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

```
import sys
import os
from pathlib import Path
import warnings
warnings.filterwarnings("ignore")

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats, integrate
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import adfuller
from statsmodels.stats.diagnostic import het_arch, acorr_ljungbox

try:
```

```python
    from arch import arch_model
except Exception as e:
    raise ImportError("arch package not installed. Run: pip install arch") from e

from scipy.stats import norm, t, skewnorm, gennorm, laplace, genextreme

DEFAULT_FILE = r"C:\Users\marco\OneDrive\Desktop\Empirical exam econometrics\Slanzi_Marco.xlsx"
date_col = "obs"
log_price_col = "lp"
is_log = True
freq_label = "D"
rolling_window = 21
ewma_span = 21
acf_lags = 50
forecast_horizon = 10
vaR_alpha = 0.1

def ensure_datetime_index(df):
    global date_col
    if date_col is None:
        return df
    if date_col in df.columns:
        df[date_col] = pd.to_datetime(df[date_col])
        df = df.sort_values(date_col).set_index(date_col)
        return df
    else:
        if isinstance(df.index, pd.DatetimeIndex):
            return df
        return df

def compute_aic_bic(ll, k, n):
    aic = 2*k - 2*ll
    bic = np.log(n)*k - 2*ll
    return aic, bic

def conditional_ES_std(dist_obj, params, alpha):
    q = dist_obj.ppf(alpha, *params)
    integrand = lambda z: z * dist_obj.pdf(z, *params)
    num, _ = integrate.quad(integrand, -np.inf, q, limit=200)
    return num / alpha



infile = DEFAULT_FILE
if not os.path.exists(infile):
    raise FileNotFoundError(f"Excel file not found: {infile}.")

df = pd.read_excel(infile, sheet_name="Foglio1", engine="openpyxl")

df = ensure_datetime_index(df)

if log_price_col not in df.columns:
    raise ValueError(f"Expected column '{log_price_col}' not found in sheet Foglio1")

data = df.copy()
```

```python

if is_log:
    L = data[log_price_col].astype(float)
else:
    price = data[log_price_col].astype(float)
    L = np.log(price)

data['r'] = L.diff()
data = data.dropna().copy()

rets = data['r']
rets.name = "r"
logp = L.loc[rets.index]

print("\nReturmmary:")
print(rets.describe())


plt.style.use('default')

fig, ax = plt.subplots(3,1, figsize=(11,9), sharex=True)
ax[0].plot(rets.index, rets); ax[0].set_title("Log-Returns (r_t)")
ax[1].plot(rets.index, rets**2); ax[1].set_title("Squared Log-Returns (r_t^2)
        volatility clustering")
ax[2].plot(logp.index, logp); ax[2].set_title("Log-Price (L_t)")
plt.tight_layout(); plt.show()

fig, ax = plt.subplots(1,2, figsize=(12,4))
ax[0].hist(rets, bins=80); ax[0].set_title("Histogram of returns")
stats.probplot(rets, dist="t", sparams=(5,), plot=ax[1]);
    ax[1].set_title("QQ-plot vs student-t(df=5) reference")
plt.tight_layout(); plt.show()

adf_stat, adf_p, *_ = adfuller(rets)
print(f"\nADF test on returns: statistic={adf_stat:.6g}, p-value={adf_p:.6g}
    (p<0.05 -> stationary)")

fig = plt.figure(figsize=(10,5)); plot_acf(rets, lags=acf_lags);
    plt.title("ACF of returns"); plt.tight_layout(); plt.show()
fig = plt.figure(figsize=(10,5)); plot_acf(rets**2, lags=acf_lags);
    plt.title("ACF of squared returns"); plt.tight_layout(); plt.show()
fig = plt.figure(figsize=(10,5)); plot_pacf(rets, lags=acf_lags,
    method='ywm'); plt.title("PACF of returns"); plt.tight_layout();
    plt.show()

arch_stat, arch_pvalue, _, _ = het_arch(rets, nlags=12)
print(f"\nEngle's ARCH LM test (12 lags): statistic={arch_stat:.6g},
    p-value={arch_pvalue:.6g}")
if arch_pvalue < 0.05:
    print(" -> Reject H0: ARCH effects detected.")
else:
    print(" -> Fail to reject H0: no strong evidence of ARCH effects at 5%
        level.")

rolling_vol = rets.rolling(window=rolling_window).std()
ewma_vol = rets.ewm(span=ewma_span).std()

fig, ax = plt.subplots(figsize=(10,4))
```

```
ax.plot(rolling_vol, label=f"Rolling {rolling_window}-obs std")
ax.plot(ewma_vol, label=f"EWMA span={ewma_span}")
ax.set_title("Nonparametric volatility estimates")
ax.legend(); plt.tight_layout(); plt.show()

rets_pct = (rets * 100.0).dropna()

models = {}

am_g11 = arch_model(rets_pct, vol='GARCH', p=1, q=1, mean='Constant',
    dist='t')
res_g11 = am_g11.fit(disp='off')
models['GARCH11'] = res_g11
print("\nGARCH(1,1) fitted.")
print(res_g11.summary())

am_eg = arch_model(rets_pct, vol='EGARCH', p=1, o=1, q=1, mean='Constant',
    dist='t')
res_eg = am_eg.fit(disp='off')
models['EGARCH'] = res_eg
print("\nEGARCH(1,1) fitted.")
print(res_eg.summary())

am_gjr = arch_model(rets_pct, vol='GARCH', p=1, o=1, q=1, mean='Constant',
    dist='t')
res_gjr = am_gjr.fit(disp='off')
models['GJR'] = res_gjr
print("\nGJR-GARCH(1,1) fitted.")
print(res_gjr.summary())

am_tgjr = arch_model(rets_pct, vol='GARCH', p=1, o=1, q=1, mean='Constant',
    dist='t')
res_tgjr = am_tgjr.fit(disp='off')
models['t-GJR'] = res_tgjr
print("\nt-GJR-GARCH(1,1) fitted.")
print(res_tgjr.summary())

am_aparch = arch_model(rets_pct, vol='APARCH', p=1, o=1, q=1,
    mean='Constant', dist='t')
res_aparch = am_aparch.fit(disp='off')
models['APARCH'] = res_aparch
print("\nAPARCH(1,1) fitted.")
print(res_aparch.summary())

print("\nModel comparison (AIC, BIC):")
for name, r in models.items():
    print(f"{name:8s} AIC={r.aic:.4f}, BIC={r.bic:.4f}")

best_name = min(models.keys(), key=lambda k: models[k].aic)
best_model = models[best_name]
print(f"\nSelected best model by AIC (among baseline family): {best_name}")

print("\n" + "="*70)
print(f"RESIDUAL DIAGNOSTICS FOR BEST MODEL: {best_name}")
print("="*70)

std_resid = pd.Series(best_model.std_resid,
    index=best_model.model.y.index).dropna()
```

```python

fig, ax = plt.subplots(1, 3, figsize=(15, 4))
ax[0].plot(std_resid, lw=0.8)
ax[0].axhline(0, color='r', linestyle='--', alpha=0.6)
ax[0].set_title("Standardized Residuals")
ax[0].grid(alpha=0.3)

ax[1].hist(std_resid, bins=40, density=True, color='skyblue', edgecolor='k',
    alpha=0.7)
x = np.linspace(std_resid.min(), std_resid.max(), 200)
ax[1].plot(x, stats.norm.pdf(x, 0, 1), 'r-', lw=2, label='N(0,1)')
ax[1].set_title("Histogram vs Normal PDF")
ax[1].legend()
ax[1].grid(alpha=0.3)

stats.probplot(std_resid, dist="norm", plot=ax[2])
ax[2].set_title("QQ-Plot vs Normal")

plt.tight_layout()
plt.show()

fig, ax = plt.subplots(1, 2, figsize=(12, 3))
plot_acf(std_resid, lags=40, ax=ax[0], alpha=0.05)
ax[0].set_title("ACF of Std Residuals")
plot_acf(std_resid**2, lags=40, ax=ax[1], alpha=0.05)
ax[1].set_title("ACF of Squared Std Residuals")
plt.tight_layout()
plt.show()

jb_stat, jb_p = stats.jarque_bera(std_resid)
lb5_p = acorr_ljungbox(std_resid, lags=[5],
    return_df=True)['lb_pvalue'].iloc[0]
lb10_p = acorr_ljungbox(std_resid, lags=[10],
    return_df=True)['lb_pvalue'].iloc[0]
lb5_sq_p = acorr_ljungbox(std_resid**2, lags=[5],
    return_df=True)['lb_pvalue'].iloc[0]
lb10_sq_p = acorr_ljungbox(std_resid**2, lags=[10],
    return_df=True)['lb_pvalue'].iloc[0]
arch_stat, arch_p, _, _ = het_arch(std_resid, nlags=12)

garch_p_models = {}
for p in [1,2,3]:
    label = f"GARCH({p},1)"
    print(f"\nFitting {label} ...")
    am = arch_model(rets_pct, vol='GARCH', p=p, q=1, mean='Constant',
        dist='t')
    res = am.fit(disp='off')
    garch_p_models[label] = res
    print(res.summary())

print("\nGARCH(p,1) comparison (AIC,BIC):")
for name, r in garch_p_models.items():
    print(f"{name:10s} AIC={r.aic:.4f}, BIC={r.bic:.4f}")

def lr_test(res_restricted, res_full):
    llr = 2.0 * (float(res_full.loglikelihood) -
        float(res_restricted.loglikelihood))
    df = len(res_full.params) - len(res_restricted.params)
```

```python
    pval = stats.chi2.sf(llr, df)
    return llr, df, pval

if 'GARCH(2,1)' in garch_p_models:
    lr21 = lr_test(garch_p_models['GARCH(1,1)'], garch_p_models['GARCH(2,1)'])
    print(f"\nLR test GARCH(2,1) vs GARCH(1,1): LR={lr21[0]:.4f}, df={lr21[1]}, p={lr21[2]:.6g}")
if 'GARCH(3,1)' in garch_p_models:
    lr32 = lr_test(garch_p_models['GARCH(2,1)'], garch_p_models['GARCH(3,1)'])
    print(f"LR test GARCH(3,1) vs GARCH(2,1): LR={lr32[0]:.4f}, df={lr32[1]}, p={lr32[2]:.6g}")

all_models = {**models, **garch_p_models}
best_name_all = min(all_models.keys(), key=lambda k: all_models[k].aic)
best_model_all = all_models[best_name_all]
print(f"\nSelected best model by AIC across all fitted models: {best_name_all}")

res = best_model_all
std_resid = pd.Series(res.std_resid, index=res.model.y.index).dropna()

fig, ax = plt.subplots(1, 3, figsize=(15, 4))
ax[0].plot(std_resid, lw=0.8)
ax[0].axhline(0, color='r', linestyle='--', alpha=0.6)
ax[0].set_title("Standardized Residuals")
ax[1].hist(std_resid, bins=40, density=True, color='skyblue', edgecolor='k', alpha=0.7)
x = np.linspace(std_resid.min(), std_resid.max(), 200)
ax[1].plot(x, stats.norm.pdf(x, 0, 1), 'r-', lw=2, label='N(0,1)')
ax[1].legend(); ax[2].grid(alpha=0.3)
stats.probplot(std_resid, dist="norm", plot=ax[2])
plt.tight_layout(); plt.show()

std_resid_best = pd.Series(best_model_all.std_resid, index=rets.index[-len(best_model_all.std_resid):]).dropna()

params = best_model_all.params
omega = float(params.get('omega', 0))
alpha1 = float(params.get('alpha[1]', 0))
beta1  = float(params.get('beta[1]', 0))
gamma1 = float(params.get('gamma[1]', 0))
mu_hat = float(params.get('mu', 0)) / 100.0

sigma_t = float(best_model_all.conditional_volatility.iloc[-1])

H = 20

irf_pos = np.zeros(H)
eps_t = sigma_t
irf_pos[0] = np.sqrt(omega + alpha1*eps_t**2 + gamma1*eps_t**2*0 + beta1*sigma_t**2)
for t in range(1, H):
    irf_pos[t] = np.sqrt(omega + beta1*irf_pos[t-1]**2)

irf_neg = np.zeros(H)
eps_t = -sigma_t
indicator = 1.0
```

```
irf_neg[0] = np.sqrt(omega + alpha1*eps_t**2 + gamma1*eps_t**2*indicator + beta1*sigma_t**2)
for t in range(1, H):
    irf_neg[t] = np.sqrt(omega + beta1*irf_neg[t-1]**2)

plt.figure(figsize=(10,5))
plt.plot(range(1,H+1), irf_pos, label='Positive Shock', marker='o')
plt.plot(range(1,H+1), irf_neg, label='Negative Shock', marker='x')
plt.title('Impulse Response Function of Conditional Volatility')
plt.xlabel('Periods ahead')
plt.ylabel('Conditional sigma')
plt.legend()
plt.grid(True)
plt.show()

print("\nFitting candidate distributions to standardized residuals...")
x = std_resid_best.values
n = len(x)
candidate_dists = {
    'normal': norm,
    't': t,
    'skewnorm': skewnorm,
    'gennorm': gennorm,
    'laplace': laplace,
    'genextreme': genextreme
}

dist_results = []
for name, dist in candidate_dists.items():
    try:
        params = dist.fit(x)
        try:
            ll = np.sum(dist.logpdf(x, *params))
        except Exception:
            ll = np.sum(np.log(dist.pdf(x, *params)))
        k = len(params)
        aic, bic = compute_aic_bic(ll, k, n)
        cdf_fun = lambda v: dist.cdf(v, *params)
        ks_stat, ks_p = stats.kstest(x, cdf_fun)
        dist_results.append({
            'dist': name, 'params': params, 'loglik': ll, 'k': k, 'AIC': aic, 'BIC': bic,
            'KS_stat': ks_stat, 'KS_p': ks_p
        })
        print(f"Fitted {name}: AIC={aic:.2f}, BIC={bic:.2f}, KS_p={ks_p:.4g}")
    except Exception as e:
        print(f"Failed to fit {name}: {e}")

dist_df = pd.DataFrame(dist_results).sort_values('AIC').reset_index(drop=True)
print("\nTop candidate distributions by AIC:")
print(dist_df[['dist','AIC','BIC','KS_p']].head())

topk = min(3, len(dist_df))
fig, ax = plt.subplots(figsize=(10,5))
ax.hist(x, bins=80, density=True, alpha=0.4, label='std_resid hist')
xs = np.linspace(x.min()*1.05, x.max()*1.05, 1000)
for i in range(topk):
    row = dist_df.loc[i]
```

```
    dd = candidate_dists[row['dist']]
    params = row['params']
    ax.plot(xs, dd.pdf(xs, *params), label=f"{row['dist']} (AIC={row['AIC']:.1f})")
ax.legend(); ax.set_title("Standardized residuals: histogram + fitted PDFs")
plt.tight_layout(); plt.show()

h = forecast_horizon
fcast = best_model_all.forecast(horizon=h, reindex=False)
last_var = fcast.variance.iloc[-1]
pred_sigma = np.sqrt(last_var) / 100.0

print(f"\n{best_model_all.model.name} - {h}-step ahead conditional std (returns units):")
for i in range(len(pred_sigma)):
    print(f" h={i+1}: sigma = {pred_sigma.iloc[i]:.6f}")

cond_sigma_series = pd.Series(best_model_all.conditional_volatility, index=rets.index) / 100.0
fig, ax = plt.subplots(figsize=(10,4))
ax.plot(cond_sigma_series, label=f"Conditional volatility ({best_name_all})")
ax.set_title("Estimated conditional volatility (sigma_t)")
ax.legend(); plt.tight_layout(); plt.show()

if len(dist_df) > 0:
    chosen = dist_df.loc[0]
    chosen_name = chosen['dist']
    chosen_dist = candidate_dists[chosen_name]
    chosen_params = chosen['params']
else:
    chosen_name = 'normal'
    chosen_dist = norm
    chosen_params = norm.fit(x)

print(f"\nUsing residual distribution for parametric VaR/ES: {chosen_name}")

q_std = chosen_dist.ppf(vaR_alpha, *chosen_params)
ES_std = conditional_ES_std(chosen_dist, chosen_params, vaR_alpha)
print(f"Std quantile (alpha={vaR_alpha}): {q_std:.6g}, Std conditional ES: {ES_std:.6g}")

mu_hat = float(best_model_all.params.get('mu', 0.0)) / 100.0
VaR_param = mu_hat + cond_sigma_series * q_std
ES_param = mu_hat + cond_sigma_series * ES_std

q_emp = np.quantile(x, vaR_alpha)
VaR_emp = mu_hat + cond_sigma_series * q_emp

viol = (rets < VaR_emp)
if viol.sum() > 0:
    ES_empirical_global = rets[viol].mean()
else:
    ES_empirical_global = np.nan

print(f"\n1-day parametric VaR (last): {VaR_param.iloc[-1]:.6f}")
print(f"1-day parametric ES (last): {ES_param.iloc[-1]:.6f}")
print(f"1-day empirical VaR (last): {VaR_emp.iloc[-1]:.6f}")
```

```python
print(f"Empirical global ES (average realized losses below empirical VaR):
    {ES_empirical_global:.6f}")

window = min(300, len(rets))
fig, ax = plt.subplots(figsize=(12,5))
ax.plot(rets[-window:], label='Returns')
