

# Codes for the models

Slanzi Marco

October 2025

## 1 Note

In the coding part AI tools were implemented since in Python there are no libraries that allow the automatic computation of GARCH, Multivariate Garch and VAR models.

The codes were tested step by step, analyzed and double checked for statistical robustness.

The links I attached below are the main resources from which I began to build the models.

The Report was written manually, analysing the outputs and giving them an interpretation, statistically and economically. The Methodological Foundations and Statistical Analysis chapter describes all the steps of the computations, while Tables Plots shows the images and graphs generated through the Python matplotlib library.

Some of the tables are refined with AI to be inserted in the full report, speeding the process.

. <https://www.machinelearningplus.com/time-series/vector-autoregression-examples-python>

<https://medium.com/@ngaridennis3/developing-a-dynamic-conditional-correlation-dcc-garch-in-python-1b9d3ddd340f>

<https://www.diva-portal.org/smash/get/diva2:1800505/FULLTEXT01.pdf>

<https://www.machinelearningplus.com/time-series/vector-autoregression-examples-python/>

## 2 Econometrics of Financial Markets

### Code for the Univariate Analysis

```
1 import sys
2 import os
3 from pathlib import Path
4 import warnings
5 warnings.filterwarnings("ignore")
6
7 import numpy as np
8 import pandas as pd
9 import matplotlib.pyplot as plt
10 from scipy import stats, integrate
11 from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
12 from statsmodels.tsa.stattools import adfuller
13 from statsmodels.stats.diagnostic import het_arch, acorr_ljungbox
14
15 try:
```

```

16     from arch import arch_model
17 except Exception as e:
18     raise ImportError("arch package not installed. Run: pip install arch")
19     from e
20
21 from scipy.stats import norm, t, skewnorm, gennorm, laplace, genextreme
22
23 DEFAULT_FILE = r"C:\Users\marco\OneDrive\Desktop\Empirical exam
24             econometrics\Slanzi_Marco.xlsx"
25 date_col = "obs"
26 log_price_col = "lp"
27 is_log = True
28 freq_label = "D"
29 rolling_window = 21
30 ewma_span = 21
31 acf_lags = 50
32 forecast_horizon = 10
33 vaR_alpha = 0.1
34
35 def ensure_datetime_index(df):
36     global date_col
37     if date_col is None:
38         return df
39     if date_col in df.columns:
40         df[date_col] = pd.to_datetime(df[date_col])
41         df = df.sort_values(date_col).set_index(date_col)
42         return df
43     else:
44         if isinstance(df.index, pd.DatetimeIndex):
45             return df
46         return df
47
48 def compute_aic_bic(ll, k, n):
49     aic = 2*k - 2*ll
50     bic = np.log(n)*k - 2*ll
51     return aic, bic
52
53 def conditional_ES_std(dist_obj, params, alpha):
54     q = dist_obj.ppf(alpha, *params)
55     integrand = lambda z: z * dist_obj.pdf(z, *params)
56     num, _ = integrate.quad(integrand, -np.inf, q, limit=200)
57     return num / alpha
58
59
60
61
62
63
64
65
66
67
68
69
70

```

```

71
72 if is_log:
73     L = data[log_price_col].astype(float)
74 else:
75     price = data[log_price_col].astype(float)
76     L = np.log(price)
77
78 data['r'] = L.diff()
79 data = data.dropna().copy()
80
81 rets = data['r']
82 rets.name = "r"
83 logp = L.loc[rets.index]
84
85 print("\nReturn summary:")
86 print(rets.describe())
87
88 plt.style.use('default')
89
90 fig, ax = plt.subplots(3,1, figsize=(11,9), sharex=True)
91 ax[0].plot(rets.index, rets); ax[0].set_title("Log>Returns (r_t)")
92 ax[1].plot(rets.index, rets**2); ax[1].set_title("Squared Log>Returns (r_t^2)
93           volatility clustering")
94 ax[2].plot(logp.index, logp); ax[2].set_title("Log-Price (L_t)")
95 plt.tight_layout(); plt.show()
96
97 fig, ax = plt.subplots(1,2, figsize=(12,4))
98 ax[0].hist(rets, bins=80); ax[0].set_title("Histogram of returns")
99 stats.probplot(rets, dist="t", sparams=(5,), plot=ax[1]);
100    ax[1].set_title("QQ-plot vs student-t(df=5) reference")
101 plt.tight_layout(); plt.show()
102
103 adf_stat, adf_p, *_ = adfuller(rets)
104 print(f"\nADF test on returns: statistic={adf_stat:.6g}, p-value={adf_p:.6g}
105       (p<0.05 -> stationary)")
106
107 fig = plt.figure(figsize=(10,5)); plot_acf(rets, lags=acf_lags);
108     plt.title("ACF of returns"); plt.tight_layout(); plt.show()
109 fig = plt.figure(figsize=(10,5)); plot_acf(rets**2, lags=acf_lags);
110     plt.title("ACF of squared returns"); plt.tight_layout(); plt.show()
111 fig = plt.figure(figsize=(10,5)); plot_pacf(rets, lags=acf_lags,
112     method='ywm'); plt.title("PACF of returns"); plt.tight_layout();
113     plt.show()
114
115 arch_stat, arch_pvalue, _, _ = het_arch(rets, nlags=12)
116 print(f"\nEngle's ARCH LM test (12 lags): statistic={arch_stat:.6g},
117       p-value={arch_pvalue:.6g}")
118 if arch_pvalue < 0.05:
119     print(" -> Reject H0: ARCH effects detected.")
120 else:
121     print(" -> Fail to reject H0: no strong evidence of ARCH effects at 5%
122           level.")
123
124 rolling_vol = rets.rolling(window=rolling_window).std()
125 ewma_vol = rets.ewm(span=ewma_span).std()
126
127 fig, ax = plt.subplots(figsize=(10,4))

```

```

120 ax.plot(rolling_vol, label=f"Rolling {rolling_window}-obs std")
121 ax.plot(ewma_vol, label=f"EWMA span={ewma_span}")
122 ax.set_title("Nonparametric volatility estimates")
123 ax.legend(); plt.tight_layout(); plt.show()
124
125 rets_pct = (rets * 100.0).dropna()
126
127 models = {}
128
129 am_g11 = arch_model(rets_pct, vol='GARCH', p=1, q=1, mean='Constant',
130     dist='t')
131 res_g11 = am_g11.fit(disp='off')
132 models['GARCH11'] = res_g11
133 print("\nGARCH(1,1) fitted.")
134 print(res_g11.summary())
135
136 am_eg = arch_model(rets_pct, vol='EGARCH', p=1, o=1, q=1, mean='Constant',
137     dist='t')
138 res_eg = am_eg.fit(disp='off')
139 models['EGARCH'] = res_eg
140 print("\nEGARCH(1,1) fitted.")
141 print(res_eg.summary())
142
143 am_gjr = arch_model(rets_pct, vol='GARCH', p=1, o=1, q=1, mean='Constant',
144     dist='t')
145 res_gjr = am_gjr.fit(disp='off')
146 models['GJR'] = res_gjr
147 print("\nGJR-GARCH(1,1) fitted.")
148 print(res_gjr.summary())
149
150 am_tgjr = arch_model(rets_pct, vol='GARCH', p=1, o=1, q=1, mean='Constant',
151     dist='t')
152 res_tgjr = am_tgjr.fit(disp='off')
153 models['t-GJR'] = res_tgjr
154 print("\nt-GJR-GARCH(1,1) fitted.")
155 print(res_tgjr.summary())
156
157 am_aparch = arch_model(rets_pct, vol='APARCH', p=1, o=1, q=1,
158     mean='Constant', dist='t')
159 res_aparch = am_aparch.fit(disp='off')
160 models['APARCH'] = res_aparch
161 print("\nAPARCH(1,1) fitted.")
162 print(res_aparch.summary())
163
164 print("\nModel comparison (AIC, BIC):")
165 for name, r in models.items():
166     print(f"{name:8s} AIC={r.aic:.4f}, BIC={r.bic:.4f}")
167
168 best_name = min(models.keys(), key=lambda k: models[k].aic)
169 best_model = models[best_name]
170 print(f"\nSelected best model by AIC (among baseline family): {best_name}")
171 print("*" * 70)
172 print(f"RESIDUAL DIAGNOSTICS FOR BEST MODEL: {best_name}")
173 print("*" * 70)
174
175 std_resid = pd.Series(best_model.std_resid,
176     index=best_model.model.y.index).dropna()

```

```

172 fig, ax = plt.subplots(1, 3, figsize=(15, 4))
173 ax[0].plot(std_resid, lw=0.8)
174 ax[0].axhline(0, color='r', linestyle='--', alpha=0.6)
175 ax[0].set_title("Standardized Residuals")
176 ax[0].grid(alpha=0.3)
177
178 ax[1].hist(std_resid, bins=40, density=True, color='skyblue', edgecolor='k',
179             alpha=0.7)
180 x = np.linspace(std_resid.min(), std_resid.max(), 200)
181 ax[1].plot(x, stats.norm.pdf(x, 0, 1), 'r-', lw=2, label='N(0,1)')
182 ax[1].set_title("Histogram vs Normal PDF")
183 ax[1].legend()
184 ax[1].grid(alpha=0.3)
185
186 stats.probplot(std_resid, dist="norm", plot=ax[2])
187 ax[2].set_title("QQ-Plot vs Normal")
188
189 plt.tight_layout()
190 plt.show()
191
192 fig, ax = plt.subplots(1, 2, figsize=(12, 3))
193 plot_acf(std_resid, lags=40, ax=ax[0], alpha=0.05)
194 ax[0].set_title("ACF of Std Residuals")
195 plot_acf(std_resid**2, lags=40, ax=ax[1], alpha=0.05)
196 ax[1].set_title("ACF of Squared Std Residuals")
197 plt.tight_layout()
198 plt.show()
199
200 jb_stat, jb_p = stats.jarque_bera(std_resid)
201 lb5_p = acorr_ljungbox(std_resid, lags=[5],
202                         return_df=True)[‘lb_pvalue’].iloc[0]
203 lb10_p = acorr_ljungbox(std_resid, lags=[10],
204                         return_df=True)[‘lb_pvalue’].iloc[0]
205 lb5_sq_p = acorr_ljungbox(std_resid**2, lags=[5],
206                         return_df=True)[‘lb_pvalue’].iloc[0]
207 lb10_sq_p = acorr_ljungbox(std_resid**2, lags=[10],
208                         return_df=True)[‘lb_pvalue’].iloc[0]
209 arch_stat, arch_p, _, _ = het_arch(std_resid, nlags=12)
210
211 garch_p_models = {}
212 for p in [1,2,3]:
213     label = f"GARCH({p},1)"
214     print(f"\nFitting {label} ...")
215     am = arch_model(rets_pct, vol='GARCH', p=p, q=1, mean='Constant',
216                      dist='t')
217     res = am.fit(disp='off')
218     garch_p_models[label] = res
219     print(res.summary())
220
221 print("\nGARCH(p,1) comparison (AIC,BIC):")
222 for name, r in garch_p_models.items():
223     print(f"{name:10s} AIC={r.aic:.4f}, BIC={r.bic:.4f}")
224
225 def lr_test(res_restricted, res_full):
226     llr = 2.0 * (float(res_full.loglikelihood) -
227                  float(res_restricted.loglikelihood))
228     df = len(res_full.params) - len(res_restricted.params)

```

```

223     pval = stats.chi2.sf(llr, df)
224     return llr, df, pval
225
226 if 'GARCH(2,1)' in garch_p_models:
227     lr21 = lr_test(garch_p_models['GARCH(1,1)'], garch_p_models['GARCH(2,1)'])
228     print(f"\nLR test GARCH(2,1) vs GARCH(1,1): LR={lr21[0]:.4f},
229           df={lr21[1]}, p={lr21[2]:.6g}")
230 if 'GARCH(3,1)' in garch_p_models:
231     lr32 = lr_test(garch_p_models['GARCH(2,1)'], garch_p_models['GARCH(3,1)'])
232     print(f"LR test GARCH(3,1) vs GARCH(2,1): LR={lr32[0]:.4f}, df={lr32[1]},
233           p={lr32[2]:.6g}")
234
235 all_models = {**models, **garch_p_models}
236 best_name_all = min(all_models.keys(), key=lambda k: all_models[k].aic)
237 best_model_all = all_models[best_name_all]
238 print(f"\nSelected best model by AIC across all fitted models:
239       {best_name_all}")
240
241 res = best_model_all
242 std_resid = pd.Series(res.std_resid, index=res.model.y.index).dropna()
243
244 fig, ax = plt.subplots(1, 3, figsize=(15, 4))
245 ax[0].plot(std_resid, lw=0.8)
246 ax[0].axhline(0, color='r', linestyle='--', alpha=0.6)
247 ax[0].set_title("Standardized Residuals")
248 ax[1].hist(std_resid, bins=40, density=True, color='skyblue', edgecolor='k',
249             alpha=0.7)
250 x = np.linspace(std_resid.min(), std_resid.max(), 200)
251 ax[1].plot(x, stats.norm.pdf(x, 0, 1), 'r-', lw=2, label='N(0,1)')
252 ax[1].legend(); ax[2].grid(alpha=0.3)
253 stats.probplot(std_resid, dist="norm", plot=ax[2])
254 plt.tight_layout(); plt.show()
255
256 std_resid_best = pd.Series(best_model_all.std_resid,
257                             index=rets.index[-len(best_model_all.std_resid):]).dropna()
258
259 params = best_model_all.params
260 omega = float(params.get('omega', 0))
261 alpha1 = float(params.get('alpha[1]', 0))
262 beta1 = float(params.get('beta[1]', 0))
263 gamma1 = float(params.get('gamma[1]', 0))
264 mu_hat = float(params.get('mu', 0)) / 100.0
265
266 sigma_t = float(best_model_all.conditional_volatility.iloc[-1])
267
268 H = 20
269
270 irf_pos = np.zeros(H)
271 eps_t = sigma_t
272 irf_pos[0] = np.sqrt(omega + alpha1*eps_t**2 + gamma1*eps_t**2*0 +
273                      beta1*sigma_t**2)
274 for t in range(1, H):
275     irf_pos[t] = np.sqrt(omega + beta1*irf_pos[t-1]**2)
276
277 irf_neg = np.zeros(H)
278 eps_t = -sigma_t
279 indicator = 1.0

```

```

274     irf_neg[0] = np.sqrt(omega + alpha1*eps_t**2 + gamma1*eps_t**2*indicator +
275                           beta1*sigma_t**2)
276     for t in range(1, H):
277         irf_neg[t] = np.sqrt(omega + beta1*irf_neg[t-1]**2)
278
279     plt.figure(figsize=(10,5))
280     plt.plot(range(1,H+1), irf_pos, label='Positive Shock', marker='o')
281     plt.plot(range(1,H+1), irf_neg, label='Negative Shock', marker='x')
282     plt.title('Impulse Response Function of Conditional Volatility')
283     plt.xlabel('Periods ahead')
284     plt.ylabel('Conditional sigma')
285     plt.legend()
286     plt.grid(True)
287     plt.show()
288
289     print("\nFitting candidate distributions to standardized residuals...")
290     x = std_resid_best.values
291     n = len(x)
292     candidate_dists = {
293         'normal': norm,
294         't': t,
295         'skewnorm': skewnorm,
296         'gennorm': gennorm,
297         'laplace': laplace,
298         'genextreme': genextreme
299     }
300
301     dist_results = []
302     for name, dist in candidate_dists.items():
303         try:
304             params = dist.fit(x)
305             try:
306                 ll = np.sum(dist.logpdf(x, *params))
307             except Exception:
308                 ll = np.sum(np.log(dist.pdf(x, *params)))
309             k = len(params)
310             aic, bic = compute_aic_bic(ll, k, n)
311             cdf_fun = lambda v: dist.cdf(v, *params)
312             ks_stat, ks_p = stats.kstest(x, cdf_fun)
313             dist_results.append({
314                 'dist': name, 'params': params, 'loglik': ll, 'k': k, 'AIC': aic,
315                 'BIC': bic,
316                 'KS_stat': ks_stat, 'KS_p': ks_p
317             })
318             print(f"Fitted {name}: AIC={aic:.2f}, BIC={bic:.2f}, KS_p={ks_p:.4g}")
319         except Exception as e:
320             print(f"Failed to fit {name}: {e}")
321
322     dist_df = pd.DataFrame(dist_results).sort_values('AIC').reset_index(drop=True)
323     print("\nTop candidate distributions by AIC:")
324     print(dist_df[['dist', 'AIC', 'BIC', 'KS_p']].head())
325
326     topk = min(3, len(dist_df))
327     fig, ax = plt.subplots(figsize=(10,5))
328     ax.hist(x, bins=80, density=True, alpha=0.4, label='std_resid hist')
329     xs = np.linspace(x.min()*1.05, x.max()*1.05, 1000)
     for i in range(topk):
         row = dist_df.loc[i]

