Econometrics Group Homework 2

In [1]:
```
import os
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import statsmodels.api as sm
from statsmodels.tsa.api import ExponentialSmoothing, Holt, SimpleExpSmoothing
from statsmodels.tsa.stattools import (
    acf, pacf, q_stat, adfuller, kpss
)
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.statespace.sarimax import SARIMAX
from arch import arch_model
from arch.univariate import EWMAVariance, ConstantMean
from arch.unitroot import PhillipsPerron
from scipy.stats import jarque_bera
from scipy.optimize import minimize
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import seaborn as sns
from scipy.stats import norm
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.stats.diagnostic import acorr_ljungbox
```

In [2]:
```
plt.rcParams['font.size'] = 15
plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.weight'] = 'normal'
```

Functions

In [3]:
```
def SACF_SPACF(series, lag_max = 24, alpha_level = 0.05, model_df = 0):
    """
    Compute the sample autocorrelation function (SACF), sample partial autocorrelation function (SPACF)
    and Ljung-Box Q-statistics for a time series.

    This function calculates the ACF and PACF values along with their corresponding confidence interval
    for lags 1 through `lag_max` using the provided significance level (`alpha_level`). In addition, it
    computes the Ljung-Box Q-statistic and associated p-values (excluding lag 0). Set `model_df`
    to the number of dof lost.

    """

    # Calculate ACF and PACF with confidence intervals
    acf_vals, acf_confint = acf(series, nlags=lag_max, alpha=alpha_level)
    pacf_vals, pacf_confint = pacf(series, nlags=lag_max, alpha=alpha_level, method='ols')

    # Calculate Ljung-Box statistics and p-values
    lb_results = sm.stats.acorr_ljungbox(
        series,
        lags=range(1, lag_max + 1),
        model_df=model_df,
        return_df=True
    )

    # Build the results DataFrame
    df_acf_pacf = pd.DataFrame({
        "Lag": np.arange(1, lag_max + 1),
        "ACF": acf_vals[1:],
        "ACF_lower": acf_confint[1:, 0],
        "ACF_upper": acf_confint[1:, 1],
        "PACF": pacf_vals[1:],
        "PACF_lower": pacf_confint[1:, 0],
        "PACF_upper": pacf_confint[1:, 1],
        "Q-stat": lb_results["lb_stat"].values,
        "Q-stat Prob": lb_results["lb_pvalue"].values.round(6)
    })

    # Set the index to 'Lag' and extract the main columns
    df_acf_pacf.set_index("Lag", inplace=True)
    df_acf_pacf_small = df_acf_pacf[["ACF", "PACF", "Q-stat", "Q-stat Prob"]].copy()

    return df_acf_pacf_small
```

In [4]:
```
def SACF_SPACF_plot (series, lag_max = 24, ylim = [-0.15, 0.15]):
  """
    Generate plots for the Sample Autocorrelation (SACF) and Sample Partial Autocorrelation (SPACF)
    of a time series.

  """
  fig, axes = plt.subplots(1, 2, figsize=(12, 4))

  # Sample Autocorrelation (SACF) Plot
  plot_acf(series, lags=lag_max, ax=axes[0], zero=False)
  axes[0].set_title("SACF")
  axes[0].set_ylim(ylim)

  # Sample Partial Autocorrelation (SPACF) Plot
  plot_pacf(series, lags=lag_max, ax=axes[1], method='ols', zero=False)
  axes[1].set_title("SPACF")
  axes[1].set_ylim(ylim)

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

In [5]:
```
def extract_roots(results):
  ar_roots = results.arroots
  ar_roots_arr = np.array(ar_roots)
  modulus = np.abs(ar_roots_arr)
  return ar_roots_arr, modulus
```

In [6]:
```
def unit_root_tests(series, name="Series"):
    """
    Runs multiple unit-root tests on a given Pandas Series:
      1) ADF (autolag=BIC, constant only)
      2) ADF (autolag=BIC, no constant)
      3) ADF (autolag=AIC, constant only)
      4) ADF (exactly 12 lags, constant only)
      5) ADF (autolag=BIC, constant + linear trend)
      6) ADF (autolag=BIC, constant + linear + quadratic trend)
```

```
  7) Phillips–Perron (constant only)
  8) KPSS (constant only)
"""

def format_crit_vals(crit_dict):
  parts = []
  for k, v in crit_dict.items():
      formatted = f"{k}: {v:.4f}"
      parts.append(formatted)
  result = "; ".join(parts)
  return result

results_list = []

print(f"\n=== UNIT-ROOT TESTS FOR: {name} ===\n")

###############################################################################
# 1) ADF Test (BIC, constant only)
###############################################################################
adf_bic = adfuller(series, autolag='BIC', regression='c')
print("=== ADF Test (BIC, constant only) ===")
print(f"ADF statistic: {adf_bic[0]:.4f}")
print(f"p-value:       {adf_bic[1]:.4f}")
print(f"Lags used:     {adf_bic[2]}")
print("Critical values:")
for k, v in adf_bic[4].items():
    print(f"    {k}: {v:.4f}")
print(f"IC Best (BIC): {adf_bic[5]:.4f}\n")

results_list.append({
    "Test": "ADF (BIC, c)",
    "ADF stat": adf_bic[0],
    "p-value": adf_bic[1],
    "Lags used": adf_bic[2],
    "IC Best": adf_bic[5],
    "Critical values": format_crit_vals(adf_bic[4])
})

###############################################################################
# 2) ADF Test (BIC, no constant)
###############################################################################
adf_bic_nc = adfuller(series, autolag='BIC', regression='n')
print("=== ADF Test (BIC, no constant) ===")
print(f"ADF statistic: {adf_bic_nc[0]:.4f}")
print(f"p-value:       {adf_bic_nc[1]:.4f}")
print(f"Lags used:     {adf_bic_nc[2]}")
print("Critical values:")
for k, v in adf_bic[4].items():
    print(f"    {k}: {v:.4f}")
print(f"IC Best (BIC): {adf_bic[5]:.4f}\n")

results_list.append({
    "Test": "ADF (BIC, nc)",
    "ADF stat": adf_bic_nc[0],
    "p-value": adf_bic_nc[1],
    "Lags used": adf_bic_nc[2],
    "IC Best": adf_bic_nc[5],
    "Critical values": format_crit_vals(adf_bic_nc[4])
})

###############################################################################
# 3) ADF Test (AIC, constant only)
###############################################################################
adf_aic = adfuller(series, autolag='AIC', regression='c')
print("=== ADF Test (AIC, constant only) ===")
print(f"ADF statistic: {adf_aic[0]:.4f}")
print(f"p-value:       {adf_aic[1]:.4f}")
print(f"Lags used:     {adf_aic[2]}")
print("Critical values:")
for k, v in adf_aic[4].items():
    print(f"    {k}: {v:.4f}")
print(f"IC Best (AIC): {adf_aic[5]:.4f}\n")

results_list.append({
    "Test": "ADF (AIC, c)",
    "ADF stat": adf_aic[0],
    "p-value": adf_aic[1],
    "Lags used": adf_aic[2],
    "IC Best": adf_aic[5],
    "Critical values": format_crit_vals(adf_aic[4])
})

###############################################################################
```

```
# 4) ADF Test (exactly 12 lags, constant only)
###########################################################################
adf_12 = adfuller(series, maxlag=12, autolag=None, regression='c')
print("=== ADF Test (exactly 12 lags, constant only) ===")
print(f"ADF statistic: {adf_12[0]:.4f}")
print(f"p-value:       {adf_12[1]:.4f}")
print(f"Lags used:     {adf_12[2]}")
print("Critical values:")
for k, v in adf_12[4].items():
    print(f"    {k}: {v:.4f}")
print()

results_list.append({
    "Test": "ADF (10 lags, c)",
    "ADF stat": adf_12[0],
    "p-value": adf_12[1],
    "Lags used": adf_12[2],
    "IC Best": None,
    "Critical values": format_crit_vals(adf_12[4])
})

###########################################################################
# 5) ADF Test (BIC, constant + linear trend)
###########################################################################
adf_trend_l = adfuller(series, autolag='BIC', regression='ct')
print("=== ADF Test (BIC, constant + linear trend) ===")
print(f"ADF statistic: {adf_trend_l[0]:.4f}")
print(f"p-value:       {adf_trend_l[1]:.4f}")
print(f"Lags used:     {adf_trend_l[2]}")
print("Critical values:")
for k, v in adf_trend_l[4].items():
    print(f"    {k}: {v:.4f}")
print(f"IC Best (BIC): {adf_trend_l[5]:.4f}\n")

results_list.append({
    "Test": "ADF (BIC, c+t)",
    "ADF stat": adf_trend_l[0],
    "p-value": adf_trend_l[1],
    "Lags used": adf_trend_l[2],
    "IC Best": adf_trend_l[5],
    "Critical values": format_crit_vals(adf_trend_l[4])
})

###########################################################################
# 6) ADF Test (BIC, constant + linear + quadratic trend)
###########################################################################
adf_trend_q = adfuller(series, autolag='BIC', regression='ctt')
print("=== ADF Test (BIC, constant + linear + quadratic trend) ===")
print(f"ADF statistic: {adf_trend_q[0]:.4f}")
print(f"p-value:       {adf_trend_q[1]:.4f}")
print(f"Lags used:     {adf_trend_q[2]}")
print("Critical values:")
for k, v in adf_trend_q[4].items():
    print(f"    {k}: {v:.4f}")
print(f"IC Best (BIC): {adf_trend_q[5]:.4f}")

results_list.append({
    "Test": "ADF (BIC, c+t+q)",
    "ADF stat": adf_trend_q[0],
    "p-value": adf_trend_q[1],
    "Lags used": adf_trend_q[2],
    "IC Best": adf_trend_q[5],
    "Critical values": format_crit_vals(adf_trend_q[4])
})

print()

###########################################################################
# 7) Phillips–Perron Test (constant only)
###########################################################################
pp_test = PhillipsPerron(series, trend='c')
print("=== Phillips–Perron Test ===")
print(f"PP Statistic: {pp_test.stat:.4f}")
print(f"p-value:      {pp_test.pvalue:.4f}")
print("Critical values:")
# 'critical_values' is a dict
for k, v in pp_test.critical_values.items():
    print(f"    {k}: {v:.4f}")

results_list.append({
    "Test": "Phillips–Perron (c)",
    "ADF stat": pp_test.stat,
    "p-value": pp_test.pvalue,
```

```
        "Lags used": None,
        "IC Best": None,
        "Critical values": format_crit_vals(pp_test.critical_values)
    })

    print()

