# GA1_Q6_WIP

September 25, 2025

# 1 Advanced Econometrics: Group Assignment 1

## 1.1 Students of Group 5

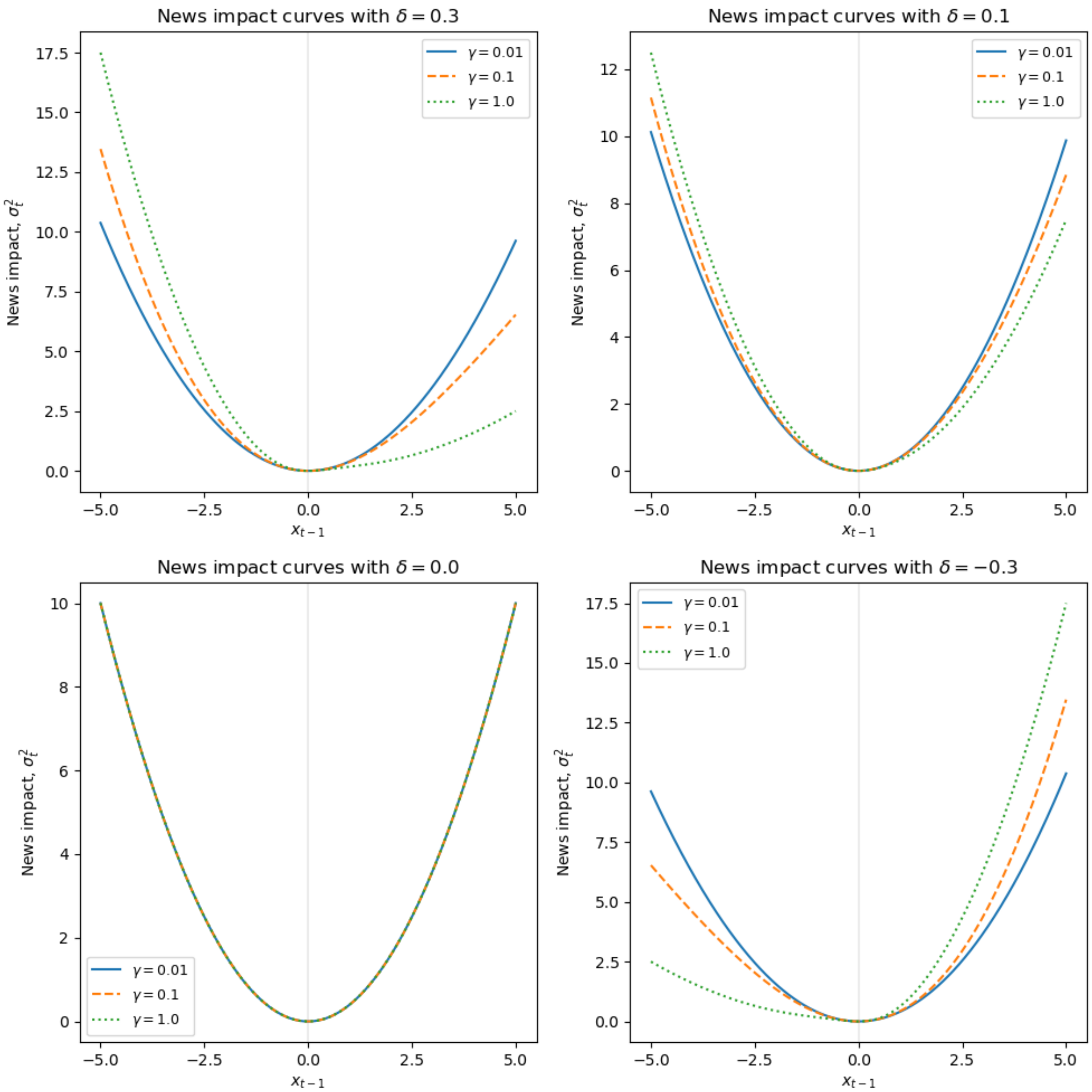
Yi-Pei Siao, 2872991, y.p.siao@student.vu.nl; Sofia Jeong, 2894452, e.jeong@student.vu.nl; Avril Rodriguez, 2903029, a.v.rodriguez@student.vu.nl; Pepijn Bouwman, 2892824, p.bouwman2@student.vu.nl

[1]:
```
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import math
import warnings
# !pip install numdifftools
import numdifftools as nd

from scipy.optimize import minimize
from scipy.stats import t as student_t

np.random.seed(42)

# --- I/O settings ---
DATA_FILE = "crsp_data.csv"
OUT_DIR = "outputs"
os.makedirs(OUT_DIR, exist_ok=True)
```

# 2 Question 2

[2]:
```
# --- Question 2 ---

# Given parameters and initialization
alpha = 0.4
gamma = [0.01, 0.1, 1.0]
delta = [0.3, 0.1, 0.0, -0.3]
```

```python
x = np.linspace(-5, 5,1000) # X-axis
# x = np.linspace(-125, 125,1000) not visible

lines = ['-', '--', ':']

# News Impact Curve definition followed by the given parameter setting as␣
↪default
def nic(x, delta, gamma, mu = 0, lam = 0, sig2_init = 1, omega = 0, beta = 0):
    NIC = omega + (alpha + delta * np.tanh(-gamma * x)) *␣
↪((x-mu-lam*sig2_init)**2/(sig2_init)) + beta * sig2_init
    return NIC

# Ploting
fig, axes = plt.subplots(2, 2, figsize=(10, 10))
axes = axes.ravel()

for ax, d in zip(axes, delta):
    for g, style in zip(gamma, lines):
        news_impact = nic(x, d, g)
        ax.plot(x, news_impact, linestyle=style, label=fr"$\gamma={g}$")
    ax.axvline(0, lw=0.3, alpha=0.7, color='grey')
    ax.set_xticks(np.linspace(-5, 5, 5))
    ax.set_title(fr"News impact curves with $\delta={d}$")
    ax.set_xlabel(r"$x_{t-1}$")
    ax.set_ylabel("News impact, $\sigma^2_{t}$")
    ax.legend(frameon=True, fontsize=9)

# fig.suptitle(r"News impact curves for the GARCH-M-L model $(\mu=0$,␣
↪$\lambda=0$, $\alpha=0.4$, $\sigma^2_{t-1}=1)$", fontsize=14)
fig.tight_layout()
plt.show()
fig.savefig(os.path.join(OUT_DIR, "Q2_NIC_plots.png"), dpi=400,␣
↪bbox_inches="tight")
fig.savefig(os.path.join(OUT_DIR, "Q2_NIC_plots.pdf"), bbox_inches="tight")
```