ax.plot(VaR_param[-window:], label='Parametric VaR')
ax.plot(ES_param[-window:], label='Parametric ES')
ax.set_title("Returns vs Parametric VaR & ES (recent)")
ax.legend(); plt.tight_layout(); plt.show()



fig, axes = plt.subplots(1, 2, figsize=(12, 5))

window = min(300, len(rets))
axes[0].plot(rets[-window:].values, label='Returns', color='black',
    alpha=0.7, linewidth=1)
axes[0].plot(VaR_param[-window:].values, label=f'Parametric VaR
    ({VaR_param.iloc[-1]:.4f})',
               color='red', linestyle='--', linewidth=1.5)
axes[0].plot(VaR_emp[-window:].values, label=f'Empirical VaR
    ({VaR_emp.iloc[-1]:.4f})',
               color='blue', linestyle='--', linewidth=1.5)

violations_param = rets[-window:] < VaR_param[-window:]
violations_emp = rets[-window:] < VaR_emp[-window:]

for i in range(window):
    if violations_param.iloc[i]:
        axes[0].axvspan(i-0.5, i+0.5, alpha=0.3, color='red')
    if violations_emp.iloc[i]:
        axes[0].axvspan(i-0.5, i+0.5, alpha=0.2, color='blue')

axes[0].set_xlabel('Time')
axes[0].set_ylabel('Returns')
axes[0].set_title('RETURNS vs VaR METHODS (Recent Period)')
axes[0].legend(fontsize=9)
axes[0].grid(True, alpha=0.3)

x_range = np.linspace(x.min(), x.max(), 1000)
parametric_pdf = chosen_dist.pdf(x_range, *chosen_params) *
    cond_sigma_series.iloc[-1]
axes[1].hist(x, bins=100, density=True, alpha=0.6, color='lightgray',
    label='Std Residuals')
axes[1].plot(x_range, parametric_pdf, 'r-', linewidth=2, label=f'Parametric
    ({chosen_name})')

axes[1].axvline(VaR_param.iloc[-1], color='red', linestyle='--', linewidth=2,
    label=f'Parametric VaR')
axes[1].axvline(VaR_emp.iloc[-1], color='blue', linestyle='--', linewidth=2,
    label=f'Empirical VaR')

axes[1].set_xlabel('Returns')
axes[1].set_ylabel('Density')
axes[1].set_title('DISTRIBUTION COMPARISON')
axes[1].legend(fontsize=9)
axes[1].grid(True, alpha=0.3)
```

```
plt.tight_layout()
plt.show()



\section{Code for the Multivariate Analysis}






\section{Code for the VAR model}
\end{document} make it suitable for latex
ChatGPT said:
Thought for 15s
You said:
\documentclass{article}
\usepackage{graphicx} % Required for inserting images

\title{Codes for the models}
\subsection{Econometrics of Financial Markets}
\author{Slanzi Marco}
\date{October 2025}

\begin{document}

\maketitle

\section{Code for the Univariate Analysis}
import sys
import os
from pathlib import Path
import warnings
warnings.filterwarnings("ignore")

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats, integrate
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import adfuller
from statsmodels.stats.diagnostic import het_arch, acorr_ljungbox

try:
    from arch import arch_model
except Exception as e:
    raise ImportError("arch package not installed. Run: pip install arch")
        from e

from scipy.stats import norm, t, skewnorm, gennorm, laplace, genextreme

DEFAULT_FILE = r"C:\Users\marco\OneDrive\Desktop\Empirical exam
    econometrics\Slanzi_Marco.xlsx"
date_col = "obs"
log_price_col = "lp"
```

```
is_log = True
freq_label = "D"
rolling_window = 21
ewma_span = 21
acf_lags = 50
forecast_horizon = 10
vaR_alpha = 0.1

def ensure_datetime_index(df):
    global date_col
    if date_col is None:
        return df
    if date_col in df.columns:
        df[date_col] = pd.to_datetime(df[date_col])
        df = df.sort_values(date_col).set_index(date_col)
        return df
    else:
        if isinstance(df.index, pd.DatetimeIndex):
            return df
        return df

def compute_aic_bic(ll, k, n):
    aic = 2*k - 2*ll
    bic = np.log(n)*k - 2*ll
    return aic, bic

def conditional_ES_std(dist_obj, params, alpha):
    q = dist_obj.ppf(alpha, *params)
    integrand = lambda z: z * dist_obj.pdf(z, *params)
    num, _ = integrate.quad(integrand, -np.inf, q, limit=200)
    return num / alpha

print("Setup done. Ready to run blocks.")

infile = DEFAULT_FILE
if not os.path.exists(infile):
    raise FileNotFoundError(f"Excel file not found: {infile}.")

print(f"Loading file: {infile} (sheet Foglio1)")
df = pd.read_excel(infile, sheet_name="Foglio1", engine="openpyxl")
print("Columns:", df.columns.tolist())

df = ensure_datetime_index(df)

if log_price_col not in df.columns:
    raise ValueError(f"Expected column '{log_price_col}' not found in sheet
        Foglio1")

data = df.copy()

if is_log:
    L = data[log_price_col].astype(float)
else:
    price = data[log_price_col].astype(float)
    L = np.log(price)

data['r'] = L.diff()
data = data.dropna().copy()
```

```
rets = data['r']
rets.name = "r"
logp = L.loc[rets.index]

print(f"\nObservations after diff: {len(data)}")
print("\nReturmmary:")
print(rets.describe())


plt.style.use('default')

fig, ax = plt.subplots(3,1, figsize=(11,9), sharex=True)
ax[0].plot(rets.index, rets); ax[0].set_title("Log-Returns (r_t)")
ax[1].plot(rets.index, rets**2); ax[1].set_title("Squared Log-Returns (r_t^2) volatility clustering")
ax[2].plot(logp.index, logp); ax[2].set_title("Log-Price (L_t)")
plt.tight_layout(); plt.show()

fig, ax = plt.subplots(1,2, figsize=(12,4))
ax[0].hist(rets, bins=80); ax[0].set_title("Histogram of returns")
stats.probplot(rets, dist="t", sparams=(5,), plot=ax[1]); ax[1].set_title("QQ-plot vs student-t(df=5) reference")
plt.tight_layout(); plt.show()

adf_stat, adf_p, *_ = adfuller(rets)
print(f"\nADF test on returns: statistic={adf_stat:.6g}, p-value={adf_p:.6g} (p<0.05 -> stationary)")

fig = plt.figure(figsize=(10,5)); plot_acf(rets, lags=acf_lags); plt.title("ACF of returns"); plt.tight_layout(); plt.show()
fig = plt.figure(figsize=(10,5)); plot_acf(rets**2, lags=acf_lags); plt.title("ACF of squared returns"); plt.tight_layout(); plt.show()
fig = plt.figure(figsize=(10,5)); plot_pacf(rets, lags=acf_lags, method='ywm'); plt.title("PACF of returns"); plt.tight_layout(); plt.show()

arch_stat, arch_pvalue, _, _ = het_arch(rets, nlags=12)
print(f"\nEngle's ARCH LM test (12 lags): statistic={arch_stat:.6g}, p-value={arch_pvalue:.6g}")
if arch_pvalue < 0.05:
    print(" -> Reject H0: ARCH effects detected.")
else:
    print(" -> Fail to reject H0: no strong evidence of ARCH effects at 5% level.")

rolling_vol = rets.rolling(window=rolling_window).std()
ewma_vol = rets.ewm(span=ewma_span).std()

fig, ax = plt.subplots(figsize=(10,4))
ax.plot(rolling_vol, label=f"Rolling {rolling_window}-obs std")
ax.plot(ewma_vol, label=f"EWMA span={ewma_span}")
ax.set_title("Nonparametric volatility estimates")
ax.legend(); plt.tight_layout(); plt.show()

rets_pct = (rets * 100.0).dropna()

models = {}
```

```
am_g11 = arch_model(rets_pct, vol='GARCH', p=1, q=1, mean='Constant',
    dist='t')
res_g11 = am_g11.fit(disp='off')
models['GARCH11'] = res_g11
print("\nGARCH(1,1) fitted.")
print(res_g11.summary())

am_eg = arch_model(rets_pct, vol='EGARCH', p=1, o=1, q=1, mean='Constant',
    dist='t')
res_eg = am_eg.fit(disp='off')
models['EGARCH'] = res_eg
print("\nEGARCH(1,1) fitted.")
print(res_eg.summary())

am_gjr = arch_model(rets_pct, vol='GARCH', p=1, o=1, q=1, mean='Constant',
    dist='t')
res_gjr = am_gjr.fit(disp='off')
models['GJR'] = res_gjr
print("\nGJR-GARCH(1,1) fitted.")
print(res_gjr.summary())

am_tgjr = arch_model(rets_pct, vol='GARCH', p=1, o=1, q=1, mean='Constant',
    dist='t')
res_tgjr = am_tgjr.fit(disp='off')
models['t-GJR'] = res_tgjr
print("\nt-GJR-GARCH(1,1) fitted.")
print(res_tgjr.summary())

am_aparch = arch_model(rets_pct, vol='APARCH', p=1, o=1, q=1,
    mean='Constant', dist='t')
res_aparch = am_aparch.fit(disp='off')
models['APARCH'] = res_aparch
print("\nAPARCH(1,1) fitted.")
print(res_aparch.summary())

print("\nModel comparison (AIC, BIC):")
for name, r in models.items():
    print(f"{name:8s} AIC={r.aic:.4f}, BIC={r.bic:.4f}")

best_name = min(models.keys(), key=lambda k: models[k].aic)
best_model = models[best_name]
print(f"\nSelected best model by AIC (among baseline family): {best_name}")

print("\n" + "="*70)
print(f"RESIDUAL DIAGNOSTICS FOR BEST MODEL: {best_name}")
print("="*70)

std_resid = pd.Series(best_model.std_resid,
    index=best_model.model.y.index).dropna()

fig, ax = plt.subplots(1, 3, figsize=(15, 4))
ax[0].plot(std_resid, lw=0.8)
ax[0].axhline(0, color='r', linestyle='--', alpha=0.6)
ax[0].set_title("Standardized Residuals")
ax[0].grid(alpha=0.3)
```

```python
ax[1].hist(std_resid, bins=40, density=True, color='skyblue', edgecolor='k',
    alpha=0.7)
x = np.linspace(std_resid.min(), std_resid.max(), 200)
ax[1].plot(x, stats.norm.pdf(x, 0, 1), 'r-', lw=2, label='N(0,1)')
ax[1].set_title("Histogram vs Normal PDF")
ax[1].legend()
ax[1].grid(alpha=0.3)

stats.probplot(std_resid, dist="norm", plot=ax[2])
ax[2].set_title("QQ-Plot vs Normal")

plt.tight_layout()
plt.show()

fig, ax = plt.subplots(1, 2, figsize=(12, 3))
plot_acf(std_resid, lags=40, ax=ax[0], alpha=0.05)
ax[0].set_title("ACF of Std Residuals")
plot_acf(std_resid**2, lags=40, ax=ax[1], alpha=0.05)
ax[1].set_title("ACF of Squared Std Residuals")
plt.tight_layout()
plt.show()

jb_stat, jb_p = stats.jarque_bera(std_resid)
lb5_p = acorr_ljungbox(std_resid, lags=[5],
    return_df=True)['lb_pvalue'].iloc[0]
lb10_p = acorr_ljungbox(std_resid, lags=[10],
    return_df=True)['lb_pvalue'].iloc[0]
lb5_sq_p = acorr_ljungbox(std_resid**2, lags=[5],
    return_df=True)['lb_pvalue'].iloc[0]
lb10_sq_p = acorr_ljungbox(std_resid**2, lags=[10],
    return_df=True)['lb_pvalue'].iloc[0]
arch_stat, arch_p, _, _ = het_arch(std_resid, nlags=12)
garch_p_models = {}
for p in [1,2,3]:
    label = f"GARCH({p},1)"
    print(f"\nFitting {label} ...")
    am = arch_model(rets_pct, vol='GARCH', p=p, q=1, mean='Constant',
        dist='t')
    res = am.fit(disp='off')
    garch_p_models[label] = res
    print(res.summary())

print("\nGARCH(p,1) comparison (AIC,BIC):")
for name, r in garch_p_models.items():
    print(f"{name:10s} AIC={r.aic:.4f}, BIC={r.bic:.4f}")

def lr_test(res_restricted, res_full):
    llr = 2.0 * (float(res_full.loglikelihood) -
        float(res_restricted.loglikelihood))
    df = len(res_full.params) - len(res_restricted.params)
    pval = stats.chi2.sf(llr, df)
    return llr, df, pval

if 'GARCH(2,1)' in garch_p_models:
    lr21 = lr_test(garch_p_models['GARCH(1,1)'], garch_p_models['GARCH(2,1)'])
    print(f"\nLR test GARCH(2,1) vs GARCH(1,1): LR={lr21[0]:.4f},
        df={lr21[1]}, p={lr21[2]:.6g}")
if 'GARCH(3,1)' in garch_p_models:
```

```python
    lr32 = lr_test(garch_p_models['GARCH(2,1)'], garch_p_models['GARCH(3,1)'])
    print(f"LR test GARCH(3,1) vs GARCH(2,1): LR={lr32[0]:.4f}, df={lr32[1]},
        p={lr32[2]:.6g}")

all_models = {**models, **garch_p_models}
best_name_all = min(all_models.keys(), key=lambda k: all_models[k].aic)
best_model_all = all_models[best_name_all]
print(f"\nSelected best model by AIC across all fitted models:
    {best_name_all}")

res = best_model_all
std_resid = pd.Series(res.std_resid, index=res.model.y.index).dropna()

fig, ax = plt.subplots(1, 3, figsize=(15, 4))
ax[0].plot(std_resid, lw=0.8)
ax[0].axhline(0, color='r', linestyle='--', alpha=0.6)
ax[0].set_title("Standardized Residuals")
ax[1].hist(std_resid, bins=40, density=True, color='skyblue', edgecolor='k',
    alpha=0.7)
x = np.linspace(std_resid.min(), std_resid.max(), 200)
ax[1].plot(x, stats.norm.pdf(x, 0, 1), 'r-', lw=2, label='N(0,1)')
ax[1].legend(); ax[2].grid(alpha=0.3)
stats.probplot(std_resid, dist="norm", plot=ax[2])
plt.tight_layout(); plt.show()

std_resid_best = pd.Series(best_model_all.std_resid,
    index=rets.index[-len(best_model_all.std_resid):]).dropna()

params = best_model_all.params
omega = float(params.get('omega', 0))
alpha1 = float(params.get('alpha[1]', 0))
beta1  = float(params.get('beta[1]', 0))
gamma1 = float(params.get('gamma[1]', 0))
mu_hat = float(params.get('mu', 0)) / 100.0

sigma_t = float(best_model_all.conditional_volatility.iloc[-1])

H = 20

irf_pos = np.zeros(H)
eps_t = sigma_t
irf_pos[0] = np.sqrt(omega + alpha1*eps_t**2 + gamma1*eps_t**2*0 +
    beta1*sigma_t**2)
for t in range(1, H):
    irf_pos[t] = np.sqrt(omega + beta1*irf_pos[t-1]**2)

irf_neg = np.zeros(H)
eps_t = -sigma_t
indicator = 1.0
irf_neg[0] = np.sqrt(omega + alpha1*eps_t**2 + gamma1*eps_t**2*indicator +
    beta1*sigma_t**2)
for t in range(1, H):
    irf_neg[t] = np.sqrt(omega + beta1*irf_neg[t-1]**2)

plt.figure(figsize=(10,5))
plt.plot(range(1,H+1), irf_pos, label='Positive Shock', marker='o')
plt.plot(range(1,H+1), irf_neg, label='Negative Shock', marker='x')
plt.title('Impulse Response Function of Conditional Volatility')
```

```
plt.xlabel('Periods ahead')
plt.ylabel('Conditional sigma')
plt.legend()
plt.grid(True)
plt.show()

print("\nFitting candidate distributions to standardized residuals...")
x = std_resid_best.values
n = len(x)
candidate_dists = {
    'normal': norm,
    't': t,
    'skewnorm': skewnorm,
    'gennorm': gennorm,
    'laplace': laplace,
    'genextreme': genextreme
}

dist_results = []
for name, dist in candidate_dists.items():
    try:
        params = dist.fit(x)
        try:
            ll = np.sum(dist.logpdf(x, *params))
        except Exception:
            ll = np.sum(np.log(dist.pdf(x, *params)))
        k = len(params)
        aic, bic = compute_aic_bic(ll, k, n)
        cdf_fun = lambda v: dist.cdf(v, *params)
        ks_stat, ks_p = stats.kstest(x, cdf_fun)
        dist_results.append({
            'dist': name, 'params': params, 'loglik': ll, 'k': k, 'AIC': aic,
                'BIC': bic,
            'KS_stat': ks_stat, 'KS_p': ks_p
        })
        print(f"Fitted {name}: AIC={aic:.2f}, BIC={bic:.2f}, KS_p={ks_p:.4g}")
    except Exception as e:
        print(f"Failed to fit {name}: {e}")

dist_df = pd.DataFrame(dist_results).sort_values('AIC').reset_index(drop=True)


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

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