```

```

330     dd = candidate_dists[row['dist']]
331     params = row['params']
332     ax.plot(xs, dd.pdf(xs, *params), label=f'{row["dist"]}\n    (AIC={row["AIC"]:.1f})')
333 ax.legend(); ax.set_title("Standardized residuals: histogram + fitted PDFs")
334 plt.tight_layout(); plt.show()
335
336 h = forecast_horizon
337 fcast = best_model_all.forecast(horizon=h, reindex=False)
338 last_var = fcast.variance.iloc[-1]
339 pred_sigma = np.sqrt(last_var) / 100.0
340
341 print(f"\n{best_model_all.model.name} - {h}-step ahead conditional std\n    (returns units):")
342 for i in range(len(pred_sigma)):
343     print(f" h={i+1}: sigma = {pred_sigma.iloc[i]:.6f}")
344
345 cond_sigma_series = pd.Series(best_model_all.conditional_volatility,
346                                index=rets.index) / 100.0
347 fig, ax = plt.subplots(figsize=(10,4))
348 ax.plot(cond_sigma_series, label=f"Conditional volatility ({best_name_all})")
349 ax.set_title("Estimated conditional volatility (sigma_t)")
350 ax.legend(); plt.tight_layout(); plt.show()
351
352 if len(dist_df) > 0:
353     chosen = dist_df.loc[0]
354     chosen_name = chosen['dist']
355     chosen_dist = candidate_dists[chosen_name]
356     chosen_params = chosen['params']
357 else:
358     chosen_name = 'normal'
359     chosen_dist = norm
360     chosen_params = norm.fit(x)
361
362 print(f"\nUsing residual distribution for parametric VaR/ES: {chosen_name}")
363
364 q_std = chosen_dist.ppf(vaR_alpha, *chosen_params)
365 ES_std = conditional_ES_std(chosen_dist, chosen_params, vaR_alpha)
366 print(f"Std quantile (alpha={vaR_alpha}): {q_std:.6g}, Std conditional ES:\n    {ES_std:.6g}")
367
368 mu_hat = float(best_model_all.params.get('mu', 0.0)) / 100.0
369 VaR_param = mu_hat + cond_sigma_series * q_std
370 ES_param = mu_hat + cond_sigma_series * ES_std
371
372 q_emp = np.quantile(x, vaR_alpha)
373 VaR_emp = mu_hat + cond_sigma_series * q_emp
374
375 viol = (rets < VaR_emp)
376 if viol.sum() > 0:
377     ES_empirical_global = rets[viol].mean()
378 else:
379     ES_empirical_global = np.nan
380
381 print(f"\n1-day parametric VaR (last): {VaR_param.iloc[-1]:.6f}")
382 print(f"1-day parametric ES (last): {ES_param.iloc[-1]:.6f}")
383 print(f"1-day empirical VaR (last): {VaR_emp.iloc[-1]:.6f}")

```

```

383 print(f"Empirical global ES (average realized losses below empirical VaR):"
384     f'{ES_empirical_global:.6f}"')
385
386 window = min(300, len(rets))
387 fig, ax = plt.subplots(figsize=(12,5))
388 ax.plot(rets[-window:], label='Returns')
389 ax.plot(VaR_param[-window:], label='Parametric VaR')
390 ax.plot(ES_param[-window:], label='Parametric ES')
391 ax.set_title("Returns vs Parametric VaR & ES (recent)")
392 ax.legend(); plt.tight_layout(); plt.show()
393
394
395 fig, axes = plt.subplots(1, 2, figsize=(12, 5))
396
397 window = min(300, len(rets))
398 axes[0].plot(rets[-window:].values, label='Returns', color='black',
399                 alpha=0.7, linewidth=1)
400 axes[0].plot(VaR_param[-window:].values, label=f'Parametric VaR
401 ({VaR_param.iloc[-1]:.4f})',
402                 color='red', linestyle='--', linewidth=1.5)
403 axes[0].plot(VaR_emp[-window:].values, label=f'Empirical VaR
404 ({VaR_emp.iloc[-1]:.4f})',
405                 color='blue', linestyle='--', linewidth=1.5)
406
407 violations_param = rets[-window:] < VaR_param[-window:]
408 violations_emp = rets[-window:] < VaR_emp[-window:]
409
410 for i in range(window):
411     if violations_param.iloc[i]:
412         axes[0].axvspan(i-0.5, i+0.5, alpha=0.3, color='red')
413     if violations_emp.iloc[i]:
414         axes[0].axvspan(i-0.5, i+0.5, alpha=0.2, color='blue')
415
416 axes[0].set_xlabel('Time')
417 axes[0].set_ylabel('Returns')
418 axes[0].set_title('RETURNS vs VaR METHODS (Recent Period)')
419 axes[0].legend(fontsize=9)
420 axes[0].grid(True, alpha=0.3)
421
422 x_range = np.linspace(x.min(), x.max(), 1000)
423 parametric_pdf = chosen_dist.pdf(x_range, *chosen_params) *
424     cond_sigma_series.iloc[-1]
425 axes[1].hist(x, bins=100, density=True, alpha=0.6, color='lightgray',
426                 label='Std Residuals')
427 axes[1].plot(x_range, parametric_pdf, 'r-', linewidth=2, label=f'Parametric
428 ({chosen_name})')
429
430 axes[1].axvline(VaR_param.iloc[-1], color='red', linestyle='--', linewidth=2,
431                 label=f'Parametric VaR')
432 axes[1].axvline(VaR_emp.iloc[-1], color='blue', linestyle='--', linewidth=2,
433                 label=f'Empirical VaR')
434
435 axes[1].set_xlabel('Returns')
436 axes[1].set_ylabel('Density')
437 axes[1].set_title('DISTRIBUTION COMPARISON')
438 axes[1].legend(fontsize=9)
439 axes[1].grid(True, alpha=0.3)

```

```

432 plt.tight_layout()
433 plt.show()
434
435
436
437
438
439 \section{Code for the Multivariate Analysis}
440
441
442
443
444
445
446 \section{Code for the VAR model}
447 \end{document} make it suitable for latex
ChatGPT said:
Thought for 15s
You said:
451 \documentclass{article}
452 \usepackage{graphicx} % Required for inserting images
453
454 \title{Codes for the models}
455 \subsection{Econometrics of Financial Markets}
456 \author{Slanzi Marco}
457 \date{October 2025}
458
459 \begin{document}
460
461 \maketitle
462
463 \section{Code for the Univariate Analysis}
464 import sys
465 import os
466 from pathlib import Path
467 import warnings
468 warnings.filterwarnings("ignore")
469
470 import numpy as np
471 import pandas as pd
472 import matplotlib.pyplot as plt
473 from scipy import stats, integrate
474 from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
475 from statsmodels.tsa.stattools import adfuller
476 from statsmodels.stats.diagnostic import het_arch, acorr_ljungbox
477
478 try:
479     from arch import arch_model
480 except Exception as e:
481     raise ImportError("arch package not installed. Run: pip install arch")
482     from e
483
484 from scipy.stats import norm, t, skewnorm, gennorm, laplace, genextreme
485
486 DEFAULT_FILE = r"C:\Users\marco\OneDrive\Desktop\Empirical exam
econometrics\Slanzi_Marco.xlsx"
487 date_col = "obs"
log_price_col = "lp"

```

```

488 is_log = True
489 freq_label = "D"
490 rolling_window = 21
491 ewma_span = 21
492 acf_lags = 50
493 forecast_horizon = 10
494 var_alpha = 0.1
495
496 def ensure_datetime_index(df):
497     global date_col
498     if date_col is None:
499         return df
500     if date_col in df.columns:
501         df[date_col] = pd.to_datetime(df[date_col])
502         df = df.sort_values(date_col).set_index(date_col)
503         return df
504     else:
505         if isinstance(df.index, pd.DatetimeIndex):
506             return df
507     return df
508
509 def compute_aic_bic(ll, k, n):
510     aic = 2*k - 2*ll
511     bic = np.log(n)*k - 2*ll
512     return aic, bic
513
514 def conditional_ES_std(dist_obj, params, alpha):
515     q = dist_obj.ppf(alpha, *params)
516     integrand = lambda z: z * dist_obj.pdf(z, *params)
517     num, _ = integrate.quad(integrand, -np.inf, q, limit=200)
518     return num / alpha
519
520 print("Setup done. Ready to run blocks.")
521
522 infile = DEFAULT_FILE
523 if not os.path.exists(infile):
524     raise FileNotFoundError(f"Excel file not found: {infile}.")
525
526 print(f"Loading file: {infile} (sheet Foglio1)")
527 df = pd.read_excel(infile, sheet_name="Foglio1", engine="openpyxl")
528 print("Columns:", df.columns.tolist())
529
530 df = ensure_datetime_index(df)
531
532 if log_price_col not in df.columns:
533     raise ValueError(f"Expected column '{log_price_col}' not found in sheet
534     Foglio1")
535
536 data = df.copy()
537
538 if is_log:
539     L = data[log_price_col].astype(float)
540 else:
541     price = data[log_price_col].astype(float)
542     L = np.log(price)
543
544 data['r'] = L.diff()
545 data = data.dropna().copy()

```

```

545
546     rets = data['r']
547     rets.name = "r"
548     logp = L.loc[rets.index]
549
550     print(f"\nObservations after diff: {len(data)}")
551     print("\nReturn summary:")
552     print(rets.describe())
553
554
555     plt.style.use('default')
556
557     fig, ax = plt.subplots(3,1, figsize=(11,9), sharex=True)
558     ax[0].plot(rets.index, rets); ax[0].set_title("Log-Returns (r_t)")
559     ax[1].plot(rets.index, rets**2); ax[1].set_title("Squared Log-Returns (r_t^2)  
volatility clustering")
560     ax[2].plot(logp.index, logp); ax[2].set_title("Log-Price (L_t)")
561     plt.tight_layout(); plt.show()
562
563
564     fig, ax = plt.subplots(1,2, figsize=(12,4))
565     ax[0].hist(rets, bins=80); ax[0].set_title("Histogram of returns")
566     stats.probplot(rets, dist="t", sparams=(5,), plot=ax[1]);
567         ax[1].set_title("QQ-plot vs student-t(df=5) reference")
568     plt.tight_layout(); plt.show()
569
570     adf_stat, adf_p, *_ = adfuller(rets)
571     print(f"\nADF test on returns: statistic={adf_stat:.6g}, p-value={adf_p:.6g}  
(p<0.05 -> stationary)")
572
573
574     fig = plt.figure(figsize=(10,5)); plot_acf(rets, lags=acf_lags);
575         plt.title("ACF of returns"); plt.tight_layout(); plt.show()
576     fig = plt.figure(figsize=(10,5)); plot_acf(rets**2, lags=acf_lags);
577         plt.title("ACF of squared returns"); plt.tight_layout(); plt.show()
578     fig = plt.figure(figsize=(10,5)); plot_pacf(rets, lags=acf_lags,
579         method='ywm'); plt.title("PACF of returns"); plt.tight_layout();
580         plt.show()
581
582     arch_stat, arch_pvalue, _, _ = het_arch(rets, nlags=12)
583     print(f"\nEngle's ARCH LM test (12 lags): statistic={arch_stat:.6g},  
p-value={arch_pvalue:.6g}")
584     if arch_pvalue < 0.05:
585         print(" -> Reject H0: ARCH effects detected.")
586     else:
587         print(" -> Fail to reject H0: no strong evidence of ARCH effects at 5%  
level.")
588
589     rolling_vol = rets.rolling(window=rolling_window).std()
590     ewma_vol = rets.ewm(span=ewma_span).std()
591
592     fig, ax = plt.subplots(figsize=(10,4))
593     ax.plot(rolling_vol, label=f"Rolling {rolling_window}-obs std")
594     ax.plot(ewma_vol, label=f"EWMA span={ewma_span}")
595     ax.set_title("Nonparametric volatility estimates")
596     ax.legend(); plt.tight_layout(); plt.show()
597
598     rrets_pct = (rets * 100.0).dropna()
599
600     models = {}

```

```

594 am_g11 = arch_model(rets_pct, vol='GARCH', p=1, q=1, mean='Constant',
595     dist='t')
596 res_g11 = am_g11.fit(disp='off')
597 models['GARCH11'] = res_g11
598 print("\nGARCH(1,1) fitted.")
599 print(res_g11.summary())
600
601 am_eg = arch_model(rets_pct, vol='EGARCH', p=1, o=1, q=1, mean='Constant',
602     dist='t')
603 res_eg = am_eg.fit(disp='off')
604 models['EGARCH'] = res_eg
605 print("\nEGARCH(1,1) fitted.")
606 print(res_eg.summary())
607
608 am_gjr = arch_model(rets_pct, vol='GARCH', p=1, o=1, q=1, mean='Constant',
609     dist='t')
610 res_gjr = am_gjr.fit(disp='off')
611 models['GJR'] = res_gjr
612 print("\nGJR-GARCH(1,1) fitted.")
613 print(res_gjr.summary())
614
615 am_tgjr = arch_model(rets_pct, vol='GARCH', p=1, o=1, q=1, mean='Constant',
616     dist='t')
617 res_tgjr = am_tgjr.fit(disp='off')
618 models['t-GJR'] = res_tgjr
619 print("\nt-GJR-GARCH(1,1) fitted.")
620 print(res_tgjr.summary())
621
622 am_aparch = arch_model(rets_pct, vol='APARCH', p=1, o=1, q=1,
623     mean='Constant', dist='t')
624 res_aparch = am_aparch.fit(disp='off')
625 models['APARCH'] = res_aparch
626 print("\nAPARCH(1,1) fitted.")
627 print(res_aparch.summary())
628
629 print("\nModel comparison (AIC, BIC):")
630 for name, r in models.items():
631     print(f"{name:8s} AIC={r.aic:.4f}, BIC={r.bic:.4f}")
632
633 best_name = min(models.keys(), key=lambda k: models[k].aic)
634 best_model = models[best_name]
635 print(f"\nSelected best model by AIC (among baseline family): {best_name}")
636
637 std_resid = pd.Series(best_model.std_resid,
638     index=best_model.model.y.index).dropna()
639
640 fig, ax = plt.subplots(1, 3, figsize=(15, 4))
641 ax[0].plot(std_resid, lw=0.8)
642 ax[0].axhline(0, color='r', linestyle='--', alpha=0.6)
643 ax[0].set_title("Standardized Residuals")
644 ax[0].grid(alpha=0.3)