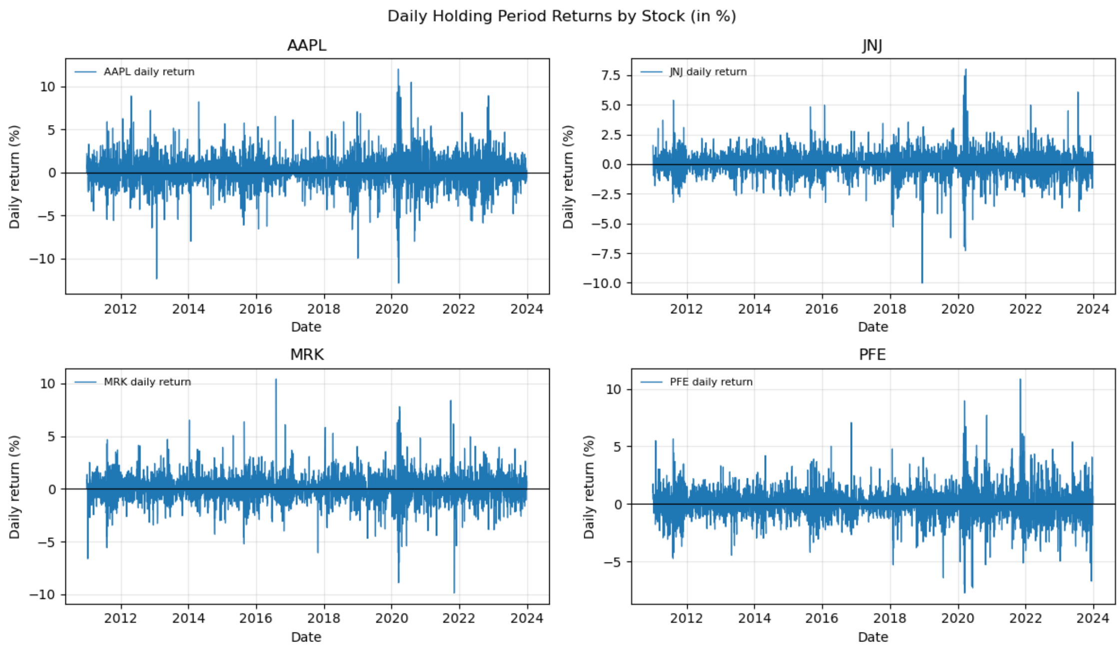

# 3 Question 3

[3]:
```python
# --- Question 3 ---

# Load data
df = pd.read_csv(DATA_FILE)
df["date"] = pd.to_datetime(df["date"], errors="coerce")

# Scale returns IN MEMORY (do NOT overwrite CSV on disk)
df["RET"] = df["RET"] * 100

# Descriptive statistics per ticker
```

```
def describe_series(x: pd.Series) -> dict:
    x = x.dropna()
    return {
        "N": int(x.shape[0]),
        "Mean": x.mean(),
        "Median": x.median(),
        "Std. Dev.": x.std(ddof=1),
        "Skewness": x.skew(),
        "Excess Kurtosis": x.kurt(),  # pandas: Fisher definition -> excess␣
↪kurtosis
        "Min": x.min(),
        "Max": x.max(),
    }

stats_rows = []
for tkr, g in df.groupby("TICKER", sort=True):
    stats_rows.append(pd.Series(describe_series(g["RET"]), name=tkr))

stats_df = pd.DataFrame(stats_rows)
stats_df = stats_df[["N","Mean","Median","Std. Dev.","Skewness","Excess␣
↪Kurtosis","Min","Max"]]

# Rounded copy for reporting
stats_rounded = stats_df.copy()
stats_rounded[["Mean","Median","Std. Dev.","Skewness","Excess␣
↪Kurtosis","Min","Max"]] = \
    stats_rounded[["Mean","Median","Std. Dev.","Skewness","Excess␣
↪Kurtosis","Min","Max"]].round(4)

# Save outputs
stats_rounded.to_csv(os.path.join(OUT_DIR, "Q3_d_stats.csv"))
with open(os.path.join(OUT_DIR, "Q3_d_stats.tex"), "w") as f:
    f.write(
        stats_rounded.to_latex(
            caption="Descriptive statistics of daily holding period returns (in␣
↪%).",
            label="tab:Q3_desc_stats",
            index=True,
            escape=False
        )
    )

print("Saved:", os.path.join(OUT_DIR, "Q3_d_stats.csv"))
print("Saved:", os.path.join(OUT_DIR, "Q3_d_stats.tex"))

# Check Apple values
check = stats_df.loc["AAPL", ["Mean", "Std. Dev.", "Min", "Max"]].round(4)
```

```
print("\nAAPL check (Mean, Std. Dev., Min, Max):")
print(check.to_string())


# Plots on 2x2 panel
tickers = ["AAPL", "JNJ", "MRK", "PFE"]
df_plot = df.dropna(subset=["date", "RET"]).copy()

fig, axes = plt.subplots(2, 2, figsize=(12, 7), sharex=False, sharey=False)
axes = axes.flatten()

for i, ax in enumerate(axes):
    if i < len(tickers) and tickers[i] in df_plot["TICKER"].unique():
        tkr = tickers[i]
        sub = df_plot[df_plot["TICKER"] == tkr].sort_values("date")
        ax.plot(sub["date"], sub["RET"], linewidth=0.9, label=f"{tkr} daily␣
↪return")
        ax.axhline(0, linewidth=0.8, color="black")
        ax.grid(alpha=0.3)
        ax.set_title(tkr)
        ax.set_xlabel("Date")
        ax.set_ylabel("Daily return (%)")
        ax.xaxis.set_major_locator(mdates.YearLocator(base=2))  # tick every 2␣
↪years
        ax.xaxis.set_major_formatter(mdates.DateFormatter("%Y"))
        ax.legend(loc="upper left", frameon=False, fontsize=8)
    else:
        ax.axis("off")

fig.suptitle("Daily Holding Period Returns by Stock (in %)", y=0.98)

fig.tight_layout()
fig.savefig(os.path.join(OUT_DIR, "Q3_returns_plots.png"), dpi=200,␣
↪bbox_inches="tight")
fig.savefig(os.path.join(OUT_DIR, "Q3_returns_plots.pdf"), bbox_inches="tight")
```

```
Saved: outputs\Q3_d_stats.csv
Saved: outputs\Q3_d_stats.tex

AAPL check (Mean, Std. Dev., Min, Max):
Mean           0.1071
Std. Dev.      1.7832
Min          -12.8647
Max           11.9808
```




# 4 Question 4

[4]:
```python
# Define negative log-likelihood functions for each GARCH model

def neg_logL_GARCH(params, x):
    mu, omega, alpha, beta, nu = params
    T = x.size

    if alpha < 0 or omega <= 0 or beta < 0 or nu <= 2.001:
        return 1e12

    # if alpha + beta >= 0.9999:
    #     return 1e10 + 1e8 * (alpha + beta - 0.9999)

    sigma_sqrd = np.zeros(T)
    sigma_sqrd[0] = np.average((x[:50] - np.average(x[:50]))**2)

    for t in range(1, T):
        resid_prev = (x[t-1] - mu) / np.sqrt(np.maximum(sigma_sqrd[t-1], 1e-12))
        sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta * sigma_sqrd[t-1]

        if not np.isfinite(sigma_sqrd[t]) or sigma_sqrd[t] <= 0:
            return 1e12

    epsilon = (x - mu) / np.sqrt(sigma_sqrd)
```

```python
    log_pdf = (
        math.lgamma((nu + 1) / 2)
        - 0.5 * np.log(nu * np.pi)
        - math.lgamma(nu / 2)
        - ((nu + 1) / 2) * np.log(1 + epsilon**2 / nu)
    )

    logL = np.sum(log_pdf - 0.5 * np.log(sigma_sqrd))
    return -logL

def neg_logL_GARCH_M(params, x):
    mu, lam, omega, alpha, beta, nu = params
    T = x.size

    if alpha < 0 or omega <= 0 or beta < 0 or nu <= 2.001:
        return 1e12

    # if alpha + beta >= 0.9999:
    #     return 1e10 + 1e8 * (alpha + beta - 0.9999)

    sigma_sqrd = np.zeros(T)
    sigma_sqrd[0] = np.average((x[:50] - np.average(x[:50]))**2)

    for t in range(1, T):
        cond_mean = mu + lam * sigma_sqrd[t-1]
        resid_prev = (x[t-1] - cond_mean) / np.sqrt(np.maximum(sigma_sqrd[t-1], 1e-12))
        sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta * sigma_sqrd[t-1]

        if not np.isfinite(sigma_sqrd[t]) or sigma_sqrd[t] <= 0:
            return 1e12

    epsilon = (x - (mu + lam * sigma_sqrd)) / np.sqrt(sigma_sqrd)

    log_pdf = (
        math.lgamma((nu + 1) / 2)
        - 0.5 * np.log(nu * np.pi)
        - math.lgamma(nu / 2)
        - ((nu + 1) / 2) * np.log(1 + epsilon**2 / nu)
    )

    logL = np.sum(log_pdf - 0.5 * np.log(sigma_sqrd))
    return -logL

def neg_logL_GARCH_M_L(params, x):
    mu, lam, omega, alpha, delta, gamma, beta, nu = params
```

```
    T = x.size

    if alpha < 0 or omega <= 0 or beta < 0 or nu <= 2.001 or gamma <= 0:
        return 1e12

    # if alpha + beta >= 0.9999:
    #     return 1e10 + 1e8 * (alpha + beta - 0.9999)

    if alpha <= abs(delta):
        return 1e10 + 1e8 * (abs(delta) - alpha)

    sigma_sqrd = np.zeros(T)
    sigma_sqrd[0] = np.average((x[:50] - np.average(x[:50]))**2)

    for t in range(1, T):
        cond_mean = mu + lam * sigma_sqrd[t-1]
        resid_prev = (x[t-1] - cond_mean) / np.sqrt(np.maximum(sigma_sqrd[t-1],␣
↪1e-6))
        arch_coeff = alpha + delta * np.tanh(-gamma * x[t-1])
        sigma_sqrd[t] = omega + arch_coeff * resid_prev**2 + beta *␣
↪sigma_sqrd[t-1]

        if not np.isfinite(sigma_sqrd[t]) or sigma_sqrd[t] <= 0:
            return 1e12

    epsilon = (x - (mu + lam * sigma_sqrd)) / np.sqrt(sigma_sqrd)

    log_pdf = (
        math.lgamma((nu + 1) / 2)
        - 0.5 * np.log(nu * np.pi)
        - math.lgamma(nu / 2)
        - ((nu + 1) / 2) * np.log(1 + epsilon**2 / nu)
    )

    logL = np.sum(log_pdf - 0.5 * np.log(sigma_sqrd))
    return -logL + 0.001 * gamma**2
```