```python
pred_sigma = np.sqrt(last_var) / 100.0

print(f"\n{best_model_all.model.name} - {h}-step ahead conditional std
    (returns units):")
for i in range(len(pred_sigma)):
    print(f" h={i+1}: sigma = {pred_sigma.iloc[i]:.6f}")

cond_sigma_series = pd.Series(best_model_all.conditional_volatility,
    index=rets.index) / 100.0
fig, ax = plt.subplots(figsize=(10,4))
ax.plot(cond_sigma_series, label=f"Conditional volatility ({best_name_all})")
ax.set_title("Estimated conditional volatility (sigma_t)")
ax.legend(); plt.tight_layout(); plt.show()

if len(dist_df) > 0:
    chosen = dist_df.loc[0]
    chosen_name = chosen['dist']
    chosen_dist = candidate_dists[chosen_name]
    chosen_params = chosen['params']
else:
    chosen_name = 'normal'
    chosen_dist = norm
    chosen_params = norm.fit(x)

print(f"\nUsing residual distribution for parametric VaR/ES: {chosen_name}")

q_std = chosen_dist.ppf(vaR_alpha, *chosen_params)
ES_std = conditional_ES_std(chosen_dist, chosen_params, vaR_alpha)
print(f"Std quantile (alpha={vaR_alpha}): {q_std:.6g}, Std conditional ES:
    {ES_std:.6g}")

mu_hat = float(best_model_all.params.get('mu', 0.0)) / 100.0
VaR_param = mu_hat + cond_sigma_series * q_std
ES_param = mu_hat + cond_sigma_series * ES_std

q_emp = np.quantile(x, vaR_alpha)
VaR_emp = mu_hat + cond_sigma_series * q_emp

viol = (rets < VaR_emp)
if viol.sum() > 0:
    ES_empirical_global = rets[viol].mean()
else:
    ES_empirical_global = np.nan


window = min(300, len(rets))
fig, ax = plt.subplots(figsize=(12,5))
ax.plot(rets[-window:], label='Returns')
ax.plot(VaR_param[-window:], label='Parametric VaR')
ax.plot(ES_param[-window:], label='Parametric ES')
ax.set_title("Returns vs Parametric VaR & ES (recent)")
ax.legend(); plt.tight_layout(); plt.show()

print("=" * 80)
print("FINAL VaR & ES COMPARISON: PARAMETRIC vs EMPIRICAL")
print("=" * 80)

```

```python
print(f"{'Method':<20} {'VaR ('+str(vaR_alpha*100)+'%)':<20} {'ES
    ('+str(vaR_alpha*100)+'%)':<20}")
print("-" * 70)
print(f"{'Parametric':<20} {VaR_param.iloc[-1]:.6f} {ES_param.iloc[-1]:.6f}")
print(f"{'Empirical':<20} {VaR_emp.iloc[-1]:.6f} {ES_empirical_global:.6f}")

print(f"\nReality Check:")
print(f"Actual returns 10th percentile: {np.percentile(rets, 10):.6f}")
print(f"Worst historical return: {rets.min():.6f}")

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

window = min(300, len(rets))
axes[0].plot(rets[-window:].values, label='Returns', color='black',
    alpha=0.7, linewidth=1)
axes[0].plot(VaR_param[-window:].values, label=f'Parametric VaR
    ({VaR_param.iloc[-1]:.4f})',
               color='red', linestyle='--', linewidth=1.5)
axes[0].plot(VaR_emp[-window:].values, label=f'Empirical VaR
    ({VaR_emp.iloc[-1]:.4f})',
               color='blue', linestyle='--', linewidth=1.5)

violations_param = rets[-window:] < VaR_param[-window:]
violations_emp = rets[-window:] < VaR_emp[-window:]

for i in range(window):
    if violations_param.iloc[i]:
        axes[0].axvspan(i-0.5, i+0.5, alpha=0.3, color='red')
    if violations_emp.iloc[i]:
        axes[0].axvspan(i-0.5, i+0.5, alpha=0.2, color='blue')

axes[0].set_xlabel('Time')
axes[0].set_ylabel('Returns')
axes[0].set_title('RETURNS vs VaR METHODS (Recent Period)')
axes[0].legend(fontsize=9)
axes[0].grid(True, alpha=0.3)

x_range = np.linspace(x.min(), x.max(), 1000)
parametric_pdf = chosen_dist.pdf(x_range, *chosen_params) *
    cond_sigma_series.iloc[-1]
axes[1].hist(x, bins=100, density=True, alpha=0.6, color='lightgray',
    label='Std Residuals')
axes[1].plot(x_range, parametric_pdf, 'r-', linewidth=2, label=f'Parametric
    ({chosen_name})')

axes[1].axvline(VaR_param.iloc[-1], color='red', linestyle='--', linewidth=2,
    label=f'Parametric VaR')
axes[1].axvline(VaR_emp.iloc[-1], color='blue', linestyle='--', linewidth=2,
    label=f'Empirical VaR')

axes[1].set_xlabel('Returns')
axes[1].set_ylabel('Density')
axes[1].set_title('DISTRIBUTION COMPARISON')
axes[1].legend(fontsize=9)
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

Notes:

3 Code for the Multivariate Analysis

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy.stats as stats
import seaborn as sns
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.stats.diagnostic import het_arch, acorr_ljungbox
from scipy.optimize import minimize
from arch import arch_model
import scipy.stats as st

DEFAULT_FILE = r"C:\Users\marco\OneDrive\Desktop\Empirical exam
    econometrics\Slanzi_Marco.xlsx"

infile = DEFAULT_FILE

print(f"Loading file: {infile} (sheet Sheet2)")
df = pd.read_excel(infile, sheet_name="Foglio2", engine="openpyxl")
df.columns = ["date", "Price1", "Price2", "Price3"]

df['date'] = pd.to_datetime(df['date'], dayfirst=True)
df = df.set_index('date')

print("Columns:", df.columns.tolist())
print(df.head())

for col in ["Price1", "Price2", "Price3"]:
    df[f"log_{col}"] = np.log(df[col])
    df[f"logret_{col}"] = df[f"log_{col}"].diff()

df = df.replace([np.inf, -np.inf], np.nan).dropna()
print(f"Final data shape: {df.shape}")

logp_cols = ["log_Price1", "log_Price2", "log_Price3"]
logret_cols = ["logret_Price1", "logret_Price2", "logret_Price3"]
acf_lags = 40

plt.style.use('default')

distribution_stats = []

for i in range(3):
    series_name = logret_cols[i]
    price_name = logp_cols[i]
    rets_series = df[series_name]
    logp_series = df[price_name]

    print(f"\nAnalysis for {series_name}")
    print("-" * 40)

    print(f"Mean: {rets_series.mean():.6f}")
```

```python
    print(f"Std: {rets_series.std():.6f}")
    print(f"Min: {rets_series.min():.6f}")
    print(f"Max: {rets_series.max():.6f}")

    adf_stat, adf_p, _, _, _, _ = adfuller(rets_series)
    print(f"ADF test: statistic={adf_stat:.4f}, p-value={adf_p:.4g}")

    skew_ret = stats.skew(rets_series, bias=False)
    kurt_ret = stats.kurtosis(rets_series, fisher=False, bias=False)
    jb_stat_ret, jb_p_ret = stats.jarque_bera(rets_series)

    distribution_stats.append({
        'Series': series_name,
        'Skewness': skew_ret,
        'Kurtosis': kurt_ret,
        'JB_Statistic': jb_stat_ret,
        'JB_p_value': jb_p_ret,
        'Normality': 'Reject' if jb_p_ret < 0.05 else 'Cannot reject'
    })

    print(f"Skewness: {skew_ret:.6f}")
    print(f"Kurtosis: {kurt_ret:.6f}")
    print(f"Jarque-Bera test: stat={jb_stat_ret:.6f}, p-value={jb_p_ret:.6g}")
    if jb_p_ret < 0.05:
        print(" -> Reject normality (returns are not normal).")
    else:
        print(" -> Cannot reject normality at 5%.")

    arch_stat, arch_pvalue, _, _ = het_arch(rets_series, nlags=12)
    print(f"ARCH LM test: stat={arch_stat:.4f}, p-value={arch_pvalue:.4g}")
    if arch_pvalue < 0.05:
        print(" -> Reject H0: ARCH effects detected.")
    else:
        print(" -> Fail to reject H0: no strong ARCH evidence.")