    ############################################################################
    # 8) KPSS Test (constant only)
    ############################################################################
    kpss_stat, kpss_pvalue, kpss_lags, kpss_critvals = kpss(series, regression='c')
    print("=== KPSS Test ===")
    print(f"Test Statistic: {kpss_stat:.4f}")
    print(f"p-value:        {kpss_pvalue:.4f}")
    print(f"Lags used:      {kpss_lags}")
    print("Critical Values:", kpss_critvals)
    print()

    results_list.append({
        "Test": "KPSS (c)",
        "ADF stat": kpss_stat,
        "p-value": kpss_pvalue,
        "Lags used": kpss_lags,
        "IC Best": None,
        "Critical values": format_crit_vals(kpss_critvals)
    })

    results_df = pd.DataFrame(results_list)

    return results_df
```

Task 1: Plot and descriptive statistics

In [7]:
```
df = pd.read_excel('Data HW2.xlsx', sheet_name='Subset')
df.rename(columns={
    'Date': 'date',
    'Price': 'price_index',
    'Dividend': 'dividend_index'
}, inplace=True)
df['date'] = pd.to_datetime(df['date'])
df.set_index('date', inplace = True)
df
```

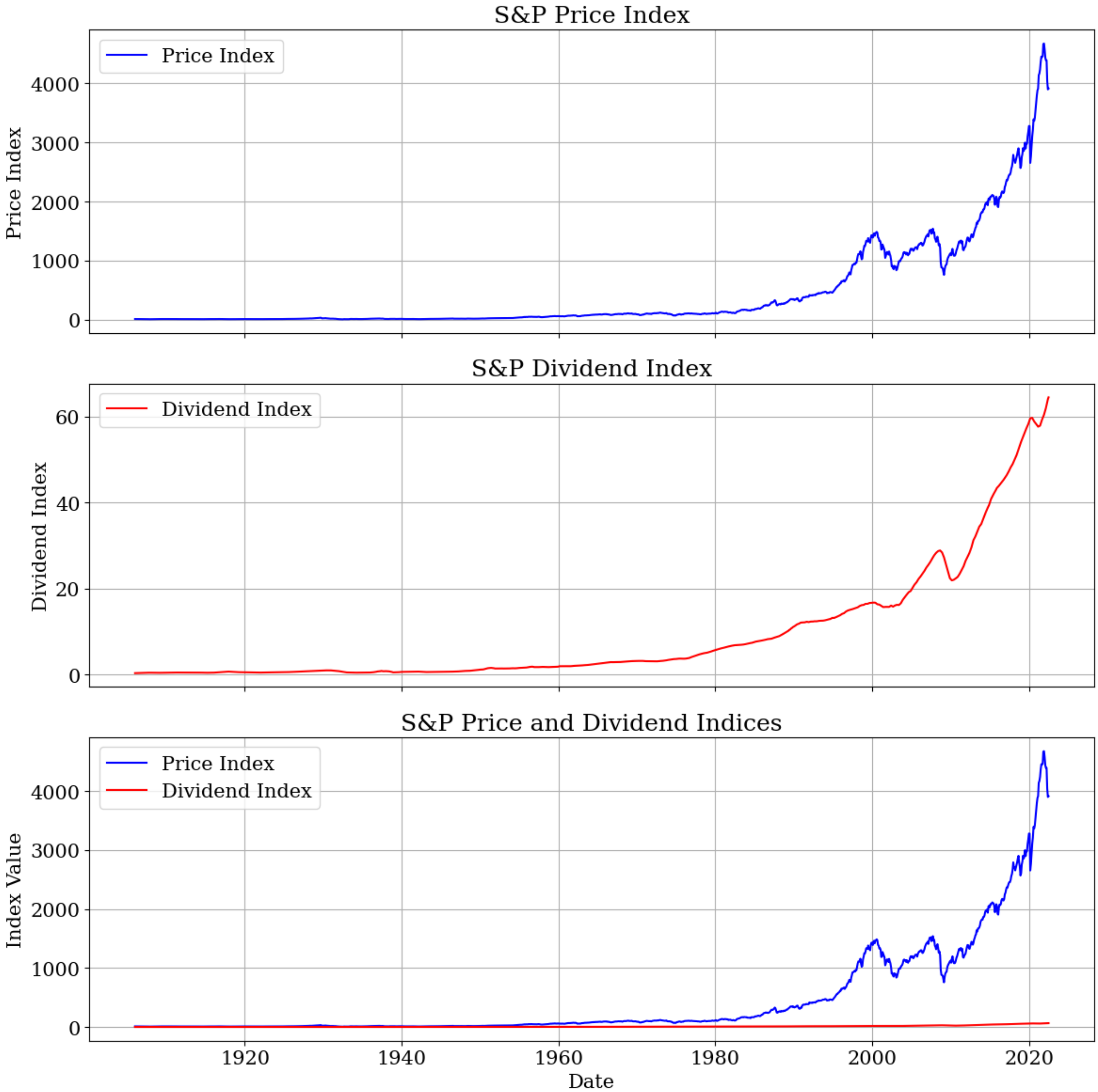
Out[7]:

date	price_index	dividend_index
1906-01-01	9.870000	0.335800
1906-02-01	9.800000	0.341700
1906-03-01	9.560000	0.347500
1906-04-01	9.430000	0.353300
1906-05-01	9.180000	0.359200
...	...	...
2022-03-01	4391.265217	61.969974
2022-04-01	4391.296000	62.653316
2022-05-01	4040.360000	63.336658
2022-06-01	3898.946667	64.020000
2022-07-01	3911.729500	64.452768

1399 rows × 2 columns

In [8]:
```
fig, axs = plt.subplots(3, 1, figsize=(12, 12), sharex=True)

# Plot 1: Price Index
axs[0].plot(df.index, df['price_index'], color='blue', label='Price Index')
axs[0].set_ylabel('Price Index')
axs[0].set_title('S&P Price Index')
axs[0].legend()
axs[0].grid()
```

```
# Plot 2: Dividend Index
axs[1].plot(df.index, df['dividend_index'], color='red', label='Dividend Index')
axs[1].set_ylabel('Dividend Index')
axs[1].set_title('S&P Dividend Index')
axs[1].legend()
axs[1].grid()

# Plot 3: Combined
axs[2].plot(df.index, df['price_index'], color='blue', label='Price Index')
axs[2].plot(df.index, df['dividend_index'], color='red', label='Dividend Index')
axs[2].set_xlabel('Date')
axs[2].set_ylabel('Index Value')
axs[2].set_title('S&P Price and Dividend Indices')
axs[2].legend()
axs[2].grid()

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



In [9]:
```
# Summary statistics price index
summary = df['price_index'].describe()
skewness = df['price_index'].skew()
kurtosis = (df['price_index'].kurt() + 3)

additional_stats = pd.DataFrame({'skewness': [skewness], 'kurtosis': [kurtosis]})

additional_stats = additional_stats.applymap(lambda x: '{:.6f}'.format(x))

all_stats_price = pd.concat([summary, additional_stats.T], axis=0)
pd.set_option('display.float_format', '{:.4f}'.format)
```

In [10]:
```
all_stats_price
```

Out[10]:

	0
count	1399.0000
mean	451.0885
std	817.5087
min	4.7700
25%	12.0450
50%	73.0300
75%	447.2600
max	4674.7727
skewness	2.536466
kurtosis	9.950432

In [11]:
```
# Summary statistics dividend index
summary = df['dividend_index'].describe()
skewness = df['dividend_index'].skew()
kurtosis = (df['dividend_index'].kurt() + 3)

additional_stats = pd.DataFrame({'skewness': [skewness], 'kurtosis': [kurtosis]})

additional_stats = additional_stats.applymap(lambda x: '{:.6f}'.format(x))

all_stats_dividend = pd.concat([summary, additional_stats.T], axis=0)
pd.set_option('display.float_format', '{:.4f}'.format)
```

In [12]:
```
all_stats_dividend
```

Out[12]:

	0
count	1399.0000
mean	9.1810
std	14.0380
min	0.3358
25%	0.6550
50%	2.3467
75%	12.5133
max	64.4528
skewness	2.138757
kurtosis	7.071887

In [13]:
```
correlation = df['price_index'].corr(df['dividend_index'])
print(f"\nCorrelation between Price and Dividend Index: {correlation:.3f}")
```

```
Correlation between Price and Dividend Index: 0.975
```

The Price Index and Dividend Index exhibit strong co-movement, as indicated by their high correlation of 0.975. This suggests that as the Price Index rises or falls, the Dividend Index tends to follow a similar trend. We would expect these two series to move together because stock prices and dividends are fundamentally linked. Indeed, stock prices are usually computed as the present value of future expected dividends (which, in turn, are often a reflection of a company's profitability and financial health). Therefore, if a company raises its dividend payout, the stock price will almost automatically go up, both due to an increase in its fundamental value (i.e. the present value of future dividends) and due to the fact that increasing dividends is usually interpreted by investors as a positive signal of higher future profitability of the company itself.

Task 2: Stationarity and order of integration PRICE INDEX

In [14]:
```
unit_root_tests(df['price_index'], "Price Index")
```

```
=== UNIT-ROOT TESTS FOR: Price Index ===

=== ADF Test (BIC, constant only) ===
ADF statistic: 3.5767
p-value:       1.0000
Lags used:     13
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679
IC Best (BIC): 13548.0303

=== ADF Test (BIC, no constant) ===
ADF statistic: 4.1303
p-value:       1.0000
Lags used:     13
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679
IC Best (BIC): 13548.0303

=== ADF Test (AIC, constant only) ===
ADF statistic: 4.5498
p-value:       1.0000
Lags used:     24
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679
IC Best (AIC): 13438.2810

=== ADF Test (exactly 12 lags, constant only) ===
ADF statistic: 2.1535
p-value:       0.9988
Lags used:     12
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679

=== ADF Test (BIC, constant + linear trend) ===
ADF statistic: 2.0389
p-value:       1.0000
Lags used:     13
Critical values:
    1%: -3.9653
    5%: -3.4137
    10%: -3.1289
IC Best (BIC): 13554.6220

=== ADF Test (BIC, constant + linear + quadratic trend) ===
ADF statistic: -0.2143
p-value:       0.9984
Lags used:     13
Critical values:
    1%: -4.3795
    5%: -3.8367
    10%: -3.5559
IC Best (BIC): 13558.4152

=== Phillips-Perron Test ===
PP Statistic: 4.2456
p-value:      1.0000
Critical values:
```

```
    1%: -3.4350
    5%: -2.8636
    10%: -2.5679

=== KPSS Test ===
Test Statistic: 3.2979
p-value:        0.0100
Lags used:      24
Critical Values: {'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.739}


/var/folders/zw/v8b7q_qx17l227tjwg7pz2p40000gn/T/ipykernel_46556/4126655903.py:186: InterpolationWarni
ng: The test statistic is outside of the range of p-values available in the
look-up table. The actual p-value is smaller than the p-value returned.

  kpss_stat, kpss_pvalue, kpss_lags, kpss_critvals = kpss(series, regression='c')
```

Out[14]:

	Test	ADF stat	p-value	Lags used	IC Best	Critical values
0	ADF (BIC, c)	3.5767	1.0000	13.0000	13548.0303	1%: -3.4351; 5%: -2.8636; 10%: -2.5679
1	ADF (BIC, nc)	4.1303	1.0000	13.0000	13540.8810	1%: -2.5674; 5%: -1.9412; 10%: -1.6166
2	ADF (AIC, c)	4.5498	1.0000	24.0000	13438.2810	1%: -3.4351; 5%: -2.8636; 10%: -2.5679
3	ADF (10 lags, c)	2.1535	0.9988	12.0000	NaN	1%: -3.4351; 5%: -2.8636; 10%: -2.5679
4	ADF (BIC, c+t)	2.0389	1.0000	13.0000	13554.6220	1%: -3.9653; 5%: -3.4137; 10%: -3.1289
5	ADF (BIC, c+t+q)	-0.2143	0.9984	13.0000	13558.4152	1%: -4.3795; 5%: -3.8367; 10%: -3.5559
6	Phillips–Perron (c)	4.2456	1.0000	NaN	NaN	1%: -3.4350; 5%: -2.8636; 10%: -2.5679
7	KPSS (c)	3.2979	0.0100	24.0000	NaN	10%: 0.3470; 5%: 0.4630; 2.5%: 0.5740; 1%: 0.7390

