewidth=3) # Floor line
1017
1018    # Annotate statistics
1019    hist_ax.annotate(f"Mean: ${int(tw_mean)}", xy=(0.7, 0.9), xycoords='axes fraction',
fontsize=24)
1020    hist_ax.annotate(f"Median: ${int(tw_median)}", xy=(0.7, 0.85), xycoords='axes
fraction', fontsize=24)
1021
1022    # If there are violations, display them, otherwise show 0 violations
1023    if n_failures == 0:
1024        hist_ax.annotate(f"Violations: 0 (0.00%)", xy=(0.7, 0.7), xycoords="axes
fraction", fontsize=24)
1025        hist_ax.annotate(f"Expected Shortfall: $0.00", xy=(0.7, 0.6), xycoords="axes
fraction", fontsize=24) # Expected shortfall 0 if no violation
1026    else:
1027        hist_ax.annotate(f"Violations: {n_failures} ({p_fail *
100:.2f}%) \n Expected Shortfall: ${e_shortfall:.2f}",
xy=(0.7, 0.7), xycoords="axes fraction", fontsize=24)
1028
1029
1030    plt.title("Monte Carlo Simulation of CPPI Strategy with Terminal Wealth
Distribution")
1031    plt.show()
1032
1033    print(f"Configuration: μ={mu}, σ={sigma}")
1034    print(f" Mean Terminal Wealth: ${tw_mean:.2f}")
1035    print(f" Median Terminal Wealth: ${tw_median:.2f}")
1036    print(f" Probability of Violating the Floor: {p_fail * 100:.2f}%")
1037    print(f" Expected Shortfall (if violated): ${e_shortfall:.2f}")
1038    print("-" * 50)
1039

```

```

1040 # Now, run the simulation with your desired parameters
1041 show_cppi(n_scenarios=50, mu=0.07, sigma=0.15, m=3, floor=0.0, riskfree_rate=0.03,
1042 y_max=100)
1043 # R2. You are going to simulate returns and apply CPPI. First, explain how GBM is used
1044 # and computed, and how it can be used into CPPI. Then, perform 5 different simulations of
1045 # returns using GBM with
1046 # different configurations for 1000 scenarios for periods of 40 years. Apply CPPI to the
1047 # simulated returns. Plot a histogram of the terminal wealth at the end of the simulation.
1048 # Also, include additional statistics,
1049 # such as mean, median, the probability of falling below the floor (which is computed
1050 # from the times the floor was violated during the simulations), and the expected shortfall
1051 # if the floor is violated.
1052 # Analyze the results and give some key conclusions about GBM usage for Portfolio
1053 Management and CPPI
1054
1055 # 1. Simulation with mu=0.12, sigma=0.25
1056 mu_1 = 0.12
1057 sigma_1 = 0.25
1058 show_cppi(n_scenarios=1000, mu=mu_1, sigma=sigma_1, m=8, floor=0.9, riskfree_rate=0.03,
1059 y_max=100)
1060
1061 # 2. Simulation with mu=0.07, sigma=0.18
1062 mu_2 = 0.07
1063 sigma_2 = 0.18
1064 show_cppi(n_scenarios=1000, mu=mu_2, sigma=sigma_2, m=6, floor=0.8, riskfree_rate=0.03,
1065 y_max=100)
1066
1067 # 3. Simulation with mu=0.03, sigma=0.10
1068 mu_3 = 0.03
1069 sigma_3 = 0.10
1070 show_cppi(n_scenarios=1000, mu=mu_3, sigma=sigma_3, m=4, floor=0.7, riskfree_rate=0.03,
1071 y_max=100)
1072
1073 # 4. Simulation with mu=0.05, sigma=0.15
1074 mu_4 = 0.05
1075 sigma_4 = 0.15
1076 show_cppi(n_scenarios=1000, mu=mu_4, sigma=sigma_4, m=7, floor=0.8, riskfree_rate=0.03,
1077 y_max=100)
1078
1079 # 5. Simulation with mu=0.02, sigma=0.05
1080 mu_5 = 0.02
1081 sigma_5 = 0.05
1082 show_cppi(n_scenarios=1000, mu=mu_5, sigma=sigma_5, m=10, floor=0.9, riskfree_rate=0.03,
1083 y_max=100)
1084
1085 # 3 Present Value of Liabilities and Funding Ratio
1086 # E12. Implement fucntions to compute the the price of a
1087 # pure discount bond, the PV of liabilities, and the
1088 # funding ratio
1089
1090 def discount(t, r):
1091     """
1092         Compute the price of a pure discount bond that pays $1 at time t,
1093         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
1100     liability amounts at each time.
1101     r: the annual interest rate.
1102
1103     Returns the present value of the liabilities.
1104     """
1105     total_pv = 0
1106     for t, amount in liabilities.items():
1107         total_pv += amount / (1 + r)**t # Discount the amount at time t
1108     return total_pv
1109
1110 #Define a set of liabilities and calculate their present value with a 3% discount rate.
1111 liabilities = pd.Series(data=[1, 1.5, 2, 2.5], index=[3, 3.5, 4, 4.5])
1112 pv(liabilities, 0.03) # Returns 6.233320315080045
1113
1114 def funding_ratio(assets, liabilities, r):
1115     """
1116     Computes the funding ratio given assets, liabilities, and interest rate.
1117
1118     assets: amount of available money to cover the liabilities.
1119     liabilities: a dictionary where keys are times (in years) and values are the
1120     liability amounts at each time.
1121     r: annual interest rate.
1122
1123     Returns the funding ratio (assets / present value of liabilities).
1124     """
1125     pv_liabilities = pv(liabilities, r) # Calculate the present value of the liabilities
1126     return assets / pv_liabilities
1127
1128 #To calculate the funding ratio with $5 in assets and a
1129 # 3% interest rate
1130 funding_ratio(5, liabilities, 0.03) # Returns 0.8021407126958777
1131
1132 #To observe the effect of interest rate changes on the funding ratio, we can recalculate
1133 # it with different rates. A drop in the interest rate generally lowers the funding ratio
1134 # because liabilities' present value increases when discounted at a lower rate.
1135 # 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

```

```

1231     num_steps = int(n_years * steps_per_year) + 1
1232
1233     # Simulate normal variables (Wiener process increments)
1234     shock = np.random.normal(0, scale=np.sqrt(dt), size=(num_steps, n_scenarios))
1235
1236     # Array to hold simulated rates and prices
1237     rates = np.empty_like(shock)
1238     prices = np.empty_like(shock)
1239
1240     # Initial rate
1241     rates[0] = r_0
1242
1243     # CIR model dynamics simulation
1244     h = np.sqrt(a ** 2 + 2 * sigma ** 2)
1245
1246     # Function to calculate the price of a zero-coupon bond
1247     def price(ttm, r): # ttm: time to maturity, r: current interest rate
1248         # Compute A(t, T)
1249         _A = ((2 * h * np.exp((h + a) * ttm / 2)) / (2 * h + (h + a) * (np.exp(h * ttm) - 1))) ** (2 * a * b / sigma ** 2)
1250
1251         # Compute B(t, T)
1252         _B = (2 * (np.exp(h * ttm) - 1)) / (2 * h + (h + a) * (np.exp(h * ttm) - 1))
1253
1254         # Zero-coupon bond price
1255         return _A * np.exp(-_B * r)
1256
1257     # Calculate bond prices at the initial time
1258     prices[0] = price(n_years, rates[0])
1259
1260     # Simulate the rates and prices over time
1261     for step in range(1, num_steps):
1262         r_t = rates[step - 1]
1263         # Simulate the changes in interest rates (Cox-Ingersoll-Ross)
1264         d_r_t = a * (b - r_t) * dt + sigma * np.sqrt(r_t) * shock[step]
1265         rates[step] = abs(r_t + d_r_t)
1266
1267         # Calculate bond prices
1268         prices[step] = price(n_years - step * dt, rates[step]) # Compute the price based
on time to maturity
1269
1270     # Convert rates from force of interest to annualized rates
1271     rates = pd.DataFrame(data=np.exp(rates) - 1, index=range(num_steps))
1272
1273     # Convert prices to a DataFrame
1274     prices = pd.DataFrame(data=prices, index=range(num_steps))
1275
1276     return rates, prices
1277
1278 #Apply the cir model
1279 cir(r_0=0.03,a=0.5,b=0.03,sigma=0.05,n_scenarios=5)[1].plot(legend=False)
1280 plt.show()
1281
1282 # R3. Explain the CIR model and its applications.

```