    fig, ax = plt.subplots(3, 1, figsize=(11, 9), sharex=True)
    ax[0].plot(rets_series.index, rets_series)
    ax[0].set_title(f"{series_name}: Log-Returns (r_t)")

    ax[1].plot(rets_series.index, rets_series**2)
    ax[1].set_title("Squared Log-Returns (r_t )     volatility clustering")

    ax[2].plot(logp_series.index, logp_series)
    ax[2].set_title(f"{price_name}: Log-Price (L_t)")
    plt.tight_layout()
    plt.show()

    fig, ax = plt.subplots(1, 2, figsize=(12, 4))
    ax[0].hist(rets_series, bins=80)
    ax[0].set_title(f"Histogram of {series_name}")
    stats.probplot(rets_series, dist="t", sparams=(5,), plot=ax[1])
    ax[1].set_title("QQ-plot vs t(df=5) reference")
    plt.tight_layout()
    plt.show()

    fig = plt.figure(figsize=(10, 5))
    plot_acf(rets_series, lags=acf_lags)
    plt.title(f"ACF of {series_name}")
```

```python
    plt.tight_layout()
    plt.show()

    fig = plt.figure(figsize=(10, 5))
    plot_acf(rets_series**2, lags=acf_lags)
    plt.title(f"ACF of {series_name}   (squared returns)")
    plt.tight_layout()
    plt.show()

print("\n" + "="*80)
print("SUMMARY TABLE: Distributional Statistics for All Series")
print("="*80)
summary_df = pd.DataFrame(distribution_stats)
print("\n" + summary_df.to_string(index=False, float_format="%.6f"))

print("\n" + "="*80)
print("FORMATTED SUMMARY TABLE")
print("="*80)
print(f"{'Series':<15} {'Skewness':<10} {'Kurtosis':<10} {'JB Stat':<12} {'JB p-value':<12} {'Normality':<15}")
print("-" * 80)
for stat in distribution_stats:
    print(f"{stat['Series']:<15} {stat['Skewness']:>9.6f} {stat['Kurtosis']:>9.6f} {stat['JB_Statistic']:>11.6f} {stat['JB_p_value']:>11.6g} {stat['Normality']:>15}")

candidate_specs = [
    {"name":"GARCH11-t",  "vol":"GARCH",  "p":1, "o":0, "q":1, "dist":"StudentsT"},
    {"name":"GJR11-t",    "vol":"GARCH",  "p":1, "o":1, "q":1, "dist":"StudentsT"},
    {"name":"EGARCH11-t", "vol":"EGARCH", "p":1, "o":1, "q":1, "dist":"StudentsT"}
]

eps_reg = 1e-8

selected_models = {}
model_summaries = []

def run_diagnostics(res, rets_index, lags=[10,20]):
    std_resid = pd.Series(res.std_resid, index=rets_index[-len(res.std_resid):])
    jb_s, jb_p = stats.jarque_bera(std_resid)
    lb1 = acorr_ljungbox(std_resid, lags=[lags[0]], return_df=True)['lb_pvalue'].iloc[0]
    lb1_sq = acorr_ljungbox(std_resid**2, lags=[lags[0]], return_df=True)['lb_pvalue'].iloc[0]
    lb2 = acorr_ljungbox(std_resid, lags=[lags[1]], return_df=True)['lb_pvalue'].iloc[0]
    lb2_sq = acorr_ljungbox(std_resid**2, lags=[lags[1]], return_df=True)['lb_pvalue'].iloc[0]
    arch_stat, arch_p, _, _ = het_arch(std_resid, nlags=12)
    return {
        "JB_p": jb_p,
        "LB_resid_p_lag10": lb1,
        "LB_sq_p_lag10": lb1_sq,
        "LB_resid_p_lag20": lb2,
```

```
        "LB_sq_p_lag20": lb2_sq,
        "ARCH_LM_p": arch_p
    }

print("\n1) Fitting candidate univariate models and selecting best by AIC\n"
    + "-"*70)

for series_name in logret_cols:
    df[f"{series_name}_scaled"] = df[series_name] * 100.0

for series_name in logret_cols:
    print(f"\nSeries: {series_name}")
    rets = df[f"{series_name}_scaled"].dropna()

    if rets.empty:
        print("  WARNING: no data for this series, skipping.")
        continue

    fitted = {}
    errors = {}
    for spec in candidate_specs:
        label = spec["name"]
        try:
            o = spec.get("o", 0)
            am = arch_model(rets, mean='Constant', vol=spec["vol"],
                p=spec["p"],
                            o=o if o>0 else 0, q=spec["q"], dist=spec["dist"])
            res = am.fit(disp='off', show_warning=False, options={'maxiter':
                1000})
            fitted[label] = res
            print(f"  {label} fitted   llf={res.loglikelihood:.2f}
                AIC={res.aic:.4f}")
        except Exception as e:
            errors[label] = str(e)
            print(f"  {label} FAILED: {e}")

    comp = []
    for name, r in fitted.items():
        comp.append({"model": name, "AIC": r.aic, "BIC": r.bic, "LLF":
            r.loglikelihood})
    if len(comp) == 0:
        print("  No successful fits for this series. Errors:", errors)
        continue
    comp_df = pd.DataFrame(comp).sort_values("AIC").reset_index(drop=True)
    print("\n  Model comparison (sorted by AIC):")
    print(comp_df.to_string(index=False, float_format="%.4f"))

    best_name = comp_df.loc[0, "model"]
    best_res = fitted[best_name]
    selected_models[series_name] = best_res

    print(f"\n  Selected: {best_name} (AIC={comp_df.loc[0,'AIC']:.4f})")

    diag = run_diagnostics(best_res, rets.index)
    print("  Diagnostics on standardized residuals:")
    print(f"    Jarque-Bera p: {diag['JB_p']:.4g}")
    print(f"    Ljung-Box resid p (lag10): {diag['LB_resid_p_lag10']:.4g}")
    print(f"    Ljung-Box resid^2 p (lag10): {diag['LB_sq_p_lag10']:.4g}")
```

```python
    print(f"    ARCH LM p (nlags=12): {diag['ARCH_LM_p']:.4g}")

    std_resid = pd.Series(best_res.std_resid,
        index=rets.index[-len(best_res.std_resid):])
    plt.figure(figsize=(8,3))
    plt.hist(std_resid, bins=80)
    plt.title(f"{series_name} standardized residuals ({best_name})")
    plt.tight_layout(); plt.show()

    cond_vol = best_res.conditional_volatility
    plt.figure(figsize=(10,3))
    plt.plot(cond_vol.index, cond_vol, label=f"{best_name} cond vol
        (sigma_t)")
    plt.title(f"{series_name} conditional volatility ({best_name})")
    plt.legend(); plt.tight_layout(); plt.show()

    model_summaries.append({
        "Series": series_name,
        "Best_model": best_name,
        "AIC": float(comp_df.loc[0, "AIC"]),
        "BIC": float(comp_df.loc[0, "BIC"]),
        "LLF": float(comp_df.loc[0, "LLF"]),
        "JB_p": diag["JB_p"],
        "LB_sq_p_lag10": diag["LB_sq_p_lag10"],
        "ARCH_LM_p": diag["ARCH_LM_p"]
    })

summary_sel_df = pd.DataFrame(model_summaries)
print("\nFinal selection summary:")
print(summary_sel_df.to_string(index=False, float_format="%.6g"))

print("\n" + "="*60)
print("COMPREHENSIVE STANDARDIZED RESIDUALS DIAGNOSTICS")
print("="*60)

fig, axes = plt.subplots(3, 3, figsize=(15, 12))

for i, series_name in enumerate(logret_cols):
    if series_name in selected_models:
        res = selected_models[series_name]
        best_name = summary_sel_df[summary_sel_df['Series'] ==
            series_name]['Best_model'].values[0]

        std_resid = pd.Series(res.std_resid, index=res.model.y.index)

        axes[0, i].plot(std_resid.index, std_resid, alpha=0.7, linewidth=0.8)
        axes[0, i].set_title(f'{series_name}\nStandardized Residuals
            ({best_name})')
        axes[0, i].axhline(y=0, color='red', linestyle='--', alpha=0.5)
        axes[0, i].grid(True, alpha=0.3)

        axes[1, i].hist(std_resid, bins=60, density=True, alpha=0.7,
            color='skyblue', edgecolor='black')

        x = np.linspace(std_resid.min(), std_resid.max(), 100)
        normal_pdf = stats.norm.pdf(x, 0, 1)
        axes[1, i].plot(x, normal_pdf, 'r-', linewidth=2, label='N(0,1)')
        axes[1, i].set_title(f'Distribution vs Normal')
```

```
        axes[1, i].legend()
        axes[1, i].grid(True, alpha=0.3)

        stats.probplot(std_resid, dist="norm", plot=axes[2, i])
        axes[2, i].set_title(f'QQ Plot vs Normal')
        axes[2, i].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("\nACF plots for standardized residuals...")
fig, axes = plt.subplots(3, 2, figsize=(15, 12))

for i, series_name in enumerate(logret_cols):
    if series_name in selected_models:
        res = selected_models[series_name]
        best_name = summary_sel_df[summary_sel_df['Series'] ==
            series_name]['Best_model'].values[0]
        std_resid = pd.Series(res.std_resid, index=res.model.y.index)

        plot_acf(std_resid, lags=40, ax=axes[i, 0], alpha=0.05)
        axes[i, 0].set_title(f'{series_name}\nACF of Std Residuals
            ({best_name})')

        plot_acf(std_resid**2, lags=40, ax=axes[i, 1], alpha=0.05)
        axes[i, 1].set_title(f'ACF of Squared Std Residuals')

plt.tight_layout()
plt.show()

print("\n" + "="*60)
print("COMPREHENSIVE DIAGNOSTICS ON STANDARDIZED RESIDUALS")
print("="*60)

diagnostics_summary = []

for series_name in logret_cols:
    if series_name in selected_models:
        res = selected_models[series_name]
        best_name = summary_sel_df[summary_sel_df['Series'] ==
            series_name]['Best_model'].values[0]
        std_resid = pd.Series(res.std_resid, index=res.model.y.index)

        mean_val = std_resid.mean()
        std_val = std_resid.std()
        skew_val = stats.skew(std_resid)
        kurt_val = stats.kurtosis(std_resid, fisher=False)

        jb_stat, jb_p = stats.jarque_bera(std_resid)

        lb_resid_5 = acorr_ljungbox(std_resid, lags=[5], return_df=True)
        lb_resid_10 = acorr_ljungbox(std_resid, lags=[10], return_df=True)
        lb_sq_5 = acorr_ljungbox(std_resid**2, lags=[5], return_df=True)
        lb_sq_10 = acorr_ljungbox(std_resid**2, lags=[10], return_df=True)

        arch_stat, arch_p, _, _ = het_arch(std_resid, nlags=12)

        diagnostics_summary.append({
```

```
            'Series': series_name,
            'Model': best_name,
            'Mean': mean_val,
            'Std': std_val,
            'Skewness': skew_val,
            'Kurtosis': kurt_val,
            'JB_p': jb_p,
            'LB5_resid_p': lb_resid_5['lb_pvalue'].iloc[0],
            'LB10_resid_p': lb_resid_10['lb_pvalue'].iloc[0],
            'LB5_sq_p': lb_sq_5['lb_pvalue'].iloc[0],
            'LB10_sq_p': lb_sq_10['lb_pvalue'].iloc[0],
            'ARCH_p': arch_p
        })

diag_df = pd.DataFrame(diagnostics_summary)

def run_diagnostics_robust(res, rets_index, lags=[10,20]):
    std_resid = pd.Series(res.std_resid,
        index=rets_index[-len(res.std_resid):])

    std_resid_clean = std_resid.replace([np.inf, -np.inf], np.nan).dropna()

    if len(std_resid_clean) < 50:
        return {
            "JB_p": np.nan, "LB_resid_p_lag10": np.nan, "LB_sq_p_lag10":
                np.nan,
            "LB_resid_p_lag20": np.nan, "LB_sq_p_lag20": np.nan, "ARCH_LM_p":
                np.nan
        }

    try:
        jb_s, jb_p = stats.jarque_bera(std_resid_clean)
    except:
        jb_p = np.nan

    try:
        lb1 = acorr_ljungbox(std_resid_clean, lags=[lags[0]],
            return_df=True)['lb_pvalue'].iloc[0]
        lb1_sq = acorr_ljungbox(std_resid_clean**2, lags=[lags[0]],
            return_df=True)['lb_pvalue'].iloc[0]
        lb2 = acorr_ljungbox(std_resid_clean, lags=[lags[1]],
            return_df=True)['lb_pvalue'].iloc[0]
        lb2_sq = acorr_ljungbox(std_resid_clean**2, lags=[lags[1]],
            return_df=True)['lb_pvalue'].iloc[0]
    except:
        lb1, lb1_sq, lb2, lb2_sq = np.nan, np.nan, np.nan, np.nan

    try:
        arch_stat, arch_p, _, _ = het_arch(std_resid_clean, nlags=12)
    except:
        arch_p = np.nan

    return {
        "JB_p": jb_p,
        "LB_resid_p_lag10": lb1,
        "LB_sq_p_lag10": lb1_sq,
        "LB_resid_p_lag20": lb2,
        "LB_sq_p_lag20": lb2_sq,
```

```python
        "ARCH_LM_p": arch_p
    }

candidate_specs_arma = [
    {"name":"AR1-GARCH11-t",  "mean": "AR", "lags": 1, "vol":"GARCH",  "p":1,
        "o":0, "q":1, "dist":"StudentsT"},
    {"name":"AR1-GJR11-t",    "mean": "AR", "lags": 1, "vol":"GARCH",  "p":1,
        "o":1, "q":1, "dist":"StudentsT"},
    {"name":"AR1-EGARCH11-t", "mean": "AR", "lags": 1, "vol":"EGARCH", "p":1,
        "o":1, "q":1, "dist":"StudentsT"},
]

improved_models = {}
improved_summaries = []

for series_name in logret_cols:
    print(f"\n--- Improving {series_name} with ARMA structure ---")
    rets = df[f"{series_name}_scaled"].dropna()

    if rets.isna().any() or np.isinf(rets).any():
        print(f"         Numerical issues in returns, cleaning...")
        rets = rets.replace([np.inf, -np.inf], np.nan).dropna()

    fitted_improved = {}
    errors_improved = {}

    for spec in candidate_specs_arma:
        label = spec["name"]
        try:
            o = spec.get("o", 0)
            lags = spec["lags"]

            am = arch_model(
                rets,
                mean=spec["mean"],
                lags=lags,
                vol=spec["vol"],
                p=spec["p"],
                o=o if o>0 else 0,
                q=spec["q"],
                dist=spec["dist"]
            )
            res = am.fit(disp='off', show_warning=False, options={'maxiter':
                1000})
            fitted_improved[label] = res
            print(f"  {label} fitted   AIC={res.aic:.4f}")

        except Exception as e:
            errors_improved[label] = str(e)
            print(f"  {label} FAILED: {e}")

    if fitted_improved:
        comp_improved = []
        for name, r in fitted_improved.items():
            comp_improved.append({
                "model": name,
                "AIC": r.aic,
                "BIC": r.bic,
```