```

```

645 ax[1].hist(std_resid, bins=40, density=True, color='skyblue', edgecolor='k',
646     alpha=0.7)
647 x = np.linspace(std_resid.min(), std_resid.max(), 200)
648 ax[1].plot(x, stats.norm.pdf(x, 0, 1), 'r-', lw=2, label='N(0,1)')
649 ax[1].set_title("Histogram vs Normal PDF")
650 ax[1].legend()
651 ax[1].grid(alpha=0.3)
652
653 stats.probplot(std_resid, dist="norm", plot=ax[2])
654 ax[2].set_title("QQ-Plot vs Normal")
655
656 plt.tight_layout()
657 plt.show()
658
659 fig, ax = plt.subplots(1, 2, figsize=(12, 3))
660 plot_acf(std_resid, lags=40, ax=ax[0], alpha=0.05)
661 ax[0].set_title("ACF of Std Residuals")
662 plot_acf(std_resid**2, lags=40, ax=ax[1], alpha=0.05)
663 ax[1].set_title("ACF of Squared Std Residuals")
664 plt.tight_layout()
665 plt.show()
666
667 jb_stat, jb_p = stats.jarque_bera(std_resid)
668 lb5_p = acorr_ljungbox(std_resid, lags=[5],
669     return_df=True)[‘lb_pvalue’].iloc[0]
670 lb10_p = acorr_ljungbox(std_resid, lags=[10],
671     return_df=True)[‘lb_pvalue’].iloc[0]
672 lb5_sq_p = acorr_ljungbox(std_resid**2, lags=[5],
673     return_df=True)[‘lb_pvalue’].iloc[0]
674 lb10_sq_p = acorr_ljungbox(std_resid**2, lags=[10],
675     return_df=True)[‘lb_pvalue’].iloc[0]
676 arch_stat, arch_p, _, _ = het_arch(std_resid, nlags=12)
677 garch_p_models = {}
678 for p in [1,2,3]:
679     label = f"GARCH({p},1)"
680     print(f"\nFitting {label} ...")
681     am = arch_model(rets_pct, vol='GARCH', p=p, q=1, mean='Constant',
682         dist='t')
683     res = am.fit(disp='off')
684     garch_p_models[label] = res
685     print(res.summary())
686
687 print("\nGARCH(p,1) comparison (AIC,BIC):")
688 for name, r in garch_p_models.items():
689     print(f"{name:10s} AIC={r.aic:.4f}, BIC={r.bic:.4f}")
690
691 def lr_test(res_restricted, res_full):
692     llr = 2.0 * (float(res_full.loglikelihood) -
693         float(res_restricted.loglikelihood))
694     df = len(res_full.params) - len(res_restricted.params)
695     pval = stats.chi2.sf(llr, df)
696     return llr, df, pval
697
698 if 'GARCH(2,1)' in garch_p_models:
699     lr21 = lr_test(garch_p_models['GARCH(1,1)'], garch_p_models['GARCH(2,1)'])
700     print(f"\nLR test GARCH(2,1) vs GARCH(1,1): LR={lr21[0]:.4f},
701           df={lr21[1]}, p={lr21[2]:.6g}")
702 if 'GARCH(3,1)' in garch_p_models:

```

```

695     lr32 = lr_test(garch_p_models['GARCH(2,1)'], garch_p_models['GARCH(3,1)'])
696     print(f"LR test GARCH(3,1) vs GARCH(2,1): LR={lr32[0]:.4f}, df={lr32[1]},"
697           p={lr32[2]:.6g}")
698
699     all_models = {**models, **garch_p_models}
700     best_name_all = min(all_models.keys(), key=lambda k: all_models[k].aic)
701     best_model_all = all_models[best_name_all]
702     print(f"\nSelected best model by AIC across all fitted models:"
703           f"\n{best_name_all}")
704
705     res = best_model_all
706     std_resid = pd.Series(res.std_resid, index=res.model.y.index).dropna()
707
708     fig, ax = plt.subplots(1, 3, figsize=(15, 4))
709     ax[0].plot(std_resid, lw=0.8)
710     ax[0].axhline(0, color='r', linestyle='--', alpha=0.6)
711     ax[0].set_title("Standardized Residuals")
712     ax[1].hist(std_resid, bins=40, density=True, color='skyblue', edgecolor='k',
713                alpha=0.7)
714     x = np.linspace(std_resid.min(), std_resid.max(), 200)
715     ax[1].plot(x, stats.norm.pdf(x, 0, 1), 'r-', lw=2, label='N(0,1)')
716     ax[1].legend(); ax[2].grid(alpha=0.3)
717     stats.probplot(std_resid, dist="norm", plot=ax[2])
718     plt.tight_layout(); plt.show()
719
720     std_resid_best = pd.Series(best_model_all.std_resid,
721                               index=rets.index[-len(best_model_all.std_resid):]).dropna()
722
723     params = best_model_all.params
724     omega = float(params.get('omega', 0))
725     alpha1 = float(params.get('alpha[1]', 0))
726     beta1 = float(params.get('beta[1]', 0))
727     gamma1 = float(params.get('gamma[1]', 0))
728     mu_hat = float(params.get('mu', 0)) / 100.0
729
730     sigma_t = float(best_model_all.conditional_volatility.iloc[-1])
731
732     H = 20
733
734     irf_pos = np.zeros(H)
735     eps_t = sigma_t
736     irf_pos[0] = np.sqrt(omega + alpha1*eps_t**2 + gamma1*eps_t**2*0 +
737                          beta1*sigma_t**2)
738     for t in range(1, H):
739         irf_pos[t] = np.sqrt(omega + beta1*irf_pos[t-1]**2)
740
741     irf_neg = np.zeros(H)
742     eps_t = -sigma_t
743     indicator = 1.0
744     irf_neg[0] = np.sqrt(omega + alpha1*eps_t**2 + gamma1*eps_t**2*indicator +
745                          beta1*sigma_t**2)
746     for t in range(1, H):
747         irf_neg[t] = np.sqrt(omega + beta1*irf_neg[t-1]**2)
748
749     plt.figure(figsize=(10,5))
750     plt.plot(range(1,H+1), irf_pos, label='Positive Shock', marker='o')
751     plt.plot(range(1,H+1), irf_neg, label='Negative Shock', marker='x')
752     plt.title('Impulse Response Function of Conditional Volatility')

```

```

747 plt.xlabel('Periods ahead')
748 plt.ylabel('Conditional sigma')
749 plt.legend()
750 plt.grid(True)
751 plt.show()

752 print("\nFitting candidate distributions to standardized residuals...")
753 x = std_resid_best.values
754 n = len(x)
755 candidate_dists = {
756     'normal': norm,
757     't': t,
758     'skewnorm': skewnorm,
759     'gennorm': gennorm,
760     'laplace': laplace,
761     'genextreme': genextreme
762 }
763
764 dist_results = []
765 for name, dist in candidate_dists.items():
766     try:
767         params = dist.fit(x)
768         try:
769             ll = np.sum(dist.logpdf(x, *params))
770         except Exception:
771             ll = np.sum(np.log(dist.pdf(x, *params)))
772         k = len(params)
773         aic, bic = compute_aic_bic(ll, k, n)
774         cdf_fun = lambda v: dist.cdf(v, *params)
775         ks_stat, ks_p = stats.kstest(x, cdf_fun)
776         dist_results.append({
777             'dist': name, 'params': params, 'loglik': ll, 'k': k, 'AIC': aic,
778             'BIC': bic,
779             'KS_stat': ks_stat, 'KS_p': ks_p
780         })
781         print(f"Fitted {name}: AIC={aic:.2f}, BIC={bic:.2f}, KS_p={ks_p:.4g}")
782     except Exception as e:
783         print(f"Failed to fit {name}: {e}")
784
785 dist_df = pd.DataFrame(dist_results).sort_values('AIC').reset_index(drop=True)

786
787
788 topk = min(3, len(dist_df))
789 fig, ax = plt.subplots(figsize=(10,5))
790 ax.hist(x, bins=80, density=True, alpha=0.4, label='std_resid hist')
791 xs = np.linspace(x.min()*1.05, x.max()*1.05, 1000)
792 for i in range(topk):
793     row = dist_df.loc[i]
794     dd = candidate_dists[row['dist']]
795     params = row['params']
796     ax.plot(xs, dd.pdf(xs, *params), label=f"{row['dist']}\n(AIC={row['AIC']:.1f})")
797 ax.legend(); ax.set_title("Standardized residuals: histogram + fitted PDFs")
798 plt.tight_layout(); plt.show()

799
800 h = forecast_horizon
801 fcast = best_model_all.forecast(horizon=h, reindex=False)
802 last_var = fcast.variance.iloc[-1]

```

```

803 pred_sigma = np.sqrt(last_var) / 100.0
804
805 print(f"\n{best_model_all.model.name} - {h}-step ahead conditional std
806     (returns units):")
807 for i in range(len(pred_sigma)):
808     print(f" h={i+1}: sigma = {pred_sigma.iloc[i]:.6f}")
809
810 cond_sigma_series = pd.Series(best_model_all.conditional_volatility,
811     index=rets.index) / 100.0
812 fig, ax = plt.subplots(figsize=(10,4))
813 ax.plot(cond_sigma_series, label=f"Conditional volatility ({best_name_all})")
814 ax.set_title("Estimated conditional volatility (sigma_t)")
815 ax.legend(); plt.tight_layout(); plt.show()
816
817 if len(dist_df) > 0:
818     chosen = dist_df.loc[0]
819     chosen_name = chosen['dist']
820     chosen_dist = candidate_dists[chosen_name]
821     chosen_params = chosen['params']
822 else:
823     chosen_name = 'normal'
824     chosen_dist = norm
825     chosen_params = norm.fit(x)
826
827 print(f"\nUsing residual distribution for parametric VaR/ES: {chosen_name}")
828
829 q_std = chosen_dist.ppf(vaR_alpha, *chosen_params)
830 ES_std = conditional_ES_std(chosen_dist, chosen_params, vaR_alpha)
831 print(f"Std quantile (alpha={vaR_alpha}): {q_std:.6g}, Std conditional ES:
832     {ES_std:.6g}")
833
834 mu_hat = float(best_model_all.params.get('mu', 0.0)) / 100.0
835 VaR_param = mu_hat + cond_sigma_series * q_std
836 ES_param = mu_hat + cond_sigma_series * ES_std
837
838 q_emp = np.quantile(x, vaR_alpha)
839 VaR_emp = mu_hat + cond_sigma_series * q_emp
840
841 viol = (rets < VaR_emp)
842 if viol.sum() > 0:
843     ES_empirical_global = rets[viol].mean()
844 else:
845     ES_empirical_global = np.nan
846
847 window = min(300, len(rets))
848 fig, ax = plt.subplots(figsize=(12,5))
849 ax.plot(rets[-window:], label='Returns')
850 ax.plot(VaR_param[-window:], label='Parametric VaR')
851 ax.plot(ES_param[-window:], label='Parametric ES')
852 ax.set_title("Returns vs Parametric VaR & ES (recent)")
853 ax.legend(); plt.tight_layout(); plt.show()
854
855 print("=" * 80)
856 print("FINAL VaR & ES COMPARISON: PARAMETRIC vs EMPIRICAL")
857 print("=" * 80)

```

```

857 print(f"{'Method':<20} {'VaR ('+str(vaR_alpha*100)+')':<20} {'ES
858     ('+str(vaR_alpha*100)+')':<20}")
859 print("-" * 70)
860 print(f"{'Parametric':<20} {VaR_param.iloc[-1]:.6f} {ES_param.iloc[-1]:.6f}")
861 print(f"{'Empirical':<20} {VaR_emp.iloc[-1]:.6f} {ES_empirical_global:.6f}")
862
863 print("\nReality Check:")
864 print(f"Actual returns 10th percentile: {np.percentile(rets, 10):.6f}")
865 print(f"Worst historical return: {rets.min():.6f}")
866
867 fig, axes = plt.subplots(1, 2, figsize=(12, 5))
868
869 window = min(300, len(rets))
870 axes[0].plot(rets[-window:].values, label='Returns', color='black',
871                 alpha=0.7, linewidth=1)
872 axes[0].plot(VaR_param[-window:].values, label=f'Parametric VaR
873     ({VaR_param.iloc[-1]:.4f})',
874                 color='red', linestyle='--', linewidth=1.5)
875 axes[0].plot(VaR_emp[-window:].values, label=f'Empirical VaR
876     ({VaR_emp.iloc[-1]:.4f})',
877                 color='blue', linestyle='--', linewidth=1.5)
878
879 violations_param = rets[-window:] < VaR_param[-window:]
880 violations_emp = rets[-window:] < VaR_emp[-window:]
881
882 for i in range(window):
883     if violations_param.iloc[i]:
884         axes[0].axvspan(i-0.5, i+0.5, alpha=0.3, color='red')
885     if violations_emp.iloc[i]:
886         axes[0].axvspan(i-0.5, i+0.5, alpha=0.2, color='blue')
887
888 axes[0].set_xlabel('Time')
889 axes[0].set_ylabel('Returns')
890 axes[0].set_title('RETURNS vs VaR METHODS (Recent Period)')
891 axes[0].legend(fontsize=9)
892 axes[0].grid(True, alpha=0.3)
893
894 x_range = np.linspace(x.min(), x.max(), 1000)
895 parametric_pdf = chosen_dist.pdf(x_range, *chosen_params) *
896     cond_sigma_series.iloc[-1]
897 axes[1].hist(x, bins=100, density=True, alpha=0.6, color='lightgray',
898                 label='Std Residuals')
899 axes[1].plot(x_range, parametric_pdf, 'r-', linewidth=2, label=f'Parametric
900     ({chosen_name})')
901
902 axes[1].axvline(VaR_param.iloc[-1], color='red', linestyle='--', linewidth=2,
903                 label=f'Parametric VaR')
904 axes[1].axvline(VaR_emp.iloc[-1], color='blue', linestyle='--', linewidth=2,
905                 label=f'Empirical VaR')
906
907 axes[1].set_xlabel('Returns')
908 axes[1].set_ylabel('Density')
909 axes[1].set_title('DISTRIBUTION COMPARISON')
910 axes[1].legend(fontsize=9)
911 axes[1].grid(True, alpha=0.3)
912
913 plt.tight_layout()
914 plt.show()

```

Notes:

### 3 Code for the Multivariate Analysis

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import scipy.stats as stats
5 import seaborn as sns
6 from statsmodels.tsa.stattools import adfuller
7 from statsmodels.graphics.tsaplots import plot_acf
8 from statsmodels.stats.diagnostic import het_arch, acorr_ljungbox
9 from scipy.optimize import minimize
10 from arch import arch_model
11 import scipy.stats as st
12
13 DEFAULT_FILE = r"C:\Users\marco\OneDrive\Desktop\Empirical exam
   econometrics\Slanzi_Marco.xlsx"
14
15 infile = DEFAULT_FILE
16
17 print(f"Loading file: {infile} (sheet Sheet2)")
18 df = pd.read_excel(infile, sheet_name="Foglio2", engine="openpyxl")
19 df.columns = ["date", "Price1", "Price2", "Price3"]
20
21 df['date'] = pd.to_datetime(df['date'], dayfirst=True)
22 df = df.set_index('date')
23
24 print("Columns:", df.columns.tolist())
25 print(df.head())
26
27 for col in ["Price1", "Price2", "Price3"]:
28     df[f"log_{col}"] = np.log(df[col])
29     df[f"logret_{col}"] = df[f"log_{col}"].diff()
30
31 df = df.replace([np.inf, -np.inf], np.nan).dropna()
32 print(f"Final data shape: {df.shape}")
33
34 logp_cols = ["log_Price1", "log_Price2", "log_Price3"]
35 logret_cols = ["logret_Price1", "logret_Price2", "logret_Price3"]
36 acf_lags = 40
37
38 plt.style.use('default')
39
40 distribution_stats = []
41
42 for i in range(3):
43     series_name = logret_cols[i]
44     price_name = logp_cols[i]
45     rets_series = df[series_name]
46     logp_series = df[price_name]
47
48     print(f"\nAnalysis for {series_name}")
49     print("-" * 40)
50
51     print(f"Mean: {rets_series.mean():.6f}")
```