In [15]:

```python
differenced_price= df['price_index'] - df['price_index'].shift(1)
differenced_price.dropna(inplace=True)
differenced_price.head()
```

Out[15]:

```
date
1906-02-01   -0.0700
1906-03-01   -0.2400
1906-04-01   -0.1300
1906-05-01   -0.2500
1906-06-01    0.1200
Name: price_index, dtype: float64
```

In [16]:

```python
unit_root_df_price = unit_root_tests(differenced_price,"Price Index - First difference")
```

```
=== UNIT-ROOT TESTS FOR: Price Index - First difference ===

=== ADF Test (BIC, constant only) ===
ADF statistic: -9.7434
p-value:       0.0000
Lags used:     12
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679
IC Best (BIC): 13544.7302

=== ADF Test (BIC, no constant) ===
ADF statistic: -9.5179
p-value:       0.0000
Lags used:     12
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679
IC Best (BIC): 13544.7302

=== ADF Test (AIC, constant only) ===
ADF statistic: -8.1204
p-value:       0.0000
Lags used:     22
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679
IC Best (AIC): 13448.0267
```

```
=== ADF Test (exactly 12 lags, constant only) ===
ADF statistic: -9.7434
p-value:       0.0000
Lags used:     12
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679

=== ADF Test (BIC, constant + linear trend) ===
ADF statistic: -10.2068
p-value:       0.0000
Lags used:     12
Critical values:
    1%: -3.9653
    5%: -3.4137
    10%: -3.1289
IC Best (BIC): 13542.6957

=== ADF Test (BIC, constant + linear + quadratic trend) ===
ADF statistic: -10.5379
p-value:       0.0000
Lags used:     12
Critical values:
    1%: -4.3795
    5%: -3.8367
    10%: -3.5559
IC Best (BIC): 13542.4525

=== Phillips-Perron Test ===
PP Statistic: -31.8851
p-value:      0.0000
Critical values:
    1%: -3.4350
    5%: -2.8636
    10%: -2.5679

=== KPSS Test ===
Test Statistic: 0.9740
p-value:        0.0100
Lags used:      14
Critical Values: {'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.739}


/var/folders/zw/v8b7q_qx17l227tjwg7pz2p40000gn/T/ipykernel_46556/4126655903.py:186: InterpolationWarni
ng: The test statistic is outside of the range of p-values available in the
look-up table. The actual p-value is smaller than the p-value returned.

  kpss_stat, kpss_pvalue, kpss_lags, kpss_critvals = kpss(series, regression='c')
```

The ADF, Phillips-Perron, and KPSS tests confirm the price index series is non-stationary. The ADF and PP test p-values are all equal or very close to 1, failing to reject the unit root null hypothesis. The KPSS statistic (3.2979, p-value = 0.01) exceeds critical values (e.g., 1%: 0.7390), rejecting stationarity at all selected confidence levels. All tests thus agree. To examine the presence of a unit root, we compute the first difference (price at t minus price at t−1) and run the same tests. The ADF and PP p-values for the differenced series are all zero, rejecting non-stationarity. However, the KPSS p-value remains 0.01, rejecting the null of stationarity. Despite this contradiction, we conclude the differenced price index is stationary, and thus the price index is integrated of order 1 (I(1)).

Task 3: Stationarity and order of integration DIVIDEND INDEX

In [17]:
```
unit_root_tests(df['dividend_index'], "Dividend Index")
```

```
=== UNIT-ROOT TESTS FOR: Dividend Index ===

=== ADF Test (BIC, constant only) ===
ADF statistic: 5.7498
p-value:       1.0000
Lags used:     19
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679
IC Best (BIC): -5455.9281

=== ADF Test (BIC, no constant) ===
ADF statistic: 6.1756
p-value:       1.0000
```

```
Lags used:     19
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679
IC Best (BIC): -5455.9281

=== ADF Test (AIC, constant only) ===
ADF statistic: 6.5391
p-value:       1.0000
Lags used:     24
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679
IC Best (AIC): -5569.7506

=== ADF Test (exactly 12 lags, constant only) ===
ADF statistic: 7.0874
p-value:       1.0000
Lags used:     12
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679

=== ADF Test (BIC, constant + linear trend) ===
ADF statistic: 4.2886
p-value:       1.0000
Lags used:     19
Critical values:
    1%: -3.9653
    5%: -3.4137
    10%: -3.1289
IC Best (BIC): -5448.7095

=== ADF Test (BIC, constant + linear + quadratic trend) ===
ADF statistic: 1.8552
p-value:       1.0000
Lags used:     19
Critical values:
    1%: -4.3796
    5%: -3.8367
    10%: -3.5559
IC Best (BIC): -5442.3166

=== Phillips–Perron Test ===
PP Statistic: 7.9456
p-value:      1.0000
Critical values:
    1%: -3.4350
    5%: -2.8636
    10%: -2.5679

=== KPSS Test ===
Test Statistic: 3.6552
p-value:        0.0100
Lags used:      24
Critical Values: {'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.739}


/var/folders/zw/v8b7q_qx17l227tjwg7pz2p40000gn/T/ipykernel_46556/4126655903.py:186: InterpolationWarni
ng: The test statistic is outside of the range of p-values available in the
look-up table. The actual p-value is smaller than the p-value returned.

  kpss_stat, kpss_pvalue, kpss_lags, kpss_critvals = kpss(series, regression='c')
```

Out[17]:

	Test	ADF stat	p-value	Lags used	IC Best	Critical values
0	ADF (BIC, c)	5.7498	1.0000	19.0000	-5455.9281	1%: -3.4351; 5%: -2.8636; 10%: -2.5679
1	ADF (BIC, nc)	6.1756	1.0000	19.0000	-5463.0778	1%: -2.5674; 5%: -1.9412; 10%: -1.6166
2	ADF (AIC, c)	6.5391	1.0000	24.0000	-5569.7506	1%: -3.4351; 5%: -2.8636; 10%: -2.5679
3	ADF (10 lags, c)	7.0874	1.0000	12.0000	NaN	1%: -3.4351; 5%: -2.8636; 10%: -2.5679
4	ADF (BIC, c+t)	4.2886	1.0000	19.0000	-5448.7095	1%: -3.9653; 5%: -3.4137; 10%: -3.1289
5	ADF (BIC, c+t+q)	1.8552	1.0000	19.0000	-5442.3166	1%: -4.3796; 5%: -3.8367; 10%: -3.5559
6	Phillips–Perron (c)	7.9456	1.0000	NaN	NaN	1%: -3.4350; 5%: -2.8636; 10%: -2.5679

7	KPSS (c)	3.6552	0.0100	24.0000	NaN	10%: 0.3470; 5%: 0.4630; 2.5%: 0.5740; 1%: 0.7390

In [18]:
```
differenced_dividend= df['dividend_index'] - df['dividend_index'].shift(1)
differenced_dividend.dropna(inplace=True)
differenced_dividend.head()
```

Out[18]:
```
date
1906-02-01   0.0059
1906-03-01   0.0058
1906-04-01   0.0058
1906-05-01   0.0059
1906-06-01   0.0058
Name: dividend_index, dtype: float64
```

In [19]:
```
unit_root_df_dividend = unit_root_tests(differenced_dividend,"Dividend Index - First difference")
```

```
=== UNIT-ROOT TESTS FOR: Dividend Index - First difference ===

=== ADF Test (BIC, constant only) ===
ADF statistic: -3.1904
p-value:       0.0205
Lags used:     24
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679
IC Best (BIC): -5505.2402

=== ADF Test (BIC, no constant) ===
ADF statistic: -2.7051
p-value:       0.0066
Lags used:     24
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679
IC Best (BIC): -5505.2402

=== ADF Test (AIC, constant only) ===
ADF statistic: -3.1904
p-value:       0.0205
Lags used:     24
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679
IC Best (AIC): -5641.0838

=== ADF Test (exactly 12 lags, constant only) ===
ADF statistic: -4.8967
p-value:       0.0000
Lags used:     12
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679

=== ADF Test (BIC, constant + linear trend) ===
ADF statistic: -4.3431
p-value:       0.0027
Lags used:     24
Critical values:
    1%: -3.9654
    5%: -3.4137
    10%: -3.1289
IC Best (BIC): -5507.0141

=== ADF Test (BIC, constant + linear + quadratic trend) ===
ADF statistic: -5.4056
p-value:       0.0002
Lags used:     24
Critical values:
    1%: -4.3796
    5%: -3.8367
    10%: -3.5559
IC Best (BIC): -5510.2635

=== Phillips-Perron Test ===
PP Statistic: -5.0716
p-value:      0.0000
```

```
Critical values:
    1%: -3.4350
    5%: -2.8636
    10%: -2.5679

=== KPSS Test ===
Test Statistic: 1.9654
p-value:        0.0100
Lags used:      24
Critical Values: {'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.739}


/var/folders/zw/v8b7q_qx17l227tjwg7pz2p40000gn/T/ipykernel_46556/4126655903.py:186: InterpolationWarni
ng: The test statistic is outside of the range of p-values available in the
look-up table. The actual p-value is smaller than the p-value returned.

  kpss_stat, kpss_pvalue, kpss_lags, kpss_critvals = kpss(series, regression='c')
```

The ADF, Phillips-Perron, and KPSS tests confirm that the dividend index series is non-stationary. The ADF and PP test p-values are all equal or extremely close to 1, failing to reject the unit root null hypothesis. The KPSS statistic (3.6552, p-value = 0.01) exceeds critical values (e.g., 1%: 0.739), rejecting stationarity at all selected confidence levels. All tests therefore agree. As before, to check for a unit root, we compute the first difference of the dividend index and run the same tests. The ADF and PP test p-values for the differenced series are all equal or very close to zero, thus rejecting non-stationarity. However, the KPSS p-value remains 0.01, rejecting the null of stationarity. Despite this contradiction, we again conclude that the differenced dividend index is stationary, so also the dividend index is integrated of order 1 (I(1)).