```
                "LLF": r.loglikelihood
            })

        comp_improved_df = pd.DataFrame(comp_improved).sort_values("AIC").reset_index(drop=True)
        print("\n  Improved model comparison (sorted by AIC):")
        print(comp_improved_df.to_string(index=False, float_format="%.4f"))

        best_improved_name = comp_improved_df.loc[0, "model"]
        best_improved_res = fitted_improved[best_improved_name]
        improved_models[series_name] = best_improved_res

        diag_improved = run_diagnostics_robust(best_improved_res, rets.index)


        improved_summaries.append({
            "Series": series_name,
            "Best_model": best_improved_name,
            "AIC": float(comp_improved_df.loc[0, "AIC"]),
            "BIC": float(comp_improved_df.loc[0, "BIC"]),
            "LLF": float(comp_improved_df.loc[0, "LLF"]),
            "LB_resid_p_lag10": diag_improved["LB_resid_p_lag10"],
            "LB_sq_p_lag10": diag_improved["LB_sq_p_lag10"],
            "ARCH_LM_p": diag_improved["ARCH_LM_p"]
        })

        original_aic = summary_sel_df[summary_sel_df['Series'] == series_name]['AIC'].values[0]
        improvement = original_aic - comp_improved_df.loc[0, "AIC"]
        print(f"  AIC improvement over original: {improvement:.4f}")

        if diag_improved['LB_resid_p_lag10'] > 0.05:
            print("      Autocorrelation successfully addressed!")
        else:
            print("         Some autocorrelation may still remain")

if improved_summaries:
    print("\n" + "="*60)
    print("IMPROVED MODELS SUMMARY")
    print("="*60)
    improved_df = pd.DataFrame(improved_summaries)
    print(improved_df.to_string(float_format="%.6f", index=False))

    print("\nReplacing original models with improved ARMA-GARCH models for DCC estimation...")
    selected_models.update(improved_models)

    print("\nUpdating standardized residuals for DCC with improved models...")
    std_resid_list = []
    cond_vol_list = []
    series_list = []

    for series_name in logret_cols:
        if series_name in selected_models:
            res = selected_models[series_name]
            std_resid = pd.Series(res.std_resid, index=res.model.y.index)
            std_resid_list.append(std_resid)
```

```python
            cond_vol = pd.Series(res.conditional_volatility, index=res.model.y.index)
            cond_vol_list.append(cond_vol)
            series_list.append(series_name)

    common_idx = std_resid_list[0].index
    for i in range(1, len(std_resid_list)):
        common_idx = common_idx.intersection(std_resid_list[i].index)

    std_resid_aligned = []
    cond_vol_aligned = []
    for i in range(len(std_resid_list)):
        std_resid_aligned.append(std_resid_list[i].loc[common_idx])
        cond_vol_aligned.append(cond_vol_list[i].loc[common_idx])

    std_resid_df = pd.DataFrame({series_list[i]: std_resid_aligned[i] for i in range(len(series_list))})
    cond_vol_df = pd.DataFrame({series_list[i]: cond_vol_aligned[i] for i in range(len(series_list))})

    print(f"Updated standardized residuals shape: {std_resid_df.shape}")
    print("    Models updated successfully for DCC estimation")

else:
    print("\nNo improved models were successfully fitted. Using original models.")

print("\n2) Preparing standardized residuals for DCC estimation")

std_resid_list = []
cond_vol_list = []
series_list = []

for series_name in logret_cols:
    if series_name in selected_models:
        res = selected_models[series_name]
        std_resid = pd.Series(res.std_resid, index=res.model.y.index)
        std_resid_list.append(std_resid)
        cond_vol = pd.Series(res.conditional_volatility, index=res.model.y.index)
        cond_vol_list.append(cond_vol)
        series_list.append(series_name)

common_idx = std_resid_list[0].index
for i in range(1, len(std_resid_list)):
    common_idx = common_idx.intersection(std_resid_list[i].index)

std_resid_aligned = []
cond_vol_aligned = []
for i in range(len(std_resid_list)):
    std_resid_aligned.append(std_resid_list[i].loc[common_idx])
    cond_vol_aligned.append(cond_vol_list[i].loc[common_idx])

std_resid_df = pd.DataFrame({series_list[i]: std_resid_aligned[i] for i in range(len(series_list))})
cond_vol_df = pd.DataFrame({series_list[i]: cond_vol_aligned[i] for i in range(len(series_list))})

```

```
std_resid_df_clean = std_resid_df.dropna()
cond_vol_df_clean = cond_vol_df.loc[std_resid_df_clean.index]



std_resid_df = std_resid_df_clean
cond_vol_df = cond_vol_df_clean



Z = std_resid_df.values
T, N = Z.shape



fig, axes = plt.subplots(3, 3, figsize=(15, 12))
fig.suptitle('COMPREHENSIVE DIAGNOSTICS: OPTIMIZED ARMA-GARCH STANDARDIZED RESIDUALS',
             fontsize=14, y=1.02)

for i, series_name in enumerate(logret_cols):
    if series_name in improved_models:
        res = improved_models[series_name]
        best_name = improved_df[improved_df['Series'] == series_name]['Best_model'].values[0]

        std_resid = std_resid_df[series_name]

        axes[0, i].plot(std_resid.index, std_resid, alpha=0.7, linewidth=0.8)
        axes[0, i].set_title(f'{series_name}\nOptimized Std Residuals ({best_name})', fontsize=10)
        axes[0, i].axhline(y=0, color='red', linestyle='--', alpha=0.5)
        axes[0, i].axhline(y=2, color='orange', linestyle=':', alpha=0.5, label=' 2  std')
        axes[0, i].axhline(y=-2, color='orange', linestyle=':', alpha=0.5)
        axes[0, i].grid(True, alpha=0.3)

        axes[1, i].hist(std_resid, bins=60, density=True, alpha=0.7, color='lightgreen', edgecolor='black')

        x = np.linspace(std_resid.min(), std_resid.max(), 100)
        normal_pdf = stats.norm.pdf(x, 0, 1)
        axes[1, i].plot(x, normal_pdf, 'r-', linewidth=2, label='N(0,1)')
        axes[1, i].set_title(f'Distribution vs Normal', fontsize=10)
        axes[1, i].legend(fontsize=8)
        axes[1, i].grid(True, alpha=0.3)

        stats.probplot(std_resid, dist="norm", plot=axes[2, i])
        axes[2, i].set_title(f'QQ Plot vs Normal', fontsize=10)
        axes[2, i].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("\nAUTOCORRELATION ANALYSIS: OPTIMIZED STANDARDIZED RESIDUALS")
```

```python
fig, axes = plt.subplots(3, 2, figsize=(15, 12))

for i, series_name in enumerate(logret_cols):
    if series_name in improved_models:
        res = improved_models[series_name]
        best_name = improved_df[improved_df['Series'] ==
            series_name]['Best_model'].values[0]
        std_resid = std_resid_df[series_name]

        plot_acf(std_resid, lags=40, ax=axes[i, 0], alpha=0.05,
                title=f'{series_name}\nACF of Optimized Std Residuals')
        axes[i, 0].set_title(f'{series_name}\nACF of Optimized Std
            Residuals', fontsize=10)

        plot_acf(std_resid**2, lags=40, ax=axes[i, 1], alpha=0.05,
                title=f'ACF of Squared Optimized Std Residuals')
        axes[i, 1].set_title(f'ACF of Squared Optimized Std Residuals',
            fontsize=10)

plt.tight_layout()
plt.show()

print("\n" + "="*60)
print("STATISTICAL SUMMARY: OPTIMIZED STANDARDIZED RESIDUALS")
print("="*60)

optimized_stats = []
for series_name in logret_cols:
    if series_name in improved_models:
        std_resid = std_resid_df[series_name]
        best_name = improved_df[improved_df['Series'] ==
            series_name]['Best_model'].values[0]

        mean_val = std_resid.mean()
        std_val = std_resid.std()
        skew_val = stats.skew(std_resid)
        kurt_val = stats.kurtosis(std_resid, fisher=False)

        jb_stat, jb_p = stats.jarque_bera(std_resid)

        lb_resid_10 = acorr_ljungbox(std_resid, lags=[10], return_df=True)
        lb_sq_10 = acorr_ljungbox(std_resid**2, lags=[10], return_df=True)

        arch_stat, arch_p, _, _ = het_arch(std_resid, nlags=12)

        optimized_stats.append({
            'Series': series_name,
            'Model': best_name,
            'Mean': mean_val,
            'Std': std_val,
            'Skewness': skew_val,
            'Kurtosis': kurt_val,
            'JB_p': jb_p,
            'LB_resid_p': lb_resid_10['lb_pvalue'].iloc[0],
            'LB_sq_p': lb_sq_10['lb_pvalue'].iloc[0],
            'ARCH_p': arch_p
        })

```

```
opt_stats_df = pd.DataFrame(optimized_stats)


R_ccc = std_resid_df.corr().values



plt.figure(figsize=(8, 6))
sns.heatmap(ccc_df, annot=True, cmap='coolwarm', center=0,
            square=True, fmt='.4f', cbar_kws={'shrink': 0.8})
plt.title('CONSTANT CONDITIONAL CORRELATION (CCC) MATRIX\n(Using ARMA-GARCH
    Standardized Residuals)')
plt.tight_layout()
plt.show()

def build_Ht_CCC(cond_vol_df, R):
    Dt = cond_vol_df.values
    Tloc, Nloc = Dt.shape
    Hs = np.empty((Tloc, Nloc, Nloc))
    for t in range(Tloc):
        D = np.diag(Dt[t])
        Hs[t] = D @ R @ D
    return Hs

H_ccc_all = build_Ht_CCC(cond_vol_df, R_ccc)
print(f"CCC conditional covariance matrices shape: {H_ccc_all.shape}")

print("\nCCC CORRELATION STATISTICS:")
ccc_stats = []
for i in range(N):
    for j in range(i+1, N):
        pair = f"{series_list[i]} vs {series_list[j]}"
        corr_val = R_ccc[i, j]
        ccc_stats.append({
            'Pair': pair,
            'CCC_Correlation': corr_val
        })

ccc_stats_df = pd.DataFrame(ccc_stats)
print(ccc_stats_df.to_string(index=False, float_format="%.6f"))



R_hat = (Z.T @ Z) / T

def dcc_negloglike_paper(params, epsilon, R_unconditional, eps_reg=1e-8):
    zeta, xi = params

    if zeta <= 1e-10 or xi <= 1e-10 or (zeta + xi) >= 1.0:
        return 1e12

    Tloc, Nloc = epsilon.shape
    Q_t = R_unconditional.copy()
    negll = 0.0

    for t in range(Tloc):
        diag_q = np.sqrt(np.diag(Q_t))
        if np.any(diag_q <= eps_reg):
```

```
            return 1e12

        Z_t = np.diag(1.0 / diag_q)
        R_t = Z_t @ Q_t @ Z_t

        try:
            sign, logdet = np.linalg.slogdet(R_t)
            if sign <= 0:
                return 1e12

            diag_R = np.diag(R_t)
            if np.any(np.abs(diag_R - 1.0) > 1e-6):
                return 1e12

            R_inv = np.linalg.inv(R_t)
            eps_t = epsilon[t, :]
            quad_form = eps_t @ (R_inv @ eps_t)

            negll += 0.5 * (logdet + quad_form)

        except np.linalg.LinAlgError:
            return 1e12

        if t < Tloc - 1:
            eps_outer = np.outer(eps_t, eps_t)
            Q_t_next = (1.0 - zeta - xi) * R_unconditional + zeta * eps_outer + xi * Q_t

            try:
                min_eigval = np.min(np.real(np.linalg.eigvals(Q_t_next)))
                if min_eigval <= 1e-8:
                    Q_t_next = 0.95 * Q_t_next + 0.05 * R_unconditional
            except np.linalg.LinAlgError:
                Q_t_next = 0.95 * Q_t + 0.05 * R_unconditional

            Q_t = Q_t_next

    return negll

print("\nSTEP 3: ESTIMATING DCC PARAMETERS (  ,  ) BY MAXIMUM LIKELIHOOD")
print("-" * 60)

bounds = [(1e-8, 0.5), (1e-8, 0.999)]
cons = ({'type': 'ineq', 'fun': lambda x: 1.0 - x[0] - x[1] - 1e-8})

starting_points = [
    np.array([0.02, 0.97]),
    np.array([0.05, 0.90]),
    np.array([0.10, 0.85]),
    np.array([0.01, 0.98]),
]

best_result = None
best_negll = np.inf

print("Optimizing DCC parameters (  ,  ) with multiple starting points...")

for i, x0 in enumerate(starting_points):
```

```
    zeta_0, xi_0 = x0
    print(f"\nAttempt {i+1}/{len(starting_points)}:      ={zeta_0:.3f},      ={xi_0:.3f},      +     ={zeta_0+xi_0:.3f}")

    try:
        res_opt = minimize(dcc_negloglike_paper, x0, args=(Z, R_hat, eps_reg),
                          method='SLSQP', bounds=bounds, constraints=cons,
                          options={'ftol': 1e-8, 'disp': False, 'maxiter': 500})

        if res_opt.success:
            zeta_opt, xi_opt = res_opt.x
            print(f"      SUCCESS:    ={zeta_opt:.6f},    ={xi_opt:.6f},   +  ={(zeta_opt+xi_opt):.6f}, LL={-res_opt.fun:.2f}")
            if res_opt.fun < best_negll:
                best_result = res_opt
                best_negll = res_opt.fun
        else:
            print(f"      FAILED: {res_opt.message}")

    except Exception as e:
        print(f"      EXCEPTION: {e}")

print("\nSTEP 4: DCC ESTIMATION RESULTS (Paper Specification)")
print("-" * 60)

if best_result is not None:
    zeta_dcc, xi_dcc = best_result.x
    negll_dcc = float(best_result.fun)


else:
    print("       DCC optimization failed, using sensible defaults")
    zeta_dcc, xi_dcc = 0.02, 0.97
    negll_dcc = dcc_negloglike_paper([zeta_dcc, xi_dcc], Z, R_hat, eps_reg)
    print(f"  Using fallback:   ={zeta_dcc:.4f},   ={xi_dcc:.4f}")

persistence = zeta_dcc + xi_dcc




def compute_dcc_correlations_paper(epsilon, R_hat, zeta, xi, eps_reg=1e-8):
    T, N = epsilon.shape
    R_dynamic = np.zeros((T, N, N))
    Q_dynamic = np.zeros((T, N, N))
    Z_dynamic = np.zeros((T, N, N))