```

1283 # Use it to simulate 5 interest rates with 10
1284 # scenarios each and different configurations that allow you to
1285 # see how it behaves under different scenarios. Be wise when
1286 # choosing your configurations. Analyze the results and provide
1287 # insights about interest rate modelling.
1288
1289 import matplotlib.ticker as ticker
1290
1291 # Simulating interest rate paths for different configurations
1292
1293 # Configuration 1: High mean reversion speed, low volatility
1294 rates_config1, _ = cir(a=1.0, b=0.03, sigma=0.02, n_scenarios=10)
1295
1296 # Configuration 2: Low mean reversion speed, high volatility
1297 rates_config2, _ = cir(a=0.1, b=0.03, sigma=0.05, n_scenarios=10)
1298
1299 # Configuration 3: Moderate mean reversion speed, high volatility
1300 rates_config3, _ = cir(a=0.5, b=0.03, sigma=0.1, n_scenarios=10)
1301
1302 # Configuration 4: Very high volatility
1303 rates_config4, _ = cir(a=0.5, b=0.03, sigma=0.2, n_scenarios=10)
1304
1305 # Configuration 5: Low mean reversion speed, low volatility
1306 rates_config5, _ = cir(a=0.1, b=0.03, sigma=0.02, n_scenarios=10)
1307
1308 # Plotting the results
1309 fig, axes = plt.subplots(5, 1, figsize=(12, 12), sharex=True)
1310
1311 configs = [
1312     (rates_config1, "Configuration 1: High Mean Reversion & Low Volatility"),
1313     (rates_config2, "Configuration 2: Low Mean Reversion & High Volatility"),
1314     (rates_config3, "Configuration 3: Moderate Mean Reversion & High Volatility"),
1315     (rates_config4, "Configuration 4: Very High Volatility"),
1316     (rates_config5, "Configuration 5: Low Mean Reversion & Low Volatility")
1317 ]
1318
1319 for ax, (rates, title) in zip(axes, configs):
1320     rates.plot(ax=ax, legend=False)
1321     ax.set_title(title, fontsize=10)
1322     ax.set_ylabel("Interest Rate", fontsize=8)
1323     ax.yaxis.set_major_locator(ticker.MaxNLocator(5)) # Reduce number of Y ticks
1324     ax.xaxis.set_major_locator(ticker.MaxNLocator(10)) # Reduce number of X ticks
1325     ax.tick_params(axis="both", which="major", labelsize=8) # Adjust font size
1326
1327 axes[-1].set_xlabel("Time (Months)", fontsize=10)
1328 plt.xticks(rotation=45) # Rotate X labels for better readability
1329
1330 plt.tight_layout()
1331 plt.show()
1332
1333 #In class, we also talked about the risk perspective of using cash vs bonds to fund
liabilities.
1334 #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.

1335
1336 import pandas as pd
1337
1338 # Initial cash on hand (in millions)
1339 a_0 = 0.75
1340
1341 # Simulate interest rates and bond prices using the CIR model
1342 rates, bond_prices = cir(r_0=0.03, b=0.03, n_scenarios=10)
1343
1344 # Assume that liabilities are the bond prices
1345 liabilities = bond_prices
1346
1347 # Present value of a Zero-Coupon Bond (ZCB) maturing in 10 years
1348 zcbond_10 = pd.Series(data=[1], index=[10]) # $1 received in year 10
1349
1350 # Compute the present value of the ZC bond today with a 3% interest rate
1351 zc_0 = pv(zcbond_10, 0.03) # The present value of $1 in 10 years at 3% discount rate
1352
1353 # How many bonds can we buy?
1354 n_bonds = a_0 / zc_0
1355
1356 # Asset value assuming we invest in Zero-Coupon Bonds
1357 av_zc_bonds = n_bonds * bond_prices
1358
1359 # Asset value assuming we invest in cash
1360 av_cash = a_0 * (rates / 12 + 1).cumprod()
1361
1362 av_cash.plot(legend=False)
1363 plt.show()
1364
1365 #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.
1366 av_zc_bonds.plot(legend=False)
1367 plt.show()
1368
1369 #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.
1370 (av_cash/liabilities).pct_change().plot(title='Returns of Funding Ratio with Cash (10
#scenarios)', legend=False, figsize=(12,5))
1371 plt.show()
1372
1373 #As we said in class it's very risky.
1374 (av_zc_bonds/liabilities).pct_change().plot(title='Returns of Funding Ratio with Bonds
#(10 scenarios)', legend=False, figsize=(12,5))
1375 plt.show()
1376
1377 # E15. Compute the final funding ratio using the CIR model
1378
1379 # Initial cash on hand (in millions)
1380 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
1411 # million. Repeat this time stating with 0.5
1412
1413 # Initial cash on hand (0.5 million)
1414 a_0 = 0.5
1415
1416 # Simulate interest rates and bond prices using the CIR model
1417 rates, bond_prices = cir(r_0=0.03, b=0.03, n_scenarios=10000)
1418
1419 # Assume liabilities are the bond prices
1420 liabilities = bond_prices
1421
1422 # Define a zero-coupon bond that pays $1 in 3 years
1423 zcbond_10 = pd.Series(data=[1], index=[3])
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

```

```

1434 # Asset value assuming we invest in cash
1435 av_cash = a_0 * (rates / 12 + 1).cumprod()
1436
1437 # Compute the terminal funding ratio (TFR) for cash and bonds
1438 tfr_cash = av_cash.iloc[-1] / liabilities.iloc[-1]
1439 tfr_zc_bonds = av_zc_bonds.iloc[-1] / liabilities.iloc[-1]
1440
1441 # Plot the histogram of the terminal funding ratios
1442 ax = tfr_cash.plot.hist(label="Cash", figsize=(15,6), bins=100, legend=True)
1443 tfr_zc_bonds.plot.hist(ax=ax, label="ZC Bonds", bins=100, legend=True, secondary_y=True)
1444 plt.show()
1445
1446 #R4. Assume that your liability is in Cash. Use the previous 5 scenarios you defined in
1447 # R3 for the CIR modelling, and compute the funding ratio. Provide an analysis on how to
1448 # use CIR to model liabilities and the funding ratio
1449 #Configuration 1
1450
1451 # Initial cash on hand (0.75 million)
1452 a_0 = 0.75
1453
1454 # Simulate interest rates and bond prices using the CIR model
1455 rates_1, bond_prices_1 = cir(a=0.7, b=0.03, sigma=0.02, n_scenarios=10000)
1456
1457 # Assume liabilities are the bond prices
1458 liabilities_1 = bond_prices_1
1459
1460 # Define a zero-coupon bond that pays $1 in 10 years
1461 zcbond_10 = pd.Series(data=[1], index=[10])
1462
1463 # Compute the present value of the zero-coupon bond at a 3% interest rate
1464 zc_0 = pv(zcbond_10, 0.03)
1465
1466 # Determine how many bonds can be purchased
1467 n_bonds = a_0 / zc_0
1468
1469 # Asset value assuming we buy bonds
1470 av_zc_bonds_1 = n_bonds * bond_prices_1
1471
1472 # Asset value assuming we invest in cash
1473 av_cash_1 = a_0 * (rates_1 / 12 + 1).cumprod()
1474
1475 # Compute the terminal funding ratio (TFR) for cash and bonds
1476 tfr_cash_1 = av_cash_1.iloc[-1] / liabilities_1.iloc[-1]
1477 tfr_zc_bonds_1 = av_zc_bonds_1.iloc[-1] / liabilities_1.iloc[-1]
1478
1479 # Plot the histogram of the terminal funding ratios
1480 ax = tfr_cash_1.plot.hist(label="Cash", figsize=(15,6), bins=100, legend=True)
1481 tfr_zc_bonds_1.plot.hist(ax=ax, label="ZC Bonds", bins=100, legend=True,
1482 secondary_y=True)
1483 plt.title("Funding Ratio Distribution - High Mean Reversion & Low Volatility")
1484 plt.xlabel("Funding Ratio")
1485 plt.show()
1486
1487 #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

```