```

52     print(f"Std: {rets_series.std():.6f}")
53     print(f"Min: {rets_series.min():.6f}")
54     print(f"Max: {rets_series.max():.6f}")
55
56     adf_stat, adf_p, _, _, _ = adfuller(rets_series)
57     print(f"ADF test: statistic={adf_stat:.4f}, p-value={adf_p:.4g}")
58
59     skew_ret = stats.skew(rets_series, bias=False)
60     kurt_ret = stats.kurtosis(rets_series, fisher=False, bias=False)
61     jb_stat_ret, jb_p_ret = stats.jarque_bera(rets_series)
62
63     distribution_stats.append({
64         'Series': series_name,
65         'Skewness': skew_ret,
66         'Kurtosis': kurt_ret,
67         'JB_Statistic': jb_stat_ret,
68         'JB_p_value': jb_p_ret,
69         'Normality': 'Reject' if jb_p_ret < 0.05 else 'Cannot reject',
70     })
71
72     print(f"Skewness: {skew_ret:.6f}")
73     print(f"Kurtosis: {kurt_ret:.6f}")
74     print(f"Jarque-Bera test: stat={jb_stat_ret:.6f}, p-value={jb_p_ret:.6g}")
75     if jb_p_ret < 0.05:
76         print(" -> Reject normality (returns are not normal).")
77     else:
78         print(" -> Cannot reject normality at 5%.")
79
80     arch_stat, arch_pvalue, _, _ = het_arch(rets_series, nlags=12)
81     print(f"ARCH LM test: stat={arch_stat:.4f}, p-value={arch_pvalue:.4g}")
82     if arch_pvalue < 0.05:
83         print(" -> Reject H0: ARCH effects detected.")
84     else:
85         print(" -> Fail to reject H0: no strong ARCH evidence.")
86
87     fig, ax = plt.subplots(3, 1, figsize=(11, 9), sharex=True)
88     ax[0].plot(rets_series.index, rets_series)
89     ax[0].set_title(f"{series_name}: Log-Returns (r_t)")
90
91     ax[1].plot(rets_series.index, rets_series**2)
92     ax[1].set_title("Squared Log-Returns (r_t) volatility clustering")
93
94     ax[2].plot(logp_series.index, logp_series)
95     ax[2].set_title(f"{price_name}: Log-Price (L_t)")
96     plt.tight_layout()
97     plt.show()
98
99     fig, ax = plt.subplots(1, 2, figsize=(12, 4))
100    ax[0].hist(rets_series, bins=80)
101    ax[0].set_title(f"Histogram of {series_name}")
102    stats.probplot(rets_series, dist="t", sparams=(5,), plot=ax[1])
103    ax[1].set_title("QQ-plot vs t(df=5) reference")
104    plt.tight_layout()
105    plt.show()
106
107    fig = plt.figure(figsize=(10, 5))
108    plot_acf(rets_series, lags=acf_lags)
109    plt.title(f"ACF of {series_name}")

```

```

110     plt.tight_layout()
111     plt.show()
112
113     fig = plt.figure(figsize=(10, 5))
114     plot_acf(rets_series**2, lags=acf_lags)
115     plt.title(f"ACF of {series_name} (squared returns)")
116     plt.tight_layout()
117     plt.show()
118
119 print("\n" + "="*80)
120 print("SUMMARY TABLE: Distributional Statistics for All Series")
121 print("="*80)
122 summary_df = pd.DataFrame(distribution_stats)
123 print("\n" + summary_df.to_string(index=False, float_format=".6f"))
124
125 print("\n" + "="*80)
126 print("FORMATTED SUMMARY TABLE")
127 print("="*80)
128 print(f'{Series':<15} {'Skewness':<10} {'Kurtosis':<10} {'JB Stat':<12} {'JB
129     p-value':<12} {'Normality':<15}')
130 print("-" * 80)
131 for stat in distribution_stats:
132     print(f'{stat["Series"]:<15} {stat["Skewness"]:>9.6f}
133         {stat["Kurtosis"]:>9.6f} {stat["JB_Statistic"]:>11.6f}
134         {stat["JB_p_value"]:>11.6g} {stat["Normality"]:>15}')
135
136 candidate_specs = [
137     {"name": "GARCH11-t", "vol": "GARCH", "p": 1, "o": 0, "q": 1,
138      "dist": "StudentsT"}, ,
139     {"name": "GJR11-t", "vol": "GARCH", "p": 1, "o": 1, "q": 1,
140      "dist": "StudentsT"}, ,
141     {"name": "EGARCH11-t", "vol": "EGARCH", "p": 1, "o": 1, "q": 1,
142      "dist": "StudentsT"}]
143
144 eps_reg = 1e-8
145
146 selected_models = []
147 model_summaries = []
148
149 def run_diagnostics(res, rets_index, lags=[10, 20]):
150     std_resid = pd.Series(res.std_resid,
151                           index=rets_index[-len(res.std_resid):])
152     jb_s, jb_p = stats.jarque_bera(std_resid)
153     lb1 = acorr_ljungbox(std_resid, lags=[lags[0]],
154                           return_df=True)[['lb_pvalue']].iloc[0]
155     lb1_sq = acorr_ljungbox(std_resid**2, lags=[lags[0]],
156                           return_df=True)[['lb_pvalue']].iloc[0]
157     lb2 = acorr_ljungbox(std_resid, lags=[lags[1]],
158                           return_df=True)[['lb_pvalue']].iloc[0]
159     lb2_sq = acorr_ljungbox(std_resid**2, lags=[lags[1]],
160                           return_df=True)[['lb_pvalue']].iloc[0]
161     arch_stat, arch_p, _, _ = het_arch(std_resid, nlags=12)
162     return {
163         "JB_p": jb_p,
164         "LB_resid_p_lag10": lb1,
165         "LB_sq_p_lag10": lb1_sq,
166         "LB_resid_p_lag20": lb2,

```

```

157         "LB_sq_p_lag20": lb2_sq,
158         "ARCH_LM_p": arch_p
159     }
160
161 print("\n1) Fitting candidate univariate models and selecting best by AIC\n"
162      + "-"*70)
163
164 for series_name in logret_cols:
165     df[f"{series_name}_scaled"] = df[series_name] * 100.0
166
167 for series_name in logret_cols:
168     print(f"\nSeries: {series_name}")
169     rets = df[f"{series_name}_scaled"].dropna()
170
171     if rets.empty:
172         print("  WARNING: no data for this series, skipping.")
173         continue
174
175     fitted = {}
176     errors = {}
177     for spec in candidate_specs:
178         label = spec["name"]
179         try:
180             o = spec.get("o", 0)
181             am = arch_model(rets, mean='Constant', vol=spec["vol"],
182                             p=spec["p"],
183                             o=o if o>0 else 0, q=spec["q"], dist=spec["dist"])
184             res = am.fit(disp='off', show_warning=False, options={'maxiter':
185                         1000})
186             fitted[label] = res
187             print(f"  {label} fitted    llf={res.loglikelihood:.2f}
188                   AIC={res.aic:.4f}")
189         except Exception as e:
190             errors[label] = str(e)
191             print(f"  {label} FAILED: {e}")
192
193     comp = []
194     for name, r in fitted.items():
195         comp.append({"model": name, "AIC": r.aic, "BIC": r.bic, "LLF":
196                     r.loglikelihood})
197     if len(comp) == 0:
198         print("  No successful fits for this series. Errors:", errors)
199         continue
200     comp_df = pd.DataFrame(comp).sort_values("AIC").reset_index(drop=True)
201     print("\n  Model comparison (sorted by AIC):")
202     print(comp_df.to_string(index=False, float_format=".4f"))
203
204     best_name = comp_df.loc[0, "model"]
205     best_res = fitted[best_name]
206     selected_models[series_name] = best_res
207
208     print(f"\n  Selected: {best_name} (AIC={comp_df.loc[0, 'AIC']:.4f})")
209
210     diag = run_diagnostics(best_res, rets.index)
211     print("  Diagnostics on standardized residuals:")
212     print(f"    Jarque-Bera p: {diag['JB_p']:.4g}")
213     print(f"    Ljung-Box resid p (lag10): {diag['LB_resid_p_lag10']:.4g}")
214     print(f"    Ljung-Box resid^2 p (lag10): {diag['LB_sq_p_lag10']:.4g}")

```

```

210     print(f"      ARCH LM p (nlags=12): {diag['ARCH_LM_p']:.4g}")
211
212     std_resid = pd.Series(best_res.std_resid,
213         index=rets.index[-len(best_res.std_resid):])
214     plt.figure(figsize=(8,3))
215     plt.hist(std_resid, bins=80)
216     plt.title(f"{series_name} standardized residuals ({best_name})")
217     plt.tight_layout(); plt.show()
218
219     cond_vol = best_res.conditional_volatility
220     plt.figure(figsize=(10,3))
221     plt.plot(cond_vol.index, cond_vol, label=f"{best_name} cond vol
222         ({sigma_t})")
223     plt.title(f"{series_name} conditional volatility ({best_name})")
224     plt.legend(); plt.tight_layout(); plt.show()
225
226     model_summaries.append({
227         "Series": series_name,
228         "Best_model": best_name,
229         "AIC": float(comp_df.loc[0, "AIC"]),
230         "BIC": float(comp_df.loc[0, "BIC"]),
231         "LLF": float(comp_df.loc[0, "LLF"]),
232         "JB_p": diag["JB_p"],
233         "LB_sq_p_lag10": diag["LB_sq_p_lag10"],
234         "ARCH_LM_p": diag["ARCH_LM_p"]
235     })
236
237 summary_sel_df = pd.DataFrame(model_summaries)
238 print("\nFinal selection summary:")
239 print(summary_sel_df.to_string(index=False, float_format=".6g"))
240
241 print("\n" + "="*60)
242 print("COMPREHENSIVE STANDARDIZED RESIDUALS DIAGNOSTICS")
243 print("="*60)
244
245 fig, axes = plt.subplots(3, 3, figsize=(15, 12))
246
247 for i, series_name in enumerate(logret_cols):
248     if series_name in selected_models:
249         res = selected_models[series_name]
250         best_name = summary_sel_df[summary_sel_df['Series'] ==
251             series_name]['Best_model'].values[0]
252
253         std_resid = pd.Series(res.std_resid, index=res.model.y.index)
254
255         axes[0, i].plot(std_resid.index, std_resid, alpha=0.7, linewidth=0.8)
256         axes[0, i].set_title(f'{series_name}\nStandardized Residuals
257             ({best_name})')
258         axes[0, i].axhline(y=0, color='red', linestyle='--', alpha=0.5)
259         axes[0, i].grid(True, alpha=0.3)
260
261         axes[1, i].hist(std_resid, bins=60, density=True, alpha=0.7,
262             color='skyblue', edgecolor='black')
263
264         x = np.linspace(std_resid.min(), std_resid.max(), 100)
265         normal_pdf = stats.norm.pdf(x, 0, 1)
266         axes[1, i].plot(x, normal_pdf, 'r-', linewidth=2, label='N(0,1)')
267         axes[1, i].set_title(f'Distribution vs Normal')

```

```

263     axes[1, i].legend()
264     axes[1, i].grid(True, alpha=0.3)
265
266     stats.probplot(std_resid, dist="norm", plot=axes[2, i])
267     axes[2, i].set_title(f'QQ Plot vs Normal')
268     axes[2, i].grid(True, alpha=0.3)
269
270 plt.tight_layout()
271 plt.show()
272
273 print("\nACF plots for standardized residuals...")
274 fig, axes = plt.subplots(3, 2, figsize=(15, 12))
275
276 for i, series_name in enumerate(logret_cols):
277     if series_name in selected_models:
278         res = selected_models[series_name]
279         best_name = summary_sel_df[summary_sel_df['Series'] ==
280             series_name]['Best_model'].values[0]
281         std_resid = pd.Series(res.std_resid, index=res.model.y.index)
282
283         plot_acf(std_resid, lags=40, ax=axes[i, 0], alpha=0.05)
284         axes[i, 0].set_title(f'{series_name}\nACF of Std Residuals
285             ({best_name})')
286
287         plot_acf(std_resid**2, lags=40, ax=axes[i, 1], alpha=0.05)
288         axes[i, 1].set_title(f'ACF of Squared Std Residuals')
289
290 plt.tight_layout()
291 plt.show()
292
293 print("\n" + "="*60)
294 print("COMPREHENSIVE DIAGNOSTICS ON STANDARDIZED RESIDUALS")
295 print("="*60)
296
297 diagnostics_summary = []
298
299 for series_name in logret_cols:
300     if series_name in selected_models:
301         res = selected_models[series_name]
302         best_name = summary_sel_df[summary_sel_df['Series'] ==
303             series_name]['Best_model'].values[0]
304         std_resid = pd.Series(res.std_resid, index=res.model.y.index)
305
306         mean_val = std_resid.mean()
307         std_val = std_resid.std()
308         skew_val = stats.skew(std_resid)
309         kurt_val = stats.kurtosis(std_resid, fisher=False)
310
311         jb_stat, jb_p = stats.jarque_bera(std_resid)
312
313         lb_resid_5 = acorr_ljungbox(std_resid, lags=[5], return_df=True)
314         lb_resid_10 = acorr_ljungbox(std_resid, lags=[10], return_df=True)
315         lb_sq_5 = acorr_ljungbox(std_resid**2, lags=[5], return_df=True)
316         lb_sq_10 = acorr_ljungbox(std_resid**2, lags=[10], return_df=True)
317
318         arch_stat, arch_p, _, _ = het_arch(std_resid, nlags=12)
319
320         diagnostics_summary.append({

```

```

318         'Series': series_name,
319         'Model': best_name,
320         'Mean': mean_val,
321         'Std': std_val,
322         'Skewness': skew_val,
323         'Kurtosis': kurt_val,
324         'JB_p': jb_p,
325         'LB5_resid_p': lb_resid_5['lb_pvalue'].iloc[0],
326         'LB10_resid_p': lb_resid_10['lb_pvalue'].iloc[0],
327         'LB5_sq_p': lb_sq_5['lb_pvalue'].iloc[0],
328         'LB10_sq_p': lb_sq_10['lb_pvalue'].iloc[0],
329         'ARCH_p': arch_p
330     })
331
332 diag_df = pd.DataFrame(iagnostics_summary)
333
334 def run_diagnostics_robust(res, rets_index, lags=[10,20]):
335     std_resid = pd.Series(res.std_resid,
336                           index=rets_index[-len(res.std_resid):])
337
338     std_resid_clean = std_resid.replace([np.inf, -np.inf], np.nan).dropna()
339
340     if len(std_resid_clean) < 50:
341         return {
342             "JB_p": np.nan, "LB_resid_p_lag10": np.nan, "LB_sq_p_lag10":
343                 np.nan,
344             "LB_resid_p_lag20": np.nan, "LB_sq_p_lag20": np.nan, "ARCH_LM_p":
345                 np.nan
346         }
347
348     try:
349         jb_s, jb_p = stats.jarque_bera(std_resid_clean)
350     except:
351         jb_p = np.nan
352
353     try:
354         lb1 = acorr_ljungbox(std_resid_clean, lags=[lags[0]],
355                               return_df=True)[‘lb_pvalue’].iloc[0]
356         lb1_sq = acorr_ljungbox(std_resid_clean**2, lags=[lags[0]],
357                               return_df=True)[‘lb_pvalue’].iloc[0]
358         lb2 = acorr_ljungbox(std_resid_clean, lags=[lags[1]],
359                               return_df=True)[‘lb_pvalue’].iloc[0]
360         lb2_sq = acorr_ljungbox(std_resid_clean**2, lags=[lags[1]],
361                               return_df=True)[‘lb_pvalue’].iloc[0]
362     except:
363         lb1, lb1_sq, lb2, lb2_sq = np.nan, np.nan, np.nan, np.nan
364
365     try:
366         arch_stat, arch_p, _, _ = het_arch(std_resid_clean, nlags=12)
367     except:
368         arch_p = np.nan
369
370     return {
371         "JB_p": jb_p,
372         "LB_resid_p_lag10": lb1,
373         "LB_sq_p_lag10": lb1_sq,
374         "LB_resid_p_lag20": lb2,
375         "LB_sq_p_lag20": lb2_sq,

```