    Q_t = R_hat.copy()

    for t in range(T):
        Q_dynamic[t] = Q_t.copy()

        diag_q = np.sqrt(np.diag(Q_t))
        Z_t = np.diag(1.0 / (diag_q + eps_reg))
        Z_dynamic[t] = Z_t

        R_t = Z_t @ Q_t @ Z_t
```

```
        R_dynamic[t] = R_t

        if t < T - 1:
            eps_t = epsilon[t, :]
            eps_outer = np.outer(eps_t, eps_t)
            Q_t_next = (1.0 - zeta - xi) * R_hat + zeta * eps_outer + xi * Q_t

            try:
                min_eigval = np.min(np.real(np.linalg.eigvals(Q_t_next)))
                if min_eigval <= 1e-8:
                    Q_t_next = 0.95 * Q_t_next + 0.05 * R_hat
            except np.linalg.LinAlgError:
                Q_t_next = 0.95 * Q_t + 0.05 * R_hat

            Q_t = Q_t_next

    return R_dynamic, Q_dynamic, Z_dynamic

R_dcc_dynamic, Q_dcc_dynamic, Z_dcc_dynamic = compute_dcc_correlations_paper(
    Z, R_hat, zeta_dcc, xi_dcc
)



def ccc_loglikelihood(epsilon, R):
    T, N = epsilon.shape
    ll = 0.0

    try:
        R_inv = np.linalg.inv(R)
        sign, logdet = np.linalg.slogdet(R)
        if sign <= 0:
            return -1e12
    except np.linalg.LinAlgError:
        return -1e12

    for t in range(T):
        eps_t = epsilon[t, :]
        quad_form = eps_t @ R_inv @ eps_t
        ll += -0.5 * (logdet + quad_form)

    return ll

print("\n" + "="*80)
print("COMPREHENSIVE MODEL COMPARISON: CCC vs DCC")
print("="*80)

R_ccc = std_resid_df.corr().values

R_dcc_unconditional = np.mean(R_dcc_dynamic, axis=0)



def compute_aic_bic(ll, n_params, n_obs):
    aic = -2 * ll + 2 * n_params
    bic = -2 * ll + n_params * np.log(n_obs)
    return aic, bic

```

```python
n_ccc_params = 3
n_dcc_params = 2

ccc_aic, ccc_bic = compute_aic_bic(ccc_ll, n_ccc_params, T)
dcc_aic, dcc_bic = compute_aic_bic(dcc_ll, n_dcc_params, T)



def likelihood_ratio_test(ll_restricted, ll_unrestricted, df):
    lr_stat = 2 * (ll_unrestricted - ll_restricted)
    p_value = 1 - stats.chi2.cdf(lr_stat, df)
    return lr_stat, p_value

lr_stat, lr_pvalue = likelihood_ratio_test(ccc_ll, dcc_ll, df=2)

print(f"\nLIKELIHOOD RATIO TEST:")
print(f"LR Statistic: {lr_stat:.4f}")
print(f"P-value: {lr_pvalue:.6f}")
if lr_pvalue < 0.05:
    print("    DCC significantly better than CCC at 5% level")
    if lr_pvalue < 0.01:
        print("    DCC significantly better than CCC at 1% level")
else:
    print("    Cannot reject that CCC is adequate")

print("\n" + "="*80)
print("MODEL COMPARISON COMPLETE")
print("="*80)

print("\n" + "="*80)
print("CCC vs DCC COMPREHENSIVE VISUALIZATION & ANALYSIS")
print("="*80)

time_index = std_resid_df.index
N = len(series_list)
colors = ['#1f77b4', '#ff7f0e', '#2ca02c']
pair_names = ['Price1 vs Price2', 'Price1 vs Price3', 'Price2 vs Price3']
pair_indices = [(0, 1), (0, 2), (1, 2)]

fig, axes = plt.subplots(3, 1, figsize=(14, 10))
fig.suptitle('Dynamic Correlations: DCC vs CCC (Separated by Pair)', fontsize=15, fontweight='bold')

for idx, (i, j) in enumerate(pair_indices):
    dcc_corr = R_dcc_dynamic[:, i, j]
    ccc_corr = R_ccc[i, j]

    axes[idx].plot(time_index, dcc_corr, color=colors[idx], lw=1.8, alpha=0.85, label='DCC Correlation')
    axes[idx].axhline(y=ccc_corr, color='red', linestyle='--', lw=2, label='CCC Constant')

    axes[idx].set_title(f'{pair_names[idx]}', fontweight='bold')
    axes[idx].set_ylabel('Correlation')
    axes[idx].grid(alpha=0.3)
    axes[idx].legend(fontsize=9)
    axes[idx].tick_params(axis='x', rotation=45)

```

```
axes[-1].set_xlabel('Time')
plt.tight_layout()
plt.show()

fig, ax = plt.subplots(figsize=(10, 6))
for idx, (i, j) in enumerate(pair_indices):
    dcc_corr = R_dcc_dynamic[:, i, j]
    ccc_corr = R_ccc[i, j]
    ax.hist(dcc_corr, bins=50, alpha=0.6, color=colors[idx], density=True,
        label=f'DCC {pair_names[idx]}')
    ax.axvline(x=ccc_corr, color=colors[idx], linestyle='--', lw=2,
        label=f'CCC {pair_names[idx]}')

ax.set_title('Correlation Distributions: DCC vs CCC', fontsize=12,
    fontweight='bold')
ax.set_xlabel('Correlation')
ax.set_ylabel('Density')
ax.legend(fontsize=9)
ax.grid(alpha=0.3)
plt.show()

fig, axes = plt.subplots(1, 2, figsize=(14, 6))
fig.suptitle('Correlation Matrices Comparison', fontsize=15,
    fontweight='bold')

im1 = axes[0].imshow(R_ccc, cmap='RdYlBu_r', vmin=-0.5, vmax=1.0)
axes[0].set_title('CCC Correlation Matrix', fontsize=12, fontweight='bold')
axes[0].set_xticks(range(N)); axes[0].set_yticks(range(N))
axes[0].set_xticklabels([s.replace('logret_', '') for s in series_list])
axes[0].set_yticklabels([s.replace('logret_', '') for s in series_list])
for i in range(N):
    for j in range(N):
        axes[0].text(j, i, f'{R_ccc[i, j]:.3f}', ha="center", va="center",
            color="black", fontsize=10)

im2 = axes[1].imshow(R_dcc_unconditional, cmap='RdYlBu_r', vmin=-0.5,
    vmax=1.0)
axes[1].set_title('DCC Average Correlation Matrix', fontsize=12,
    fontweight='bold')
axes[1].set_xticks(range(N)); axes[1].set_yticks(range(N))
axes[1].set_xticklabels([s.replace('logret_', '') for s in series_list])
axes[1].set_yticklabels([s.replace('logret_', '') for s in series_list])
for i in range(N):
    for j in range(N):
        axes[1].text(j, i, f'{R_dcc_unconditional[i, j]:.3f}', ha="center",
            va="center", color="black", fontsize=10)

plt.colorbar(im1, ax=axes[0], fraction=0.046, pad=0.04)
plt.colorbar(im2, ax=axes[1], fraction=0.046, pad=0.04)
plt.tight_layout()
plt.show()

print("\nROLLING CORRELATION STATISTICS (252-day windows)")

fig, axes = plt.subplots(3, 1, figsize=(14, 10))
fig.suptitle('Rolling Correlation Statistics (252-day windows)', fontsize=14,
    fontweight='bold')

```

```python
for idx, (i, j) in enumerate(pair_indices):
    dcc_corr = pd.Series(R_dcc_dynamic[:, i, j], index=time_index)
    rolling_mean = dcc_corr.rolling(window=252).mean()
    rolling_std = dcc_corr.rolling(window=252).std()
    ccc_corr = R_ccc[i, j]

    axes[idx].plot(time_index, dcc_corr, alpha=0.3, color=colors[idx], label='Daily DCC')
    axes[idx].plot(time_index, rolling_mean, color='black', lw=2, label='252-day Rolling Mean')
    axes[idx].fill_between(time_index, rolling_mean - rolling_std, rolling_mean + rolling_std,
                           alpha=0.2, color=colors[idx], label=' 1  Std Dev')
    axes[idx].axhline(y=ccc_corr, color='red', linestyle='--', lw=2, label='CCC Constant')

    axes[idx].set_title(pair_names[idx], fontweight='bold')
    axes[idx].set_ylabel('Correlation')
    axes[idx].legend()
    axes[idx].grid(alpha=0.3)

axes[-1].set_xlabel('Time')
plt.tight_layout()
plt.show()

print("\n" + "="*80)
print("DCC DYNAMIC CORRELATION SUMMARY STATISTICS")
print("="*80)

summary_data = []
for idx, (i, j) in enumerate(pair_indices):
    dcc_corr = R_dcc_dynamic[:, i, j]
    ccc_corr = R_ccc[i, j]
    summary_data.append({
        'Pair': pair_names[idx],
        'CCC_Correlation': ccc_corr,
        'DCC_Mean': np.mean(dcc_corr),
        'DCC_Std': np.std(dcc_corr),
        'DCC_Min': np.min(dcc_corr),
        'DCC_Max': np.max(dcc_corr),
        'DCC_Range': np.max(dcc_corr) - np.min(dcc_corr),
        'Volatility_Ratio': np.std(dcc_corr) / np.mean(np.abs(dcc_corr))
    })

summary_df = pd.DataFrame(summary_data)
print("\n" + summary_df.to_string(index=False, float_format="%.4f"))

print("\n" + "="*80)
print("MODEL SELECTION CONCLUSION")
print("="*80)