Task 4: Regression of price index on dividend index

In [20]:
```python
y = df['price_index']
X = df['dividend_index']
X_with_constant = sm.add_constant(X)
model = sm.OLS(y, X_with_constant)
results = model.fit()
print(results.summary())
```

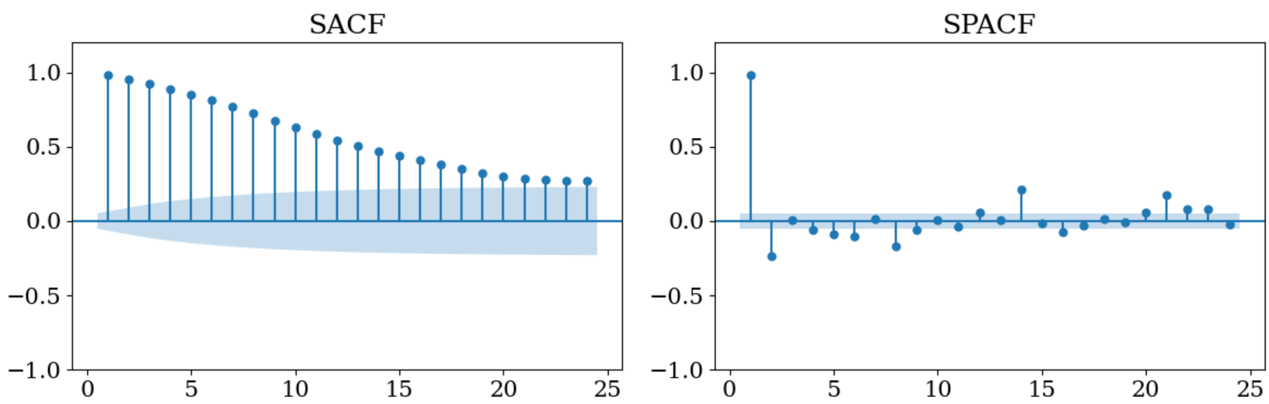
```
                            OLS Regression Results
==============================================================================
Dep. Variable:            price_index   R-squared:                       0.950
Model:                            OLS   Adj. R-squared:                  0.950
Method:                 Least Squares   F-statistic:                 2.639e+04
Date:                Thu, 17 Apr 2025   Prob (F-statistic):               0.00
Time:                        16:08:48   Log-Likelihood:                -9275.1
No. Observations:                1399   AIC:                         1.855e+04
Df Residuals:                    1397   BIC:                         1.856e+04
Df Model:                           1
Covariance Type:            nonrobust
==================================================================================
                     coef    std err          t      P>|t|      [0.025      0.975]
----------------------------------------------------------------------------------
const            -69.9526      5.859    -11.940      0.000     -81.446     -58.459
dividend_index    56.7524      0.349    162.438      0.000      56.067      57.438
==============================================================================
Omnibus:                      729.208   Durbin-Watson:                   0.036
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            11487.647
Skew:                           2.047   Prob(JB):                         0.00
Kurtosis:                      16.428   Cond. No.                         20.1
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

In [21]:
```
residuals = results.resid
residuals_acf_pacf = SACF_SPACF(residuals)
residuals_acf_pacf
```

Out[21]:

Lag	ACF	PACF	Q-stat	Q-stat Prob
1	0.9807	0.9829	1348.3956	0.0000
2	0.9531	-0.2381	2622.8558	0.0000
3	0.9232	0.0072	3819.3753	0.0000
4	0.8893	-0.0587	4930.4513	0.0000
5	0.8527	-0.0854	5952.6533	0.0000
6	0.8122	-0.1050	6880.8930	0.0000
7	0.7710	0.0168	7717.8597	0.0000
8	0.7254	-0.1692	8459.3789	0.0000
9	0.6778	-0.0589	9107.2679	0.0000
10	0.6323	0.0077	9671.4921	0.0000
11	0.5872	-0.0351	10158.3731	0.0000
12	0.5444	0.0565	10577.2476	0.0000
13	0.5037	0.0054	10936.0082	0.0000
14	0.4706	0.2087	11249.4118	0.0000
15	0.4404	-0.0145	11524.1424	0.0000
16	0.4091	-0.0760	11761.2923	0.0000
17	0.3789	-0.0323	11964.9549	0.0000
18	0.3505	0.0128	12139.2740	0.0000
19	0.3251	-0.0043	12289.4098	0.0000
20	0.3035	0.0548	12420.2965	0.0000
21	0.2893	0.1751	12539.3150	0.0000
22	0.2806	0.0783	12651.3636	0.0000
23	0.2743	0.0803	12758.5285	0.0000
24	0.2678	-0.0234	12860.7866	0.0000

In [22]:
```
SACF_SPACF_plot(residuals, ylim=[-1,1.2])
```



In [23]:
```
unit_root_tests(residuals, "Residuals")
```

```
=== UNIT-ROOT TESTS FOR: Residuals ===

=== ADF Test (BIC, constant only) ===
ADF statistic: -4.5793
```

```
p-value:       0.0001
Lags used:     13
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679
IC Best (BIC): 13582.7758

=== ADF Test (BIC, no constant) ===
ADF statistic: -4.5854
p-value:       0.0000
Lags used:     13
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679
IC Best (BIC): 13582.7758

=== ADF Test (AIC, constant only) ===
ADF statistic: -3.0274
p-value:       0.0324
Lags used:     24
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679
IC Best (AIC): 13460.0594

=== ADF Test (exactly 12 lags, constant only) ===
ADF statistic: -5.9683
p-value:       0.0000
Lags used:     12
Critical values:
    1%: -3.4351
    5%: -2.8636
    10%: -2.5679

=== ADF Test (BIC, constant + linear trend) ===
ADF statistic: -4.6421
p-value:       0.0009
Lags used:     13
Critical values:
    1%: -3.9653
    5%: -3.4137
    10%: -3.1289
IC Best (BIC): 13589.1743

=== ADF Test (BIC, constant + linear + quadratic trend) ===
ADF statistic: -4.6823
p-value:       0.0034
Lags used:     13
Critical values:
    1%: -4.3795
    5%: -3.8367
    10%: -3.5559
IC Best (BIC): 13595.5381

=== Phillips–Perron Test ===
PP Statistic: -4.3479
p-value:      0.0004
Critical values:
    1%: -3.4350
    5%: -2.8636
    10%: -2.5679

=== KPSS Test ===
Test Statistic: 0.3640
p-value:        0.0927
Lags used:      24
Critical Values: {'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.739}
```

Out[23]:

	Test	ADF stat	p-value	Lags used	IC Best	Critical values
0	ADF (BIC, c)	-4.5793	0.0001	13.0000	13582.7758	1%: -3.4351; 5%: -2.8636; 10%: -2.5679
1	ADF (BIC, nc)	-4.5854	0.0000	13.0000	13575.5534	1%: -2.5674; 5%: -1.9412; 10%: -1.6166
2	ADF (AIC, c)	-3.0274	0.0324	24.0000	13460.0594	1%: -3.4351; 5%: -2.8636; 10%: -2.5679
3	ADF (10 lags, c)	-5.9683	0.0000	12.0000	NaN	1%: -3.4351; 5%: -2.8636; 10%: -2.5679
4	ADF (BIC, c+t)	-4.6421	0.0009	13.0000	13589.1743	1%: -3.9653; 5%: -3.4137; 10%: -3.1289

5	ADF (BIC, c+t+q)	-4.6823	0.0034	13.0000	13595.5381	1%: -4.3795; 5%: -3.8367; 10%: -3.5559
6	Phillips–Perron (c)	-4.3479	0.0004	NaN	NaN	1%: -3.4350; 5%: -2.8636; 10%: -2.5679
7	KPSS (c)	0.3640	0.0927	24.0000	NaN	10%: 0.3470; 5%: 0.4630; 2.5%: 0.5740; 1%: 0.7390

The regression of the price index on the dividend index shows a high R^2 (0.950) and statistically significant coefficients, suggesting a strong relationship. However, both series are non-stationary and integrated of order 1 (I(1)), raising concerns about spurious regression and, therefore, about the validity of the results of the regression. This concern is reinforced by the very low Durbin-Watson statistic (0.036), indicating strong autocorrelation in the residuals.

To assess the validity of the regression, we test the residuals for stationarity. The results of the ADF tests (e.g., statistic: -4.5793, p-value: 0.0001) and the Phillips-Perron test (statistic: -4.3479, p-value: 0.0004) strongly reject the null of a unit root. Meanwhile, the KPSS test fails to reject the null of stationarity (statistic: 0.3640, p = 0.0927). Although SACF and SPACF plots do not resemble those of a white noise process, the results of the tests suggest that the residuals are stationary, indicating the presence of cointegration, and thus a valid long-run equilibrium relationship, between the two variables.

However, due to residual autocorrelation, which violates the OLS assumptions, we recommend modeling the autocorrelation structure explicitly (e.g., with ARIMA errors) or applying heteroskedasticity and autocorrelation-consistent (HAC) standard errors. In addition, it might also be useful to rerun the stationarity tests and regression with the differenced series (i.e., using Δprice_index and Δdividend_index) to achieve stationarity, as both are I(1).

Task 5: Log returns

In [24]:
```
df['price_index_log_returns'] = np.log(df['price_index'] / df['price_index'].shift(1))
```

In [25]:
```
df
```

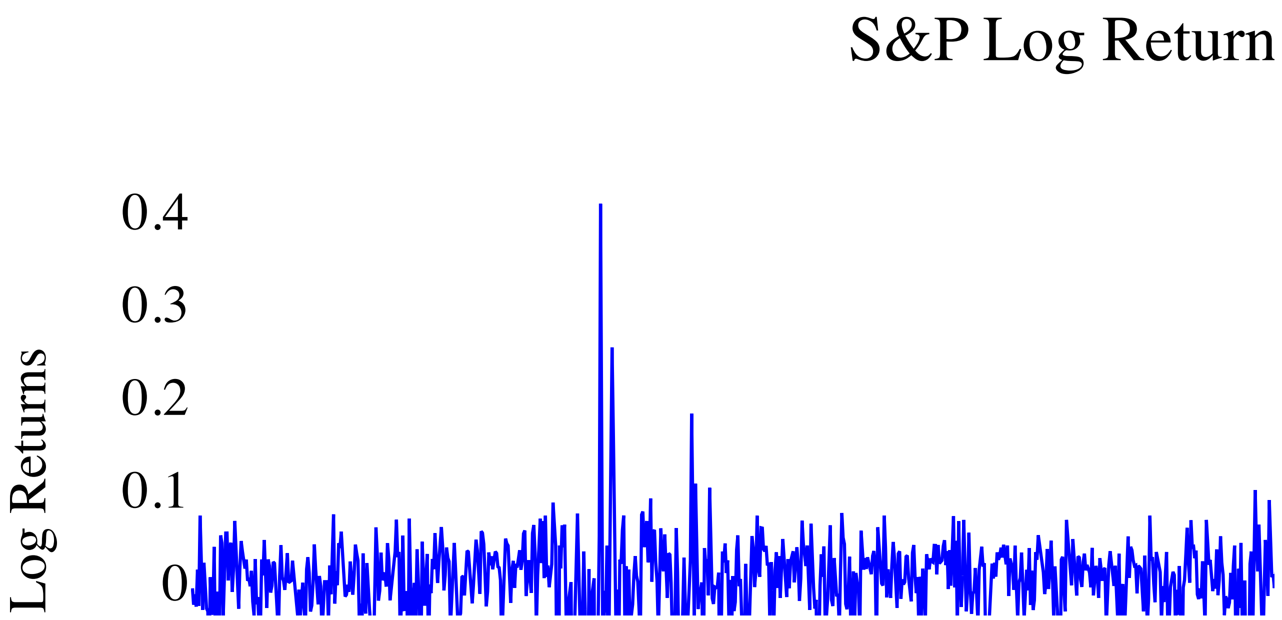
Out[25]:

date	price_index	dividend_index	price_index_log_returns
1906-01-01	9.8700	0.3358	NaN
1906-02-01	9.8000	0.3417	-0.0071
1906-03-01	9.5600	0.3475	-0.0248
1906-04-01	9.4300	0.3533	-0.0137
1906-05-01	9.1800	0.3592	-0.0269
...	...	...	...
2022-03-01	4391.2652	61.9700	-0.0101
2022-04-01	4391.2960	62.6533	0.0000
2022-05-01	4040.3600	63.3367	-0.0833
2022-06-01	3898.9467	64.0200	-0.0356
2022-07-01	3911.7295	64.4528	0.0033

1399 rows × 3 columns

In [26]:
```
fig_returns = go.Figure()
fig_returns.add_trace(go.Scatter(
    x=df.index, y=df['price_index_log_returns'],
    mode='lines',
    name='Log Returns',
    line=dict(color='blue', width=1.5),
))
fig_returns.update_layout(
    title=dict(
        text='S&P Log Returns',
        x=0.5,
        font=dict(size=30, family='Serif', color='black')
    ),