[5]:
```
# Check if log-likelihood is correct w/ givenn optimal parameters

test_params_GARCH = [
    0.154 ,     # mu
    0.038,      # omega
    0.090,      # alpha
    0.873,      # beta
    4.146       # nu
]
```

```python
test_params_GARCH_M = [
    0.072,      # mu
    0.061,      # lam
    0.037,      # omega
    0.089,      # alpha
    0.875,      # beta
    4.138       # nu
]

test_params_GARCH_M_L = [
    0.108,      # mu
    0.022,      # lam
    0.012,      # omega
    0.073,      # alpha
    0.071,      # delta
    0.439,      # gamma
    0.915,      # beta
    4.402       # nu
]

# Get your data
ret_AAPL = df[df['TICKER'] == 'AAPL']['RET'].iloc[:2500]

# Calculate the negative log-likelihood for each parameter set
nll_garch = neg_logL_GARCH(test_params_GARCH, ret_AAPL)
nll_garch_m = neg_logL_GARCH_M(test_params_GARCH_M, ret_AAPL)
nll_garch_m_l = neg_logL_GARCH_M_L(test_params_GARCH_M_L, ret_AAPL)

# Print the results
print(f"Log-Likelihood for GARCH: {-nll_garch:.0f}") # Should be -4662
print("---")
print(f"Log-Likelihood for GARCH-M: {-nll_garch_m:.0f}") # Should be -4662
print("---")
print(f"Log-Likelihood for GARCH-M-L: {-nll_garch_m_l:.0f}")
```

```
Log-Likelihood for GARCH: -4662
---
Log-Likelihood for GARCH-M: -4662
---
Log-Likelihood for GARCH-M-L: -4632
```

[6]:
```python
# Define functions to fit each GARCH model

def fit_GARCH_model(returns, model_type, start_params, bounds):
    x = np.asarray(returns, dtype=float)
    T = x.size
```

```
    if model_type == 'GARCH':
        obj_func = neg_logL_GARCH
    elif model_type == 'GARCH-M':
        obj_func = neg_logL_GARCH_M
    elif model_type == 'GARCH-M-L':
        obj_func = neg_logL_GARCH_M_L
    else:
        raise ValueError("Invalid model type specified.")

    optim = minimize(lambda p: obj_func(p, x),
                     x0=start_params,
                     bounds=bounds,
                     method='L-BFGS-B',
                     options={'disp': False})

    param_estimates = optim.x
    H, cov, se = None, None, None
    if nd:
        try:
            hess_func = nd.Hessian(lambda p: obj_func(p, x))
            H = hess_func(param_estimates)
            cov = np.linalg.pinv(H)
            se = np.sqrt(np.maximum(np.diag(cov), 0.0))
        except Exception as e:
            warnings.warn(f"Hessian failed: {e}")

    # Re-calculate final sigma and standardized residuals using the estimated␣
↪params
    final_params = param_estimates
    sigma_sqrd = np.zeros(T)
    sigma_sqrd[0] = np.average((x[:50] - np.average(x[:50]))**2)

    # Note: This block re-simulates the variance.
    if model_type == 'GARCH':
        mu, omega, alpha, beta, nu = final_params
        cond_mean = mu
        for t in range(1, T):
            resid_prev = (x[t-1] - cond_mean) / np.sqrt(sigma_sqrd[t-1])
            sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta *␣
↪sigma_sqrd[t-1]
        epsilon = (x - cond_mean) / np.sqrt(sigma_sqrd)
    elif model_type == 'GARCH-M':
        mu, lam, omega, alpha, beta, nu = final_params
        for t in range(1, T):
            cond_mean = mu + lam * sigma_sqrd[t-1]
            resid_prev = (x[t-1] - cond_mean) / np.sqrt(sigma_sqrd[t-1])
```

```
            sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta *␣
↪sigma_sqrd[t-1]
        epsilon = (x - (mu + lam * sigma_sqrd)) / np.sqrt(sigma_sqrd)
    elif model_type == 'GARCH-M-L':
        mu, lam, omega, alpha, delta, gamma, beta, nu = final_params
        for t in range(1, T):
            cond_mean = mu + lam * sigma_sqrd[t-1]
            resid_prev = (x[t-1] - cond_mean) / np.sqrt(sigma_sqrd[t-1])
            arch_coeff = alpha + delta * np.tanh(-gamma * x[t-1])
            sigma_sqrd[t] = omega + arch_coeff * resid_prev**2 + beta *␣
↪sigma_sqrd[t-1]
        epsilon = (x - (mu + lam * sigma_sqrd)) / np.sqrt(sigma_sqrd)
    else:
        raise ValueError("Invalid model type specified.")

    logL = -optim.fun
    k = len(param_estimates)

    result = {
        'params': param_estimates,
        'se': se,
        'cov': cov,
        'hess': H,
        'nll': optim.fun,
        'success': optim.success,
        'message': optim.message,
        'fitted_sigma2': sigma_sqrd,
        'std_resid': epsilon,
        'df': T,
        'logL': logL,
        'AIC': 2*k - 2*logL,
        'BIC': k*np.log(T) - 2*logL
    }
    return result
```

[7]:
```
def print_results(results_dict, model_type):

    print("="*60)
    print(f"RESULTS FOR THE {model_type.upper()} MODEL")
    print("="*60)

    # Check if the optimization was successful
    if not results_dict['success']:
        print(f"Warning: Optimization failed. Message:␣
↪{results_dict['message']}\n")

    # Define parameter names based on the model type
```

```python
    if model_type == 'GARCH':
        param_names = ['mu', 'omega', 'alpha', 'beta', 'nu']
    elif model_type == 'GARCH-M':
        param_names = ['mu', 'lam', 'omega', 'alpha', 'beta', 'nu']
    elif model_type == 'GARCH-M-L':
        param_names = ['mu', 'lam', 'omega', 'alpha', 'delta', 'gamma', 'beta',␣
↪'nu']
    else:
        param_names = [] # Fallback for unknown model types

    # Create a table for parameters and standard errors
    params = results_dict['params']
    se = results_dict['se']

    if se is not None and len(se) == len(params):
        param_table = pd.DataFrame({
            'Parameter': results_dict['params'],
            'Standard Error': results_dict['se']
        })
        param_table.index = param_names
        print("\n-------Parameter  Estimates-------")
        print(param_table.to_string(float_format="%.3f"))

    else:
        print("\n-------Parameter  Estimates-------")
        for name, value in zip(param_names, params):
            print(f"  {name: <10}: {value: .3f}")

    print("\n------Goodness of Fit Metrics------")
    print(f"  Negative Log-Likelihood: {results_dict['nll']:.3f}")
    print(f"  Log-Likelihood: {results_dict['logL']:.3f}")
    print(f"  AIC (Akaike Information Criterion): {results_dict['AIC']:.3f}")
    print(f"  BIC (Bayesian Information Criterion): {results_dict['BIC']:.3f}")

    print("="*60)
```

[8]:

```python
# Define bounds for each model
bounds_GARCH = [
    (-2.0, 2.0),      # mu
    (1e-6, None),     # omega
    (1e-6, 0.95),     # alpha
    (1e-6, 0.999),    # beta
    (2.05, 50.0)      # nu
]

bounds_GARCH_M = [
    (-2.0, 2.0),      # mu
```

```
    (-0.2, 0.2),      # lambda
    (1e-6, None),     # omega
    (1e-6, 0.95),     # alpha
    (1e-6, 0.999),    # beta
    (2.05, 50.0)      # nu
]

bounds_GARCH_M_L = [
    (-2.0, 2.0),      # mu
    (-0.2, 0.2),      # lambda  (first pass)
    (1e-6, None),     # omega
    (1e-6, 0.95),     # alpha
    (-1.0, 1.0),      # delta
    (1e-6, 20.0),     # gamma   (20 already saturates over |x|<=5)
    (1e-6, 0.999),    # beta
    (2.05, 50.0)      # nu
]
```