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

4 Code for the VAR model

```
import warnings
warnings.filterwarnings("ignore")

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import jarque_bera

from statsmodels.tsa.api import VAR
from statsmodels.tsa.stattools import adfuller, kpss, grangercausalitytests
from statsmodels.tsa.vector_ar.vecm import coint_johansen
from statsmodels.stats.stattools import durbin_watson
from statsmodels.stats.diagnostic import acorr_ljungbox, het_arch

from sklearn.metrics import mean_squared_error

sns.set(style="whitegrid")
colors = ['tab:blue', 'tab:orange', 'tab:green']

print("="*70)
print("VECTOR AUTOREGRESSION (VAR) + GARCH ANALYSIS - CORRECTED")
print("Oil Price Shocks on Macroeconomic Variables")
print("="*70)

DEFAULT_FILE = r"C:\Users\marco\OneDrive\Desktop\Empirical exam
    econometrics\Slanzi_Marco.xlsx"
df = pd.read_excel(DEFAULT_FILE, sheet_name="Foglio3", engine="openpyxl")

df.columns = ["obs", "oil_price", "inflation", "gdp_growth"]
df = df.rename(columns={'obs': 'date'})

df['date'] = pd.to_datetime(df['date'].astype(str).str.replace('M', '-'),
    format='%Y-%m')
df = df.set_index('date').sort_index()
try:
    df.index = pd.DatetimeIndex(df.index, freq='MS')
except Exception:
    pass



fig, axes = plt.subplots(3, 1, figsize=(12, 9), sharex=True)
for i, col in enumerate(df.columns):
    axes[i].plot(df.index, df[col], color=colors[i], linewidth=1.2)
    axes[i].set_title(f'{col} - Time Series')
    axes[i].set_ylabel(col)
    axes[i].grid(alpha=0.3)
plt.xlabel('Date')
plt.tight_layout()
plt.show()

plt.figure(figsize=(8, 6))
corr_matrix = df.corr()
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', center=0,
```

```
            square=True, fmt='.3f', cbar_kws={'shrink': 0.8})
plt.title('Correlation Matrix')
plt.tight_layout()
plt.show()

print("\nCorrelation matrix:")
print(corr_matrix.round(3))

for col in df.columns:
    print(f"\n{col} distribution - skew={df[col].skew():.3f} kurt={df[col].kurtosis():.3f}")
    print(f"Rolling stats (12 months): mean={df[col].rolling(12).mean().iloc[-1]:.4f}, std={df[col].rolling(12).std().iloc[-1]:.4f}")

def granger_causality_matrix(data, variables, max_lag=8, verbose=False):
    df_gc = pd.DataFrame(np.nan, index=variables, columns=variables, dtype=float)
    for response in variables:
        for predictor in variables:
            if response == predictor:
                df_gc.loc[response, predictor] = np.nan
                continue
            try:
                test_result = grangercausalitytests(data[[response, predictor]].dropna(),
                                                    maxlag=max_lag, verbose=verbose)
                p_values = [test_result[lag][0]['ssr_chi2test'][1] for lag in range(1, max_lag + 1)]
                df_gc.loc[response, predictor] = np.min(p_values)
            except Exception as e:
                print(f"Granger test failed for {predictor} -> {response}: {e}")
                df_gc.loc[response, predictor] = np.nan
    return df_gc

print("\nPerforming Granger causality tests (min p-value over lags 1..8):")
vars_ = list(df.columns)
granger_pvals = granger_causality_matrix(df, vars_, max_lag=8, verbose=False)
print(granger_pvals.round(4))

def adf_test(series, title=''):
    res = adfuller(series.dropna(), autolag='AIC')
    return res[1]

def kpss_test(series, title=''):
    stat, p_value, lags, crit = kpss(series.dropna(), regression='c', nlags="auto")
    return p_value

print("\n" + "="*50)
print("STATIONARITY ANALYSIS - DATA DRIVEN DECISION")
print("="*50)

stationarity_results = []
for col in df.columns:
    adf_p = adf_test(df[col], title=col)
```

```
    kpss_p = kpss_test(df[col], title=col)

    is_stationary = (adf_p < 0.05) and (kpss_p > 0.05)
    stationarity_results.append(is_stationary)

    print(f"{col:12} - ADF p-value: {adf_p:.4f}, KPSS p-value: {kpss_p:.4f}
        -> {'STATIONARY' if is_stationary else 'NON-STATIONARY'}")

use_diff = not all(stationarity_results)

print(f"\nSTATIONARITY DECISION: {'DIFFERENCING' if use_diff else 'LEVELS'}")
print(f"Rationale: {sum(stationarity_results)}/{len(stationarity_results)}
    series are stationary")

if use_diff:
    df_var = df.diff().dropna()
    print("Using first differences for VAR modeling")
else:
    df_var = df.copy()
    print("Using levels for VAR modeling (all series stationary)")

print("\nData used for VAR (first 5 rows):")
print(df_var.head())

model = VAR(df_var)
maxlags_search = 12
lag_res = model.select_order(maxlags_search)
print("\nLag order selection (all criteria):")
print(lag_res.summary())

criteria = ['aic', 'bic', 'hqic']
selected_lags = {c: lag_res.selected_orders[c] for c in criteria}
print("\nLag selection across criteria:")
for crit, lag in selected_lags.items():
    print(f"  {crit.upper()}: {lag}")

lag_counts = {}
for lag in selected_lags.values():
    lag_counts[lag] = lag_counts.get(lag, 0) + 1

if max(lag_counts.values()) >= 2:
    selected_lag = max(lag_counts, key=lag_counts.get)
    print(f"Selected lag (majority): {selected_lag}")
else:
    selected_lag = selected_lags['bic']
    print(f"Selected lag (BIC - parsimony): {selected_lag}")

n_obs = 12
train = df_var[:-n_obs]
test = df_var[-n_obs:]
print(f"\nTrain shape: {train.shape}, Test shape: {test.shape}")
print(f"Train period: {train.index.min().date()} to
    {train.index.max().date()}")
print(f"Test period: {test.index.min().date()} to {test.index.max().date()}")

p = int(selected_lag) if (selected_lag is not None and not
    np.isnan(selected_lag)) else 1
var_model = VAR(train)
```

```python
var_results = var_model.fit(p)
print("\nVAR MODEL SUMMARY:")
print(var_results.summary())

print("\n" + "="*50)
print("RESIDUAL DIAGNOSTICS")
print("="*50)

print("\nDurbin-Watson statistics (close to 2 indicates no autocorrelation):")
dw = durbin_watson(var_results.resid)
for col, val in zip(train.columns, dw):
    interpretation = "no autocorrelation" if 1.5 < val < 2.5 else "possible autocorrelation"
    print(f"  {col}: {val:.3f} - {interpretation}")

print("\nLjung-Box test for residual autocorrelation:")
for col in var_results.resid.columns:
    lb = acorr_ljungbox(var_results.resid[col], lags=[p, 2*p], return_df=True)
    print(f"\n{col}:")
    for lag in lb.index:
        sig = "***" if lb.loc[lag, 'lb_pvalue'] < 0.01 else "**" if lb.loc[lag, 'lb_pvalue'] < 0.05 else "*" if lb.loc[lag, 'lb_pvalue'] < 0.1 else ""
        print(f"  Lag {lag:2d}: p-value = {lb.loc[lag, 'lb_pvalue']:.4f} {sig}")

print("\n" + "="*50)
print("ARCH EFFECTS TESTING (Engle's Test using het_arch)")
print("Necessary condition for GARCH modeling")
print("="*50)

arch_test_results = {}
garch_recommended = {}

for col in var_results.resid.columns:
    try:
        arch_test = het_arch(var_results.resid[col])
        arch_pvalue = arch_test[1]
        arch_test_results[col] = arch_pvalue
        garch_recommended[col] = arch_pvalue < 0.05

        print(f"{col:12} - ARCH test p-value: {arch_pvalue:.4f} -> {'GARCH RECOMMENDED' if garch_recommended[col] else 'No significant ARCH effects'}")
    except Exception as e:
        print(f"{col:12} - ARCH test failed: {e}")
        arch_test_results[col] = 1.0
        garch_recommended[col] = False

series_for_garch = [col for col in var_results.resid.columns if garch_recommended[col]]
print(f"\nProceeding with GARCH modeling for: {series_for_garch}")

lag_order = var_results.k_ar
last_obs_for_forecast = train.values[-lag_order:]
fc = var_results.forecast(last_obs_for_forecast, steps=n_obs)
fc_df = pd.DataFrame(fc, index=test.index, columns=train.columns)
print("\nVAR forecast (model scale; first 5 rows):")
```

```
print(fc_df.head().round(4))

if use_diff:
    orig_forecast = pd.DataFrame(index=fc_df.index, columns=fc_df.columns,
        dtype=float)
    for col in fc_df.columns:
        last_level = df[col].loc[train.index[-1]]
        orig_forecast[col] = last_level + fc_df[col].cumsum()
else:
    orig_forecast = fc_df.copy()

print("\nForecast on original scale (first 5 rows):")
print(orig_forecast.head().round(4))

actual_levels = df.loc[orig_forecast.index, orig_forecast.columns]

fig, axes = plt.subplots(3, 1, figsize=(12, 10), sharex=True)
for i, col in enumerate(orig_forecast.columns):
    train_extended = df[col].loc[:test.index[0]].iloc[-24:]

    axes[i].plot(train_extended.index, train_extended, color=colors[i],
        linewidth=1.5, label='Historical')
    axes[i].plot(actual_levels.index, actual_levels[col], color=colors[i],
        linewidth=2, label='Actual')
    axes[i].plot(orig_forecast.index, orig_forecast[col], color='red',
        linestyle='--',
                linewidth=2, label='VAR Forecast')
    axes[i].set_title(f'VAR Forecast vs Actual: {col}')
    axes[i].set_ylabel(col)
    axes[i].legend()
    axes[i].grid(alpha=0.3)

    axes[i].axvspan(test.index[0], test.index[-1], alpha=0.1, color='gray')

plt.xlabel('Date')
plt.tight_layout()
plt.show()

print("\nFORECAST ACCURACY (RMSE):")
rmse_results = {}
for col in orig_forecast.columns:
    rmse = np.sqrt(mean_squared_error(actual_levels[col], orig_forecast[col]))
    rmse_results[col] = rmse
    print(f"  {col:12} RMSE: {rmse:.4f}")

print("\n" + "="*50)
print("GARCH MODELING")
print("="*50)

try:
    from arch import arch_model
except Exception as e:
    raise ImportError("To run the GARCH stage install the 'arch' package: pip
        install arch") from e

residuals = var_results.resid
garch_results = {}
```

```
garch_forecasts_var = pd.DataFrame(index=test.index,
    columns=residuals.columns, dtype=float)

if series_for_garch:
    print(f"Fitting GARCH(1,1) models for: {series_for_garch}")

    for col in series_for_garch:
        print(f"\n{'='*30}")
        print(f"GARCH for: {col}")
        print(f"{'='*30}")

        series = residuals[col].dropna()

        best_aic = np.inf
        best_model = None
        best_dist = None

        for dist in ['normal', 't', 'skewt']:
            try:
                am = arch_model(series, mean='Zero', vol='Garch', p=1, q=1,
                    dist=dist)
                gres = am.fit(disp='off', show_warning=False)

                if gres.aic < best_aic:
                    best_aic = gres.aic
                    best_model = gres
                    best_dist = dist

            except Exception as e:
                print(f"  {dist} distribution failed: {e}")
                continue

        if best_model is not None:
            garch_results[col] = best_model
            print(f"Best distribution: {best_dist}")
            print(best_model.summary())

            horizon = len(test.index)
            fc_g = best_model.forecast(horizon=horizon, reindex=False)

            try:
                var_fc = fc_g.variance.iloc[-1].values
            except Exception:
                try:
                    var_array = np.asarray(fc_g.variance)
                    var_fc = var_array.reshape(-1)[-horizon:]
                except Exception:
                    var_fc = np.full(horizon, np.nan)

            if len(var_fc) < horizon:
                var_fc = np.resize(var_fc, horizon)
            elif len(var_fc) > horizon:
                var_fc = var_fc[:horizon]

            garch_forecasts_var[col] = var_fc

            plt.figure(figsize=(10, 4))
            conditional_vol = best_model.conditional_volatility
```

```
            plt.plot(train.index[1:], conditional_vol, color=colors[list(train.columns).index(col)], linewidth=1)
            plt.title(f'Conditional Volatility - {col}')
            plt.ylabel('Conditional Std. Dev.')
            plt.grid(alpha=0.3)
            plt.tight_layout()
            plt.show()
        else:
            print(f"  GARCH fitting failed for {col} with all distributions")
            garch_forecasts_var[col] = np.nan
else:
    print("No significant ARCH effects detected - skipping GARCH modeling")

for col in residuals.columns:
    if col not in garch_forecasts_var or garch_forecasts_var[col].isna().all():
        constant_var = residuals[col].var()
        garch_forecasts_var[col] = constant_var

print("\nGARCH conditional variance forecasts (first 5 rows):")
print(garch_forecasts_var.head().round(6))

combined_table = orig_forecast.copy()
for col in orig_forecast.columns:
    combined_table[col + "_cond_std"] = np.sqrt(garch_forecasts_var[col].values)

print("\nCOMBINED VAR + GARCH RESULTS (first 5 rows):")
print("Mean forecasts with conditional standard deviations:")
print(combined_table.head().round(4))

print("\n" + "="*50)
print("COINTEGRATION ANALYSIS (Johansen Test)")
print("="*50)

try:
    johansen_result = coint_johansen(df, det_order=0, k_ar_diff=p)
    print("Johansen Cointegration Test Results:")
    print(f"Trace statistic critical values: 90%={johansen_result.cvt[0,0]:.3f}, 95%={johansen_result.cvt[0,1]:.3f}, 99%={johansen_result.cvt[0,2]:.3f}")

    n_coint = np.sum(johansen_result.lr1 > johansen_result.cvt[:, 1])
    print(f"Number of cointegrating relationships (95% confidence): {n_coint}")

    if n_coint > 0:
        print("Cointegration detected - consider VECM for long-run analysis")
    else:
        print("No cointegration - VAR in differences is appropriate")

except Exception as e:
    print(f"Cointegration test failed: {e}")

print("\n" + "="*60)
print("RECOMMENDATION: VECM MODEL (Due to Cointegration)")
print("="*60)
```

```python

print("""
STRONG COINTEGRATION DETECTED (3 relationships)

Recommendation: Use VECM instead of VAR for:
1. Capturing long-run equilibrium relationships
2. Better economic interpretation
3. Improved forecasting performance

Your current VAR in levels is valid but suboptimal for cointegrated data.
""")

try:
    from statsmodels.tsa.vector_ar.vecm import VECM

    print("\nFitting VECM model...")

    print("VECM code commented out - uncomment to run VECM analysis")

except ImportError:
    print("VECM modeling requires additional setup")

print("\n" + "="*70)
print("FINAL RECOMMENDATIONS")
print("="*70)

print("""
1. **METHODOLOGICAL**: Switch to VECM modeling due to cointegration
2. **ECONOMIC INTERPRETATION**:
   - Oil prices significantly drive inflation (0.327***)
   - Inflation negatively affects GDP growth (-0.120**)
   - All variables show strong persistence
3. **FORECASTING**: RMSE values provide baseline for model comparison
4. **VOLATILITY**: No GARCH effects detected - constant variance assumption valid

NEXT STEPS:
- Implement VECM model with 3 cointegrating relationships
- Analyze long-run equilibrium relationships
- Compare VECM vs VAR forecast performance
- Conduct impulse response analysis
""")

print("\n" + "="*70)
print("ANALYSIS COMPLETE - KEY FINDINGS")
print("="*70)

print(f"\n1. DATA: {df.shape[0]} monthly observations, {df.shape[1]} variables")
print(f"2. STATIONARITY: Using {'DIFFERENCES' if use_diff else 'LEVELS'}")
print(f"3. OPTIMAL LAG: {selected_lag} (selected by majority/BIC)")
print(f"4. GRANGER CAUSALITY: Oil      Inflation***, Oil      GDP*, Inflation      GDP**")

print(f"\n5. VAR FORECAST ACCURACY (RMSE):")
for col, rmse in rmse_results.items():
    print(f"   {col:12}: {rmse:.4f}")