    plot_bgcolor='rgba(0,0,0,0)',

    paper_bgcolor='rgba(0,0,0,0)',

    xaxis=dict(
```

```
        title='Date',
        titlefont=dict(size=25),
        tickfont=dict(size=25),
        showgrid=False
    ),
    yaxis=dict(
        title='S&P Log Returns',
        titlefont=dict(size=25),
        tickfont=dict(size=25),
        showgrid=False,
        zeroline=False
    ),
    font=dict(
        family="Serif",
        size=25,
        color="black"
    ),
)

fig_returns.show()
```



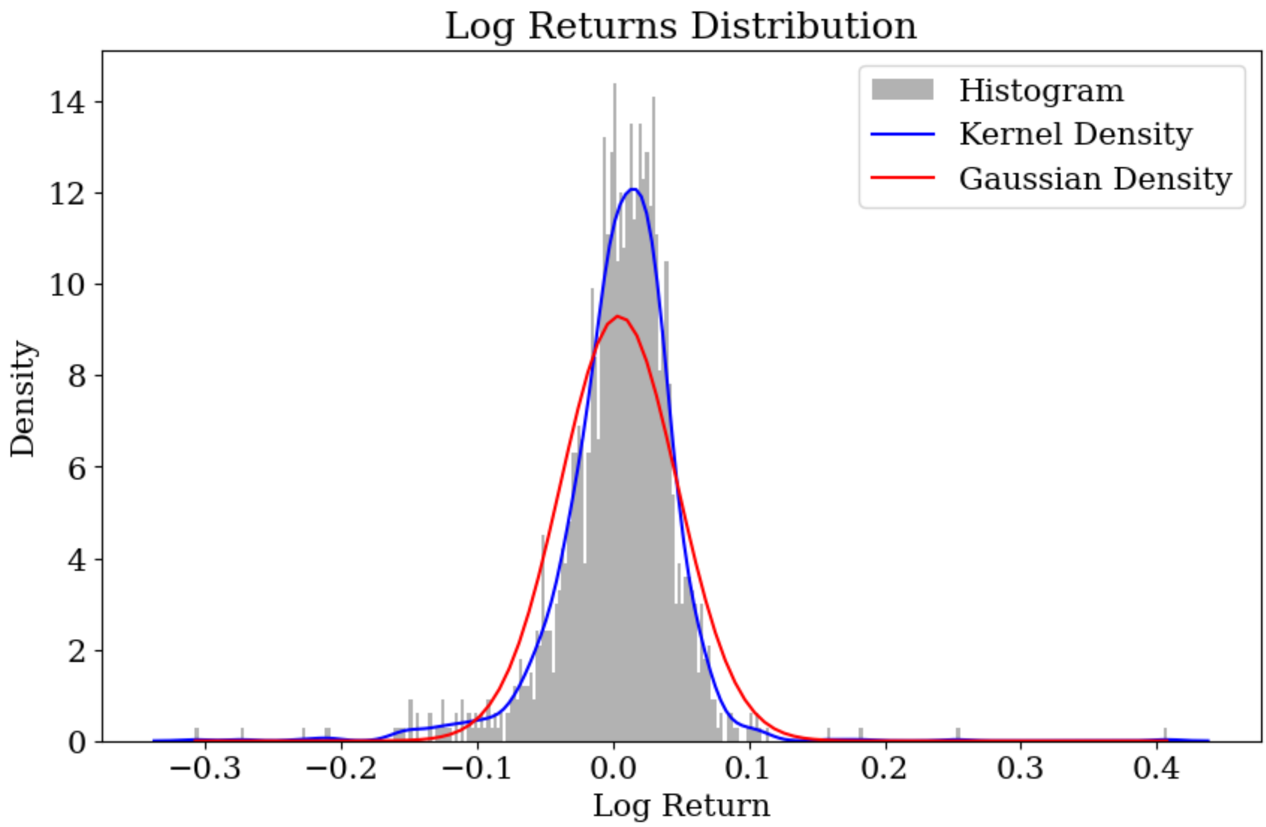
In [27]:
```
log_returns = df['price_index_log_returns'].dropna()

# Summary statistics log returns
summary = log_returns.describe()
skewness = log_returns.skew()
kurtosis = (log_returns.kurt() + 3)

additional_stats = pd.DataFrame({'skewness': [skewness], 'kurtosis': [kurtosis]})

additional_stats = additional_stats.applymap(lambda x: '{:.6f}'.format(x))

all_stats_log_returns = pd.concat([summary, additional_stats.T], axis=0)
pd.set_option('display.float_format', '{:.4f}'.format)
```

In [28]:
```
all_stats_log_returns
```

Out[28]:

	0
count	1398.0000
mean	0.0043
std	0.0429
min	-0.3075
25%	-0.0141
50%	0.0084
75%	0.0288
max	0.4075
skewness	-0.574883
kurtosis	14.724082

In [29]:
```
mean, std = log_returns.mean(), log_returns.std()
plt.figure(figsize=(10, 6))
plt.hist(log_returns, bins=300, density=True, color='gray', alpha=0.6, label='Histogram')
sns.kdeplot(log_returns, color='blue', label='Kernel Density')
x = np.linspace(min(log_returns), max(log_returns), 100)
plt.plot(x, norm.pdf(x, mean, std), 'r-', label='Gaussian Density')
plt.title('Log Returns Distribution')
plt.xlabel('Log Return')
plt.ylabel('Density')
plt.legend()
plt.show()
```



The log returns of the price index, with a mean of 0.0043 and standard deviation of 0.0429, exhibit clear evidence of volatility clustering in the time series plot, where large return fluctuations are followed by similarly large movements, and small fluctuations cluster together. This is typical in financial time series, reflecting market responses to news or economic events. The histogram with kernel density smoother reveals a leptokurtic unconditional density, as it has a sharper peak near the mean and fatter tails compared to the theoretical Gaussian density (same mean and variance). This leptokurticity indicates a higher probability of extreme returns than a normal distribution would predict.

A connection between volatility clustering and leptokurticity likely exists. Volatility clustering implies that large price movements (high volatility periods) occur in bursts, contributing to the fat tails observed in the return distribution. During these high-volatility periods, extreme returns are more frequent, increasing the kurtosis of the unconditional distribution. Conversely, low-volatility periods produce returns closer to the mean, enhancing the peakedness of the distribution. This combination of clustering and extreme events drives the leptokurtic nature of the returns.

Task 8: ARMA model selection with BIC

In [30]:
```
# Test ARMA(p, q) models with p, q from 0 to 4

max_p, max_q = 4, 4
results = []
for p in range(max_p + 1):
    for q in range(max_q + 1):
        try:
            arma_model = sm.tsa.statespace.SARIMAX(
                log_returns,
                order=(p, 0, q),
                trend='c',
                enforce_stationarity=False,
                enforce_invertibility=False
            )
            fitted_model = arma_model.fit(disp=0)
            results.append((p, q, fitted_model.bic))
        except:
            results.append((p, q, np.inf))  # Return infinity if model fails

bic_df = pd.DataFrame(results, columns=['p', 'q', 'BIC']).sort_values('BIC')

print("ARMA Model Selection based on BIC:")
print(bic_df)

best_model = bic_df.iloc[0]
print(f"\nBest Model: ARMA({int(best_model['p'])},{int(best_model['q'])}) with BIC = {best_model['BIC']

# If best is ARMA(0,0), report second best
if best_model['p'] == 0 and best_model['q'] == 0:
    second_best = bic_df.iloc[1]
    print(f"Second Best Model: ARMA({int(second_best['p'])},{int(second_best['q'])}) with BIC = {seconc
```

```
/Users/danielebiondi/anaconda3/lib/python3.11/site-packages/statsmodels/tsa/base/tsa_model.py:473: Val
ueWarning:

No frequency information was provided, so inferred frequency MS will be used.

/Users/danielebiondi/anaconda3/lib/python3.11/site-packages/statsmodels/tsa/base/tsa_model.py:473: Val
ueWarning:

No frequency information was provided, so inferred frequency MS will be used.

/Users/danielebiondi/anaconda3/lib/python3.11/site-packages/statsmodels/tsa/base/tsa_model.py:473: Val
ueWarning:

No frequency information was provided, so inferred frequency MS will be used.

/Users/danielebiondi/anaconda3/lib/python3.11/site-packages/statsmodels/tsa/base/tsa_model.py:473: Val
ueWarning:

No frequency information was provided, so inferred frequency MS will be used.