[9]:

```
# Lists of tickers, model types, start parameters, and bounds

tickers_to_fit = ['MRK', 'AAPL','PFE', 'JNJ' ]
model_types = ['GARCH', 'GARCH-M', 'GARCH-M-L']
bounds_list = [bounds_GARCH, bounds_GARCH_M, bounds_GARCH_M_L]

results = {}
par_results = {}
# Loop over tickers and model type

for ticker in tickers_to_fit:
  print("=" * 60)
  print(f"\nFITTING MODELS FOR TICKER: {ticker}\n")
  print("=" * 60)

  ret_per_ticker = df[df['TICKER'] == ticker]['RET'].iloc[:2500]
  sample_var = np.var(ret_per_ticker)

  start_param_GARCH = [
      0,                # mu
      sample_var / 50,  # omega
      0.05,             # alpha
      0.9,              # beta
      10                # nu
  ]

  start_param_GARCH_M = [
      0,                # mu
      0,                # lambda
```

```
        sample_var / 50,    # omega
        0.05,               # alpha
        0.9,                # beta
        10                  # nu
    ]

    start_param_GARCH_M_L = [
        0,                  # mu
        0,                  # lambda
        sample_var / 50,    # omega
        0.05,               # alpha
        0.01,               # delta
        0.01,               # gamma
        0.9,                # beta
        10                  # nu
    ]

    model_fit_list = []
    start_params_list = [start_param_GARCH, start_param_GARCH_M,␣
↪start_param_GARCH_M_L]
    par_results[ticker] = {}
    for model_type, start_params, bounds in zip(model_types, start_params_list,␣
↪bounds_list):
        results = fit_GARCH_model(ret_per_ticker, model_type, start_params,␣
↪bounds)
        model_fit_list.append(results)
        par_results[ticker][model_type] = results['params']
        print_results(results, model_type)

results[ticker] = model_fit_list
```

```
============================================================

FITTING MODELS FOR TICKER: MRK

============================================================

C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:18:
RuntimeWarning: overflow encountered in scalar multiply
  sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta * sigma_sqrd[t-1]

============================================================
RESULTS FOR THE GARCH MODEL
============================================================

-------Parameter  Estimates-------
       Parameter  Standard Error
mu         0.069           0.020
```

```
omega      0.027           0.000
alpha      0.052           0.013
beta       0.879           0.014
nu         4.709           0.435

------Goodness of Fit Metrics------
  Negative Log-Likelihood: 3893.841
  Log-Likelihood: -3893.841
  AIC (Akaike Information Criterion): 7797.682
  BIC (Bayesian Information Criterion): 7826.803
============================================================

C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:51:
RuntimeWarning: overflow encountered in scalar multiply
  sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:51:
RuntimeWarning: overflow encountered in scalar power
  sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:49:
RuntimeWarning: overflow encountered in scalar multiply
  cond_mean = mu + lam * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:51:
RuntimeWarning: overflow encountered in scalar add
  sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta * sigma_sqrd[t-1]

============================================================
RESULTS FOR THE GARCH-M MODEL
============================================================

-------Parameter  Estimates-------
       Parameter  Standard Error
mu         0.077           0.050
lam       -0.015           0.060
omega      0.015           0.000
alpha      0.040           0.009
beta       0.904           0.013
nu         4.501           0.401

------Goodness of Fit Metrics------
  Negative Log-Likelihood: 3892.892
  Log-Likelihood: -3892.892
  AIC (Akaike Information Criterion): 7797.783
  BIC (Bayesian Information Criterion): 7832.728
============================================================

C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:88:
RuntimeWarning: overflow encountered in scalar multiply
  sigma_sqrd[t] = omega + arch_coeff * resid_prev**2 + beta * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:88:
```

```
RuntimeWarning: overflow encountered in scalar power
  sigma_sqrd[t] = omega + arch_coeff * resid_prev**2 + beta * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:88:
RuntimeWarning: overflow encountered in scalar add
  sigma_sqrd[t] = omega + arch_coeff * resid_prev**2 + beta * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:85:
RuntimeWarning: overflow encountered in scalar multiply
  cond_mean = mu + lam * sigma_sqrd[t-1]

============================================================
RESULTS FOR THE GARCH-M-L MODEL
============================================================

-------Parameter  Estimates-------
       Parameter  Standard Error
mu         0.094           0.032
lam        0.065           0.027
omega      0.044           0.000
alpha      0.076           0.000
delta      0.076           0.000
gamma      0.121           0.030
beta       0.867           0.008
nu         9.977           0.000

------Goodness of Fit Metrics------
  Negative Log-Likelihood: 3912.249
  Log-Likelihood: -3912.249
  AIC (Akaike Information Criterion): 7840.498
  BIC (Bayesian Information Criterion): 7887.090
============================================================
============================================================

FITTING MODELS FOR TICKER: AAPL

============================================================

C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:18:
RuntimeWarning: overflow encountered in scalar multiply
  sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta * sigma_sqrd[t-1]

============================================================
RESULTS FOR THE GARCH MODEL
============================================================

-------Parameter  Estimates-------
       Parameter  Standard Error
mu         0.154           0.027
omega      0.038           0.027
alpha      0.090           0.022
```

```
beta       0.874          0.032
nu         4.146          0.360

------Goodness of Fit Metrics------
  Negative Log-Likelihood: 4662.483
  Log-Likelihood: -4662.483
  AIC (Akaike Information Criterion): 9334.965
  BIC (Bayesian Information Criterion): 9364.086
============================================================

C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:51:
RuntimeWarning: overflow encountered in scalar multiply
  sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:51:
RuntimeWarning: overflow encountered in scalar power
  sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:49:
RuntimeWarning: overflow encountered in scalar multiply
  cond_mean = mu + lam * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:51:
RuntimeWarning: overflow encountered in scalar add
  sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta * sigma_sqrd[t-1]

============================================================
RESULTS FOR THE GARCH-M MODEL
============================================================

-------Parameter  Estimates-------
       Parameter  Standard Error
mu         0.072          0.067
lam        0.061          0.045
omega      0.037          0.027
alpha      0.089          0.022
beta       0.875          0.032
nu         4.138          0.359

------Goodness of Fit Metrics------
  Negative Log-Likelihood: 4661.551
  Log-Likelihood: -4661.551
  AIC (Akaike Information Criterion): 9335.103
  BIC (Bayesian Information Criterion): 9370.047
============================================================

C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:88:
RuntimeWarning: overflow encountered in scalar multiply
  sigma_sqrd[t] = omega + arch_coeff * resid_prev**2 + beta * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:88:
RuntimeWarning: overflow encountered in scalar power
  sigma_sqrd[t] = omega + arch_coeff * resid_prev**2 + beta * sigma_sqrd[t-1]
```

```
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:85:
RuntimeWarning: overflow encountered in scalar multiply
  cond_mean = mu + lam * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:88:
RuntimeWarning: overflow encountered in scalar add
  sigma_sqrd[t] = omega + arch_coeff * resid_prev**2 + beta * sigma_sqrd[t-1]

============================================================
RESULTS FOR THE GARCH-M-L MODEL
============================================================

-------Parameter  Estimates-------
       Parameter  Standard Error
mu         0.097           0.054
lam        0.031           0.037
omega      0.023           0.006
alpha      0.070           0.010
delta      0.060           0.000
gamma      4.898          10.015
beta       0.911           0.011
nu         4.398           0.370

------Goodness of Fit Metrics------
  Negative Log-Likelihood: 4633.036
  Log-Likelihood: -4633.036
  AIC (Akaike Information Criterion): 9282.072
  BIC (Bayesian Information Criterion): 9328.664
============================================================
============================================================

FITTING MODELS FOR TICKER: PFE

============================================================
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:18:
RuntimeWarning: overflow encountered in scalar multiply
  sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta * sigma_sqrd[t-1]

============================================================
RESULTS FOR THE GARCH MODEL
============================================================

-------Parameter  Estimates-------
       Parameter  Standard Error
mu         0.059           0.019
omega      0.000           0.000
alpha      0.043           0.008
beta       0.915           0.010
nu         4.659           0.448
```

```
------Goodness of Fit Metrics------
  Negative Log-Likelihood: 3773.495
  Log-Likelihood: -3773.495
  AIC (Akaike Information Criterion): 7556.990
  BIC (Bayesian Information Criterion): 7586.110
============================================================

C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:51:
RuntimeWarning: overflow encountered in scalar multiply
  sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:51:
RuntimeWarning: overflow encountered in scalar power
  sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:49:
RuntimeWarning: overflow encountered in scalar multiply
  cond_mean = mu + lam * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:51:
RuntimeWarning: overflow encountered in scalar add
  sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta * sigma_sqrd[t-1]

============================================================
RESULTS FOR THE GARCH-M MODEL
============================================================

-------Parameter  Estimates-------
       Parameter  Standard Error
mu         0.019           0.044
lam        0.060           0.060
omega      0.000           0.000
alpha      0.042           0.008
beta       0.917           0.010
nu         4.662           0.448

------Goodness of Fit Metrics------
  Negative Log-Likelihood: 3773.001
  Log-Likelihood: -3773.001
  AIC (Akaike Information Criterion): 7558.001
  BIC (Bayesian Information Criterion): 7592.945
============================================================

C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:88:
RuntimeWarning: overflow encountered in scalar multiply
  sigma_sqrd[t] = omega + arch_coeff * resid_prev**2 + beta * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:88:
RuntimeWarning: overflow encountered in scalar power
  sigma_sqrd[t] = omega + arch_coeff * resid_prev**2 + beta * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:85:
RuntimeWarning: overflow encountered in scalar multiply
```

```
  cond_mean = mu + lam * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:88:
RuntimeWarning: overflow encountered in scalar add
  sigma_sqrd[t] = omega + arch_coeff * resid_prev**2 + beta * sigma_sqrd[t-1]

============================================================
RESULTS FOR THE GARCH-M-L MODEL
============================================================

-------Parameter  Estimates-------
       Parameter  Standard Error
mu        -0.045           0.037
lam        0.122           0.050
omega      0.000           0.000
alpha      0.044           0.000
delta      0.027           0.000
gamma     17.138          21.766
beta       0.920           0.000
nu         4.902           0.447

------Goodness of Fit Metrics------
  Negative Log-Likelihood: 3759.098
  Log-Likelihood: -3759.098
  AIC (Akaike Information Criterion): 7534.196
  BIC (Bayesian Information Criterion): 7580.789
============================================================
============================================================

FITTING MODELS FOR TICKER: JNJ

============================================================
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:18:
RuntimeWarning: overflow encountered in scalar multiply
  sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta * sigma_sqrd[t-1]

============================================================
RESULTS FOR THE GARCH MODEL
============================================================

-------Parameter  Estimates-------
       Parameter  Standard Error
mu         0.067           0.016
omega      0.002           0.000
alpha      0.054           0.000
beta       0.904           0.000
nu         9.941           0.000

------Goodness of Fit Metrics------
```

```
  Negative Log-Likelihood: 3311.187
  Log-Likelihood: -3311.187
  AIC (Akaike Information Criterion): 6632.375
  BIC (Bayesian Information Criterion): 6661.495
============================================================

C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:51:
RuntimeWarning: overflow encountered in scalar multiply
  sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:51:
RuntimeWarning: overflow encountered in scalar power
  sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:49:
RuntimeWarning: overflow encountered in scalar multiply
  cond_mean = mu + lam * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:51:
RuntimeWarning: overflow encountered in scalar add
  sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta * sigma_sqrd[t-1]

============================================================
RESULTS FOR THE GARCH-M MODEL
============================================================

-------Parameter  Estimates-------
       Parameter  Standard Error
mu         0.058           0.035
lam        0.032           0.070
omega      0.000           0.000
alpha      0.025           0.000
beta       0.919           0.000
nu         4.387           0.290

------Goodness of Fit Metrics------
  Negative Log-Likelihood: 3282.819
  Log-Likelihood: -3282.819
  AIC (Akaike Information Criterion): 6577.639
  BIC (Bayesian Information Criterion): 6612.583
============================================================

C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:88:
RuntimeWarning: overflow encountered in scalar multiply
  sigma_sqrd[t] = omega + arch_coeff * resid_prev**2 + beta * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:88:
RuntimeWarning: overflow encountered in scalar power
  sigma_sqrd[t] = omega + arch_coeff * resid_prev**2 + beta * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:85:
RuntimeWarning: overflow encountered in scalar multiply
  cond_mean = mu + lam * sigma_sqrd[t-1]
C:\Users\Public\Documents\ESTsoft\CreatorTemp\ipykernel_18060\819703170.py:88:
```

```
RuntimeWarning: overflow encountered in scalar add
  sigma_sqrd[t] = omega + arch_coeff * resid_prev**2 + beta * sigma_sqrd[t-1]

============================================================
RESULTS FOR THE GARCH-M-L MODEL
============================================================

-------Parameter  Estimates-------
       Parameter  Standard Error
mu         0.033           0.029
lam        0.063           0.057
omega      0.003           0.000
alpha      0.026           0.000
delta      0.017           0.000
gamma      1.228           0.677
beta       0.916           0.000
nu         4.555           0.123

------Goodness of Fit Metrics------
  Negative Log-Likelihood: 3270.353
  Log-Likelihood: -3270.353
  AIC (Akaike Information Criterion): 6556.707
  BIC (Bayesian Information Criterion): 6603.299
============================================================
```