```

369         "ARCH_LM_p": arch_p
370     }
371
372 candidate_specs_arma = [
373     {"name": "AR1-GARCH11-t", "mean": "AR", "lags": 1, "vol": "GARCH", "p": 1,
374      "o": 0, "q": 1, "dist": "StudentsT"}, 
375     {"name": "AR1-GJR11-t", "mean": "AR", "lags": 1, "vol": "GARCH", "p": 1,
376      "o": 1, "q": 1, "dist": "StudentsT"}, 
377     {"name": "AR1-EGARCH11-t", "mean": "AR", "lags": 1, "vol": "EGARCH", "p": 1,
378      "o": 1, "q": 1, "dist": "StudentsT"}, 
379 ]
380
381 improved_models = {}
382 improved_summaries = []
383
384 for series_name in logret_cols:
385     print(f"\n--- Improving {series_name} with ARMA structure ---")
386     rets = df[f"{series_name}_scaled"].dropna()
387
388     if rets.isna().any() or np.isinf(rets).any():
389         print(f"          Numerical issues in returns, cleaning...")
390         rets = rets.replace([np.inf, -np.inf], np.nan).dropna()
391
392     fitted_improved = {}
393     errors_improved = {}
394
395     for spec in candidate_specs_arma:
396         label = spec["name"]
397         try:
398             o = spec.get("o", 0)
399             lags = spec["lags"]
400
401             am = arch_model(
402                 rets,
403                 mean=spec["mean"],
404                 lags=lags,
405                 vol=spec["vol"],
406                 p=spec["p"],
407                 o=o if o>0 else 0,
408                 q=spec["q"],
409                 dist=spec["dist"]
410             )
411             res = am.fit(disp='off', show_warning=False, options={'maxiter': 1000})
412             fitted_improved[label] = res
413             print(f" {label} fitted AIC={res.aic:.4f}")
414
415         except Exception as e:
416             errors_improved[label] = str(e)
417             print(f" {label} FAILED: {e}")
418
419         if fitted_improved:
420             comp_improved = []
421             for name, r in fitted_improved.items():
422                 comp_improved.append({
423                     "model": name,
424                     "AIC": r.aic,
425                     "BIC": r.bic,

```

```

423         "LLF": r.loglikelihood
424     })
425
426     comp_improved_df =
427         pd.DataFrame(comp_improved).sort_values("AIC").reset_index(drop=True)
428     print("\n  Improved model comparison (sorted by AIC):")
429     print(comp_improved_df.to_string(index=False, float_format=".4f"))
430
431     best_improved_name = comp_improved_df.loc[0, "model"]
432     best_improved_res = fitted_improved[best_improved_name]
433     improved_models[series_name] = best_improved_res
434
435     diag_improved = run_diagnostics_robust(best_improved_res, rets.index)
436
437     improved_summaries.append({
438         "Series": series_name,
439         "Best_model": best_improved_name,
440         "AIC": float(comp_improved_df.loc[0, "AIC"]),
441         "BIC": float(comp_improved_df.loc[0, "BIC"]),
442         "LLF": float(comp_improved_df.loc[0, "LLF"]),
443         "LB_resid_p_lag10": diag_improved["LB_resid_p_lag10"],
444         "LB_sq_p_lag10": diag_improved["LB_sq_p_lag10"],
445         "ARCH_LM_p": diag_improved["ARCH_LM_p"]
446     })
447
448     original_aic = summary_sel_df[summary_sel_df['Series'] ==
449                     series_name]['AIC'].values[0]
450     improvement = original_aic - comp_improved_df.loc[0, "AIC"]
451     print(f"  AIC improvement over original: {improvement:.4f}")
452
453     if diag_improved['LB_resid_p_lag10'] > 0.05:
454         print("      Autocorrelation successfully addressed!")
455     else:
456         print("      Some autocorrelation may still remain")
457
458 if improved_summaries:
459     print("\n" + "*60)
460     print("IMPROVED MODELS SUMMARY")
461     print("*60)
462     improved_df = pd.DataFrame(improved_summaries)
463     print(improved_df.to_string(float_format=".6f", index=False))
464
465     print("\nReplacing original models with improved ARMA-GARCH models for
466           DCC estimation...")
467     selected_models.update(improved_models)
468
469     print("\nUpdating standardized residuals for DCC with improved models...")
470     std_resid_list = []
471     cond_vol_list = []
472     series_list = []
473
474     for series_name in logret_cols:
475         if series_name in selected_models:
476             res = selected_models[series_name]
477             std_resid = pd.Series(res.std_resid, index=res.model.y.index)
478             std_resid_list.append(std_resid)

```

```

477         cond_vol = pd.Series(res.conditional_volatility,
478                               index=res.model.y.index)
479         cond_vol_list.append(cond_vol)
480         series_list.append(series_name)
481
482         common_idx = std_resid_list[0].index
483         for i in range(1, len(std_resid_list)):
484             common_idx = common_idx.intersection(std_resid_list[i].index)
485
486         std_resid_aligned = []
487         cond_vol_aligned = []
488         for i in range(len(std_resid_list)):
489             std_resid_aligned.append(std_resid_list[i].loc[common_idx])
490             cond_vol_aligned.append(cond_vol_list[i].loc[common_idx])
491
492         std_resid_df = pd.DataFrame({series_list[i]: std_resid_aligned[i] for i
493                                     in range(len(series_list))})
494         cond_vol_df = pd.DataFrame({series_list[i]: cond_vol_aligned[i] for i in
495                                     range(len(series_list))})
496
497         print(f"Updated standardized residuals shape: {std_resid_df.shape}")
498         print("    Models updated successfully for DCC estimation")
499
500     else:
501         print("\nNo improved models were successfully fitted. Using original
502               models.")
503
504     print("\n2) Preparing standardized residuals for DCC estimation")
505
506     std_resid_list = []
507     cond_vol_list = []
508     series_list = []
509
510     for series_name in logret_cols:
511         if series_name in selected_models:
512             res = selected_models[series_name]
513             std_resid = pd.Series(res.std_resid, index=res.model.y.index)
514             std_resid_list.append(std_resid)
515             cond_vol = pd.Series(res.conditional_volatility,
516                                   index=res.model.y.index)
517             cond_vol_list.append(cond_vol)
518             series_list.append(series_name)
519
520             common_idx = std_resid_list[0].index
521             for i in range(1, len(std_resid_list)):
522                 common_idx = common_idx.intersection(std_resid_list[i].index)
523
524             std_resid_aligned = []
525             cond_vol_aligned = []
526             for i in range(len(std_resid_list)):
527                 std_resid_aligned.append(std_resid_list[i].loc[common_idx])
528                 cond_vol_aligned.append(cond_vol_list[i].loc[common_idx])
529
530             std_resid_df = pd.DataFrame({series_list[i]: std_resid_aligned[i] for i in
531                                         range(len(series_list))})
532             cond_vol_df = pd.DataFrame({series_list[i]: cond_vol_aligned[i] for i in
533                                         range(len(series_list))})

```

```

528 std_resid_df_clean = std_resid_df.dropna()
529 cond_vol_df_clean = cond_vol_df.loc[std_resid_df_clean.index]
530
531
532
533
534
535 std_resid_df = std_resid_df_clean
536 cond_vol_df = cond_vol_df_clean
537
538
539
540 Z = std_resid_df.values
541 T, N = Z.shape
542
543
544
545
546 fig, axes = plt.subplots(3, 3, figsize=(15, 12))
547 fig.suptitle('COMPREHENSIVE DIAGNOSTICS: OPTIMIZED ARMA-GARCH STANDARDIZED
548 RESIDUALS',
549             fontsize=14, y=1.02)
550
551 for i, series_name in enumerate(logret_cols):
552     if series_name in improved_models:
553         res = improved_models[series_name]
554         best_name = improved_df[improved_df['Series'] ==
555             series_name]['Best_model'].values[0]
556
557         std_resid = std_resid_df[series_name]
558
559         axes[0, i].plot(std_resid.index, std_resid, alpha=0.7, linewidth=0.8)
560         axes[0, i].set_title(f'{series_name}\nOptimized Std Residuals
561 ({best_name})', fontsize=10)
562         axes[0, i].axhline(y=0, color='red', linestyle='--', alpha=0.5)
563         axes[0, i].axhline(y=2, color='orange', linestyle=':', alpha=0.5,
564             label='2 std')
565         axes[0, i].axhline(y=-2, color='orange', linestyle=':', alpha=0.5)
566         axes[0, i].grid(True, alpha=0.3)
567
568         axes[1, i].hist(std_resid, bins=60, density=True, alpha=0.7,
569             color='lightgreen', edgecolor='black')
570
571         x = np.linspace(std_resid.min(), std_resid.max(), 100)
572         normal_pdf = stats.norm.pdf(x, 0, 1)
573         axes[1, i].plot(x, normal_pdf, 'r-', linewidth=2, label='N(0,1)')
574         axes[1, i].set_title(f'Distribution vs Normal', fontsize=10)
575         axes[1, i].legend(fontsize=8)
576         axes[1, i].grid(True, alpha=0.3)
577
578         stats.probplot(std_resid, dist="norm", plot=axes[2, i])
579         axes[2, i].set_title(f'QQ Plot vs Normal', fontsize=10)
580         axes[2, i].grid(True, alpha=0.3)
581
582 plt.tight_layout()
583 plt.show()
584
585 print("\nAUTOCORRELATION ANALYSIS: OPTIMIZED STANDARDIZED RESIDUALS")

```

```

581 fig, axes = plt.subplots(3, 2, figsize=(15, 12))
582
583 for i, series_name in enumerate(logret_cols):
584     if series_name in improved_models:
585         res = improved_models[series_name]
586         best_name = improved_df[improved_df['Series'] ==
587             series_name]['Best_model'].values[0]
588         std_resid = std_resid_df[series_name]
589
590         plot_acf(std_resid, lags=40, ax=axes[i, 0], alpha=0.05,
591                   title=f'{series_name}\nACF of Optimized Std Residuals')
592         axes[i, 0].set_title(f'{series_name}\nACF of Optimized Std
593             Residuals', fontsize=10)
594
595         plot_acf(std_resid**2, lags=40, ax=axes[i, 1], alpha=0.05,
596                   title=f'ACF of Squared Optimized Std Residuals')
597         axes[i, 1].set_title(f'ACF of Squared Optimized Std Residuals',
598                           fontsize=10)
599
600 plt.tight_layout()
601 plt.show()
602
603 print("\n" + "="*60)
604 print("STATISTICAL SUMMARY: OPTIMIZED STANDARDIZED RESIDUALS")
605 print("="*60)
606
607 optimized_stats = []
608 for series_name in logret_cols:
609     if series_name in improved_models:
610         std_resid = std_resid_df[series_name]
611         best_name = improved_df[improved_df['Series'] ==
612             series_name]['Best_model'].values[0]
613
614         mean_val = std_resid.mean()
615         std_val = std_resid.std()
616         skew_val = stats.skew(std_resid)
617         kurt_val = stats.kurtosis(std_resid, fisher=False)
618
619         jb_stat, jb_p = stats.jarque_bera(std_resid)
620
621         lb_resid_10 = acorr_ljungbox(std_resid, lags=[10], return_df=True)
622         lb_sq_10 = acorr_ljungbox(std_resid**2, lags=[10], return_df=True)
623
624         arch_stat, arch_p, _, _ = het_arch(std_resid, nlags=12)
625
626         optimized_stats.append({
627             'Series': series_name,
628             'Model': best_name,
629             'Mean': mean_val,
630             'Std': std_val,
631             'Skewness': skew_val,
632             'Kurtosis': kurt_val,
633             'JB_p': jb_p,
634             'LB_resid_p': lb_resid_10['lb_pvalue'].iloc[0],
635             'LB_sq_p': lb_sq_10['lb_pvalue'].iloc[0],
636             'ARCH_p': arch_p
637         })
638
639

```

```

635 opt_stats_df = pd.DataFrame(optimized_stats)
636
637 R_ccc = std_resid_df.corr().values
638
639
640
641 plt.figure(figsize=(8, 6))
642 sns.heatmap(ccc_df, annot=True, cmap='coolwarm', center=0,
643             square=True, fmt='.4f', cbar_kws={'shrink': 0.8})
644 plt.title('CONSTANT CONDITIONAL CORRELATION (CCC) MATRIX\n(Using ARMA-GARCH
645 Standardized Residuals)')
646 plt.tight_layout()
647 plt.show()
648
649 def build_Ht_CCC(cond_vol_df, R):
650     Dt = cond_vol_df.values
651     Tloc, Nloc = Dt.shape
652     Hs = np.empty((Tloc, Nloc, Nloc))
653     for t in range(Tloc):
654         D = np.diag(Dt[t])
655         Hs[t] = D @ R @ D
656     return Hs
657
658 H_ccc_all = build_Ht_CCC(cond_vol_df, R_ccc)
659 print(f"CCC conditional covariance matrices shape: {H_ccc_all.shape}")
660
661 print("\nCCC CORRELATION STATISTICS:")
662 ccc_stats = []
663 for i in range(N):
664     for j in range(i+1, N):
665         pair = f"{series_list[i]} vs {series_list[j]}"
666         corr_val = R_ccc[i, j]
667         ccc_stats.append({
668             'Pair': pair,
669             'CCC_Correlation': corr_val
670         })
671
672 ccc_stats_df = pd.DataFrame(ccc_stats)
673 print(ccc_stats_df.to_string(index=False, float_format=".6f"))
674
675
676
677 R_hat = (Z.T @ Z) / T
678
679 def dcc_negloglike_paper(params, epsilon, R_unconditional, eps_reg=1e-8):
680     zeta, xi = params
681
682     if zeta <= 1e-10 or xi <= 1e-10 or (zeta + xi) >= 1.0:
683         return 1e12
684
685     Tloc, Nloc = epsilon.shape
686     Q_t = R_unconditional.copy()
687     negll = 0.0
688
689     for t in range(Tloc):
690         diag_q = np.sqrt(np.diag(Q_t))
691         if np.any(diag_q <= eps_reg):

```

```

692         return 1e12
693
694     Z_t = np.diag(1.0 / diag_q)
695     R_t = Z_t @ Q_t @ Z_t
696
697     try:
698         sign, logdet = np.linalg.slogdet(R_t)
699         if sign <= 0:
700             return 1e12
701
702         diag_R = np.diag(R_t)
703         if np.any(np.abs(diag_R - 1.0) > 1e-6):
704             return 1e12
705
706         R_inv = np.linalg.inv(R_t)
707         eps_t = epsilon[t, :]
708         quad_form = eps_t @ (R_inv @ eps_t)
709
710         negll += 0.5 * (logdet + quad_form)
711
712     except np.linalg.LinAlgError:
713         return 1e12
714
715     if t < Tloc - 1:
716         eps_outer = np.outer(eps_t, eps_t)
717         Q_t_next = (1.0 - zeta - xi) * R_unconditional + zeta * eps_outer
718         + xi * Q_t
719
720         try:
721             min_eigval = np.min(np.real(np.linalg.eigvals(Q_t_next)))
722             if min_eigval <= 1e-8:
723                 Q_t_next = 0.95 * Q_t_next + 0.05 * R_unconditional
724         except np.linalg.LinAlgError:
725             Q_t_next = 0.95 * Q_t + 0.05 * R_unconditional
726
727         Q_t = Q_t_next
728
729     return negll
730
731 print("\nSTEP 3: ESTIMATING DCC PARAMETERS ( , ) BY MAXIMUM LIKELIHOOD")
732 print("-" * 60)
733 bounds = [(1e-8, 0.5), (1e-8, 0.999)]
734 cons = ({'type': 'ineq', 'fun': lambda x: 1.0 - x[0] - x[1] - 1e-8})
735
736 starting_points = [
737     np.array([0.02, 0.97]),
738     np.array([0.05, 0.90]),
739     np.array([0.10, 0.85]),
740     np.array([0.01, 0.98]),
741 ]
742
743 best_result = None
744 best_negll = np.inf
745
746 print("Optimizing DCC parameters ( , ) with multiple starting points...")
747
748 for i, x0 in enumerate(starting_points):

```