```

1538 # Determine how many bonds can be purchased
1539 n_bonds = a_0 / zc_0
1540
1541 # Asset value assuming we buy bonds
1542 av_zc_bonds_3 = n_bonds * bond_prices_3
1543
1544 # Asset value assuming we invest in cash
1545 av_cash_3 = a_0 * (rates_3 / 12 + 1).cumprod()
1546
1547 # Compute the terminal funding ratio (TFR) for cash and bonds
1548 tfr_cash_3 = av_cash_3.iloc[-1] / liabilities_3.iloc[-1]
1549 tfr_zc_bonds_3 = av_zc_bonds_3.iloc[-1] / liabilities_3.iloc[-1]
1550
1551 # Plot the histogram of the terminal funding ratios
1552 ax = tfr_cash_3.plot.hist(label="Cash", figsize=(15,6), bins=100, legend=True)
1553 tfr_zc_bonds_3.plot.hist(ax=ax, label="ZC Bonds", bins=100, legend=True,
secondary_y=True)
1554 plt.title("Funding Ratio Distribution - Moderate Mean Reversion & High Volatility")
1555 plt.xlabel("Funding Ratio")
1556 plt.show()
1557
1558 # Configuration 4
1559
1560 # Initial cash on hand (0.75 million)
1561 a_0 = 0.75
1562
1563 # Simulate interest rates and bond prices using the CIR model
1564 rates_4, bond_prices_4 = cir(a=0.5, b=0.03, sigma=0.2, n_scenarios=10000)
1565
1566 # Assume liabilities are the bond prices
1567 liabilities_4 = bond_prices_4
1568
1569 # Define a zero-coupon bond that pays $1 in 10 years
1570 zcbond_10 = pd.Series(data=[1], index=[10])
1571
1572 # Compute the present value of the zero-coupon bond at a 3% interest rate
1573 zc_0 = pv(zcbond_10, 0.03)
1574
1575 # Determine how many bonds can be purchased
1576 n_bonds = a_0 / zc_0
1577
1578 # Asset value assuming we buy bonds
1579 av_zc_bonds_4 = n_bonds * bond_prices_4
1580
1581 # Asset value assuming we invest in cash
1582 av_cash_4 = a_0 * (rates_4 / 12 + 1).cumprod()
1583
1584 # Compute the terminal funding ratio (TFR) for cash and bonds
1585 tfr_cash_4 = av_cash_4.iloc[-1] / liabilities_4.iloc[-1]
1586 tfr_zc_bonds_4 = av_zc_bonds_4.iloc[-1] / liabilities_4.iloc[-1]
1587
1588 # Plot the histogram of the terminal funding ratios
1589 ax = tfr_cash_4.plot.hist(label="Cash", figsize=(15,6), bins=100, legend=True)

```

```

1590 tfr_zc_bonds_4.plot.hist(ax=ax, label="ZC Bonds", bins=100, legend=True,
1591 secondary_y=True)
1592 plt.title("Funding Ratio Distribution - Very High Volatility")
1593 plt.xlabel("Funding Ratio")
1594 plt.show()

1595 #Configuration 5

1596

1597 # Initial cash on hand (0.75 million)
1598 a_0 = 0.75
1599

1600 # Simulate interest rates and bond prices using the CIR model
1601 rates_5, bond_prices_5 = cir(a=0.1, b=0.03, sigma=0.02, n_scenarios=10000)
1602

1603 # Assume liabilities are the bond prices
1604 liabilities_5 = bond_prices_5
1605

1606 # Define a zero-coupon bond that pays $1 in 10 years
1607 zcbond_10 = pd.Series(data=[1], index=[10])
1608

1609 # Compute the present value of the zero-coupon bond at a 3% interest rate
1610 zc_0 = pv(zcbond_10, 0.03)
1611

1612 # Determine how many bonds can be purchased
1613 n_bonds = a_0 / zc_0
1614

1615 # Asset value assuming we buy bonds
1616 av_zc_bonds_5 = n_bonds * bond_prices_5
1617

1618 # Asset value assuming we invest in cash
1619 av_cash_5 = a_0 * (rates_5 / 12 + 1).cumprod()
1620

1621 # Compute the terminal funding ratio (TFR) for cash and bonds
1622 tfr_cash_5 = av_cash_5.iloc[-1] / liabilities_5.iloc[-1]
1623 tfr_zc_bonds_5 = av_zc_bonds_5.iloc[-1] / liabilities_5.iloc[-1]
1624

1625 # Plot the histogram of the terminal funding ratios
1626 ax = tfr_cash_5.plot.hist(label="Cash", figsize=(15,6), bins=100, legend=True)
1627 tfr_zc_bonds_5.plot.hist(ax=ax, label="ZC Bonds", bins=100, legend=True,
1628 secondary_y=True)
1629 plt.title("Funding Ratio Distribution - Low Mean Reversion & Low Volatility")
1630 plt.xlabel("Funding Ratio")
1631 plt.show()

1632 # 5 GHP and Duration Matching
1633

1634 #E16. Implement a function to compute the returns of a bond. Then implement a fucntion to
1635 # use these cashflows to compute the price of the bond
1636
1637 def bond_cash_flows(maturity, principal=100, coupon_rate=0.03, coupons_per_year=12):
1638     """
1639         Returns a series of cash flows generated by a bond, indexed by coupon number.
1640     """
1641     # Calculate number of coupons

```

```

1641     n_coupons = round(maturity * coupons_per_year) # Total number of coupon payments
1642
1643     # Calculate the coupon amount
1644     coupon_amt = principal * coupon_rate / coupons_per_year # Amount paid at each coupon
date
1645
1646     # Generate the times of coupon payments
1647     coupon_times = np.arange(1, n_coupons + 1)
1648
1649     # Create the cash flows (all coupon payments)
1650     cash_flows = pd.Series(data=coupon_amt, index=coupon_times)
1651
1652     # Add the principal amount to the final cash flow (maturity)
1653     cash_flows.iloc[-1] += principal # The final payment includes the principal
1654
1655     return cash_flows
1656
1657 #To get cash flows for a 3-year bond with a 3% coupon rate paid semiannually
1658 bond_cash_flows(3, 100, 0.03, 2)
1659
1660 #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.
1661
1662 def bond_price(maturity, principal=100, coupon_rate=0.03, coupons_per_year=12,
discount_rate=0.03):
1663     """
1664     Price a bond based on bond parameters and discount rate.
1665     """
1666     # Obtain the bond's cash flows
1667     cash_flows = bond_cash_flows(maturity, principal, coupon_rate, coupons_per_year)
1668
1669     # Discount rate per period
1670     period_rate = discount_rate / coupons_per_year # Monthly discount rate
1671
1672     # Present value of the cash flows
1673     pv_cash_flows = cash_flows / (1 + period_rate) ** cash_flows.index
1674
1675     # Sum the present values of all cash flows to get the bond price
1676     bond_price = pv_cash_flows.sum()
1677
1678     return bond_price
1679
1680 #Calculate the price of a 20-year bond with a 5% coupon rate and a 4% discount rate
1681 bond_price(20, 1000, 0.05, 2, 0.04)
1682
1683 #Now, assume we have the following rates and compute the bonds with these rates. How can
you interpret the resulting plot?
1684
1685 rates=np.linspace(0.01,.1,num=20)
1686 rates
1687
1688 prices= [bond_price(10,1000,0.05,2,rate) for rate in rates]
1689 pd.DataFrame(data=prices,index=rates).plot(title="Prices 10y Bond with different Interest
Rate", legend=False)

```

```

1690
1691 #The problem with these bonds is the intermediate cash flows. Remember that a bond with
1692 coupons are multiple
1693 # zero coupon bonds together, and the problem is that some of those have a short time
1694 term.
1695
1696
1697 # We have cashflows, but it's better to get 30 today than 3 years in the future, right?
1698 Thus, how long are we waiting until we get the cashflows? We could compute the weighted
1699 average time.
1700
1701 discounts=discount(cf.index,0.06/2)
1702 discounts
1703
1704 #These are the discount factors. Now, we discounted values:
1705 dcf= cf*discounts #Multiply cashflows by the discount factors
1706 dcf
1707
1708 #This is the discounted values of the present values for the cashflows. Now, we can get
1709 the weights.
1710
1711 # Calculate the sum of the discounted cash flows
1712 total_dcf = dcf.sum() # Total present value of all cash flows
1713
1714 # Calculate the weights for each cash flow
1715 weights = dcf / total_dcf # Weight is the proportion of each discounted cash flow to the
1716 total discounted value
1717 print("Weights of the cash flows:")
1718 print(weights)
1719
1720 # Finally, we make a weighted average.
1721 (cf.index*weights).sum()
1722
1723 #E17. Implement a function that computes the Macaulay Duration
1724
1725 import numpy as np
1726
1727 def macaulay_duration(flops, discount_rate):
1728     """
1729         Computes the Macaulay Duration of a bond.
1730
1731     Arguments:
1732         - flops: pandas Series containing the bond's cash flows (index = time)
1733         - discount_rate: the periodic discount rate (decimal), NOT annual!
1734
1735     Returns:
1736         - Macaulay Duration: weighted average time until cash flows are received.
1737     """
1738
1739     # **Paso 1**: Calcular factores de descuento
1740     discounts = (1 + discount_rate) ** (-flops.index) # Descuento aplicado correctamente
1741
1742     # **Paso 2**: Calcular los flujos descontados (DCF)

```