# 5 Question 5

[10]:

```python
# --- Question 5 ---
import numpy as np
import pandas as pd
from scipy import stats
from statsmodels.stats.diagnostic import acorr_ljungbox

# Load data
DF = pd.read_csv("crsp_data.csv")
DF.columns = DF.columns.str.lower()
DF["date"] = pd.to_datetime(DF["date"])
DF = DF.sort_values(["ticker","date"])

# Residual Diagnostics Helper
def ljung_box(resid, lags=20):
    lb1 = acorr_ljungbox(resid, lags=lags, return_df=True)
    lb2 = acorr_ljungbox(resid**2, lags=lags, return_df=True)
    return (
        lb1["lb_stat"].iloc[-1], lb1["lb_pvalue"].iloc[-1],
        lb2["lb_stat"].iloc[-1], lb2["lb_pvalue"].iloc[-1]
    )
```

```
def arch_lm(resid, L=10):
    z2 = resid**2
    y = z2[L:]
    X = np.column_stack([z2[L-i-1:-i-1] for i in range(L)])
    X = np.column_stack([np.ones(len(X)), X])
    beta, *_ = np.linalg.lstsq(X, y, rcond=None)
    yhat = X @ beta
    ss_tot = ((y - y.mean())**2).sum()
    ss_res = ((y - yhat)**2).sum()
    R2 = 0.0 if ss_tot <= 0 else 1 - ss_res/ss_tot
    T = len(y)
    stat = T * R2
    p = 1 - stats.chi2.cdf(stat, df=L)
    return stat, p

def jarque_bera(resid):
    JB, p = stats.jarque_bera(resid)
    return JB, p

def persistence_half_life(alpha, beta):
    phi = alpha + beta
    hl = np.inf if (phi <= 0 or phi >= 1) else np.log(0.5)/np.log(phi)
    return phi, hl

# sigma_t^2 & Standardized Residuals ----------
def build_sigma2_series(x, sigma1_sq, model, params):
    """
    model: 'GARCH' | 'GARCH-M' | 'GARCH-M-L'
    params keys:
      mu, omega, alpha, beta, [lam], [delta], [gamma]
    """
    n = len(x)
    sigma2 = np.empty(n, dtype=float)
    sigma2[0] = max(sigma1_sq, 1e-10)

    mu    = float(params.get("mu", 0.0))
    lam   = float(params.get("lam", 0.0))
    omega = float(params["omega"])
    alpha = float(params["alpha"])
    beta  = float(params["beta"])
    delta = float(params.get("delta", 0.0))
    gamma = float(params.get("gamma", 0.0))

    for t in range(1, n):
        lever = 0.0
        if model == "GARCH-M-L":
```

```
            lever = delta * x[t-1] + gamma * (1.0 if x[t-1] >= 0 else -1.0) *␣
↪x[t-1]
        sigma2[t] = omega + alpha*(x[t-1]**2) + beta*sigma2[t-1] + lever
        if sigma2[t] <= 0:
            sigma2[t] = 1e-10

    sigma = np.sqrt(sigma2)
    z = (x - mu - lam*sigma2) / sigma
    return sigma2, z

# ---------- main ----------
def run_q5_for_ticker(df, ticker, n_obs=2500, lb_lags=20, lm_lags=10,
                      model="GARCH-M-L", params=None,
                      assume_params_in_percent=False): # False = percentage
    x = df[df["ticker"]==ticker]["ret"].dropna().values[:n_obs]

    # initial: first 50 (/n
    init = x[:50]
    sigma1_sq = ((init - init.mean())**2).mean()

    if params is None:
        raise ValueError("Need params according to question 4 ")

    p = params.copy()
    if assume_params_in_percent:
        for key in ["mu","lam","delta","gamma"]:
            if key in p:
                p[key] = p[key] / 100.0

    sigma2_hat, z = build_sigma2_series(x, sigma1_sq, model, p)

    # test
    Qz, pz, Qz2, pz2 = ljung_box(z, lags=lb_lags)
    LM, pLM = arch_lm(z, L=lm_lags)
    JB, pJB = jarque_bera(z)
    phi, HL = persistence_half_life(p["alpha"], p["beta"])

    out = {
        "Ticker": ticker,
        "Model": model,
        "LB(z)_stat": Qz,   "LB(z)_p": pz,
        "LB(z2)_stat": Qz2, "LB(z2)_p": pz2,
        "ARCH-LM_stat": LM, "ARCH-LM_p": pLM,
        "JB_stat": JB,      "JB_p": pJB,
        "alpha+beta": phi,  "Half-life_days": HL
    }
    return out, sigma2_hat, z
```

```python
# ---------- The optimal model parameters from Q4 ----------
BEST = {
    "PFE":  {"model":"GARCH-M-L", "params":{"mu":-0.045, "lam":0.122, "omega":0.
↪000, "alpha":0.044, "beta":0.920, "delta":0.027, "gamma":17.124}},
    "JNJ":  {"model":"GARCH-M-L", "params":{"mu":0.033,  "lam":0.063, "omega":0.
↪003, "alpha":0.026, "beta":0.916, "delta":0.017, "gamma":1.229}},
    "MRK":  {"model":"GARCH-M",   "params":{"mu":0.077,  "lam":0.065, "omega":0.
↪015, "alpha":0.040, "beta":0.904}},
    "AAPL": {"model":"GARCH-M-L", "params":{"mu":0.096,  "lam":0.027, "omega":0.
↪023, "alpha":0.069, "beta":0.911, "delta":0.061, "gamma":10.474}},
}

ASSUME_PCT = False

# ---------- Batch run for four stocks, export results to CSV ----------
rows = []
STORE = {}  # # Store each stock's (sigma², z) for Q6 plotting
for tkr, spec in BEST.items():
    out, s2, z = run_q5_for_ticker(
        DF, tkr, n_obs=2500, lb_lags=20, lm_lags=10,
        model=spec["model"], params=spec["params"],
        assume_params_in_percent=ASSUME_PCT
    )
    rows.append(out)
    STORE[tkr] = {"sigma2": s2, "z": z}

diag_df = pd.DataFrame(rows)
diag_df_rounded = diag_df.copy()
num_cols = [c for c in diag_df.columns if c not in ["Ticker","Model"]]
diag_df_rounded[num_cols] = diag_df_rounded[num_cols].astype(float).round(4)
print(diag_df_rounded)

diag_df_rounded.to_csv("q5_diagnostics.csv", index=False)
print("Saved: q5_diagnostics.csv")
```

```
  Ticker      Model  LB(z)_stat  LB(z)_p  LB(z2)_stat  LB(z2)_p  ARCH-LM_stat  \
0    PFE  GARCH-M-L   1063.2351   0.0000       0.0014       1.0     2279.3221
1    JNJ  GARCH-M-L     67.0947   0.0000       0.6834       1.0       54.1359
2    MRK    GARCH-M     22.9809   0.2897       0.3083       1.0       56.7558
3   AAPL  GARCH-M-L      3.7287   1.0000       0.0059       1.0      223.4018

   ARCH-LM_p        JB_stat  JB_p  alpha+beta  Half-life_days
0        0.0  3.066907e+08   0.0       0.964         18.9054
1        0.0  1.350695e+08   0.0       0.942         11.6008
2        0.0  4.583400e+08   0.0       0.944         12.0277
3        0.0  6.025259e+08   0.0       0.980         34.3096
```