```

749     zeta_0, xi_0 = x0
750     print(f"\nAttempt {i+1}/{len(starting_points)}:      ={zeta_0:.3f},
751           ={xi_0:.3f},      +      ={zeta_0+xi_0:.3f})")
752
753     try:
754         res_opt = minimize(dcc_negloglike_paper, x0, args=(Z, R_hat, eps_reg),
755                             method='SLSQP', bounds=bounds, constraints=cons,
756                             options={'ftol': 1e-8, 'disp': False, 'maxiter':
757                                       500})
758
759         if res_opt.success:
760             zeta_opt, xi_opt = res_opt.x
761             print(f"      SUCCESS:    ={zeta_opt:.6f},    ={xi_opt:.6f},
762                   +    ={(zeta_opt+xi_opt):.6f}, LL={-res_opt.fun:.2f}")
763             if res_opt.fun < best_negll:
764                 best_result = res_opt
765                 best_negll = res_opt.fun
766             else:
767                 print(f"      FAILED: {res_opt.message}")
768
769     except Exception as e:
770         print(f"      EXCEPTION: {e}")
771
772 print("\nSTEP 4: DCC ESTIMATION RESULTS (Paper Specification)")
773 print("-" * 60)
774
775 if best_result is not None:
776     zeta_dcc, xi_dcc = best_result.x
777     negll_dcc = float(best_result.fun)
778
779 else:
780     print("      DCC optimization failed, using sensible defaults")
781     zeta_dcc, xi_dcc = 0.02, 0.97
782     negll_dcc = dcc_negloglike_paper([zeta_dcc, xi_dcc], Z, R_hat, eps_reg)
783     print(f"      Using fallback:    ={zeta_dcc:.4f},    ={xi_dcc:.4f}")
784
785 persistence = zeta_dcc + xi_dcc
786
787
788 def compute_dcc_correlations_paper(epsilon, R_hat, zeta, xi, eps_reg=1e-8):
789     T, N = epsilon.shape
790     R_dynamic = np.zeros((T, N, N))
791     Q_dynamic = np.zeros((T, N, N))
792     Z_dynamic = np.zeros((T, N, N))
793
794     Q_t = R_hat.copy()
795
796     for t in range(T):
797         Q_dynamic[t] = Q_t.copy()
798
799         diag_q = np.sqrt(np.diag(Q_t))
800         Z_t = np.diag(1.0 / (diag_q + eps_reg))
801         Z_dynamic[t] = Z_t
802
803         R_t = Z_t @ Q_t @ Z_t

```

```

804     R_dynamic[t] = R_t
805
806     if t < T - 1:
807         eps_t = epsilon[t, :]
808         eps_outer = np.outer(eps_t, eps_t)
809         Q_t_next = (1.0 - zeta - xi) * R_hat + zeta * eps_outer + xi * Q_t
810
811         try:
812             min_eigval = np.min(np.real(np.linalg.eigvals(Q_t_next)))
813             if min_eigval <= 1e-8:
814                 Q_t_next = 0.95 * Q_t_next + 0.05 * R_hat
815         except np.linalg.LinAlgError:
816             Q_t_next = 0.95 * Q_t + 0.05 * R_hat
817
818         Q_t = Q_t_next
819
820     return R_dynamic, Q_dynamic, Z_dynamic
821
822 R_dcc_dynamic, Q_dcc_dynamic, Z_dcc_dynamic = compute_dcc_correlations_paper(
823     Z, R_hat, zeta_dcc, xi_dcc
824 )
825
826
827
828 def ccc_loglikelihood(epsilon, R):
829     T, N = epsilon.shape
830     ll = 0.0
831
832     try:
833         R_inv = np.linalg.inv(R)
834         sign, logdet = np.linalg.slogdet(R)
835         if sign <= 0:
836             return -1e12
837     except np.linalg.LinAlgError:
838         return -1e12
839
840     for t in range(T):
841         eps_t = epsilon[t, :]
842         quad_form = eps_t @ R_inv @ eps_t
843         ll += -0.5 * (logdet + quad_form)
844
845     return ll
846
847 print("\n" + "="*80)
848 print("COMPREHENSIVE MODEL COMPARISON: CCC vs DCC")
849 print("="*80)
850
851 R_ccc = std_resid_df.corr().values
852
853 R_dcc_unconditional = np.mean(R_dcc_dynamic, axis=0)
854
855
856
857 def compute_aic_bic(ll, n_params, n_obs):
858     aic = -2 * ll + 2 * n_params
859     bic = -2 * ll + n_params * np.log(n_obs)
860
861     return aic, bic

```

```

862 n_ccc_params = 3
863 n_dcc_params = 2
864
865 ccc_aic, ccc_bic = compute_aic_bic(ccc_ll, n_ccc_params, T)
866 dcc_aic, dcc_bic = compute_aic_bic(dcc_ll, n_dcc_params, T)
867
868
869
870 def likelihood_ratio_test(ll_restricted, ll_unrestricted, df):
871     lr_stat = 2 * (ll_unrestricted - ll_restricted)
872     p_value = 1 - stats.chi2.cdf(lr_stat, df)
873     return lr_stat, p_value
874
875 lr_stat, lr_pvalue = likelihood_ratio_test(ccc_ll, dcc_ll, df=2)
876
877 print(f"\nLIKELIHOOD RATIO TEST:")
878 print(f"LR Statistic: {lr_stat:.4f}")
879 print(f"P-value: {lr_pvalue:.6f}")
880 if lr_pvalue < 0.05:
881     print("    DCC significantly better than CCC at 5% level")
882     if lr_pvalue < 0.01:
883         print("    DCC significantly better than CCC at 1% level")
884 else:
885     print("    Cannot reject that CCC is adequate")
886
887 print("\n" + "="*80)
888 print("MODEL COMPARISON COMPLETE")
889 print("="*80)
890
891 print("\n" + "="*80)
892 print("CCC vs DCC COMPREHENSIVE VISUALIZATION & ANALYSIS")
893 print("="*80)
894
895 time_index = std_resid_df.index
896 N = len(series_list)
897 colors = ['#1f77b4', '#ff7f0e', '#2ca02c']
898 pair_names = ['Price1 vs Price2', 'Price1 vs Price3', 'Price2 vs Price3']
899 pair_indices = [(0, 1), (0, 2), (1, 2)]
900
901 fig, axes = plt.subplots(3, 1, figsize=(14, 10))
902 fig.suptitle('Dynamic Correlations: DCC vs CCC (Separated by Pair)',
903              fontsize=15, fontweight='bold')
904
905 for idx, (i, j) in enumerate(pair_indices):
906     dcc_corr = R_dcc_dynamic[:, i, j]
907     ccc_corr = R_ccc[i, j]
908
909     axes[idx].plot(time_index, dcc_corr, color=colors[idx], lw=1.8,
910                     alpha=0.85, label='DCC Correlation')
911     axes[idx].axhline(y=ccc_corr, color='red', linestyle='--', lw=2,
912                         label='CCC Constant')
913
914     axes[idx].set_title(f'{pair_names[idx]}', fontweight='bold')
915     axes[idx].set_ylabel('Correlation')
916     axes[idx].grid(alpha=0.3)
917     axes[idx].legend(fontsize=9)
918     axes[idx].tick_params(axis='x', rotation=45)

```

```

917 axes[-1].set_xlabel('Time')
918 plt.tight_layout()
919 plt.show()
920
921 fig, ax = plt.subplots(figsize=(10, 6))
922 for idx, (i, j) in enumerate(pair_indices):
923     dcc_corr = R_dcc_dynamic[:, i, j]
924     ccc_corr = R_ccc[i, j]
925     ax.hist(dcc_corr, bins=50, alpha=0.6, color=colors[idx], density=True,
926             label=f'DCC {pair_names[idx]}')
927     ax.axvline(x=ccc_corr, color=colors[idx], linestyle='--', lw=2,
928                 label=f'CCC {pair_names[idx]}')
929
930     ax.set_title('Correlation Distributions: DCC vs CCC', fontsize=12,
931                 fontweight='bold')
932     ax.set_xlabel('Correlation')
933     ax.set_ylabel('Density')
934     ax.legend(fontsize=9)
935     ax.grid(alpha=0.3)
936     plt.show()
937
938 fig, axes = plt.subplots(1, 2, figsize=(14, 6))
939 fig.suptitle('Correlation Matrices Comparison', fontsize=15,
940               fontweight='bold')
941
942 im1 = axes[0].imshow(R_ccc, cmap='RdYlBu_r', vmin=-0.5, vmax=1.0)
943 axes[0].set_title('CCC Correlation Matrix', fontsize=12, fontweight='bold')
944 axes[0].set_xticks(range(N)); axes[0].set_yticks(range(N))
945 axes[0].set_xticklabels([s.replace('logret_', '') for s in series_list])
946 axes[0].set_yticklabels([s.replace('logret_', '') for s in series_list])
947 for i in range(N):
948     for j in range(N):
949         axes[0].text(j, i, f'{R_ccc[i, j]:.3f}', ha="center", va="center",
950                     color="black", fontsize=10)
951
952 im2 = axes[1].imshow(R_dcc_unconditional, cmap='RdYlBu_r', vmin=-0.5,
953                     vmax=1.0)
954 axes[1].set_title('DCC Average Correlation Matrix', fontsize=12,
955                     fontweight='bold')
956 axes[1].set_xticks(range(N)); axes[1].set_yticks(range(N))
957 axes[1].set_xticklabels([s.replace('logret_', '') for s in series_list])
958 axes[1].set_yticklabels([s.replace('logret_', '') for s in series_list])
959 for i in range(N):
960     for j in range(N):
961         axes[1].text(j, i, f'{R_dcc_unconditional[i, j]:.3f}', ha="center",
962                     va="center", color="black", fontsize=10)
963
964 plt.colorbar(im1, ax=axes[0], fraction=0.046, pad=0.04)
965 plt.colorbar(im2, ax=axes[1], fraction=0.046, pad=0.04)
966 plt.tight_layout()
967 plt.show()
968
969 print("\nROLLING CORRELATION STATISTICS (252-day windows)")
970
971 fig, axes = plt.subplots(3, 1, figsize=(14, 10))
972 fig.suptitle('Rolling Correlation Statistics (252-day windows)', fontsize=14,
973               fontweight='bold')

```

```

966 for idx, (i, j) in enumerate(pair_indices):
967     dcc_corr = pd.Series(R_dcc_dynamic[:, i, j], index=time_index)
968     rolling_mean = dcc_corr.rolling(window=252).mean()
969     rolling_std = dcc_corr.rolling(window=252).std()
970     ccc_corr = R_ccc[i, j]
971
972     axes[idx].plot(time_index, dcc_corr, alpha=0.3, color=colors[idx],
973                     label='Daily DCC')
974     axes[idx].plot(time_index, rolling_mean, color='black', lw=2,
975                     label='252-day Rolling Mean')
976     axes[idx].fill_between(time_index, rolling_mean - rolling_std,
977                           rolling_mean + rolling_std,
978                           alpha=0.2, color=colors[idx], label=' 1 Std Dev')
979     axes[idx].axhline(y=ccc_corr, color='red', linestyle='--', lw=2,
980                         label='CCC Constant')
981
982     axes[idx].set_title(pair_names[idx], fontweight='bold')
983     axes[idx].set_ylabel('Correlation')
984     axes[idx].legend()
985     axes[idx].grid(alpha=0.3)
986
987 axes[-1].set_xlabel('Time')
988 plt.tight_layout()
989 plt.show()
990
991 print("\n" + "="*80)
992 print("DCC DYNAMIC CORRELATION SUMMARY STATISTICS")
993 print("="*80)
994
995 summary_data = []
996 for idx, (i, j) in enumerate(pair_indices):
997    dcc_corr = R_dcc_dynamic[:, i, j]
998    ccc_corr = R_ccc[i, j]
999    summary_data.append({
1000         'Pair': pair_names[idx],
1001         'CCC_Correlation': ccc_corr,
1002         'DCC_Mean': np.mean(dcc_corr),
1003         'DCC_Std': np.std(dcc_corr),
1004         'DCC_Min': np.min(dcc_corr),
1005         'DCC_Max': np.max(dcc_corr),
1006         'DCC_Range': np.max(dcc_corr) - np.min(dcc_corr),
1007         'Volatility_Ratio': np.std(dcc_corr) / np.mean(np.abs(dcc_corr))
1008     })
1009
1010 summary_df = pd.DataFrame(summary_data)
1011 print("\n" + summary_df.to_string(index=False, float_format=".4f"))
1012
1013 print("\n" + "="*80)
1014 print("MODEL SELECTION CONCLUSION")
1015 print("="*80)

```

## 4 Code for the VAR model

```
1 import warnings
2 warnings.filterwarnings("ignore")
3
4 import pandas as pd
5 import numpy as np
6 import matplotlib.pyplot as plt
7 import seaborn as sns
8 from scipy.stats import jarque_bera
9
10 from statsmodels.tsa.api import VAR
11 from statsmodels.tsa.stattools import adfuller, kpss, grangercausalitytests
12 from statsmodels.tsa.vector_ar.vecm import coint_johansen
13 from statsmodels.stats.stattools import durbin_watson
14 from statsmodels.stats.diagnostic import acorr_ljungbox, het_arch
15
16 from sklearn.metrics import mean_squared_error
17
18 sns.set(style="whitegrid")
19 colors = ['tab:blue', 'tab:orange', 'tab:green']
20
21 print("*"*70)
22 print("VECTOR AUTOREGRESSION (VAR) + GARCH ANALYSIS - CORRECTED")
23 print("Oil Price Shocks on Macroeconomic Variables")
24 print("*"*70)
25
26 DEFAULT_FILE = r"C:\Users\marco\OneDrive\Desktop\Empirical exam
27   econometrics\Slanzi_Marco.xlsx"
28 df = pd.read_excel(DEFAULT_FILE, sheet_name="Foglio3", engine="openpyxl")
29
30 df.columns = ["obs", "oil_price", "inflation", "gdp_growth"]
31 df = df.rename(columns={'obs': 'date'})
32
33 df['date'] = pd.to_datetime(df['date'].astype(str).str.replace('M', '-'),
34   format='%Y-%m')
35 df = df.set_index('date').sort_index()
36 try:
37   df.index = pd.DatetimeIndex(df.index, freq='MS')
38 except Exception:
39   pass
40
41 fig, axes = plt.subplots(3, 1, figsize=(12, 9), sharex=True)
42 for i, col in enumerate(df.columns):
43   axes[i].plot(df.index, df[col], color=colors[i], linewidth=1.2)
44   axes[i].set_title(f'{col} - Time Series')
45   axes[i].set_ylabel(col)
46   axes[i].grid(alpha=0.3)
47 plt.xlabel('Date')
48 plt.tight_layout()
49 plt.show()
50
51 plt.figure(figsize=(8, 6))
52 corr_matrix = df.corr()
53 sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', center=0,
```

```

54         square=True, fmt='%.3f', cbar_kws={'shrink': 0.8})
55 plt.title('Correlation Matrix')
56 plt.tight_layout()
57 plt.show()
58
59 print("\nCorrelation matrix:")
60 print(corr_matrix.round(3))
61
62 for col in df.columns:
63     print(f"\n{col} distribution - skew={df[col].skew():.3f}\n"
64           f"        kurt={df[col].kurtosis():.3f}")
65     print(f"Rolling stats (12 months):\n"
66           f"        mean={df[col].rolling(12).mean().iloc[-1]:.4f},\n"
67           f"        std={df[col].rolling(12).std().iloc[-1]:.4f}")
68
69 def granger_causality_matrix(data, variables, max_lag=8, verbose=False):
70     df_gc = pd.DataFrame(np.nan, index=variables, columns=variables,
71                          dtype=float)
72     for response in variables:
73         for predictor in variables:
74             if response == predictor:
75                 df_gc.loc[response, predictor] = np.nan
76                 continue
77             try:
78                 test_result = grangercausalitytests(data[[response,
79                                               predictor]].dropna(),
80                                               maxlag=max_lag,
81                                               verbose=verbose)
82                 p_values = [test_result[lag][0]['ssr_chi2test'][1] for lag in
83                             range(1, max_lag + 1)]
84                 df_gc.loc[response, predictor] = np.min(p_values)
85             except Exception as e:
86                 print(f"Granger test failed for {predictor} -> {response}:\n"
87                       f"        {e}")
88                 df_gc.loc[response, predictor] = np.nan
89     return df_gc
90
91
92 print("\nPerforming Granger causality tests (min p-value over lags 1..8):")
93 vars_ = list(df.columns)
94 granger_pvals = granger_causality_matrix(df, vars_, max_lag=8, verbose=False)
95 print(granger_pvals.round(4))
96
97 def adf_test(series, title=''):
98     res = adfuller(series.dropna(), autolag='AIC')
99     return res[1]
100
101 def kpss_test(series, title=''):
102     stat, p_value, lags, crit = kpss(series.dropna(), regression='c',
103                                       nlags="auto")
104     return p_value
105
106 print("\n" + "="*50)
107 print("STATIONARITY ANALYSIS - DATA DRIVEN DECISION")
108 print("="*50)
109
110 stationarity_results = []
111 for col in df.columns:
112     adf_p = adf_test(df[col], title=col)

```