/Users/danielebiondi/anaconda3/lib/python3.11/site-packages/statsmodels/tsa/base/tsa_model.py:473: Val
```

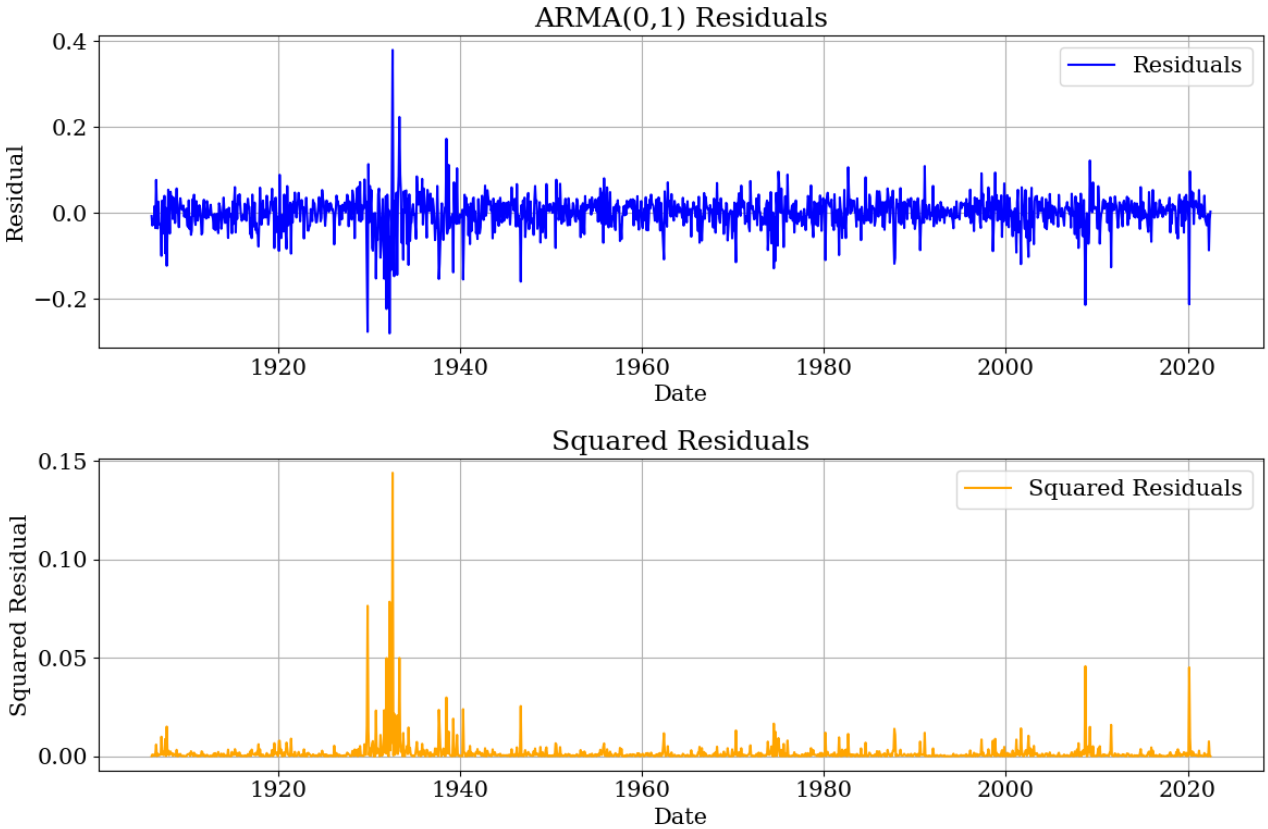
The best model based on the BIC information criterion is the ARMA(0,1)

Task 9: Residuals analysis

In [31]:
```
arma_0_1 = sm.tsa.statespace.SARIMAX(
    log_returns,
    order=(0, 0, 1),
    trend='c',
    enforce_stationarity=False,
    enforce_invertibility=False,
)
results_arma_0_1 = arma_0_1.fit()
print(results_arma_0_1.summary())
```

```
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =            3     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f= -1.76814D+00    |proj g|=  3.38665D-01

At iterate    5    f= -1.76830D+00    |proj g|=  8.16348D-03

At iterate   10    f= -1.76832D+00    |proj g|=  8.82519D-02

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
    3     14     18      1     0     0   5.340D-05  -1.768D+00
  F =  -1.7683300896681879

CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
                                  SARIMAX Results
==================================================================================
Dep. Variable:     price_index_log_returns   No. Observations:                 1398
Model:                    SARIMAX(0, 0, 1)   Log Likelihood                2472.125
Date:                     Thu, 17 Apr 2025   AIC                          -4938.251
Time:                             16:09:06   BIC                          -4922.527
Sample:                         02-01-1906   HQIC                         -4932.372
                              - 07-01-2022
Covariance Type:                       opg
=============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
-----------------------------------------------------------------------------
intercept      0.0043      0.001      3.005      0.003       0.001       0.007
ma.L1          0.2855      0.019     15.286      0.000       0.249       0.322
sigma2         0.0017   2.83e-05     59.892      0.000       0.002       0.002
==================================================================================
Ljung-Box (L1) (Q):                   0.06   Jarque-Bera (JB):              6912.19
Prob(Q):                              0.81   Prob(JB):                         0.00
Heteroskedasticity (H):               0.46   Skew:                            -0.52
Prob(H) (two-sided):                  0.00   Kurtosis:                        13.85
==================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).

/Users/danielebiondi/anaconda3/lib/python3.11/site-packages/statsmodels/tsa/base/tsa_model.py:473: Val
ueWarning:

No frequency information was provided, so inferred frequency MS will be used.

/Users/danielebiondi/anaconda3/lib/python3.11/site-packages/statsmodels/tsa/base/tsa_model.py:473: Val
ueWarning:

No frequency information was provided, so inferred frequency MS will be used.

 This problem is unconstrained.
```

In [32]:
```
residuals = results_arma_0_1.resid
squared_residuals = residuals**2

# Plot residuals and squared residuals
plt.figure(figsize=(12, 8))
plt.subplot(2, 1, 1)
plt.plot(residuals, label='Residuals', color='blue')
plt.title('ARMA(0,1) Residuals')
plt.xlabel('Date')
plt.ylabel('Residual')
plt.legend()
plt.grid(True)

plt.subplot(2, 1, 2)
plt.plot(squared_residuals, label='Squared Residuals', color='orange')
plt.title('Squared Residuals')
plt.xlabel('Date')
plt.ylabel('Squared Residual')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

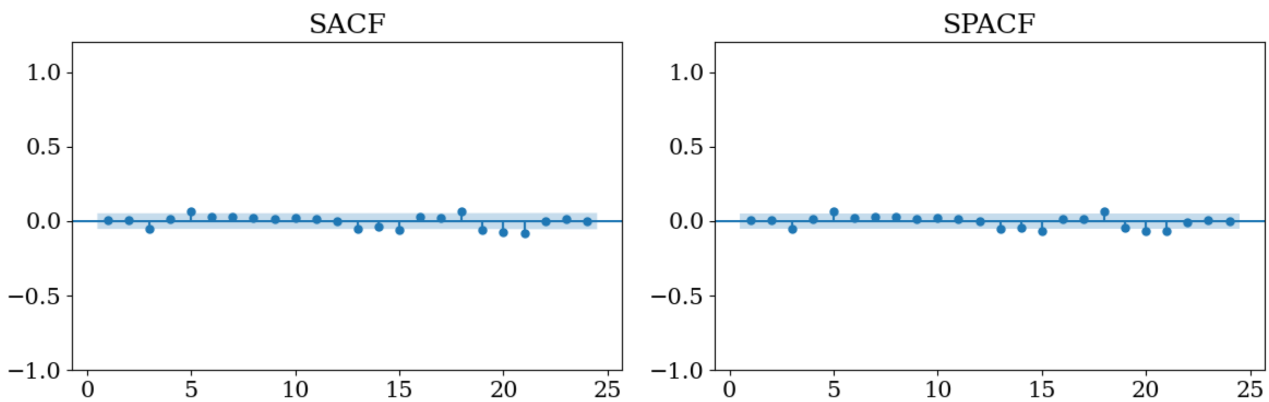


In [33]:
```
#Check for any remaning structure in the mean

residuals_acf_pacf = SACF_SPACF(residuals)
residuals_acf_pacf
```

Out[33]:

Lag	ACF	PACF	Q-stat	Q-stat Prob
1	0.0067	0.0067	0.0631	0.8016
2	0.0081	0.0080	0.1548	0.9255
3	-0.0497	-0.0499	3.6146	0.3062
4	0.0145	0.0152	3.9098	0.4183
5	0.0641	0.0651	9.6769	0.0849
6	0.0260	0.0225	10.6252	0.1007
7	0.0292	0.0294	11.8211	0.1066
8	0.0231	0.0290	12.5721	0.1274
9	0.0176	0.0177	13.0106	0.1621
10	0.0220	0.0198	13.6931	0.1875
11	0.0146	0.0128	13.9920	0.2334
12	0.0018	-0.0020	13.9965	0.3009
13	-0.0485	-0.0526	17.3145	0.1853
14	-0.0390	-0.0433	19.4667	0.1479
15	-0.0594	-0.0655	24.4665	0.0576
16	0.0264	0.0173	25.4500	0.0623
17	0.0193	0.0144	25.9791	0.0748
18	0.0668	0.0672	32.3149	0.0202
19	-0.0559	-0.0466	36.7564	0.0085
20	-0.0761	-0.0651	44.9717	0.0011
21	-0.0785	-0.0695	53.7390	0.0001
22	-0.0011	-0.0044	53.7407	0.0002
23	0.0138	0.0064	54.0116	0.0003
24	-0.0023	0.0003	54.0190	0.0004

In [34]:
```
SACF_SPACF_plot(residuals, ylim=[-1,1.2])
```

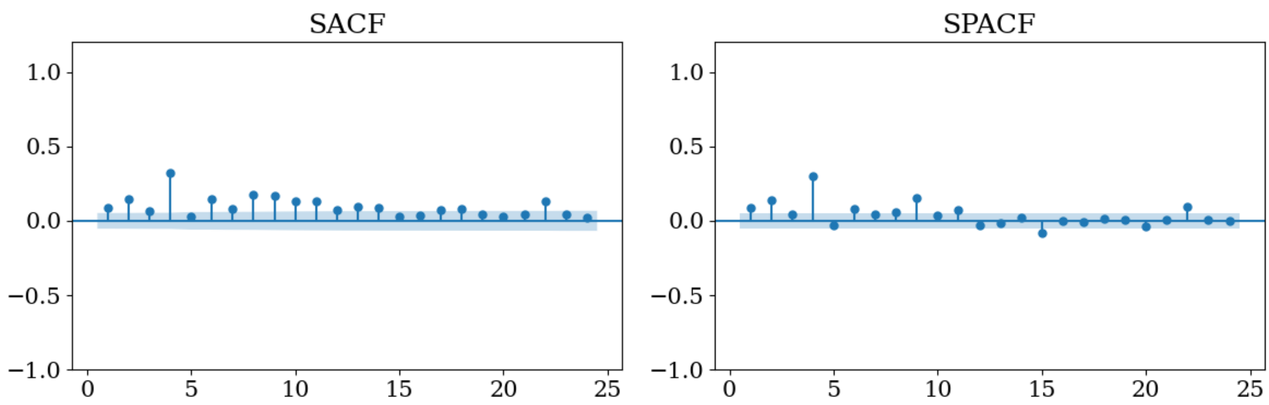


In [35]:
```
#Check for any remaning structure in the variance

residuals_squared_acf_pacf = SACF_SPACF(squared_residuals)
residuals_squared_acf_pacf
```

Out[35]:

Lag	ACF	PACF	Q-stat	Q-stat Prob
1	0.0875	0.0875	10.7324	0.0011
2	0.1441	0.1375	39.8261	0.0000
3	0.0642	0.0424	45.6005	0.0000
4	0.3226	0.3039	191.7292	0.0000
5	0.0297	-0.0276	192.9680	0.0000
6	0.1490	0.0836	224.1911	0.0000
7	0.0794	0.0442	233.0657	0.0000
8	0.1757	0.0584	276.5168	0.0000
9	0.1664	0.1570	315.5459	0.0000
10	0.1342	0.0334	340.9386	0.0000
11	0.1310	0.0764	365.1519	0.0000
12	0.0749	-0.0301	373.0670	0.0000
13	0.0967	-0.0149	386.2932	0.0000
14	0.0848	0.0199	396.4698	0.0000
15	0.0288	-0.0830	397.6455	0.0000
16	0.0397	-0.0034	399.8819	0.0000
17	0.0710	-0.0062	407.0317	0.0000
18	0.0835	0.0136	416.9191	0.0000
19	0.0446	0.0085	419.7379	0.0000
20	0.0302	-0.0392	421.0350	0.0000
21	0.0414	0.0054	423.4705	0.0000
22	0.1337	0.0953	448.8888	0.0000
23	0.0410	0.0084	451.2811	0.0000
24	0.0219	0.0020	451.9616	0.0000

In [36]:
```
SACF_SPACF_plot(squared_residuals, ylim=[-1,1.2])
```



The ARMA(0,1) model for the price index log returns has a significant MA(1) coefficient (0.2855, p-value = 0.000) and fits the data with an AIC of -4922.527. The Ljung-Box test for residuals shows no significant autocorrelation at all lags (with the p-values far exceeding 0.05 at all lags except for the very last ones) and the plots of the SACF and SPACF confirm that there is no remaining structure in the first moment of the residuals, as their correlogram is in line with that of a white noise. As far as the squared residuals are concerned, the Ljung-Box test reveals strong autocorrelation at all lags (with all the p-values being equal or very close to 0), and even their correlogram confirms this by showing that both the ACF and PACF have quite few lags outside the confidence band. Therefore, while there seems to be no remaining structure in the series of residuals (i.e. the first moment), indicating that the conditional mean is adequately modeled, the squared residuals display significant autocorrelation (as shown in the correlogram and confirmed by the Ljung-Box test), providing evidence of volatility clustering and the presence of conditional heteroskedasticity (ARCH effects). To capture this time-varying volatility, we must extend the model to a GARCH or ARCH framework.