```

1738     dcf = flows * discounts # Multiplicamos cash flows por factores de descuento
1739
1740     # **Paso 3**: Obtener la suma total de los flujos descontados
1741     total_dcf = dcf.sum() # Total de los flujos descontados
1742
1743     # **Paso 4**: Calcular los pesos (proporción de cada flujo descontado)
1744     weights = dcf / total_dcf # Normalizamos los flujos descontados
1745
1746     # **Paso 5**: Calcular la duración de Macaulay como media ponderada
1747     duration = np.dot(flows.index, weights) # Media ponderada usando producto punto
1748
1749     return duration
1750
1751 macaulay_duration(bond_cash_flows(3,1000,0.06,2), 0.06/2)
1752
1753 #If a zero-coupon bond does not match the maturity of our liabilities, we can use two
1754 bonds with different maturities to achieve the desired duration through weighted
1755 allocation.
1756 #define liabilities
1757 liabilities = pd.Series(data=[100000, 100000], index=[10,12])
1758
1759 #Now, we define bonds with different maturities and calculate their Macaulay durations.
1760 md_10 = macaulay_duration(bond_cash_flows(10, 1000, 0.05, 1), 0.04)
1761 md_20 = macaulay_duration(bond_cash_flows(20, 1000, 0.05, 1), 0.04)
1762 md_10, md_20
1763
1764 #E18.Implement the match duration function to determine the weights of each bond required
1765 to match the liability duration
1766
1767 def match_duration(cf_t, cf_s, cf_l, discount_rate):
1768     """
1769         Returns the weight W in cf_s to match duration for target cash flows cf_t.
1770
1771     Arguments:
1772         - cf_t: Target cash flows (liabilities)
1773         - cf_s: Cash flows of the short-duration bond
1774         - cf_l: Cash flows of the long-duration bond
1775         - discount_rate: Discount rate for duration calculation
1776
1777     Returns:
1778         - W: Weight of the short-duration bond needed to match liability duration
1779     """
1780
1781     d_t = macaulay_duration(cf_t, discount_rate) # Duration of target liabilities
1782     d_s = macaulay_duration(cf_s, discount_rate) # Duration of short bond
1783     d_l = macaulay_duration(cf_l, discount_rate) # Duration of long bond
1784
1785     # Compute the weight W for the short-duration bond
1786     W = (d_l - d_t) / (d_l - d_s)
1787
1788     return W # Return the computed weight
1789
1790 # Define the liabilities (payments at year 10 and 12)
1791 liabilities = pd.Series(data=[100000, 100000], index=[10, 11]) # Payments of 100,000 at
1792 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 liabilities 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

```

```

1832 # Compute funding ratio for long bond
1833 funding_ratios_long = [funding_ratio(long_bond, liabilities, rate, rate) for rate in
1834 rates]
1835
1836 # Compute funding ratio for short bond
1837 funding_ratios_short = [funding_ratio(short_bond, liabilities, rate, rate) for rate in
1838 rates]
1839
1840 # Compute funding ratio for the matched portfolio (using portfolio_flows and liabilities)
1841 funding_ratios_portfolio = [funding_ratio(portfolio_flows, liabilities, rate, rate) for
1842 rate in rates]
1843
1844 # Plot Funding Ratio for Long Bond
1845 plt.figure(figsize=(8, 5))
1846 plt.plot(rates, funding_ratios_long, label="Funding Ratio (Long Bond)", linestyle="--",
1847 color="blue")
1848 plt.xlabel("Interest Rate")
1849 plt.ylabel("Funding Ratio")
1850 plt.title("Funding Ratio Sensitivity (Long Bond)")
1851 plt.legend()
1852 plt.grid(True)
1853 plt.show()
1854
1855 # Plot Funding Ratio for Short Bond
1856 plt.figure(figsize=(8, 5))
1857 plt.plot(rates, funding_ratios_short, label="Funding Ratio (Short Bond)", linestyle="-.",
1858 color="red")
1859 plt.xlabel("Interest Rate")
1860 plt.ylabel("Funding Ratio")
1861 plt.title("Funding Ratio Sensitivity (Short Bond)")
1862 plt.legend()
1863 plt.grid(True)
1864 plt.show()
1865
1866 # Plot Funding Ratio for Matched Portfolio (with portfolio_flows)
1867 plt.figure(figsize=(8, 5))
1868 plt.plot(rates, funding_ratios_portfolio, label="Funding Ratio (Matched Portfolio)",
1869 linewidth=2, color="green")
1870 plt.xlabel("Interest Rate")
1871 plt.ylabel("Funding Ratio")
1872 plt.title("Funding Ratio Sensitivity (Matched Portfolio)")
1873 plt.legend()
1874 plt.grid(True)
1875 plt.show()
1876
1877 # 6 Simulation of Prices of Coupon Bonds using CIR
1878
1879 def discount (t,r):
1880     """
1881         Compute the price of a pure discount bond that pays a dollar at time t, g
1882         ien an interest rate r. Returns a |t|x|r| series or data frame r can be a flo
1883         at, series or data frame
1884         returns a datafram indexed by t

```

```

1880 """
1881 discounts=pd.DataFrame([(r+1)**-i for i in t])
1882 discounts.index=t
1883 return (discounts)
1884
1885 def pv(flows,r):
1886 """
1887     Computes PV of a set of liabilities
1888     flows is indexed by the time and amounts
1889     r can be a scalar, a series, or a dataframe with the number of rows matching
1890     the num of rows in flows
1891 """
1892 dates= flows.index
1893 discounts= discount(dates,r)
1894 return discounts.multiply(flows, axis='rows').sum()
1895
1896 bond_price(5,100,0.05,12,0.03)
1897
1898 #simulate rates and zc prices using CIR
1899 rates, zc_prices=cir(10,500,b=0.03,r_0=0.03)
1900 #bond prices
1901 selected_rates = rates.iloc[0, [1, 2, 3]]
1902 for rate in selected_rates:
1903     price = bond_price(5, 100, 0.05, 12, rate) # Calcular precio del bono para cada tasa
1904     print(f"Bond price with rate {rate}: {price}")
1905
1906 selected_rates = rates.iloc[1, [1, 2, 3]]
1907 for rate in selected_rates:
1908     price = bond_price(5, 100, 0.05, 12, rate) # Calcular precio del bono para cada tasa
1909     print(f"Bond price with rate {rate}: {price}")
1910
1911 rates[[1,2,3]].head()
1912
1913 #The bond_price function calculates the price of a bond that pays coupons until maturity,
1914 #with the principal returned at maturity. The function can handle a discount_rate input as
1915 #a DataFrame to simulate varying rates over time
1916
1917 def bond_price(maturity, principal=100, coupon_rate=0.03, coupons_per_year=12,
1918 discount_rate=0.03):
1919 """
1920     Computes the price of a bond that pays coupons until maturity.
1921 """
1922     if isinstance(discount_rate, pd.DataFrame):
1923         pricing_dates = discount_rate.index
1924         prices = pd.DataFrame(index=pricing_dates, columns=discount_rate.columns)
1925         for t in pricing_dates:
1926             prices.loc[t] = bond_price(maturity - t / coupons_per_year, principal,
1927             coupon_rate, coupons_per_year, discount_rate.loc[t])
1928         return prices
1929     else:
1930         if maturity <= 0:
1931             return principal + principal * coupon_rate / coupons_per_year
1932         cash_flows = bond_cash_flows(maturity, principal, coupon_rate, coupons_per_year)
1933         return pv(cash_flows, discount_rate / coupons_per_year)

```

```

1930
1931 #Using the simulated interest rates, we calculate bond prices over time with the
1932 bond_price function. The price evolution shows how interest rate fluctuations affect bond
1933 prices.
1934 bond_price(10,100,0.05,12,rates[[1,2,3,4,5]]).plot(legend=False, figsize=(12,6))
1935 plt.show()
1936
1937 #To evaluate bond performance, we calculate the annualized bond returns based on
1938 #percentage changes in bond prices. This reveals how bond returns respond to interest rate
1939 #changes
1940
1941 def annualize_rets(r, periods_per_year):
1942     """
1943         Annualizes a set of periodic returns.
1944
1945         Parameters:
1946             - r: pd.Series or pd.DataFrame of returns
1947             - periods_per_year: Number of compounding periods per year (e.g., 12 for monthly, 252
1948             for daily)
1949
1950         Returns:
1951             - Annualized return
1952             """
1953         compounded_growth = (1 + r).prod() # Cumulative growth
1954         n_periods = len(r) # Number of periods in the return series
1955         return compounded_growth ** (periods_per_year / n_periods) - 1 # Annualized return
1956 formula
1957
1958
1959 prices=bond_price(10,100,0.05,12,rates[[1,2,3,4,5]])
1960 br = prices.pct_change().dropna()
1961 annualize_rets(br, 12)
1962
1963 #E19. Compute the bond total returns
1964
1965
1966 def bond_total_return(monthly_prices, principal, coupon_rate, coupons_per_year):
1967     """
1968         Computes the total return of a bond, including coupon payments.
1969
1970         Parameters:
1971             - monthly_prices: DataFrame of monthly bond prices (each column is a different
1972               scenario)
1973             - principal: Bond principal (face value)
1974             - coupon_rate: Annual coupon rate of the bond
1975             - coupons_per_year: Number of coupon payments per year
1976
1977         Returns:
1978             - DataFrame of total returns, accounting for both price changes and coupon payments.
1979             """
1980
1981         # Initialize DataFrame for coupons, matching the structure of monthly_prices
1982         coupons = pd.DataFrame(0, index=monthly_prices.index, columns=monthly_prices.columns)
1983         t_max = monthly_prices.index.max()
1984
1985         # Define payment dates (in terms of months) for the coupons

```