```

103     kpss_p = kpss_test(df[col], title=col)
104
105     is_stationary = (adf_p < 0.05) and (kpss_p > 0.05)
106     stationarity_results.append(is_stationary)
107
108     print(f"\{col:12\} - ADF p-value: {adf_p:.4f}, KPSS p-value: {kpss_p:.4f}\n"
109           " -> {'STATIONARY' if is_stationary else 'NON-STATIONARY'}")
110
111 use_diff = not all(stationarity_results)
112
113 print(f"\nSTATIONARITY DECISION: {'DIFFERENCING' if use_diff else 'LEVELS'}")
114 print(f"Rationale: {sum(stationarity_results)}/{len(stationarity_results)}\n"
115       "series are stationary")
116
117 if use_diff:
118     df_var = df.diff().dropna()
119     print("Using first differences for VAR modeling")
120 else:
121     df_var = df.copy()
122     print("Using levels for VAR modeling (all series stationary)")
123
124 print("\nData used for VAR (first 5 rows):")
125 print(df_var.head())
126
127 model = VAR(df_var)
128 maxlags_search = 12
129 lag_res = model.select_order(maxlags_search)
130 print("\nLag order selection (all criteria):")
131 print(lag_res.summary())
132
133 criteria = ['aic', 'bic', 'hqic']
134 selected_lags = {c: lag_res.selected_orders[c] for c in criteria}
135 print("\nLag selection across criteria:")
136 for crit, lag in selected_lags.items():
137     print(f" {crit.upper()}: {lag}")
138
139 lag_counts = {}
140 for lag in selected_lags.values():
141     lag_counts[lag] = lag_counts.get(lag, 0) + 1
142
143 if max(lag_counts.values()) >= 2:
144     selected_lag = max(lag_counts, key=lag_counts.get)
145     print(f"Selected lag (majority): {selected_lag}")
146 else:
147     selected_lag = selected_lags['bic']
148     print(f"Selected lag (BIC - parsimony): {selected_lag}")
149
150 n_obs = 12
151 train = df_var[:-n_obs]
152 test = df_var[-n_obs:]
153 print(f"\nTrain shape: {train.shape}, Test shape: {test.shape}")
154 print(f"Train period: {train.index.min().date()} to\n"
155       f"{train.index.max().date()}")
156 print(f"Test period: {test.index.min().date()} to {test.index.max().date()}")
157
158 p = int(selected_lag) if (selected_lag is not None and not
159             np.isnan(selected_lag)) else 1
160 var_model = VAR(train)

```

```

157 var_results = var_model.fit(p)
158 print("\nVAR MODEL SUMMARY:")
159 print(var_results.summary())
160
161 print("\n" + "="*50)
162 print("RESIDUAL DIAGNOSTICS")
163 print("="*50)
164
165 print("\nDurbin-Watson statistics (close to 2 indicates no autocorrelation):")
166 dw = durbin_watson(var_results.resid)
167 for col, val in zip(train.columns, dw):
168     interpretation = "no autocorrelation" if 1.5 < val < 2.5 else "possible
169     autocorrelation"
170     print(f" {col}: {val:.3f} - {interpretation}")
171
172 print("\nLjung-Box test for residual autocorrelation:")
173 for col in var_results.resid.columns:
174     lb = acorr_ljungbox(var_results.resid[col], lags=[p, 2*p], return_df=True)
175     print(f"\n{col}:")
176     for lag in lb.index:
177         sig = "***" if lb.loc[lag, 'lb_pvalue'] < 0.01 else "**" if
178             lb.loc[lag, 'lb_pvalue'] < 0.05 else "*" if lb.loc[lag,
179                 'lb_pvalue'] < 0.1 else ""
180         print(f" Lag {lag:2d}: p-value = {lb.loc[lag, 'lb_pvalue']:.4f}
181             {sig}")
182
183 print("\n" + "="*50)
184 print("ARCH EFFECTS TESTING (Engle's Test using het_arch)")
185 print("Necessary condition for GARCH modeling")
186 print("="*50)
187
188 arch_test_results = {}
189 garch_recommended = {}
190
191 for col in var_results.resid.columns:
192     try:
193         arch_test = het_arch(var_results.resid[col])
194         arch_pvalue = arch_test[1]
195         arch_test_results[col] = arch_pvalue
196         garch_recommended[col] = arch_pvalue < 0.05
197
198         print(f"{col:12} - ARCH test p-value: {arch_pvalue:.4f} -> {'GARCH
199             RECOMMENDED' if garch_recommended[col] else 'No significant ARCH
200             effects'}")
201     except Exception as e:
202         print(f"{col:12} - ARCH test failed: {e}")
203         arch_test_results[col] = 1.0
204         garch_recommended[col] = False
205
206 series_for_garch = [col for col in var_results.resid.columns if
207     garch_recommended[col]]
208 print(f"\nProceeding with GARCH modeling for: {series_for_garch}")
209
210 lag_order = var_results.k_ar
211 last_obs_for_forecast = train.values[-lag_order:]
212 fc = var_results.forecast(last_obs_for_forecast, steps=n_obs)
213 fc_df = pd.DataFrame(fc, index=test.index, columns=train.columns)
214 print("\nVAR forecast (model scale; first 5 rows):")

```

```

208 print(fc_df.head().round(4))
209
210 if use_diff:
211     orig_forecast = pd.DataFrame(index=fc_df.index, columns=fc_df.columns,
212         dtype=float)
213     for col in fc_df.columns:
214         last_level = df[col].loc[train.index[-1]]
215         orig_forecast[col] = last_level + fc_df[col].cumsum()
216     else:
217         orig_forecast = fc_df.copy()
218
219 print("\nForecast on original scale (first 5 rows):")
220 print(orig_forecast.head().round(4))
221
222 actual_levels = df.loc[orig_forecast.index, orig_forecast.columns]
223
224 fig, axes = plt.subplots(3, 1, figsize=(12, 10), sharex=True)
225 for i, col in enumerate(orig_forecast.columns):
226     train_extended = df[col].loc[:test.index[0]].iloc[-24:]
227
228     axes[i].plot(train_extended.index, train_extended, color=colors[i],
229                 linewidth=1.5, label='Historical')
230     axes[i].plot(actual_levels.index, actual_levels[col], color=colors[i],
231                 linewidth=2, label='Actual')
232     axes[i].plot(orig_forecast.index, orig_forecast[col], color='red',
233                 linestyle='--',
234                 linewidth=2, label='VAR Forecast')
235     axes[i].set_title(f'VAR Forecast vs Actual: {col}')
236     axes[i].set_ylabel(col)
237     axes[i].legend()
238     axes[i].grid(alpha=0.3)
239
240     axes[i].axvspan(test.index[0], test.index[-1], alpha=0.1, color='gray')
241
242 plt.xlabel('Date')
243 plt.tight_layout()
244 plt.show()
245
246 print("\nFORECAST ACCURACY (RMSE):")
247 rmse_results = {}
248 for col in orig_forecast.columns:
249     rmse = np.sqrt(mean_squared_error(actual_levels[col], orig_forecast[col]))
250     rmse_results[col] = rmse
251     print(f" {col:12} RMSE: {rmse:.4f}")
252
253 print("\n" + "*50)
254 print("GARCH MODELING")
255 print("*50)
256
257 try:
258     from arch import arch_model
259 except Exception as e:
260     raise ImportError("To run the GARCH stage install the 'arch' package: pip
261         install arch") from e
262
263 residuals = var_results.resid
264 garch_results = {}

```

```

260 garch_forecasts_var = pd.DataFrame(index=test.index,
261                                         columns=residuals.columns, dtype=float)
262
263 if series_for_garch:
264     print(f"Fitting GARCH(1,1) models for: {series_for_garch}")
265
266     for col in series_for_garch:
267         print(f"\n{'='*30}")
268         print(f"GARCH for: {col}")
269         print(f"{'='*30}")
270
271     series = residuals[col].dropna()
272
273     best_aic = np.inf
274     best_model = None
275     best_dist = None
276
277     for dist in ['normal', 't', 'skewt']:
278         try:
279             am = arch_model(series, mean='Zero', vol='Garch', p=1, q=1,
280                             dist=dist)
281             gres = am.fit(disp='off', show_warning=False)
282
283             if gres.aic < best_aic:
284                 best_aic = gres.aic
285                 best_model = gres
286                 best_dist = dist
287
288         except Exception as e:
289             print(f" {dist} distribution failed: {e}")
290             continue
291
292     if best_model is not None:
293         garch_results[col] = best_model
294         print(f"Best distribution: {best_dist}")
295         print(best_model.summary())
296
297     horizon = len(test.index)
298     fc_g = best_model.forecast(horizon=horizon, reindex=False)
299
300     try:
301         var_fc = fc_g.variance.iloc[-1].values
302     except Exception:
303         try:
304             var_array = np.asarray(fc_g.variance)
305             var_fc = var_array.reshape(-1)[-horizon:]
306         except Exception:
307             var_fc = np.full(horizon, np.nan)
308
309     if len(var_fc) < horizon:
310         var_fc = np.resize(var_fc, horizon)
311     elif len(var_fc) > horizon:
312         var_fc = var_fc[:horizon]
313
314     garch_forecasts_var[col] = var_fc
315
316     plt.figure(figsize=(10, 4))
317     conditional_vol = best_model.conditional_volatility

```

```

316     plt.plot(train.index[1:], conditional_vol,
317               color=colors[list(train.columns).index(col)], linewidth=1)
318     plt.title(f'Conditional Volatility - {col}')
319     plt.ylabel('Conditional Std. Dev.')
320     plt.grid(alpha=0.3)
321     plt.tight_layout()
322     plt.show()
323 else:
324     print(f" GARCH fitting failed for {col} with all distributions")
325 garch_forecasts_var[col] = np.nan
326 else:
327     print("No significant ARCH effects detected - skipping GARCH modeling")
328 for col in residuals.columns:
329     if col not in garch_forecasts_var or
330         garch_forecasts_var[col].isna().all():
331         constant_var = residuals[col].var()
332         garch_forecasts_var[col] = constant_var
333
334 print("\nGARCH conditional variance forecasts (first 5 rows):")
335 print(garch_forecasts_var.head().round(6))
336
337 combined_table = orig_forecast.copy()
338 for col in orig_forecast.columns:
339     combined_table[col + "_cond_std"] =
340         np.sqrt(garch_forecasts_var[col].values)
341
342 print("\nCOMBINED VAR + GARCH RESULTS (first 5 rows):")
343 print("Mean forecasts with conditional standard deviations:")
344 print(combined_table.head().round(4))
345
346 print("\n" + "="*50)
347 print("COINTEGRATION ANALYSIS (Johansen Test)")
348 print("=". * 50)
349 try:
350     johansen_result = coint_johansen(df, det_order=0, k_ar_diff=p)
351     print("Johansen Cointegration Test Results:")
352     print(f"Trace statistic critical values:
353         90%={johansen_result.cvt[0,0]:.3f},
354         95%={johansen_result.cvt[0,1]:.3f},
355         99%={johansen_result.cvt[0,2]:.3f}")
356
357     n_coint = np.sum(johansen_result.lrt > johansen_result.cvt[:, 1])
358     print(f"Number of cointegrating relationships (95% confidence):
359         {n_coint}")
360
361     if n_coint > 0:
362         print("Cointegration detected - consider VECM for long-run analysis")
363     else:
364         print("No cointegration - VAR in differences is appropriate")
365
366 except Exception as e:
367     print(f"Cointegration test failed: {e}")
368
369 print("\n" + "="*60)
370 print("RECOMMENDATION: VECM MODEL (Due to Cointegration)")
371 print("=". * 60)

```

```

367
368 print("""
369 STRONG COINTEGRATION DETECTED (3 relationships)
370
371 Recommendation: Use VECM instead of VAR for:
372 1. Capturing long-run equilibrium relationships
373 2. Better economic interpretation
374 3. Improved forecasting performance
375
376 Your current VAR in levels is valid but suboptimal for cointegrated data.
377 """
378
379 try:
380     from statsmodels.tsa.vector_ar.vecm import VECM
381
382     print("\nFitting VECM model...")
383
384     print("VECM code commented out - uncomment to run VECM analysis")
385
386 except ImportError:
387     print("VECM modeling requires additional setup")
388
389 print("\n" + "="*70)
390 print("FINAL RECOMMENDATIONS")
391 print("="*70)
392
393 print("""
394 1. **METHODOLOGICAL**: Switch to VECM modeling due to cointegration
395 2. **ECONOMIC INTERPRETATION**:
396     - Oil prices significantly drive inflation (0.327***)
397     - Inflation negatively affects GDP growth (-0.120**)
398     - All variables show strong persistence
399 3. **FORECASTING**: RMSE values provide baseline for model comparison
400 4. **VOLATILITY**: No GARCH effects detected - constant variance assumption
401     valid
402
403 NEXT STEPS:
404 - Implement VECM model with 3 cointegrating relationships
405 - Analyze long-run equilibrium relationships
406 - Compare VECM vs VAR forecast performance
407 - Conduct impulse response analysis
408 """
409 print("\n" + "="*70)
410 print("ANALYSIS COMPLETE - KEY FINDINGS")
411 print("="*70)
412
413 print(f"\n1. DATA: {df.shape[0]} monthly observations, {df.shape[1]} variables")
414 print(f"2. STATIONARITY: Using {'DIFFERENCES' if use_diff else 'LEVELS'}")
415 print(f"3. OPTIMAL LAG: {selected_lag} (selected by majority/BIC)")
416 print(f"4. GRANGER CAUSALITY: Oil      Inflation***, Oil      GDP*, Inflation
417             GDP**")
418
419 print(f"\n5. VAR FORECAST ACCURACY (RMSE):")
420 for col, rmse in rmse_results.items():
421     print(f"    {col:12}: {rmse:.4f}")
422

```

```

422 print(f"\n6. ARCH EFFECTS: GARCH modeling {'recommended for ' + ','
423     '.join(series_for_garch) if series_for_garch else 'not recommended'}")
424
425
426 for response in granger_pvals.index:
427     for predictor in granger_pvals.columns:
428         if response != predictor and granger_pvals.loc[response, predictor] <
429             0.05:
430             sig_level = "***" if granger_pvals.loc[response, predictor] <
431                 0.01 else "**" if granger_pvals.loc[response, predictor] <
432                     0.05 else "*"
433             print(f" {predictor} {response}: p =
434                 {granger_pvals.loc[response, predictor]:.4f} {sig_level}")
435
436
437
438
439
440
441
442 print("\nResponse: inflation")
443 try:
444     coeff_oil_oil = var_results.params.iloc[1, 0] if
445         len(var_results.params.columns) > 1 else var_results.params.iloc[1]
446     coeff_inf_oil = var_results.params.iloc[2, 0] if
447         len(var_results.params.columns) > 1 else var_results.params.iloc[2]
448     coeff_gdp_oil = var_results.params.iloc[3, 0] if
449         len(var_results.params.columns) > 1 else var_results.params.iloc[3]
450
451
452
453
454     print(f" L1.oil_price: {coeff_oil_oil:.3f}***")
455     print(f" L1.inflation: {coeff_inf_oil:.3f}***")
456     print(f" L1.gdp_growth: {coeff_gdp_oil:.3f}")
457 except Exception as e:
458     print(f" Error accessing coefficients: {e}")
459
460
461
462
463 print("\nResponse: gdp_growth")
464 try:
465     coeff_oil_gdp = var_results.params.iloc[1, 2] if
466         len(var_results.params.columns) > 1 else var_results.params.iloc[7]
467     coeff_inf_gdp = var_results.params.iloc[2, 2] if
468         len(var_results.params.columns) > 1 else var_results.params.iloc[8]
469     coeff_gdp_gdp = var_results.params.iloc[3, 2] if
470         len(var_results.params.columns) > 1 else var_results.params.iloc[9]
471
472     print(f" L1.oil_price: {coeff_oil_gdp:.3f}")
473     print(f" L1.inflation: {coeff_inf_gdp:.3f}**")
474     print(f" L1.gdp_growth: {coeff_gdp_gdp:.3f}***")
475 except Exception as e:
476     print(f" Error accessing coefficients: {e}")
477
478
479

```