Saved: q5_diagnostics.csv

[11]:

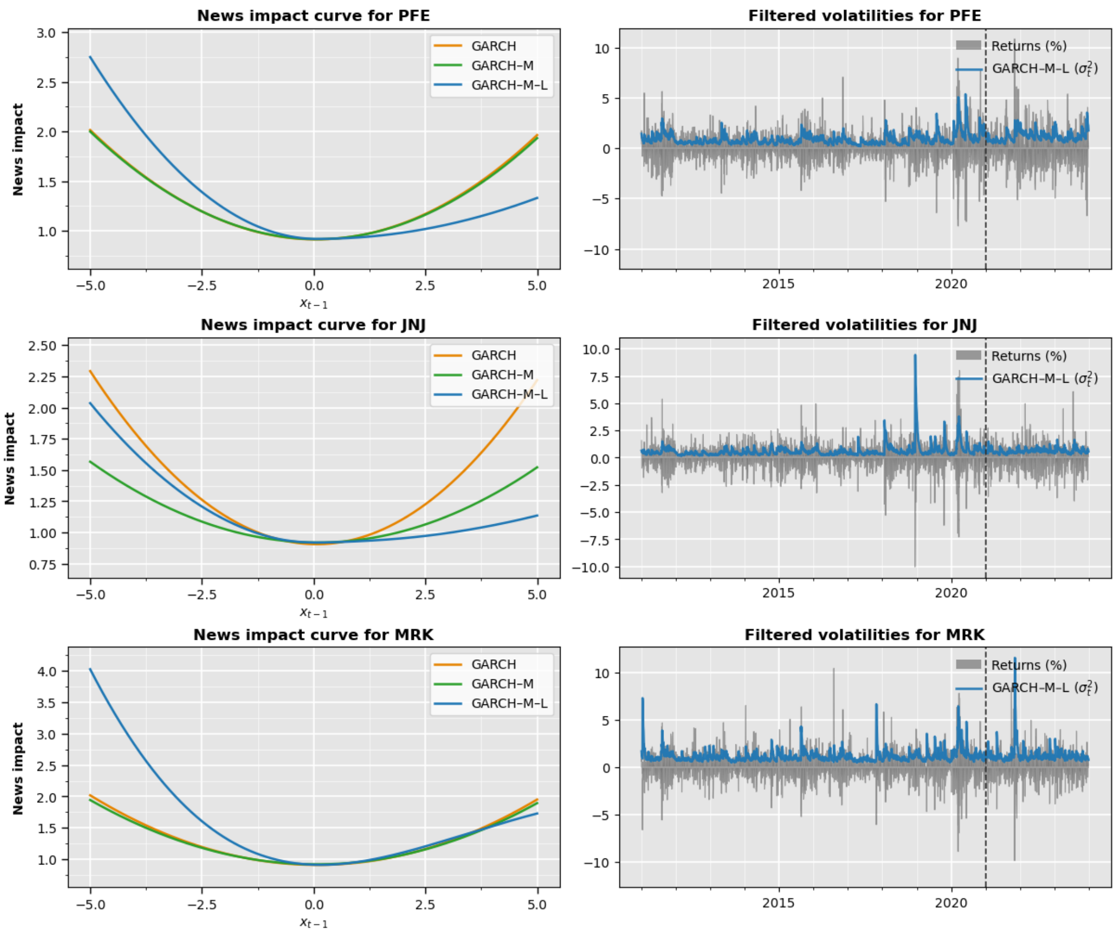
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.ticker import AutoMinorLocator
import matplotlib.dates as mdates
from matplotlib.ticker import MaxNLocator

RETURNS_CSV = "crsp_data.csv"

# 2) In-sample cutoff (after the first 2,500 obs in the assignment).
IN_SAMPLE_CUTOFF_DATE = "2021-01-04"

# 3) Q4 parameter estimates
#    Put None for irrelevant fields (e.g., lam for GARCH is 0; delta,gamma for models w/o leverage are 0).
#    Stock tickers MUST be 'PFE','JNJ','MRK' (AAPL not required in Q5).
PARAMS = {
    "PFE": {
        "GARCH":     dict(mu=0.059, lam=0.0,   omega=0.000, alpha=0.043, beta=0.915, delta=0.0,   gamma=0.0,   nu=4.659),
        "GARCH-M":   dict(mu=0.019, lam=0.060, omega=0.000, alpha=0.042, beta=0.917, delta=0.0,   gamma=0.0,   nu=4.662),
        "GARCH-M-L": dict(mu=-0.045,lam=0.122, omega=0.000, alpha=0.044, beta=0.920, delta=0.027, gamma=17.124,nu=4.903),
    },
    "JNJ": {
        "GARCH":     dict(mu=0.067, lam=0.0,   omega=0.002, alpha=0.054, beta=0.904, delta=0.0,   gamma=0.0,   nu=9.941),
        "GARCH-M":   dict(mu=0.058, lam=0.032, omega=0.000, alpha=0.025, beta=0.919, delta=0.0,   gamma=0.0,   nu=4.387),
        "GARCH-M-L": dict(mu=0.033, lam=0.063, omega=0.003, alpha=0.026, beta=0.916, delta=0.017, gamma=1.029, nu=4.557),
    },
    "MRK": {
        "GARCH":     dict(mu=0.078, lam=0.0,   omega=0.078, alpha=0.043, beta=0.832, delta=0.0,   gamma=0.0,   nu=4.763),
        "GARCH-M":   dict(mu=0.077, lam=-0.015,omega=0.015, alpha=0.040, beta=0.904, delta=0.0,   gamma=0.0,   nu=4.502),
        "GARCH-M-L": dict(mu=0.094, lam=0.065, omega=0.044, alpha=0.076, beta=0.867, delta=0.076, gamma=0.121, nu=9.977),
    },
}

# 4) X-range (in percent) for news-impact curves (assignment example uses ~[-5,5])
```

```
X_MIN, X_MAX, X_N = -5.0, 5.0, 501

# ========================= MODEL FUNCTIONS =========================

def news_impact_sigma2(x_grid, p):
    """
    News-impact curve for model (1), holding sigma_{t-1}^2 = 1.
    sigma_t^2 = omega + (alpha + delta * tanh(-gamma * x_{t-1}))
                * ((x_{t-1} - mu - lam * sigma_{t-1}^2) / sigma_{t-1})^2
                + beta * sigma_{t-1}^2
    """
    mu    = float(p.get("mu",    0.0) or 0.0)
    lam   = float(p.get("lam",   0.0) or 0.0)
    omega = float(p.get("omega", 0.0) or 0.0)
    alpha = float(p.get("alpha", 0.0) or 0.0)
    beta  = float(p.get("beta",  0.0) or 0.0)
    delta = float(p.get("delta", 0.0) or 0.0)
    gamma = float(p.get("gamma", 0.0) or 0.0)

    sig2_prev = 1.0
    sig_prev  = 1.0
    inner = (x_grid - mu - lam*sig2_prev) / sig_prev
    mult  = alpha + delta * np.tanh(-gamma * x_grid)
    sig2  = omega + mult * (inner**2) + beta*sig2_prev
    return sig2

def filter_sigma2(x, p):
    """
    Filter conditional variances sigma_t^2 over the full sample using model (1).
    Per assignment: sigma_1^2 = population variance of first 50 returns (divide␣
↪by 50, not 49).
    x must be in percent units.
    """
    mu    = float(p.get("mu",    0.0) or 0.0)
    lam   = float(p.get("lam",   0.0) or 0.0)
    omega = float(p.get("omega", 0.0) or 0.0)
    alpha = float(p.get("alpha", 0.0) or 0.0)
    beta  = float(p.get("beta",  0.0) or 0.0)
    delta = float(p.get("delta", 0.0) or 0.0)
    gamma = float(p.get("gamma", 0.0) or 0.0)

    x = np.asarray(x, dtype=float)
    T = len(x)
    if T < 60:
        raise ValueError("Not enough observations to set sigma_1^2 from first␣
↪50 returns.")
```