```

1977 pay_dates = np.linspace(12 / coupons_per_year, t_max, int(coupons_per_year * t_max / 12), dtype=int)
1978
1979 # Populate coupon payments at the defined payment dates
1980 coupons.iloc[pay_dates] = principal * coupon_rate / coupons_per_year
1981
1982 # Calculate total returns by adding the coupon payments to the bond price (monthly)
1983 price_changes = monthly_prices.pct_change() # Price changes (percentage return)
1984 total_returns = price_changes + (coupons / monthly_prices) # Add coupon yield
1985
1986 # Drop the initial row since it has no previous price for return calculation
1987 return total_returns.dropna()
1988
1989 p=bond_price(10,100,0.05,12,rates[[1,2,3,4,]])]
1990 btr=bond_total_return(p,100,0.05,12)
1991 annualize_rets(btr,12)
1992
1993 # As we can see, bond prices change over time. The dynamics of interest rates is
affecting us
1994
1995 price_10=bond_price(10,100,0.05,12,rates)
1996 price_10[[1,2,3]].tail()
1997
1998 # Assume bond_price function is defined as before, and rates for 10 years have been
generated
1999
2000 # Let's extend the rates for 30 years (as an example, we repeat the last 10 years of
rates for simplicity)
2001 extended_rates = rates.iloc[:, :10].reindex(range(30), method='ffill') # Repeat the
rates for 30 years
2002
2003 # Now we can calculate the price for 30-year bonds using the extended rates
2004 price_30 = bond_price(30, 100, 0.05, 12, extended_rates)
2005
2006 # Display the last few rows for the scenarios 1, 2, 3 to see how the price converges
2007 print(price_30[[1, 2, 3]].tail())
2008
2009 #So, bonds tend to be thought as safe, but they are not, this is clear when the maturity is
long.
2010 #Let's make a portfolio with the two bonds. Use a 60/40 combination using the 10 and 30
year bonds'
2011
2012 rets_30 = bond_total_return(price_30, 100, 0.05, 12)
2013 rets_10 = bond_total_return(price_10, 100, 0.05, 12)
2014
2015 # Rebalance mensual para mantener una asignación 60/40:
2016 rets_bonds = 0.6 * rets_10 + 0.4 * rets_30
2017
2018 # Calcular los rendimientos medios para cada mes
2019 mean_rets_bonds = rets_bonds.mean(axis=1) # Promedio para cada fila (mes)
2020
2021 # Asegurarnos de que los valores sean numéricos
2022 mean_rets_bonds = pd.to_numeric(mean_rets_bonds, errors='coerce')
2023

```

```

2024 # Llamar a la función summary_stats con los rendimientos medios
2025 summary_stats(mean_rets_bonds)
2026
2027 #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.¶
2028
2029
2030
2031 # Function to calculate summary statistics (mean, volatility, skewness, kurtosis, VaR,
CVar, etc.)
2032 def summary_stats(returns):
2033     # Calculate summary statistics
2034     mean_return = returns.mean()
2035     vol = returns.std()
2036     skew = returns.skew()
2037     kurt = returns.kurt()
2038
2039     # Historical VaR (5%)
2040     VaR_5 = np.percentile(returns, 5)
2041
2042     # CVar (5%)
2043     below_VaR = returns[returns <= VaR_5]
2044     CVar_5 = below_VaR.mean() if len(below_VaR) > 0 else np.nan
2045
2046     # Sharpe ratio
2047     sharpe_ratio = mean_return / vol if vol != 0 else np.nan
2048
2049     # Max Drawdown
2050     cumulative_returns = (1 + returns).cumprod()
2051     peak = cumulative_returns.cummax()
2052     drawdown = (cumulative_returns - peak) / peak
2053     max_drawdown = drawdown.min()
2054
2055     # Summary of statistics
2056     stats = pd.Series({
2057         'Mean Return': mean_return,
2058         'Volatility': vol,
2059         'Skewness': skew,
2060         'Kurtosis': kurt,
2061         'VaR (5%)': VaR_5,
2062         'CVar (5%)': CVar_5,
2063         'Sharpe Ratio': sharpe_ratio,
2064         'Max Drawdown': max_drawdown
2065     })
2066
2067     return stats
2068
2069 # 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 +
2087     sigma * np.sqrt(dt) * z)
2088
2089 # Convert prices to returns (percent changes)
2090 equity_returns =
2091 pd.DataFrame(equity_prices).pct_change(axis='columns').dropna(axis='columns')
2092
2093 # 2. Combine Bond Returns (rets_bonds) with Equities in a 70/30 Portfolio
2094 # Assuming rets_bonds is a previously calculated series of bond returns
2095 rets_bonds = pd.DataFrame(np.random.normal(0.03, 0.04, size=(500, 120))) # Simulated
2096 bond returns as an example
2097
2098 # Combine the bond (30%) and equity (70%) returns
2099 portfolio_returns = 0.7 * equity_returns + 0.3 * rets_bonds
2100
2101 # 3. Calculate risk-return statistics for the 70/30 Portfolio
2102 portfolio_stats = summary_stats(portfolio_returns.mean(axis=1))
2103
2104 # 4. Calculate statistics for Bonds only (60/40 Portfolio)
2105 bond_stats = summary_stats(rets_bonds.mean(axis=1))
2106
2107 # 5. Print the full statistics
2108 print("70/30 Portfolio Statistics (Equities + Bonds):")
2109 print(portfolio_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

```