```

466 print("\nFull VAR Parameters Table:")
467 print(var_results.params.round(4))
468
469 print("\n6. RESIDUAL DIAGNOSTICS")
470 print("-" * 40)
471 print("Durbin-Watson Statistics:")
472 for col, val in zip(train.columns, dw):
473     status = "No autocorrelation" if 1.5 < val < 2.5 else "Possible
474         autocorrelation"
475     print(f" {col:12}: {val:.3f} ({status})")
476
477 print("\nLjung-Box Test (p-values):")
478 for col in var_results.resid.columns:
479     lb = acorr_ljungbox(var_results.resid[col], lags=[p], return_df=True)
480     pval = lb.loc[p, 'lb_pvalue']
481     sig = " " if pval > 0.05 else " "
482     print(f" {col:12}: {pval:.4f} {sig}")
483
484 print("\n7. VOLATILITY ANALYSIS")
485 print("-" * 40)
486 print("ARCH Effects Test (p-values):")
487 for col, pval in arch_test_results.items():
488     status = "No GARCH needed" if pval > 0.05 else "GARCH recommended"
489     print(f" {col:12}: {pval:.4f} ({status})")
490
491 print("\n8. FORECAST ACCURACY")
492 print("-" * 40)
493 print("Root Mean Squared Error (RMSE):")
494 for col, rmse in rmse_results.items():
495     print(f" {col:12}: {rmse:.4f}")
496
497 print("\n9. COINTEGRATION ANALYSIS")
498 print("-" * 40)
499 print(f"Number of Cointegrating Relationships: {n_coint}")
500 if n_coint > 0:
501     print(" Strong evidence of long-run equilibrium relationships")
502     print(" Recommendation: Consider VECM model for improved analysis")
503 else:
504     print(" No cointegration - VAR model is appropriate")
505
506 validation_checks = {
507     "Stationarity": all(stationarity_results),
508     "No Residual Autocorrelation": all(1.5 < val < 2.5 for val in dw),
509     "No ARCH Effects": all(not garch_recommended[col] for col in
510         garch_recommended),
511     "Clear Lag Selection": selected_lag is not None,
512     "Cointegration Present": n_coint > 0
513 }
514
515 for check, result in validation_checks.items():
516     status = " PASS" if result else " CHECK"
517     print(f" {check:30}: {status}")
518
519
520
521

```

```

522 results_summary = pd.DataFrame({
523     'Variable': list(df.columns),
524     'Mean': [df[col].mean() for col in df.columns],
525     'Std_Dev': [df[col].std() for col in df.columns],
526     'Stationarity': stationarity_results,
527     'ARCH_Test_Pvalue': [arch_test_results[col] for col in df.columns],
528     'Forecast_RMSE': [rmse_results[col] for col in df.columns],
529     'Residual_Autocorr': [dw[i] for i in range(len(df.columns))]}
530 })
531
532 print("\nTechnical Results DataFrame:")
533 print(results_summary.round(4))
534
535 print("="*70)
536 print("GENERATING ADDITIONAL PLOTS FOR REPORT")
537 print("="*70)
538
539 print("\nGenerating Impulse Response Functions...")
540
541 try:
542     irf = var_results.irf(periods=24)
543
544     fig, axes = plt.subplots(3, 3, figsize=(15, 12))
545     fig.suptitle('Orthogonalized Impulse Response Functions (24 months)', fontsize=16, y=0.95)
546
547     responses = ['oil_price', 'inflation', 'gdp_growth']
548
549     for i, shock in enumerate(responses):
550         for j, response in enumerate(responses):
551             axes[i,j].plot(irf.irfs[:, i, j], linewidth=2)
552             axes[i,j].axhline(0, color='red', linestyle='--', alpha=0.5)
553             axes[i,j].set_title(f'{shock} {response}')
554             axes[i,j].grid(alpha=0.3)
555
556             try:
557                 axes[i,j].fill_between(range(len(irf.irfs[:, i, j])), irf.lower[:, i, j], irf.upper[:, i, j], alpha=0.2)
558             except:
559                 pass
560
561     plt.tight_layout()
562     plt.show()
563
564 except Exception as e:
565     print(f"IRF plotting failed: {e}")
566
567 print("\nGenerating Forecast Error Variance Decomposition...")
568
569 try:
570     fevd = var_results.fevd(periods=24)
571
572     fig, axes = plt.subplots(1, 3, figsize=(18, 6))
573     fig.suptitle('Forecast Error Variance Decomposition (24 months ahead)', fontsize=16)
574
575
576
577

```

```

578     periods = range(1, 25)
579     colors = ['#1f77b4', '#ff7f0e', '#2ca02c']
580
581     for i, var in enumerate(responses):
582         fevd_data = fevd.decomp[i]
583         cumulative = np.zeros(len(periods))
584
585         for j, shock in enumerate(responses):
586             axes[i].fill_between(periods, cumulative, cumulative +
587                 fevd_data[:, j],
588                 label=shock, alpha=0.7, color=colors[j])
589             cumulative += fevd_data[:, j]
590
591         axes[i].set_title(f'Variance Decomposition: {var}')
592         axes[i].set_xlabel('Months Ahead')
593         axes[i].set_ylabel('Proportion of Variance')
594         axes[i].legend()
595         axes[i].grid(alpha=0.3)
596         axes[i].set_ylim(0, 1)
597
598     plt.tight_layout()
599     plt.show()
600
601 except Exception as e:
602     print(f"FEVD plotting failed: {e}")
603
604 print("\nGenerating Historical Decomposition...")
605
606 try:
607     fig, axes = plt.subplots(3, 1, figsize=(14, 10))
608     fig.suptitle('Variable Contributions to Historical Movements',
609                  fontsize=16)
610
611     fitted_values = var_results.fittedvalues
612
613     for i, col in enumerate(df_var.columns):
614         axes[i].plot(train.index[1:], train[col].iloc[1:], label='Actual',
615                       linewidth=2, color='black')
616         axes[i].plot(train.index[1:], fitted_values[col], label='Fitted',
617                       linestyle='--', linewidth=1.5)
618         axes[i].set_title(f'{col} - Model Fit')
619         axes[i].set_ylabel(col)
620         axes[i].legend()
621         axes[i].grid(alpha=0.3)
622
623     plt.tight_layout()
624     plt.show()
625
626 except Exception as e:
627     print(f"Historical decomposition plotting failed: {e}")
628
629 print("\nGenerating Detailed Residual Analysis...")
630
631 fig, axes = plt.subplots(3, 3, figsize=(15, 12))
632 fig.suptitle('Comprehensive Residual Diagnostics', fontsize=16)
633
634 residuals = var_results.resid

```

```

632 for i, col in enumerate(residuals.columns):
633     axes[i, 0].plot(residuals.index, residuals[col], color=colors[i])
634     axes[i, 0].axhline(0, color='red', linestyle='--', alpha=0.7)
635     axes[i, 0].set_title(f'{col} - Residuals')
636     axes[i, 0].set_ylabel('Residual')
637     axes[i, 0].grid(alpha=0.3)
638
639     from statsmodels.graphics.tsaplots import plot_acf
640     plot_acf(residuals[col], ax=axes[i, 1], lags=20, alpha=0.05)
641     axes[i, 1].set_title(f'{col} - ACF')
642
643     axes[i, 2].hist(residuals[col], bins=30, density=True, alpha=0.7,
644                      color=colors[i])
645     axes[i, 2].set_title(f'{col} - Distribution')
646     axes[i, 2].set_xlabel('Residual Value')
647
648     from scipy.stats import norm
649     x = np.linspace(residuals[col].min(), residuals[col].max(), 100)
650     axes[i, 2].plot(x, norm.pdf(x, residuals[col].mean(),
651                               residuals[col].std()),
652                      'r-', linewidth=2, label='Normal')
653     axes[i, 2].legend()
654
655 plt.tight_layout()
656 plt.show()
657
658 print("\nGenerating Rolling Statistics...")
659
660 fig, axes = plt.subplots(2, 2, figsize=(15, 10))
661 fig.suptitle('Rolling Statistics and Volatility Analysis', fontsize=16)
662
663 for i, col in enumerate(df.columns):
664     rolling_mean = df[col].rolling(window=12).mean()
665     rolling_std = df[col].rolling(window=12).std()
666
667     axes[0, 0].plot(rolling_mean.index, rolling_mean, label=col, linewidth=2)
668     axes[0, 0].set_title('12-Month Rolling Means')
669     axes[0, 0].legend()
670     axes[0, 0].grid(alpha=0.3)
671
672     for i, col in enumerate(df.columns):
673         rolling_std = df[col].rolling(window=12).std()
674         axes[0, 1].plot(rolling_std.index, rolling_std, label=col, linewidth=2)
675     axes[0, 1].set_title('12-Month Rolling Standard Deviations')
676     axes[0, 1].legend()
677     axes[0, 1].grid(alpha=0.3)
678
679     for i, col in enumerate(df.columns):
680         cumulative = (1 + df[col]).cumprod() if col != 'inflation' else
681             df[col].cumsum()
682         axes[1, 0].plot(cumulative.index, cumulative, label=col, linewidth=2)
683     axes[1, 0].set_title('Cumulative Path')
684     axes[1, 0].legend()
685     axes[1, 0].grid(alpha=0.3)
686
687     rolling_corr = df['oil_price'].rolling(window=24).corr(df['inflation'])
688     axes[1, 1].plot(rolling_corr.index, rolling_corr, linewidth=2, color='purple')

```

```

686 axes[1, 1].axhline(df['oil_price'].corr(df['inflation']), color='red',
687     linestyle='--',
688     label=f'Overall corr:
689         {df["oil_price"].corr(df["inflation"]):.3f}')
690 axes[1, 1].set_title('Rolling Correlation: Oil vs Inflation (24-month
691     window)')
692 axes[1, 1].legend()
693 axes[1, 1].grid(alpha=0.3)
694 axes[1, 1].set_ylim(-1, 1)
695
696 plt.tight_layout()
697 plt.show()
698
699 print("\nGenerating Forecast Intervals...")
700
701 try:
702     forecast_intervals = var_results.forecast_interval(train.values[-p:],
703             steps=n_obs, alpha=0.05)
704
705     for i, col in enumerate(df.columns):
706         historical = df[col].loc[train.index[-36]:]
707
708         axes[i].plot(historical.index, historical, label='Historical',
709             linewidth=2, color=colors[i])
710         axes[i].plot(orig_forecast.index, orig_forecast[col],
711             label='Forecast',
712             linewidth=2, color='red', linestyle='--')
713
714         if not use_diff:
715             lower = forecast_intervals[0][:, i]
716             upper = forecast_intervals[1][:, i]
717             axes[i].fill_between(orig_forecast.index, lower, upper,
718                 alpha=0.2, color='red',
719                 label='95% Confidence Interval')
720
721         axes[i].axvline(test.index[0], color='gray', linestyle=':',
722             alpha=0.7, label='Forecast Start')
723         axes[i].set_title(f'{col} - Forecast with Uncertainty')
724         axes[i].set_ylabel(col)
725         axes[i].legend()
726         axes[i].grid(alpha=0.3)
727
728         axes[i].axvspan(test.index[0], test.index[-1], alpha=0.1,
729             color='gray')
730
731 except Exception as e:
732     print(f"Forecast interval plotting failed: {e}")
733     for i, col in enumerate(df.columns):
734         historical = df[col].loc[train.index[-36]:]
735         axes[i].plot(historical.index, historical, label='Historical',
736             linewidth=2, color=colors[i])
737         axes[i].plot(orig_forecast.index, orig_forecast[col],
738             label='Forecast',
739             linewidth=2, color='red', linestyle='--')

```

```

732         axes[i].axvline(test.index[0], color='gray', linestyle=':', alpha=0.7)
733         axes[i].set_title(f'{col} - Forecast')
734         axes[i].legend()
735         axes[i].grid(alpha=0.3)
736
737     plt.tight_layout()
738     plt.show()
739
740 print("\nGenerating Structural Break Analysis...")
741
742 try:
743     from statsmodels.stats.diagnostic import breaks_cusumolsresid
744
745     fig, axes = plt.subplots(3, 1, figsize=(14, 10))
746     fig.suptitle('CUSUM Test for Structural Breaks in Residuals', fontsize=16)
747
748     for i, col in enumerate(residuals.columns):
749         try:
750             cusum_stat, pval, crit = breaks_cusumolsresid(residuals[col])
751
752             axes[i].plot(cusum_stat, linewidth=2, color=colors[i])
753             axes[i].axhline(crit[0], color='red', linestyle='--', alpha=0.7,
754                             label='5% Critical Value')
755             axes[i].axhline(-crit[0], color='red', linestyle='--', alpha=0.7)
756             axes[i].axhline(0, color='black', linestyle='-', alpha=0.5)
757             axes[i].set_title(f'{col} - CUSUM Test (p-value: {pval:.3f})')
758             axes[i].set_ylabel('CUSUM Statistic')
759             axes[i].legend()
760             axes[i].grid(alpha=0.3)
761
762             if pval < 0.05:
763                 axes[i].text(0.02, 0.95, 'STRUCTURAL BREAK DETECTED',
764                             transform=axes[i].transAxes, color='red',
765                             fontweight='bold')
766             else:
767                 axes[i].text(0.02, 0.95, 'No structural break',
768                             transform=axes[i].transAxes, color='green')
769
770         except Exception as e:
771             axes[i].text(0.3, 0.5, f'CUSUM test failed: {str(e)[:50]}...', transform=axes[i].transAxes)
772             axes[i].set_title(f'{col} - CUSUM Test')
773
774     plt.tight_layout()
775     plt.show()
776
777 except ImportError:
778     print("CUSUM test not available - skipping structural break analysis")
779
780 print("\nGenerating Time-Varying Correlation Analysis...")
781
782 periods = [
783     ('1973-1980', '1973-01-01', '1980-12-01'),
784     ('1981-1990', '1981-01-01', '1990-12-01'),
785     ('1991-2000', '1991-01-01', '2000-12-01'),
786     ('2001-2006', '2001-01-01', '2006-04-01')
787 ]

```

```

788 fig, axes = plt.subplots(2, 2, figsize=(15, 12))
789 fig.suptitle('Evolution of Correlations Over Time', fontsize=16)
790 axes = axes.flatten()
791
792 for idx, (period_name, start, end) in enumerate(periods):
793     if idx < len(axes):
794         period_data = df.loc[start:end]
795         corr_matrix = period_data.corr()
796
797         im = axes[idx].imshow(corr_matrix.values, cmap='RdBu_r', vmin=-1,
798                             vmax=1, aspect='auto')
799         axes[idx].set_xticks(range(len(corr_matrix.columns)))
800         axes[idx].set_yticks(range(len(corr_matrix.index)))
801         axes[idx].set_xticklabels(corr_matrix.columns, rotation=45)
802         axes[idx].set_yticklabels(corr_matrix.index)
803         axes[idx].set_title(f'Correlation Matrix: {period_name}')
804
805         for i in range(len(corr_matrix.index)):
806             for j in range(len(corr_matrix.columns)):
807                 axes[idx].text(j, i, f'{corr_matrix.iloc[i, j]:.2f}',
808                                ha='center', va='center', fontweight='bold')
809
810 plt.tight_layout()
811 plt.show()

```