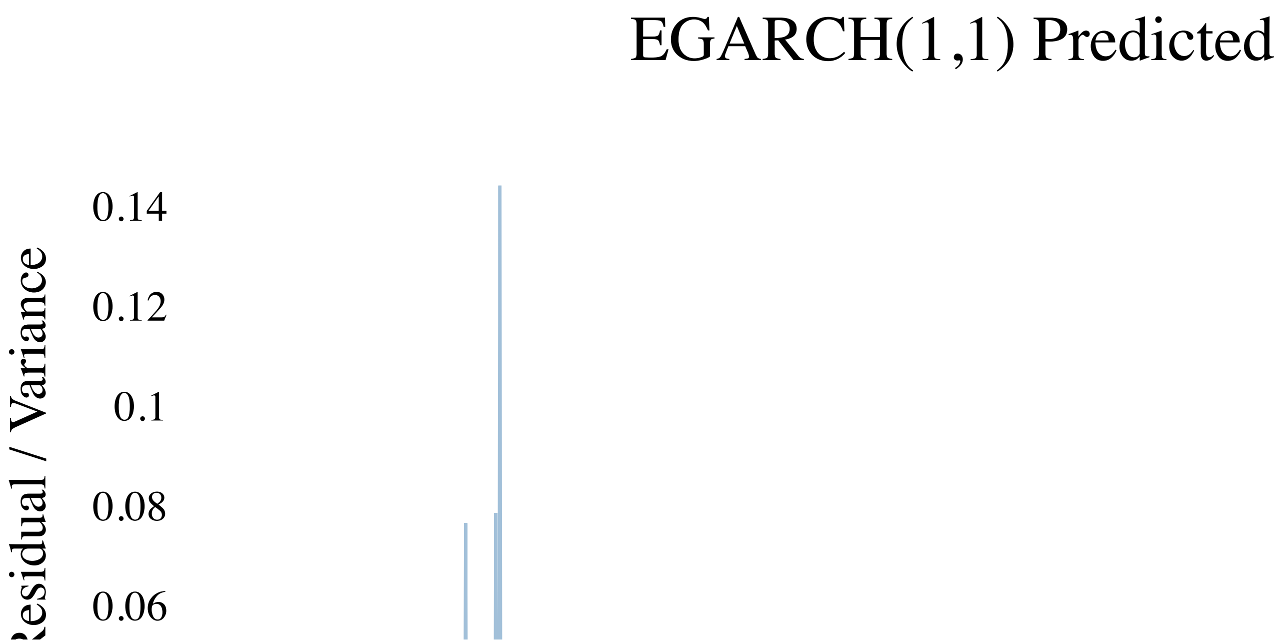
Task 10: EGARCH(1,1)

In [37]:
```
egarch = arch_model(
    # We use the residuals from the ARMA(0,1) model because we already modeled the conditional mean.
    #Setting mean='Zero' assumes no mean structure needs to be estimated in this step.
    residuals,
    mean='Zero',
    vol='EGARCH',
    p=1,
    o=1,
    q=1,
    dist='normal'
)
results_egarch = egarch.fit()
epsilons_egarch = results_egarch.resid
epsilons_egarch_squared = epsilons_egarch**2
predicted_egarch_volatility = results_egarch.conditional_volatility
predicted_egarch_variance = predicted_egarch_volatility**2
print(results_egarch.summary())
```

```
Iteration:      1,   Func. Count:      6,   Neg. LLF: 475104793.9200104
Iteration:      2,   Func. Count:     15,   Neg. LLF: 234394383.27398863
Iteration:      3,   Func. Count:     24,   Neg. LLF: 5097.437207357914
Iteration:      4,   Func. Count:     31,   Neg. LLF: -2633.819732447386
Iteration:      5,   Func. Count:     37,   Neg. LLF: -2652.3576565675603
Iteration:      6,   Func. Count:     43,   Neg. LLF: -2653.5858290357337
Iteration:      7,   Func. Count:     48,   Neg. LLF: -2653.577952181607
Iteration:      8,   Func. Count:     54,   Neg. LLF: -2653.5894034629055
Iteration:      9,   Func. Count:     59,   Neg. LLF: -2653.59054978905
Iteration:     10,   Func. Count:     64,   Neg. LLF: -2653.5905620637086
Iteration:     11,   Func. Count:     69,   Neg. LLF: -2653.5905663569656
Iteration:     12,   Func. Count:     73,   Neg. LLF: -2653.5905663581025
Optimization terminated successfully    (Exit mode 0)
            Current function value: -2653.5905663569656
            Iterations: 12
            Function evaluations: 73
            Gradient evaluations: 12
                       Zero Mean - EGARCH Model Results
==============================================================================
Dep. Variable:                   None   R-squared:                       0.000
Mean Model:                 Zero Mean   Adj. R-squared:                  0.001
Vol Model:                     EGARCH   Log-Likelihood:                2653.59
Distribution:                  Normal   AIC:                          -5299.18
Method:            Maximum Likelihood   BIC:                          -5278.21
                                        No. Observations:                 1398
Date:                Thu, Apr 17 2025   Df Residuals:                     1398
Time:                        16:09:07   Df Model:                            0
                              Volatility Model
===========================================================================
                 coef    std err          t      P>|t|     95.0% Conf. Int.
---------------------------------------------------------------------------
omega         -0.3393      0.134     -2.528  1.148e-02 [ -0.602,-7.621e-02]
alpha[1]       0.2309  4.264e-02      5.416  6.106e-08    [  0.147,  0.314]
gamma[1]      -0.0888  3.755e-02     -2.366  1.797e-02 [ -0.162,-1.526e-02]
beta[1]        0.9470  2.024e-02     46.789      0.000    [  0.907,  0.987]
===========================================================================

Covariance estimator: robust

/Users/danielebiondi/anaconda3/lib/python3.11/site-packages/arch/univariate/base.py:309: DataScaleWarn
ing:

y is poorly scaled, which may affect convergence of the optimizer when
estimating the model parameters. The scale of y is 0.001694. Parameter
estimation work better when this value is between 1 and 1000. The recommended
rescaling is 10 * y.

This warning can be disabled by either rescaling y before initializing the
model or by setting rescale=False.
```

In [38]:
```
fig_egarch = go.Figure()

fig_egarch.add_trace(go.Scatter(
    x=log_returns.index,
    y=epsilons_egarch_squared,
    mode='lines',
    name='Squared Residual',
    line=dict(color='steelblue', width=1.5),
    opacity=0.5
))
```

```
fig_egarch.add_trace(go.Scatter(
    x=log_returns.index,
    y=predicted_egarch_variance,
    mode='lines',
    name=f'EGARCH(1,1) Variance',
    line=dict(color='firebrick', width=2.0)
))

fig_egarch.update_layout(
    title=dict(
        text=f'EGARCH(1,1) Predicted Variance',
        x=0.5,
        font=dict(size=30, family='Serif', color='black')
    ),
    xaxis=dict(
        title=dict(
            text='Date',
            font=dict(size=25, family='Serif', color='black')  # Fixed titlefont
        ),
        tickfont=dict(size=20, family='Serif', color='black'),  # Adjusted for consistency
        showgrid=False
    ),
    yaxis=dict(
        title=dict(
            text='Squared Residual / Variance',
            font=dict(size=25, family='Serif', color='black')  # Fixed titlefont
        ),
        tickfont=dict(size=20, family='Serif', color='black'),  # Adjusted for consistency
        showgrid=False,
        zeroline=False,
    ),
    font=dict(
        family="Serif",
        size=20,
        color="black"
    ),
    plot_bgcolor='white',  # Changed to white
    paper_bgcolor='white',  # Changed to white
    legend=dict(
        font=dict(size=18),
        bgcolor='rgba(255,255,255,0.5)',
        bordercolor='black',
        borderwidth=1
    ),
    margin=dict(t=80, l=60, r=60, b=60)
)

fig_egarch.show()
```



The EGARCH(1,1) model captures volatility clustering effectively, with significant parameters and strong persistence ($\beta = 0.947$). The asymmetry term ($\gamma < 0$) confirms that negative shocks increase volatility more than positive ones. The conditional variance series closely follows the timing of squared residuals, reflecting changes in volatility well. However, it appears more smoothed and fails to fully capture the magnitude of extreme volatility spikes, underestimating the size of large shocks. This suggests that while the model tracks volatility dynamics, it may not entirely reflect the intensity of sudden, sharp movements observed in the squared residuals.

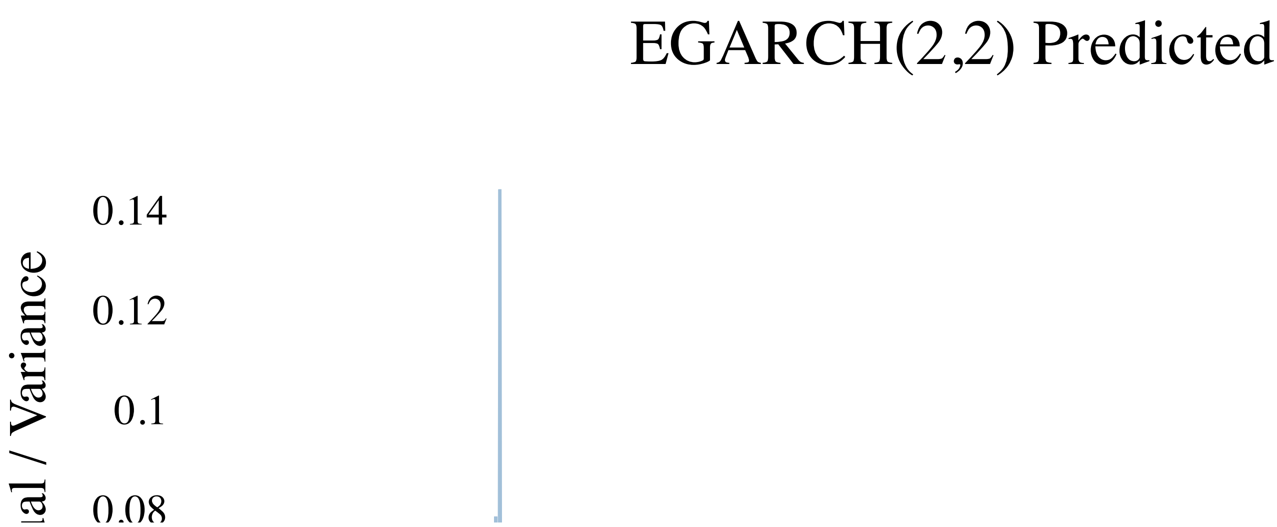
Task 11: Other models --> EGARCH(2,2)