```

2121 #7 Naive Risk Budgeting Strategies between the PSP and GHP
2122
2123 #allocator is a free function to allocate that the user gives. **kwargs allows us to take
2124 #the function and whichever variable within.
2125 def bt_mix(r1,r2, allocator, **kwargs):
2126     """
2127     Runs a back test (simulation) of allocating between a two sets of returns
2128     r1 and r2 are TxN DataFrames or returns where T is the time step and N is the number
2129     of scenarios.
2130     allocator is a function that takes two set of returns and allocator specific
2131     parameters, and produces
2132     an allocation tot the first portfolio (the rest of the money is invested in the GHP)
2133     as a T x 1 DataFrame
2134     Returns a TxN DataFrame of the resulting N portfolio scenarios.
2135     """
2136     if not r1.shape== r2.shape:
2137         raise ValueError("r1 and r2 need to have the same shape")
2138     weights=allocator(r1,r2, **kwargs) #The allocator takes r1, r2 and a bunch of other
2139     variables
2140     if not weights.shape==r1.shape:
2141         raise ValueError("Allocator results not matching r1 shape")
2142     r_mix=weights*r1+(1-weights)*r2
2143     return r_mix
2144
2145 #The following function produces a time series over T steps of allocation between the PSP
2146 #and the GHP across N scenarios.
2147
2148 def fixedmix_allocator(r1,r2, w1, **kwargs):
2149     """
2150     Produces a time series over T steps of allocation between the PSP
2151     and the GHP across N scenarios.
2152     PSP and GHP are TxN DataFrames that represent the returns of the PSP and GHP such
2153     that:
2154     each column is a scenario
2155     each row is the price for a timestep
2156     returns an TxN DataFrame of PSP Weights
2157     """
2158     return pd.DataFrame(data=w1, index=r1.index, columns=r1.columns)
2159
2160 #The following function is the same as before to compute the total bond return.
2161
2162 def bond_total_return(monthly_prices, principal, coupon_rate,coupons_per_year):
2163     """
2164     Computes the total return of a bond on monthly bond prices and coupon payments
2165     Assumes that dividends (coupons) are paid out at the end of the period (e.g. end of 3
2166     months for quaterly div) and that dividens are reinvested in the bond
2167     """
2168     coupons=pd.DataFrame(data=0, index=monthly_prices.index,
2169     columns=monthly_prices.columns)
2170     t_max=monthly_prices.index.max()
2171     pay_date=np.linspace(12/coupons_per_year,t_max, int(coupons_per_year*t_max/12),
2172     dtype=int)
2173     coupons.iloc[pay_date]=principal*coupon_rate/coupons_per_year
2174     total_returns=(monthly_prices+coupons)/monthly_prices.shift()-1
2175     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,
2185 s_0=100.0,prices=True):
2186     """
2187         Evolution of a Stock Price using GBM
2188     """
2189     dt=1/steps_per_year
2190     n_steps=int(n_years*steps_per_year)
2191
2192     if prices:
2193         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
2194         rets_plus_1[0]=1
2195         prices=s_0*pd.DataFrame(rets_plus_1).cumprod()
2196         return prices
2197     else:
2198         rets=np.random.normal(loc=mu*dt,scale=sigma*np.sqrt(dt),size=(n_steps+1,n_scenarios))
2199         return rets
2200
2201 price_eq=gbm(n_years=10,n_scenarios=500,mu=0.07,sigma=0.15)
2202 rets_eq=price_eq.pct_change().dropna()
2203 rets_zc=zc_prices.pct_change().dropna()
2204
2205 #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.
2206 # 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().
2207 Compare the results of the two approaches. Now, repeat the same using an allocation og 60/40.
2208 # Compare the results, which annualized return is greater? What are your insights about the results?
2209 # 70/30 Allocation: 70% in equities and 30% in bonds
2210 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
2233 # returns
2234 annualized_return_70_30_avg = (1 + mean_rets_70_30.mean())**12 - 1
2235 annualized_return_70_30_full = (1 + rets_70_30.mean().mean())**12 - 1
2236
2237 annualized_return_60_40_avg = (1 + mean_rets_60_40.mean())**12 - 1
2238 annualized_return_60_40_full = (1 + rets_60_40.mean().mean())**12 - 1
2239
2240 # Print the summary statistics for both portfolios
2241 print("Summary statistics for the 70/30 portfolio with averaged returns:")
2242 print(summary_stats_70_30_avg)
2243
2244 print("\nSummary statistics for the 70/30 portfolio without averaging returns:")
2245 print(summary_stats_70_30_full)
2246
2247 print("\nSummary statistics for the 60/40 portfolio with averaged returns:")
2248 print(summary_stats_60_40_avg)
2249
2250 print("\nSummary statistics for the 60/40 portfolio without averaging returns:")
2251 print(summary_stats_60_40_full)
2252
2253 # Comparison of the annualized returns
2254 print("\nAnnualized return (70/30, averaged returns):", annualized_return_70_30_avg)
2255 print("Annualized return (70/30, without averaging returns):", annualized_return_70-
2256 _30_full)
2257
2258 print("\nAnnualized return (60/40, averaged returns):", annualized_return_60_40_avg)
2259 print("Annualized return (60/40, without averaging returns):", annualized_return_60-
2260 _40_full)
2261
2262 #Now, we look at the terminal values. The following function returns the final values of
2263 # 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
2264         scenario.
2265         """
2266
2267     return (rets + 1).prod()
2268
2269
2270 def terminal_stats(rets, floor=0.8, cap=np.inf, name="Stats"):
2271     """
2272         Produce Summary Statistics on the terminal value per invested dollar across a range
2273         of N scenarios.
2274         rets is a TxN DataFrame of returns, where T is the time_step (we assume rets is
2275         sorted by time).
2276         Returns a 1-column DataFrame of Summary Stats indexed by the stat name.
2277         """
2278
2279     terminal_wealth = terminal_values(rets) # Cumulative returns: (1 + returns).prod()
2280     across time steps
2281
2282     breach = terminal_wealth < floor # Check if the terminal wealth breaches the floor
2283     reach = terminal_wealth >= cap # Check if the terminal wealth reaches or exceeds the
2284     cap
2285
2286     p_breach = breach.mean() if breach.sum() > 0 else np.nan # Probability of breach:
2287     percentage of scenarios below the floor
2288     p_reach = reach.mean() if reach.sum() > 0 else np.nan # Probability of reach:
2289     percentage of scenarios above or at the cap
2290
2291     e_short = (floor - terminal_wealth[breach]).mean() if breach.sum() > 0 else np.nan # Average shortfall in scenarios below the floor
2292     e_surplus = (terminal_wealth[reach] - cap).mean() if reach.sum() > 0 else np.nan # Average surplus in scenarios exceeding the cap
2293
2294     sum_stats = pd.DataFrame.from_dict({
2295         "mean": terminal_wealth.mean(),
2296         "std": terminal_wealth.std(),
2297         "p_breach": p_breach,
2298         "e_short": e_short,
2299         "p_reach": p_reach,
2300         "e_surplus": e_surplus
2301     }, orient="index", columns=[name])
2302
2303     return sum_stats
2304
2305
2306 #E22/R7. Compute the terminal stats for the returns of the bonds, for the equities, for
2307 # the 70/30 portfolio, and for the 60/40 portfolio. Import seaborn and use plt.figure() and
2308 sns.distplot(terminal values for equities,color="red",label="100% Equities"),
2309 distplot(terminal values for bonds,color="blue",label="100% Bonds"), etc. Analyze the
2310 results. Notice that you will obtain some NaNs, why? Interpret.
2311
2312 import seaborn as sns
2313 # Calculate terminal stats for each scenario
2314 terminal_bonds = terminal_values(rets_zc)
2315 terminal_eq = terminal_values(rets_eq)
2316 terminal_70_30 = terminal_values(rets_70_30)
2317 terminal_60_40 = terminal_values(rets_60_40)
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
3099
3100
3101
3102
3103
3104
3105
3106
3107
3108
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
3149
3150
3151
3152
3153
3154
3155
3156
3157
3158
3159
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
3179
3179
3180
3181
3182
3183
3184
3185
3186
3187
3187
3188
3189
3189
3190
3191
3192
3193
3194
3195
3196
3197
3197
3198
3199
3199
3200
3201
3202
3203
3204
3205
3206
3207
3208
3209
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
3249
3250
3251
3252
3253
3254
3255
3256
3257
3258
3259
3259
3260
3261
3262
3263
3264
3265
3266
3267
3268
3269
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3278
3279
3280
3281
3282
3283
3284
3285
3286
3287
3287
3288
3289
3289
3290
3291
3292
3293
3294
3295
3296
3297
3297
3298
3299
3299
3300
3301
3302
3303
3304
3305
3306
3307
3308
3309
3309
3310
3311
3312
3313
3314
3315
3316
3317
3318
3319
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
3349
3350
3351
3352
3353
3354
3355
3356
3357
3358
3359
3359
3360
3361
3362
3363
3364
3365
3366
3367
3368
3369
3369
3370
3371
3372
3373
3374
3375
3376
3377
3378
3378
3379
3380
3381
3382
3383
3384
3385
3386
3387
3387
3388
3389
3389
3390
3391
3392
3393
3394
3395
3396
3397
3397
3398
3399
3399
3400
3401
3402
3403
3404
3405
3406
3407
3408
3408
3409
3410
3411
3412
3413
3414
3415
3416
3417
3418
3419
3419
3420
3421
3422
3423
3424
3425
3426
3427
3428
3429
3429
3430
3431
3432
3433
3434
3435
3436
3437
3438
3438
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3448
3449
3450
3451
3452
3453
3454
3455
3456
3457
3458
3458
3459
3460
3461
3462
3463
3464
3465
3466
3467
3468
3468
3469
3470
3471
3472
3473
3474
3475
3476
3477
3478
3478
3479
3480
3481
3482
3483
3484
3485
3486
3487
3487
3488
3489
3489
3490
3491
3492
3493
3494
3495
3496
3497
3497
3498
3499
3499
3500
3501
3502
3503
3504
3505
3506
3507
3508
3508
3509
3510
3511
3512
3513
3514
3515
3516
3517
3518
3519
3519
3520
3521
3522
3523
3524
3525
3526
3527
3528
3528
3529
3530
3531
3532
3533
3534
3535
3536
3537
3538
3538
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3548
3549
3550
3551
3552
3553
3554
3555
3556
3557
3558
3558
3559
3560
3561
3562
3563
3564
3565
3566
3567
3568
3568
3569
3570
3571
3572
3573
3574
3575
3576
3577
3578
3578
3579
3580
3581
3582
3583
3584
3585
3586
3587
3587
3588
3589
3589
3590
3591
3592
3593
3594
3595
3596
3597
3597
3598
3599
3599
3600
3601
3602
3603
3604
3605
3606
3607
3608
3608
3609
3610
3611
3612
3613
3614
3615
3616
3617
3618
3619
3619
3620
3621
3622
3623
3624
3625
3626
3627
3628
3628
3629
3630
3631
3632
3633
3634
3635
3636
3637
3638
3638
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3648
3649
3650
3651
3652
3653
3654
3655
3656
3657
3658
3658
3659
3660
3661
3662
3663
3664
3665
3666
3667
3668
3668
3669
3670
3671
3672
3673
3674
3675
3676
3677
3678
3678
3679
3680
3681
3682
3683
3684
3685
3686
3687
3687
3688
3689
3689
3690
3691
3692
3693
3694
3695
3696
3697
3697
3698
3699
3699
3700
3701
3702
3703
3704
3705
3706
3707
3708
3708
3709
3710
3711
3712
3713
3714
3715
3716
3717
3718
3719
3719
3720
3721
3722
3723
3724
3725
3726
3727
3728
3728
3729
3730
3731
3732
3733
3734
3735
3736
3737
3738
3738
3739
3740
3741
3742
3743
3744
3745
3746
3747
3748
3748
3749
3750
3751
3752
3753
3754
3755
3756
3757
3758
3758
3759
3760
3761
3762
3763
3764
3765
3766
3767
3768
3769
3769
3770
3771
3772
3773
3774
3775
3776
3777
3778
3778
3779
3780
3781
3782
3783
3784
3785
3786
3787
3787
3788
3789
3789
3790
3791
3792
3793
3794
3795
3796
3797
3797
3798
3799
3799
3800
3801
3802
3803
3804
3805
3806
3807
3808
3808
3809
3810
3811
3812
3813
3814
3815
3816
3817
3818
3819
3819
3820
3821
3822
3823
3824
3825
3826
3827
3828
3828
3829
3830
3831
3832
3833
3834
3835
3836
3837
3838
3838
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3848
3849
3850
3851
3852
3853
3854
3855
3856
3857
3858
3858
3859
3860
3861
3862
3863
3864
3865
3866
3867
3868
3869
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3878
3879
3880
3881
3882
3883
3884
3885
3886
3887
3887
3888
3889
3889
3890
3891
3892
3893
3894
3895
3896
3897
3897
3898
3899
3899
3900
3901
3902
3903
3904
3905
3906
3907
3908
3908
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3928
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3938
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3948
3949
3950
3951
3952
3953
3954
3955
3956
3957
3958
3958
3959
3960
3961
3962
3963
3964
3965
3966
3967
3968
3969
3969
3970
3971
3972
3973
3974
3975
3976
3977
3978
3978
3979
3980
3981
3982
3983
3984
3985
3986
3987
3987
3988
3989
3989
3990
3991
3992
3993
3994
3995
3996
3997
3997
3998
3999
3999
4000
4001
4002
4003
4004
4005
4006
4007
4008
4008
4009
4010
4011
4012
4013
4014
4015
4016
4017
4018
4019
4019
4020
4021
4022
4023
4024
4025
4026
4027
4028
4029
4029
4030
4031
4032
4033
4034
4035
4036
4037
4038
4038
4039
4040
4041
4042
4043
4044
4045
4046
4047
4048
4048
4049
4050
4051
4052
4053
4054
4055
4056
4057
4058
4058
4059
4060
4061
4062
4063
4064
4065
4066
4067
4068
4069
4069
4070
4071
4072
4073
4074
4075
4076
4077
4078
4078
4079
4080
4081
4082
4083
4084
4085
4086
4087
4088
4088
4089
4090
4091
4092
4093
4094
4095
4096
4097
4097
4098
4099
4099
4100
4101
4102
4103
4104
4105
4106
4107
4108
4108
4109
4110
4111
4112
4113
4114
4115
4116
4117
4118
4119
4119
4120
4121
4122
4123
4124
4125
4126
4127
4128
4128
4129
4130
4131
4132
4133
4134
4135
4136
4137
4138
4138
4139
4140
4141
4142
4143
4144
4145
4146
4147
4148
4148
4149
4150
4151
4152
4153
4154
4155
4156
4157
4158
4158
4159
4160
4161
4162
4163
4164
4165
4166
4167
4168
4169
4169
4170
4171
4172
4173
4174
4175
4176
4177
4178
4178
4179
4180
4181
4182
4183
4184
4185
4186
4187
4188
4188
4189
4190
4191
4192
4193
4194
4195
4196
4197
4197
4198
4199
4199
4200
4201
4202
4203
4204
4205
4206
4207
4208
4208
4209
4210
4211
4212
4213
4214
4215
4216
4217
4218
4219
4219
4220
4221
4222
4223
4224
4225
4226
4227
4228
4229
4229
4230
4231
4232
4233
4234
4235
4236
4237
4238
4238
4239
4240
4241
4242
4243
4244
4245
4246
4247
4248
4248
4249
4250
4251
4252
4253
4254
4255
4256
4257
4258
4258
4259
4260
4261
4262
4263
4264
4265
4266
4267
4268
4269
4269
4270
4271
4272
4273
4274
4275
4276
4277
4278
4278
4279
4280
4281
4282
4283
4284
4285
4286
4287
4288
4288
4289
4290
4291
4292
4293
4294
4295
4296
4297
4298
4298
4299
4300
4301
4302
4303
4304
4305
4306
4307
4308
4308
4309
4310
4311
4312
4313
4314
4315
4316
4317
4318
4319
4319
4320
4321
4322
4323
4324
4325
4326
4327
4328
4329
4329
4330
4331
4332
4333
4334
4335
4336
4
```