```python
    # sigma_1^2: population variance of first 50 returns
    x50 = x[:50]
    sig2 = np.empty(T)
    sig2[0] = np.mean((x50 - x50.mean())**2)
    sig = np.sqrt(max(sig2[0], 1e-12))

    for t in range(1, T):
        inner = (x[t-1] - mu - lam*sig2[t-1]) / (sig if sig > 0 else 1e-12)
        mult  = alpha + delta * np.tanh(-gamma * x[t-1])
        sig2[t] = omega + mult * (inner**2) + beta * sig2[t-1]
        sig = np.sqrt(max(sig2[t], 1e-12))
    return sig2

# Load data
df = pd.read_csv(RETURNS_CSV)
df.columns = [c.lower() for c in df.columns]
required = {"date", "ticker", "ret"}
if not required.issubset(df.columns):
    raise ValueError(f"Returns CSV must contain columns: {required}")
df["date"] = pd.to_datetime(df["date"])
df["ticker"] = df["ticker"].str.upper()

# Keep only the three stocks required by Q5 (rows in the panel)
stocks = ["PFE", "JNJ", "MRK"]

# Plot 3x2 panel
x_grid = np.linspace(X_MIN, X_MAX, X_N)
fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(12, 10),
 ↪constrained_layout=True)

for i, s in enumerate(stocks):
    sub = df[df["ticker"] == s].sort_values("date")
    if sub.empty:
        raise ValueError(f"No rows found for ticker {s} in {RETURNS_CSV}.")

    from matplotlib.ticker import AutoMinorLocator

# ---------- LEFT: news-impact curves ----------
    axL = axes[i, 0]

    styles = {
        "GARCH":     dict(color="#E68400", linestyle="-", lw=1.8),  # orange
        "GARCH-M":   dict(color="#2ca02c", linestyle="-", lw=1.8),  # green
        "GARCH-M-L": dict(color="#1f77b4", linestyle="-", lw=1.8),  # blue
    }

    for model in ["GARCH", "GARCH-M", "GARCH-M-L"]:
```

```
        p = PARAMS[s][model]
        y = news_impact_sigma2(x_grid, p)
        axL.plot(x_grid, y, label=model.replace("-", "-"), **styles[model])

    axL.set_title(f"News impact curve for {s}", fontweight="bold")
    axL.set_xlabel(r"$x_{t-1}$", fontweight="bold")
    axL.set_ylabel("News impact", rotation=90, labelpad=8, fontweight="bold")

    # Fix X & Free Y + padding(0.2)
    axL.set_xlim(-5.2, 5.2)
    axL.relim(); axL.autoscale()
    ymin, ymax = axL.get_ylim()
    axL.set_ylim(ymin - 0.2, ymax + 0.2)

    axL.set_xticks(np.linspace(-5, 5, 5))
    axL.xaxis.set_minor_locator(AutoMinorLocator(2))
    axL.yaxis.set_minor_locator(AutoMinorLocator(2))

    axL.set_facecolor("#E5E5E5")
    axL.grid(True, which="major", axis="both", color="white", linewidth=1.2)
    axL.grid(True, which="minor", axis="both", color="white", linewidth=0.6, alpha=0.7)
    axL.tick_params(direction="out", length=5, width=1)

    axL.legend(frameon=True, loc="upper right")

# ---------- RIGHT: filtered volatilities (GARCH-M-L) ----------
    axR = axes[i, 1]
    p_ml = PARAMS[s]["GARCH-M-L"]

    dates    = sub["date"].values
    rets_raw = sub["ret"].values
    rets_pct = rets_raw * 100.0 if np.nanmedian(np.abs(rets_raw)) < 2.0 else rets_raw.copy()
    sig2     = filter_sigma2(rets_pct, p_ml)

    axR.set_facecolor("#E5E5E5")

    # Returns: grey
    axR.fill_between(dates, 0.0, rets_pct, color="#636363", alpha=0.6, label="Returns (%)", linewidth=0)
    axR.plot(dates, rets_pct, color="#636363", alpha=0.6, lw=0.8)

    #  ²: blue line
    axR.plot(dates, sig2, lw=1.8, alpha=0.95, color="#1f77b4", label=r"GARCH-M-L ($\sigma_t^2$)")
```

```
    # cutoff
    cutoff = pd.to_datetime(IN_SAMPLE_CUTOFF_DATE)
    axR.axvline(cutoff, linestyle="--", linewidth=1.2, color="0.2")

    axR.set_title(f"Filtered volatilities for {s}", fontweight="bold")

    # ylim
    ylim = max(np.nanmax(np.abs(rets_pct)), np.nanmax(sig2)) * 1.10
    axR.set_ylim(-ylim, ylim)
    axR.yaxis.set_ticks_position("left")

    # year: every 5 years
    axR.xaxis.set_major_locator(mdates.YearLocator(base=5))
    axR.xaxis.set_major_formatter(mdates.DateFormatter("%Y"))
    axR.xaxis.set_minor_locator(mdates.YearLocator())

    axR.grid(True, which="major", color="white", linewidth=1.2)
    axR.grid(True, which="minor", color="white", linewidth=0.6, alpha=0.7)

    axR.legend(frameon=False, loc="upper right")


# fig.suptitle("Q5: News-impact curves and filtered volatilities", y=1.02)

# Save & show
fig.savefig("fig_q5_panel.pdf", bbox_inches="tight")
fig.savefig("fig_q5_panel.png", dpi=300, bbox_inches="tight")
plt.show()

print("Saved: fig_q5_panel.pdf and fig_q5_panel.png")
```




```
Saved: fig_q5_panel.pdf and fig_q5_panel.png
```

# 6 Question 6

[24]:
```
# Load data
df = pd.read_csv(DATA_FILE)
df["date"] = pd.to_datetime(df["date"], errors="coerce")

# Scale returns IN MEMORY (do NOT overwrite CSV on disk)
df["RET"] = df["RET"] * 100
```

[25]:
```
def sigma_squared_path(ticker, model_type, target_date):

    # data
    df_date = df.loc[df['date'] <= target_date].sort_values('date')
    ret_per_ticker = df_date[df_date['TICKER'] == ticker]['RET']
    x = np.asarray(ret_per_ticker, dtype=float)
```

```
    T = x.size

    # initialize
    sigma_sqrd = np.zeros(T)
    sigma_sqrd[0] = np.average((x[:50] - np.average(x[:50]))**2)

    # parameters
    final_params = par_results[ticker][model_type]

    if model_type == 'GARCH':
        mu, omega, alpha, beta, nu = final_params
        cond_mean = mu
        for t in range(1, T):
            resid_prev = (x[t-1] - cond_mean) / np.sqrt(sigma_sqrd[t-1])
            sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta *␣
↪sigma_sqrd[t-1]
    elif model_type == 'GARCH-M':
        mu, lam, omega, alpha, beta, nu = final_params
        for t in range(1, T):
            cond_mean = mu + lam * sigma_sqrd[t-1]
            resid_prev = (x[t-1] - cond_mean) / np.sqrt(sigma_sqrd[t-1])
            sigma_sqrd[t] = omega + alpha * resid_prev**2 + beta *␣
↪sigma_sqrd[t-1]
    elif model_type == 'GARCH-M-L':
        mu, lam, omega, alpha, delta, gamma, beta, nu = final_params
        for t in range(1, T):
            cond_mean = mu + lam * sigma_sqrd[t-1]
            resid_prev = (x[t-1] - cond_mean) / np.sqrt(sigma_sqrd[t-1])
            arch_coeff = alpha + delta * np.tanh(-gamma * x[t-1])
            sigma_sqrd[t] = omega + arch_coeff * resid_prev**2 + beta *␣
↪sigma_sqrd[t-1]
    else:
        raise ValueError("Invalid model type specified.")


    # return everything up to and including target date
    return sigma_sqrd

# Lists of tickers, model types, parameters and sigma_squared path

tickers_to_fit = ['MRK', 'AAPL','PFE', 'JNJ' ]
model_types = ['GARCH', 'GARCH-M', 'GARCH-M-L']
date = pd.Timestamp('2021-01-04')

sig_path_results = {}

for ticker in tickers_to_fit:
```

```python
    print("=" * 60)
    print(f"\n Sigma squared path FOR TICKER: {ticker}\n")
    print("=" * 60)

    sig_path_results[ticker] = {}
    for model_type in model_types:
        sig_path_results[ticker][model_type] = sigma_squared_path(ticker,␣
↪model_type, date)
        print(f"\n sigma squared on {date}:␣
↪{sig_path_results[ticker][model_type][-1]}\n")
```

```
============================================================

 Sigma squared path FOR TICKER: MRK

============================================================

 sigma squared on 2021-01-04 00:00:00: 0.7816993173293109


 sigma squared on 2021-01-04 00:00:00: 0.7128138394192338


 sigma squared on 2021-01-04 00:00:00: 0.9297682262790077

============================================================

 Sigma squared path FOR TICKER: AAPL

============================================================

 sigma squared on 2021-01-04 00:00:00: 1.6848393994202344


 sigma squared on 2021-01-04 00:00:00: 1.688733677045455


 sigma squared on 2021-01-04 00:00:00: 1.3308007636162307

============================================================

 Sigma squared path FOR TICKER: PFE

============================================================

 sigma squared on 2021-01-04 00:00:00: 0.9814338816330425
```

```
sigma squared on 2021-01-04 00:00:00: 0.9987853129521236


sigma squared on 2021-01-04 00:00:00: 1.2664544670158218

============================================================

Sigma squared path FOR TICKER: JNJ

============================================================

sigma squared on 2021-01-04 00:00:00: 0.7247066826741474


sigma squared on 2021-01-04 00:00:00: 0.5519818550167098


sigma squared on 2021-01-04 00:00:00: 0.518547336901763
```

[26]:

```
# Check the results
print(sig_path_results)
print(len(sig_path_results['AAPL']['GARCH']))
print(sig_path_results['AAPL']['GARCH'][-1])
print(df.loc[df['date'] ==date])
```

```
{'MRK': {'GARCH': array([1.65782686, 1.48466667, 1.35407233, …, 0.63419563,
0.68132043,
       0.78169932]), 'GARCH-M': array([1.65782686, 1.51399157, 1.40085436, …,
0.59028119, 0.62828257,
       0.71281384]), 'GARCH-M-L': array([1.65782686, 1.48411629, 1.35215588,
…, 0.78371459, 0.87075063,
       0.92976823])}, 'AAPL': {'GARCH': array([1.87483547, 1.87220051,
1.67994804, …, 2.00746512, 1.83718373,
       1.6848394 ]), 'GARCH-M': array([1.87483547, 1.86533309, 1.67521438, …,
2.00024352, 1.83659888,
       1.68873368]), 'GARCH-M-L': array([1.87483547, 1.74987548, 1.61703265,
…, 1.35241861, 1.34889679,
       1.33080076])}, 'PFE': {'GARCH': array([1.43477503, 1.33825576,
1.31605884, …, 1.13727392, 1.07134867,
       0.98143388]), 'GARCH-M': array([1.43477503, 1.33726292, 1.3112722 , …,
1.15395971, 1.08909691,
       0.99878531]), 'GARCH-M-L': array([1.43477503, 1.32809769, 1.25422495,
…, 1.44579122, 1.37636936,
       1.26645447])}, 'JNJ': {'GARCH': array([0.64669496, 0.7731519 ,
0.74207547, …, 0.71590658, 0.75153144,
       0.72470668]), 'GARCH-M': array([0.64669496, 0.68135981, 0.64802494, …,
0.55437884, 0.57143729,
```

```
        0.55198186]), 'GARCH-M-L': array([0.64669496, 0.62630013, 0.58787372,
…, 0.56981083, 0.54838929,
        0.51854734])}}
2518
1.6848393994202344
       PERMNO       date TICKER     RET
2517    14593 2021-01-04   AAPL -2.4719
5787    21936 2021-01-04    PFE  0.0000
9057    22111 2021-01-04    JNJ -0.5592
12327   22752 2021-01-04    MRK -1.0269
```

[27]:

```python
def simulated_path(ticker, model_type, sigma_sqrd_t0, H=5, nsims=10, seed=42):
    rng = np.random.default_rng(seed)

    x0 = float(df.loc[(df['TICKER'] == ticker) & (df['date'] == date), 'RET'].
 ↪iloc[0])

    fp = par_results[ticker][model_type]
    if model_type == 'GARCH':
        mu, omega, alpha, beta, nu = fp
        lam = 0.0; delta = 0.0; gamma = 0.0
    elif model_type == 'GARCH-M':
        mu, lam, omega, alpha, beta, nu = fp
        delta = 0.0; gamma = 0.0
    elif model_type == 'GARCH-M-L':
        mu, lam, omega, alpha, delta, gamma, beta, nu = fp
    else:
        raise ValueError("Invalid model type specified.")

    x = np.empty((nsims, H+1), dtype=float)
    x[:, 0] = x0
    sigma_sqrd = np.empty((nsims, H+1), dtype=float)
    sigma_sqrd[:, 0] = float(sigma_sqrd_t0)

    eps = rng.standard_t(df=nu, size=(nsims, H))
    tiny = 1e-12

    for h in range(H):
        sig = np.sqrt(np.maximum(sigma_sqrd[:, h], tiny))
        mean_tm1 = mu + lam * sigma_sqrd[:, h]

        z_prev = (x[:, h] - mean_tm1) / np.maximum(sig, tiny)

        if model_type == 'GARCH-M-L':
            arch_coeff = alpha + delta * np.tanh(-gamma * x[:, h])
        else:
            arch_coeff = alpha
```

```
        sigma_sqrd[:, h+1] = np.maximum(omega + arch_coeff*(z_prev**2) +␣
↪beta*sigma_sqrd[:, h], tiny)

        mean_next = mu + lam * sigma_sqrd[:, h+1]
        x[:, h+1] = mean_next + np.sqrt(sigma_sqrd[:, h+1]) * eps[:, h]

    gross   = 1.0 + x[:, 1:] / 100.0
    cumprod = np.cumprod(gross, axis=1)

    comp_1  = x[:, 1]
    comp_5  = 100.0*(cumprod[:, 4]  - 1.0) if H >= 5  else None
    comp_20 = 100.0*(cumprod[:, 19] - 1.0) if H >= 20 else None

    return x, sigma_sqrd, comp_1, comp_5, comp_20
```

[28]:
```
tickers = ['MRK','AAPL','PFE','JNJ']
models  = ['GARCH','GARCH-M','GARCH-M-L']
levels  = [0.01, 0.05, 0.10]

rows = []
for s in tickers:
    for m in models:
        sigma2_t0 = sig_path_results[s][m][-1]   # filtered variance on␣
↪2021-01-04
        _, _, r1, r5, r20 = simulated_path(
            s, m, sigma2_t0,
            H=20, nsims=100000, seed=123
        )
        rows.append({
            "Stock": s, "Model": m,
            "VaR_1d_1%":  float(np.quantile(r1,  0.01)),
            "VaR_1d_5%":  float(np.quantile(r1,  0.05)),
            "VaR_1d_10%": float(np.quantile(r1,  0.10)),
            "VaR_5d_1%":  float(np.quantile(r5,  0.01)),
            "VaR_5d_5%":  float(np.quantile(r5,  0.05)),
            "VaR_5d_10%": float(np.quantile(r5,  0.10)),
            "VaR_20d_1%": float(np.quantile(r20, 0.01)),
            "VaR_20d_5%": float(np.quantile(r20, 0.05)),
            "VaR_20d_10%":float(np.quantile(r20, 0.10)),
        })

var_table = pd.DataFrame(rows).set_index(["Stock","Model"]).sort_index()
var_table.round(2)
```

[28]:
```
                 VaR_1d_1%  VaR_1d_5%  VaR_1d_10%  VaR_5d_1%  VaR_5d_5%  \
Stock Model
```

| | | | | | | |
|---|---|---|---|---|---|---|
| AAPL | GARCH | -4.90 | -2.73 | -1.91 | -9.78 | -5.74 |
| | GARCH-M | -4.91 | -2.73 | -1.91 | -9.42 | -5.57 |
| | GARCH-M-L | -4.68 | -2.71 | -1.91 | -9.92 | -5.85 |
| JNJ | GARCH | -2.21 | -1.43 | -1.07 | -4.74 | -3.00 |
| | GARCH-M | -2.52 | -1.43 | -1.02 | -5.16 | -3.11 |
| | GARCH-M-L | -2.44 | -1.40 | -1.00 | -5.20 | -3.09 |
| MRK | GARCH | -2.98 | -1.73 | -1.24 | -6.36 | -3.87 |
| | GARCH-M | -2.92 | -1.68 | -1.20 | -6.25 | -3.78 |
| | GARCH-M-L | -2.55 | -1.64 | -1.22 | -5.31 | -3.26 |
| PFE | GARCH | -3.24 | -1.90 | -1.37 | -6.67 | -4.11 |
| | GARCH-M | -3.23 | -1.88 | -1.35 | -6.53 | -4.02 |
| | GARCH-M-L | -3.53 | -2.07 | -1.49 | -7.13 | -4.43 |

| | | VaR_5d_10% | VaR_20d_1% | VaR_20d_5% | VaR_20d_10% |
|---|---|---|---|---|---|
| Stock | Model | | | | |
| AAPL | GARCH | -4.08 | -16.90 | -9.88 | -6.75 |
| | GARCH-M | -3.92 | -15.42 | -9.09 | -6.17 |
| | GARCH-M-L | -4.14 | -17.47 | -10.18 | -6.78 |
| JNJ | GARCH | -2.21 | -8.69 | -5.41 | -3.83 |
| | GARCH-M | -2.22 | -9.31 | -5.54 | -3.85 |
| | GARCH-M-L | -2.20 | -9.80 | -5.71 | -3.88 |
| MRK | GARCH | -2.79 | -12.03 | -7.41 | -5.32 |
| | GARCH-M | -2.74 | -11.91 | -7.25 | -5.19 |
| | GARCH-M-L | -2.29 | -8.80 | -4.94 | -3.09 |
| PFE | GARCH | -3.00 | -12.38 | -7.67 | -5.52 |
| | GARCH-M | -2.93 | -11.65 | -7.24 | -5.19 |
| | GARCH-M-L | -3.19 | -12.27 | -7.50 | -5.31 |