```

```
print(f"\n6. ARCH EFFECTS: GARCH modeling {'recommended for ' + ', '.join(series_for_garch) if series_for_garch else 'not recommended'}")



for response in granger_pvals.index:
    for predictor in granger_pvals.columns:
        if response != predictor and granger_pvals.loc[response, predictor] < 0.05:
            sig_level = "***" if granger_pvals.loc[response, predictor] < 0.01 else "**" if granger_pvals.loc[response, predictor] < 0.05 else "*"
            print(f"  {predictor}     {response}: p = {granger_pvals.loc[response, predictor]:.4f} {sig_level}")



try:
    coeff_oil_oil = var_results.params.iloc[1, 0] if len(var_results.params.columns) > 1 else var_results.params.iloc[1]
    coeff_inf_oil = var_results.params.iloc[2, 0] if len(var_results.params.columns) > 1 else var_results.params.iloc[2]
    coeff_gdp_oil = var_results.params.iloc[3, 0] if len(var_results.params.columns) > 1 else var_results.params.iloc[3]




print("\nResponse: inflation")
try:
    coeff_oil_inf = var_results.params.iloc[1, 1] if len(var_results.params.columns) > 1 else var_results.params.iloc[4]
    coeff_inf_inf = var_results.params.iloc[2, 1] if len(var_results.params.columns) > 1 else var_results.params.iloc[5]
    coeff_gdp_inf = var_results.params.iloc[3, 1] if len(var_results.params.columns) > 1 else var_results.params.iloc[6]

    print(f"  L1.oil_price:  {coeff_oil_inf:.3f}***")
    print(f"  L1.inflation:  {coeff_inf_inf:.3f}***")
    print(f"  L1.gdp_growth: {coeff_gdp_inf:.3f}")
except Exception as e:
    print(f"  Error accessing coefficients: {e}")

print("\nResponse: gdp_growth")
try:
    coeff_oil_gdp = var_results.params.iloc[1, 2] if len(var_results.params.columns) > 1 else var_results.params.iloc[7]
    coeff_inf_gdp = var_results.params.iloc[2, 2] if len(var_results.params.columns) > 1 else var_results.params.iloc[8]
    coeff_gdp_gdp = var_results.params.iloc[3, 2] if len(var_results.params.columns) > 1 else var_results.params.iloc[9]

    print(f"  L1.oil_price:  {coeff_oil_gdp:.3f}")
    print(f"  L1.inflation:  {coeff_inf_gdp:.3f}**")
    print(f"  L1.gdp_growth: {coeff_gdp_gdp:.3f}***")
except Exception as e:
    print(f"  Error accessing coefficients: {e}")
```

```python
print("\nFull VAR Parameters Table:")
print(var_results.params.round(4))

print("\n6. RESIDUAL DIAGNOSTICS")
print("-" * 40)
print("Durbin-Watson Statistics:")
for col, val in zip(train.columns, dw):
    status = "    No autocorrelation" if 1.5 < val < 2.5 else "    Possible autocorrelation"
    print(f"  {col:12}: {val:.3f} ({status})")

print("\nLjung-Box Test (p-values):")
for col in var_results.resid.columns:
    lb = acorr_ljungbox(var_results.resid[col], lags=[p], return_df=True)
    pval = lb.loc[p, 'lb_pvalue']
    sig = "    " if pval > 0.05 else "    "
    print(f"  {col:12}: {pval:.4f} {sig}")

print("\n7. VOLATILITY ANALYSIS")
print("-" * 40)
print("ARCH Effects Test (p-values):")
for col, pval in arch_test_results.items():
    status = "No GARCH needed" if pval > 0.05 else "GARCH recommended"
    print(f"  {col:12}: {pval:.4f} ({status})")

print("\n8. FORECAST ACCURACY")
print("-" * 40)
print("Root Mean Squared Error (RMSE):")
for col, rmse in rmse_results.items():
    print(f"  {col:12}: {rmse:.4f}")

print("\n9. COINTEGRATION ANALYSIS")
print("-" * 40)
print(f"Number of Cointegrating Relationships: {n_coint}")
if n_coint > 0:
    print("    Strong evidence of long-run equilibrium relationships")
    print("    Recommendation: Consider VECM model for improved analysis")
else:
    print("    No cointegration - VAR model is appropriate")



validation_checks = {
    "Stationarity": all(stationarity_results),
    "No Residual Autocorrelation": all(1.5 < val < 2.5 for val in dw),
    "No ARCH Effects": all(not garch_recommended[col] for col in garch_recommended),
    "Clear Lag Selection": selected_lag is not None,
    "Cointegration Present": n_coint > 0
}

for check, result in validation_checks.items():
    status = "    PASS" if result else "    CHECK"
    print(f"  {check:30}: {status}")
```

```
results_summary = pd.DataFrame({
    'Variable': list(df.columns),
    'Mean': [df[col].mean() for col in df.columns],
    'Std_Dev': [df[col].std() for col in df.columns],
    'Stationary': stationarity_results,
    'ARCH_Test_Pvalue': [arch_test_results[col] for col in df.columns],
    'Forecast_RMSE': [rmse_results[col] for col in df.columns],
    'Residual_Autocorr': [dw[i] for i in range(len(df.columns))]
})

print("\nTechnical Results DataFrame:")
print(results_summary.round(4))

print("\n" + "="*70)
print("GENERATING ADDITIONAL PLOTS FOR REPORT")
print("="*70)

print("\nGenerating Impulse Response Functions...")

try:
    irf = var_results.irf(periods=24)

    fig, axes = plt.subplots(3, 3, figsize=(15, 12))
    fig.suptitle('Orthogonalized Impulse Response Functions (24 months)',
        fontsize=16, y=0.95)

    responses = ['oil_price', 'inflation', 'gdp_growth']

    for i, shock in enumerate(responses):
        for j, response in enumerate(responses):
            axes[i,j].plot(irf.irfs[:, i, j], linewidth=2)
            axes[i,j].axhline(0, color='red', linestyle='--', alpha=0.5)
            axes[i,j].set_title(f'{shock}     {response}')
            axes[i,j].grid(alpha=0.3)

            try:
                axes[i,j].fill_between(range(len(irf.irfs[:, i, j])),
                                     irf.lower[:, i, j],
                                     irf.upper[:, i, j],
                                     alpha=0.2)
            except:
                pass

    plt.tight_layout()
    plt.show()

except Exception as e:
    print(f"IRF plotting failed: {e}")

print("\nGenerating Forecast Error Variance Decomposition...")

try:
    fevd = var_results.fevd(periods=24)

    fig, axes = plt.subplots(1, 3, figsize=(18, 6))
    fig.suptitle('Forecast Error Variance Decomposition (24 months ahead)',
        fontsize=16)

```

```
    periods = range(1, 25)
    colors = ['#1f77b4', '#ff7f0e', '#2ca02c']

    for i, var in enumerate(responses):
        fevd_data = fevd.decomp[i]
        cumulative = np.zeros(len(periods))

        for j, shock in enumerate(responses):
            axes[i].fill_between(periods, cumulative, cumulative +
                fevd_data[:, j],
                               label=shock, alpha=0.7, color=colors[j])
            cumulative += fevd_data[:, j]

        axes[i].set_title(f'Variance Decomposition: {var}')
        axes[i].set_xlabel('Months Ahead')
        axes[i].set_ylabel('Proportion of Variance')
        axes[i].legend()
        axes[i].grid(alpha=0.3)
        axes[i].set_ylim(0, 1)

    plt.tight_layout()
    plt.show()

except Exception as e:
    print(f"FEVD plotting failed: {e}")

print("\nGenerating Historical Decomposition...")

try:
    fig, axes = plt.subplots(3, 1, figsize=(14, 10))
    fig.suptitle('Variable Contributions to Historical Movements',
        fontsize=16)

    fitted_values = var_results.fittedvalues

    for i, col in enumerate(df_var.columns):
        axes[i].plot(train.index[1:], train[col].iloc[1:], label='Actual',
            linewidth=2, color='black')
        axes[i].plot(train.index[1:], fitted_values[col], label='Fitted',
            linestyle='--', linewidth=1.5)
        axes[i].set_title(f'{col} - Model Fit')
        axes[i].set_ylabel(col)
        axes[i].legend()
        axes[i].grid(alpha=0.3)

    plt.tight_layout()
    plt.show()

except Exception as e:
    print(f"Historical decomposition plotting failed: {e}")

print("\nGenerating Detailed Residual Analysis...")

fig, axes = plt.subplots(3, 3, figsize=(15, 12))
fig.suptitle('Comprehensive Residual Diagnostics', fontsize=16)

residuals = var_results.resid

```

```python
for i, col in enumerate(residuals.columns):
    axes[i, 0].plot(residuals.index, residuals[col], color=colors[i])
    axes[i, 0].axhline(0, color='red', linestyle='--', alpha=0.7)
    axes[i, 0].set_title(f'{col} - Residuals')
    axes[i, 0].set_ylabel('Residual')
    axes[i, 0].grid(alpha=0.3)

    from statsmodels.graphics.tsaplots import plot_acf
    plot_acf(residuals[col], ax=axes[i, 1], lags=20, alpha=0.05)
    axes[i, 1].set_title(f'{col} - ACF')

    axes[i, 2].hist(residuals[col], bins=30, density=True, alpha=0.7, color=colors[i])
    axes[i, 2].set_title(f'{col} - Distribution')
    axes[i, 2].set_xlabel('Residual Value')

    from scipy.stats import norm
    x = np.linspace(residuals[col].min(), residuals[col].max(), 100)
    axes[i, 2].plot(x, norm.pdf(x, residuals[col].mean(), residuals[col].std()),
                   'r-', linewidth=2, label='Normal')
    axes[i, 2].legend()

plt.tight_layout()
plt.show()

print("\nGenerating Rolling Statistics...")

fig, axes = plt.subplots(2, 2, figsize=(15, 10))
fig.suptitle('Rolling Statistics and Volatility Analysis', fontsize=16)

for i, col in enumerate(df.columns):
    rolling_mean = df[col].rolling(window=12).mean()
    rolling_std = df[col].rolling(window=12).std()

    axes[0, 0].plot(rolling_mean.index, rolling_mean, label=col, linewidth=2)
axes[0, 0].set_title('12-Month Rolling Means')
axes[0, 0].legend()
axes[0, 0].grid(alpha=0.3)

for i, col in enumerate(df.columns):
    rolling_std = df[col].rolling(window=12).std()
    axes[0, 1].plot(rolling_std.index, rolling_std, label=col, linewidth=2)
axes[0, 1].set_title('12-Month Rolling Standard Deviations')
axes[0, 1].legend()
axes[0, 1].grid(alpha=0.3)

for i, col in enumerate(df.columns):
    cumulative = (1 + df[col]).cumprod() if col != 'inflation' else df[col].cumsum()
    axes[1, 0].plot(cumulative.index, cumulative, label=col, linewidth=2)
axes[1, 0].set_title('Cumulative Path')
axes[1, 0].legend()
axes[1, 0].grid(alpha=0.3)

rolling_corr = df['oil_price'].rolling(window=24).corr(df['inflation'])
axes[1, 1].plot(rolling_corr.index, rolling_corr, linewidth=2, color='purple')
```

```
axes[1, 1].axhline(df['oil_price'].corr(df['inflation']), color='red', linestyle='--',
                  label=f'Overall corr: {df["oil_price"].corr(df["inflation"]):.3f}')
axes[1, 1].set_title('Rolling Correlation: Oil vs Inflation (24-month window)')
axes[1, 1].legend()
axes[1, 1].grid(alpha=0.3)
axes[1, 1].set_ylim(-1, 1)

plt.tight_layout()
plt.show()

print("\nGenerating Forecast Intervals...")

fig, axes = plt.subplots(3, 1, figsize=(14, 12))
fig.suptitle('VAR Forecasts with Historical Context and Confidence Intervals', fontsize=16)

try:
    forecast_intervals = var_results.forecast_interval(train.values[-p:], steps=n_obs, alpha=0.05)

    for i, col in enumerate(df.columns):
        historical = df[col].loc[train.index[-36]:]

        axes[i].plot(historical.index, historical, label='Historical', linewidth=2, color=colors[i])
        axes[i].plot(orig_forecast.index, orig_forecast[col], label='Forecast',
                    linewidth=2, color='red', linestyle='--')

        if not use_diff:
            lower = forecast_intervals[0][:, i]
            upper = forecast_intervals[1][:, i]
            axes[i].fill_between(orig_forecast.index, lower, upper, alpha=0.2, color='red',
                               label='95% Confidence Interval')

        axes[i].axvline(test.index[0], color='gray', linestyle=':', alpha=0.7, label='Forecast Start')
        axes[i].set_title(f'{col} - Forecast with Uncertainty')
        axes[i].set_ylabel(col)
        axes[i].legend()
        axes[i].grid(alpha=0.3)

        axes[i].axvspan(test.index[0], test.index[-1], alpha=0.1, color='gray')

except Exception as e:
    print(f"Forecast interval plotting failed: {e}")
    for i, col in enumerate(df.columns):
        historical = df[col].loc[train.index[-36]:]
        axes[i].plot(historical.index, historical, label='Historical', linewidth=2, color=colors[i])
        axes[i].plot(orig_forecast.index, orig_forecast[col], label='Forecast',
                    linewidth=2, color='red', linestyle='--')
```

```python
        axes[i].axvline(test.index[0], color='gray', linestyle=':', alpha=0.7)
        axes[i].set_title(f'{col} - Forecast')
        axes[i].legend()
        axes[i].grid(alpha=0.3)

plt.tight_layout()
plt.show()

print("\nGenerating Structural Break Analysis...")

try:
    from statsmodels.stats.diagnostic import breaks_cusumolsresid

    fig, axes = plt.subplots(3, 1, figsize=(14, 10))
    fig.suptitle('CUSUM Test for Structural Breaks in Residuals', fontsize=16)

    for i, col in enumerate(residuals.columns):
        try:
            cusum_stat, pval, crit = breaks_cusumolsresid(residuals[col])

            axes[i].plot(cusum_stat, linewidth=2, color=colors[i])
            axes[i].axhline(crit[0], color='red', linestyle='--', alpha=0.7, label='5% Critical Value')
            axes[i].axhline(-crit[0], color='red', linestyle='--', alpha=0.7)
            axes[i].axhline(0, color='black', linestyle='-', alpha=0.5)
            axes[i].set_title(f'{col} - CUSUM Test (p-value: {pval:.3f})')
            axes[i].set_ylabel('CUSUM Statistic')
            axes[i].legend()
            axes[i].grid(alpha=0.3)

            if pval < 0.05:
                axes[i].text(0.02, 0.95, 'STRUCTURAL BREAK DETECTED',
                           transform=axes[i].transAxes, color='red', fontweight='bold')
            else:
                axes[i].text(0.02, 0.95, 'No structural break',
                           transform=axes[i].transAxes, color='green')

        except Exception as e:
            axes[i].text(0.3, 0.5, f'CUSUM test failed: {str(e)[:50]}...',
                       transform=axes[i].transAxes)
            axes[i].set_title(f'{col} - CUSUM Test')

    plt.tight_layout()
    plt.show()

except ImportError:
    print("CUSUM test not available - skipping structural break analysis")

print("\nGenerating Time-Varying Correlation Analysis...")

periods = [
    ('1973-1980', '1973-01-01', '1980-12-01'),
    ('1981-1990', '1981-01-01', '1990-12-01'),
    ('1991-2000', '1991-01-01', '2000-12-01'),
    ('2001-2006', '2001-01-01', '2006-04-01')
]

```

```python
fig, axes = plt.subplots(2, 2, figsize=(15, 12))
fig.suptitle('Evolution of Correlations Over Time', fontsize=16)
axes = axes.flatten()

for idx, (period_name, start, end) in enumerate(periods):
    if idx < len(axes):
        period_data = df.loc[start:end]
        corr_matrix = period_data.corr()

        im = axes[idx].imshow(corr_matrix.values, cmap='RdBu_r', vmin=-1,
            vmax=1, aspect='auto')
        axes[idx].set_xticks(range(len(corr_matrix.columns)))
        axes[idx].set_yticks(range(len(corr_matrix.index)))
        axes[idx].set_xticklabels(corr_matrix.columns, rotation=45)
        axes[idx].set_yticklabels(corr_matrix.index)
        axes[idx].set_title(f'Correlation Matrix: {period_name}')

        for i in range(len(corr_matrix.index)):
            for j in range(len(corr_matrix.columns)):
                axes[idx].text(j, i, f'{corr_matrix.iloc[i, j]:.2f}',
                               ha='center', va='center', fontweight='bold')

plt.tight_layout()
plt.show()
```