In [39]:
```
egarch22 = arch_model(
    # We use the residuals from the ARMA(0,1) model because we already modeled the conditional mean.
    #Setting mean='Zero' assumes no mean structure needs to be estimated in this step.
    residuals,
    mean='Zero',
    vol='EGARCH',
    p=2,
    o=2,
    q=2,
    dist='normal'
)
results_egarch22 = egarch22.fit()
epsilons_egarch22 = results_egarch22.resid
epsilons_egarch22_squared = epsilons_egarch22**2
predicted_egarch22_volatility = results_egarch22.conditional_volatility
predicted_egarch22_variance = predicted_egarch22_volatility**2
print(results_egarch22.summary())
```

```
Iteration:      1,   Func. Count:      9,   Neg. LLF: 283337007.893505
Iteration:      2,   Func. Count:     21,   Neg. LLF: 410058584.3638823
Iteration:      3,   Func. Count:     32,   Neg. LLF: 11962882.97432759
Iteration:      4,   Func. Count:     43,   Neg. LLF: 5820.877851316743
Iteration:      5,   Func. Count:     54,   Neg. LLF: 122762873.15827334
Iteration:      6,   Func. Count:     65,   Neg. LLF: -2639.945308115658
Iteration:      7,   Func. Count:     74,   Neg. LLF: -2604.935057760572
Iteration:      8,   Func. Count:     83,   Neg. LLF: -2648.1587699328816
Iteration:      9,   Func. Count:     92,   Neg. LLF: 2231390.4631497385
Iteration:     10,   Func. Count:    102,   Neg. LLF: -2661.191539923663
Iteration:     11,   Func. Count:    111,   Neg. LLF: -2668.9731233944167
Iteration:     12,   Func. Count:    119,   Neg. LLF: -2653.2568217302414
Iteration:     13,   Func. Count:    128,   Neg. LLF: -2669.3536934932235
Iteration:     14,   Func. Count:    137,   Neg. LLF: -2670.119892051491
Iteration:     15,   Func. Count:    145,   Neg. LLF: -2670.5513458249743
Iteration:     16,   Func. Count:    153,   Neg. LLF: -2670.642112904234
Iteration:     17,   Func. Count:    161,   Neg. LLF: -2670.6566942235795
Iteration:     18,   Func. Count:    169,   Neg. LLF: -2670.658421052619
Iteration:     19,   Func. Count:    177,   Neg. LLF: -2670.6584268247757
Iteration:     20,   Func. Count:    184,   Neg. LLF: -2670.658426588362
Optimization terminated successfully    (Exit mode 0)
            Current function value: -2670.6584268247757
            Iterations: 20
            Function evaluations: 184
            Gradient evaluations: 20
                       Zero Mean - EGARCH Model Results
==============================================================================
Dep. Variable:                   None   R-squared:                       0.000
Mean Model:                 Zero Mean   Adj. R-squared:                  0.001
Vol Model:                     EGARCH   Log-Likelihood:                2670.66
Distribution:                  Normal   AIC:                          -5327.32
Method:            Maximum Likelihood   BIC:                          -5290.62
                                        No. Observations:                 1398
Date:                Thu, Apr 17 2025   Df Residuals:                     1398
Time:                        16:09:07   Df Model:                            0
                              Volatility Model
===========================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
---------------------------------------------------------------------------
omega         -0.2230      0.152     -1.469      0.142  [ -0.521,7.459e-02]
alpha[1]       0.0627      0.108      0.580      0.562    [ -0.149,  0.275]
alpha[2]       0.1407  8.230e-02      1.710  8.734e-02 [-2.061e-02,  0.302]
gamma[1]      -0.2871  6.958e-02     -4.126  3.698e-05    [ -0.423, -0.151]
gamma[2]       0.2309      0.101      2.285  2.233e-02  [3.282e-02,  0.429]
beta[1]        0.9651      0.504      1.915  5.551e-02 [-2.272e-02,  1.953]
beta[2]        0.0000      0.484      0.000      1.000    [ -0.948,  0.948]
===========================================================================

Covariance estimator: robust

/Users/danielebiondi/anaconda3/lib/python3.11/site-packages/arch/univariate/base.py:309: DataScaleWarn
ing:

y is poorly scaled, which may affect convergence of the optimizer when
estimating the model parameters. The scale of y is 0.001694. Parameter
estimation work better when this value is between 1 and 1000. The recommended
rescaling is 10 * y.

This warning can be disabled by either rescaling y before initializing the
model or by setting rescale=False.
```

In [40]:

```python
fig_egarch22 = go.Figure()

fig_egarch22.add_trace(go.Scatter(
    x=log_returns.index,
    y=epsilons_egarch22_squared,
    mode='lines',
    name='Squared Residual',
    line=dict(color='steelblue', width=1.5),
    opacity=0.5
))

fig_egarch22.add_trace(go.Scatter(
    x=log_returns.index,
    y=predicted_egarch22_variance,
    mode='lines',
    name=f'EGARCH(2,2) Variance',
    line=dict(color='firebrick', width=2.0)
))

fig_egarch22.update_layout(
    title=dict(
        text=f'EGARCH(2,2) Predicted Variance',
        x=0.5,
        font=dict(size=30, family='Serif', color='black')
    ),
    xaxis=dict(
        title=dict(
            text='Date',
            font=dict(size=25, family='Serif', color='black')  # Fixed titlefont
        ),
        tickfont=dict(size=20, family='Serif', color='black'),  # Adjusted for consistency
        showgrid=False
    ),
    yaxis=dict(
        title=dict(
            text='Squared Residual / Variance',
            font=dict(size=25, family='Serif', color='black')  # Fixed titlefont
        ),
        tickfont=dict(size=20, family='Serif', color='black'),  # Adjusted for consistency
        showgrid=False,
        zeroline=False,
    ),
    font=dict(
        family="Serif",
        size=20,
        color="black"
    ),
    plot_bgcolor='white',  # Changed to white
    paper_bgcolor='white',  # Changed to white
    legend=dict(
        font=dict(size=18),
        bgcolor='rgba(255,255,255,0.5)',
        bordercolor='black',
        borderwidth=1
    ),
    margin=dict(t=80, l=60, r=60, b=60)
)

fig_egarch22.show()
```



To answer this question we estimated an EGARCH(2,2). When looking at the output, more precisely the log-likelihood and the information criteria, we see that it performs better than an EGARCH(1,1). The log-likelihood is in fact higher (2670.66 vs 2653.59) and both the AIC and BIC are lower (-5327.32 vs -5299.18 and -5290.62 vs -5278.21 respectively). Also looking at how the model tracks squared residuals, we see that it performs a better job, even though there is still room for improvement.

Task 12: Winning model analysis

In [41]:

```
pred_df = pd.DataFrame(log_returns)
pred_df.rename(columns={pred_df.columns[0]: "Returns"}, inplace=True)

pred_df["Residual"] = residuals
pred_df["Residual Squared"] = squared_residuals
pred_df["Predicted variance EGARCH(2,2)"] = predicted_egarch22_variance

pred_df.dropna(inplace=True)
pred_df.head()
```

Out[41]:

	Returns	Residual	Residual Squared	Predicted variance EGARCH(2,2)
date				
1906-02-01	-0.0071	-0.0071	0.0001	0.0017
1906-03-01	-0.0248	-0.0291	0.0008	0.0017
1906-04-01	-0.0137	-0.0180	0.0003	0.0018
1906-05-01	-0.0269	-0.0264	0.0007	0.0017
1906-06-01	0.0130	0.0162	0.0003	0.0017

In [42]:
```python
Y = pred_df['Residual Squared']
X = pred_df['Predicted variance EGARCH(2,2)']
X = sm.add_constant(X)
model = sm.OLS(Y, X).fit()
results=model
test=dict()
test["Predicted variance EGARCH(2,2)"] = {
            'summary': results.summary().as_text(),
            'f_test': None,
            't_test': None
        }
try:
  f_test = results.f_test(f"const = 0, Predicted variance EGARCH(2,2) = 1")
  test["Predicted variance EGARCH(2,2)"]['f_test'] = f_test
except Exception as e:
  test["Predicted variance EGARCH(2,2)"]['f_test'] = f"Error performing F-test: {e}"

        # Perform and store the T-test
try:
  t_test = results.t_test(f"Predicted variance EGARCH(2,2) = 1")
  test["Predicted variance EGARCH(2,2)"]['t_test'] = t_test
except Exception as e:
  test["Predicted variance EGARCH(2,2)"]['t_test'] = f"Error performing T-test: {e}"

print(test["Predicted variance EGARCH(2,2)"]['summary'])
print("\nF-test for hypothesis that intercept = 0 and coefficient of regressor = 1:")
print(test["Predicted variance EGARCH(2,2)"]['f_test'])
print("\nT-test for hypothesis that the coefficient of regressor = 1:")
print(test["Predicted variance EGARCH(2,2)"]['t_test'])
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:       Residual Squared   R-squared:                       0.078
Model:                            OLS   Adj. R-squared:                  0.077
Method:                 Least Squares   F-statistic:                     118.0
Date:                Thu, 17 Apr 2025   Prob (F-statistic):           1.95e-26
Time:                        16:09:07   Log-Likelihood:                 5207.9
No. Observations:                1398   AIC:                        -1.041e+04
Df Residuals:                    1396   BIC:                        -1.040e+04
Df Model:                           1
Covariance Type:            nonrobust
==================================================================================================
                                     coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------------------------
const                          -1.899e-05      0.000     -0.086      0.932      -0.000       0.000
Predicted variance EGARCH(2,2)     1.0464      0.096     10.861      0.000       0.857       1.235
==============================================================================
Omnibus:                     2582.595   Durbin-Watson:                   2.153
Prob(Omnibus):                  0.000   Jarque-Bera (JB):          3616940.253
Skew:                          13.115   Prob(JB):                         0.00
Kurtosis:                     250.801   Cond. No.                         617.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

F-test for hypothesis that intercept = 0 and coefficient of regressor = 1:
<F test: F=0.1826929798076157, p=0.833043785017947, df_denom=1.4e+03, df_num=2>

T-test for hypothesis that the coefficient of regressor = 1:
                             Test for Constraints
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
c0             1.0464      0.096      0.482      0.630       0.857       1.235
==============================================================================
```

For this question, we used our previously estimated EGARCH(2,2), as it was the better performer. In order to test whether the model provides us with an efficient and unbiased predictor of the squared residuals from the ARMA(0,1) we used, we run an auxiliary regression of the squared residuals over the variance predicted by the model. The output of our regression shows a constant extremely close to 0 and a a coefficient of 1.0464, with an R-squared of 0.078. Performing an F-test with null hp intercept=0 and coefficient of predicted variance=1, we fail to reject them, reporting a p-value of 0.833. This result highlights that the chosen model is an unbiased and efficient predictor of future variance.