```

2302 # Print terminal stats for each asset and portfolio
2303 print("Terminal Stats for Bonds:")
2304 print(terminal_stats(rets_zc))
2305
2306 print("\nTerminal Stats for Equities:")
2307 print(terminal_stats(rets_eq))
2308
2309 print("\nTerminal Stats for 70/30 Portfolio:")
2310 print(terminal_stats(rets_70_30))
2311
2312 print("\nTerminal Stats for 60/40 Portfolio:")
2313 print(terminal_stats(rets_60_40))
2314
2315 # Plot terminal values for each asset and portfolio
2316 plt.figure(figsize=(10, 6))
2317 sns.distplot(terminal_eq, color="red", label="100% Equities")
2318 sns.distplot(terminal_bonds, color="blue", label="100% Bonds")
2319 sns.distplot(terminal_70_30, color="green", label="70/30 Portfolio")
2320 sns.distplot(terminal_60_40, color="purple", label="60/40 Portfolio")
2321 plt.legend()
2322 plt.title("Terminal Values Distribution")
2323 plt.show()
2324
2325 # 8 Glide Paths for Allocation
2326
2327 def glidepath_allocator(r1,r2,start.glide=1,end.glide=0):
2328     """
2329         Simulates a Target-Date-Fund Style gradual move from r1 to r2.
2330     """
2331     n_points=r1.shape[0]
2332     n_col=r1.shape[1]
2333     path=pd.Series(data=np.linspace(start.glide,end.glide, num=n_points))
2334     paths= pd.concat([path]*n_col, axis=1) #we replicate our list// we concatenate n_col
2335     copies of path
2336     paths.index=r1.index
2337     paths.columns=r1.columns
2338     return paths
2339
2340 #E23/R8. Add to your previous analysis the terminal state of an allocation of 80/20 using
2341 # a Glide Allocator with start.glide=.8,end.glide=.2. Interpret the results. What is the
2342 # results for the probability of breach?
2343
2344 def glidepath_allocator(r1, r2, start.glide=1, end.glide=0):
2345     """
2346         Simulates a Target-Date-Fund Style gradual move from r1 to r2.
2347     """
2348     n_points = r1.shape[0]
2349     n_col = r1.shape[1]
2350     path = pd.Series(data=np.linspace(start.glide, end.glide, num=n_points))
2351     paths = pd.concat([path] * n_col, axis=1) # replicate the glide path for each
2352     scenario
2353     paths.index = r1.index
2354     paths.columns = r1.columns
2355     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
2401 # between PSP and GHP using a CPPI-style dynamic risk budgeting. To the previous
2402 # computation of terminal
2403 # states of ZC bonds, equities, and 70/30, add the terminal state for the floor
2404 # allocation using a floor of 75%. Analyze the results. What happens to the probability of
2405 # breach? Is this good or bad?

```

```

2402
2403 def floor_allocator(bsp_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 bsp/ghp representing the weights in the
2410         PSP.
2411         """
2412         if zc_prices.shape != bsp_r.shape:
2413             raise ValueError("PSP and ZC Prices must have the same shape")
2414
2415         n_steps, n_scenarios = bsp_r.shape
2416         account_value = np.repeat(1, n_scenarios) # Starting with $1
2417         floor_value = np.repeat(1, n_scenarios)
2418         w_history = pd.DataFrame(index=bsp_r.index, columns=bsp_r.columns)
2419
2420         for step in range(n_steps):
2421             floor_value = floor * zc_prices.iloc[step] # PV of the Floor assuming today's
2422             rates and flat YC
2423             cushion = account_value - floor_value # Cushion is the amount above the floor
2424             psp_w = (m * cushion / account_value).clip(0, 1) # Apply CPPI rule (multiplier *
2425             cushion)
2426             ghp_w = 1 - psp_w # The remaining weight goes to GHP
2427
2428             # Allocation to PSP and GHP
2429             psp_alloc = psp_w * account_value # Amount allocated to PSP
2430             ghp_alloc = ghp_w * account_value # Amount allocated to GHP
2431
2432             # Recompute the new account value at the end of this step using the new
2433             # allocations
2434             account_value = psp_alloc * (1 + bsp_r.iloc[step]) + ghp_alloc * (1 +
2435             ghp_r.iloc[step])
2436
2437             # Save the weight of PSP at this step
2438             w_history.iloc[step] = psp_w
2439
2440         return w_history
2441
2442
2443 # 1. Simulating the data for ZC prices and returns for PSP (equities) and GHP (bonds)
2444 rates, zc_prices = cir(n_years=10, n_scenarios=500, b=0.03, r_0=0.03)
2445 price_eq = gbm(n_years=10, n_scenarios=500, mu=0.07, sigma=0.15)
2446 rets_eq = price_eq.pct_change().dropna()
2447 rets_zc = zc_prices.pct_change().dropna()
2448
2449 # 2. Ensure that both the PSP (rets_eq) and ZC (zc_prices) have the same time index
2450 # Aligning the indices of zc_prices and rets_eq
2451 zc_prices = zc_prices.reindex(rets_eq.index)
2452
2453 # 3. Allocate 70/30 using the fixed mix allocator
2454 def fixedmix_allocator(r1, r2, w1, **kwargs):
2455     """
2456         Fixed mix allocator: w1 portion goes to r1 (PSP) and the rest to r2 (GHP).

```

```

2451     """
2452     return pd.DataFrame(data=w1, index=r1.index, columns=r1.columns)
2453
2454 # Perform the backtest with 70/30 allocation
2455 rets_7030b = bt_mix(rets_eq, rets_zc, allocator=fixedmix_allocator, w1=0.7)
2456
2457 # 4. Now use the floor allocator for 75% floor
2458 floor_value = 0.75 # The floor is 75%
2459
2460 # Calculate terminal states for ZC, Equities, and 70/30 using floor allocator
2461 zc_terminal = terminal_stats(rets_zc, name="ZC", floor=floor_value)
2462 eq_terminal = terminal_stats(rets_eq, name="Equities", floor=floor_value)
2463 alloc_7030_terminal = terminal_stats(rets_7030b, name="70/30", floor=floor_value)
2464
2465 # Calculate the floor allocation using the floor_allocator function
2466 floor_alloc = floor_allocator(rets_eq, rets_zc, floor=floor_value, zc_prices=zc_prices,
m=3)
2467 floor_alloc_terminal = terminal_stats(floor_alloc, name="Floor Allocation",
floor=floor_value)
2468
2469 # Display the terminal statistics for each allocation
2470 pd.concat([zc_terminal, eq_terminal, alloc_7030_terminal, floor_alloc_terminal],
axis=1).round(2)
2471
2472 #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.
2473
2474 # Drawdown-Based Allocator
2475 def drawdown_allocator(pps_r, ghp_r, maxdd, m=3):
2476     """Implements a CPPI-style allocation with a max drawdown constraint."""
2477     n_steps, n_scenarios = pps_r.shape
2478     account_value = np.ones(n_scenarios)
2479     peak_value = np.ones(n_scenarios)
2480     w_history = pd.DataFrame(index=pps_r.index, columns=pps_r.columns)
2481
2482     for step in range(n_steps):
2483         floor_value = (1 - maxdd) * peak_value
2484         cushion = (account_value - floor_value) / account_value
2485         pps_w = (m * cushion).clip(0, 1)
2486         ghp_w = 1 - pps_w
2487
2488         # Update account and peak values
2489         account_value *= (pps_w * (1 + pps_r.iloc[step]) + ghp_w * (1 +
ghp_r.iloc[step]))
2490         peak_value = np.maximum(peak_value, account_value)
2491
2492         # Store allocation history
2493         w_history.iloc[step] = pps_w
2494
2495     return w_history
2496
2497 # Define Cash Returns (since bonds can't be used for drawdown constraint)
2498 cash_rate = 0.02

```

```

2499 monthly_cash_return = (1 + cash_rate) ** (1 / 12) - 1
2500 rets_cash = pd.DataFrame(data=monthly_cash_return, index=rets_eq.index,
columns=rets_eq.columns)
2501
2502 # Compute the Allocation with Drawdown Constraint (Max DD 25%)
2503 w_maxdd25 = drawdown_allocator(rets_eq, rets_cash, maxdd=0.25)
2504 rets_maxdd25 = w_maxdd25 * rets_eq + (1 - w_maxdd25) * rets_cash
2505
2506 # Compute the Allocation with a Floor at 75%
2507 w_floor75 = floor_allocator(rets_eq, rets_cash, floor=0.75, zc_prices=zc_prices)
2508 rets_floor75 = w_floor75 * rets_eq + (1 - w_floor75) * rets_cash
2509
2510 # Compute Terminal Values
2511 tv_eq = terminal_values(rets_eq)
2512 tv_zc = terminal_values(rets_zc)
2513 tv_7030b = terminal_values(rets_7030b)
2514 tv_floor75 = terminal_values(rets_floor75)
2515 tv_maxdd25 = terminal_values(rets_maxdd25)
2516
2517 # Compute and Print Summary Stats
2518 print("Terminal Stats for Floor at 75%:")
2519 print(terminal_stats(rets_floor75, name="Floor 75%"))
2520 print("\nTerminal Stats for Max Drawdown 25%:")
2521 print(terminal_stats(rets_maxdd25, name="Max Drawdown 25%"))
2522
2523 #PLOT:
2524 plt.figure(figsize=(12,6))
2525 sns.distplot(tv_eq,color="red",label="100% Equities", bins=100)
2526 plt.axvline(tv_eq.mean(), ls="--", color="red")
2527 sns.distplot(tv_7030b, color="orange", label="70/30 Equities/Bonds", bins=100)
2528 plt.axvline(tv_7030b.mean(), ls="--", color="orange")
2529 sns.distplot(tv_floor75,color="green",label="Floor at 75%", bins=100)
2530 plt.axvline(tv_floor75.mean(), ls="--", color="green")
2531 sns.distplot(tv_maxdd25, color="yellow",label="MaxDD 25%", bins=100)
2532 plt.axvline(tv_maxdd25.mean(),ls="--", color="yellow")
2533 plt.legend();
2534 plt.show()
2535
2536
2537 #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 porpose 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.
2538
2539 # Function to compute drawdowns
2540 def drawdown(return_series: pd.Series):
2541     """Computes the wealth index, previous peaks, and percentage drawdowns."""
2542     wealth_index = 1000 * (1 + return_series).cumprod()
2543     previous_peaks = wealth_index.cummax()
2544     drawdowns = (wealth_index - previous_peaks) / previous_peaks

```

```
2545     return pd.DataFrame({"Wealth": wealth_index,
2546                           "Previous Peak": previous_peaks,
2547                           "Drawdown": drawdowns})
2548
2549 # Compute Drawdowns of Historical Market Returns
2550 rets_tmi = total_market_return["1990":] # Market returns from 1990 onwards
2551 dd_tmi = drawdown(rets_tmi)
2552
2553 # Ensure index is in Datetime format before plotting
2554 dd_tmi = dd_tmi.copy()
2555 dd_tmi.index = dd_tmi.index.to_timestamp()
2556
2557 # Now it should plot correctly
2558 ax = dd_tmi["Wealth"].plot(figsize=(12, 6), ls="-", color="goldenrod", label="Market Wealth")
2559 dd_tmi["Previous Peak"].plot(ax=ax, ls=":", color="red", label="Previous Peak")
2560
2561 plt.legend()
2562 plt.title("Market Wealth and Drawdowns (1990-Present)")
2563 plt.show()
2564
2565 # Apply Max Drawdown 25% LDI Allocation Strategy
2566 cash_rate = 0.03
2567 monthly_cash_return = (1 + cash_rate) ** (1 / 12) - 1
2568 rets_cash = pd.DataFrame(data=monthly_cash_return, index=rets_tmi.index, columns=[0])
2569
2570 # Compute returns for MaxDD 25% strategy using the historical market returns
2571 rets_maxdd25 = bt_mix(pd.DataFrame(rets_tmi), rets_cash, allocator=drawdown_allocator,
maxdd=0.25, m=5)
2572 dd_25 = drawdown(rets_maxdd25[0])
2573
2574 # Plot Drawdowns of Market vs MaxDD 25% Strategy
2575 plt.figure(figsize=(12, 6))
2576 plt.plot(dd_tmi["Drawdown"], label="Market Drawdowns", color="red")
2577 plt.plot(dd_25["Drawdown"], label="MaxDD 25% Drawdowns", color="blue")
2578 plt.title("Drawdowns: Market vs. MaxDD 25% Strategy")
2579 plt.legend()
2580 plt.show()
2581
```