RELATIONSHIP BETWEEN US/CANADA EXCHANGE RATE AND COMMODITY PRICES

ECONOMETRICS II: PROJECT III

NANA OSEI SARPONG
11371873

Contents

INTRODUCTION

This project serves to apply techniques of simulations to improve inferences from our conventional tests. In the previous project, we run a number of models and diagnostic tests on those models to assess the robustness of the results we obtained. In this project, we go a step further to try to make improvements to the tests we run. To keep our work brief and simple, we shall focus on the Vector Autoregressive model for the logged first differences of the Canadian exchange rate and the real total commodity price index. We shall look at ways in which we can improve inferences made from the various diagnostic tests that were run on the model. Since the augmented Dickey-Fuller test makes use of critical values that were obtained from simulations, we do not stand to improve inference by dwelling on it. However, we can improve inference for our model diagnostic tests, among others.

NORMALITY TEST

We run the Jarque Bera normality test for our VAR model. The table below shows the results of the conventional test.

Test Statistic	P-Value
1226.1737	0.0000
292.5785	0.0000

Based on our conventional test for normality, we reject the null hypothesis of normally distributed errors for both equations of our VAR model.

MONTE-CARLO SIMULATION FOR JARQUE-BERA

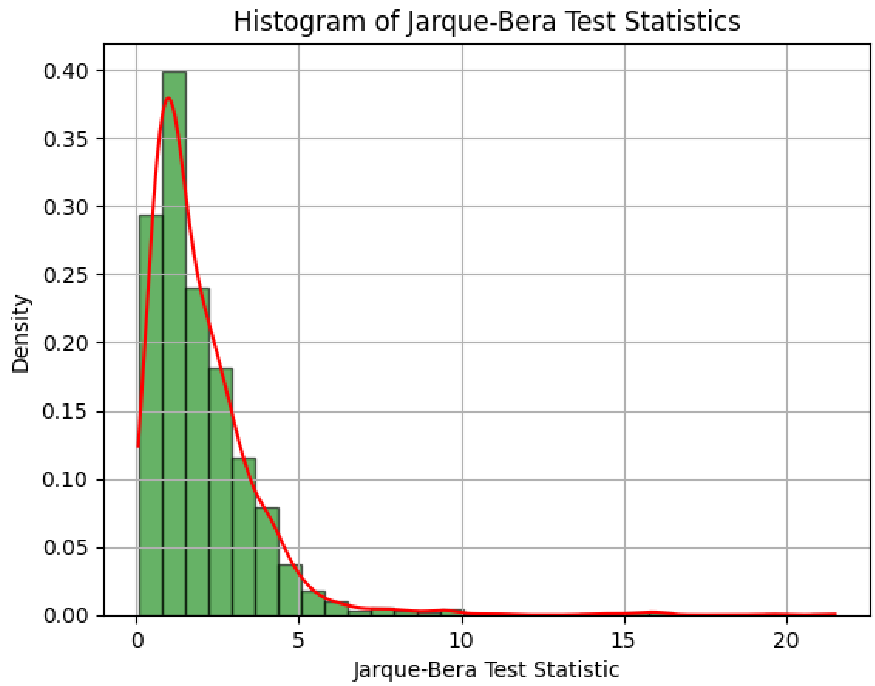
The Jarque-Bera test for normality is not a pivotal test since it relies on the skewness and kurtosis computed based on our data which is a single realization of a random process, a data generating process which is not known. In finite samples, our conclusions from inference will be heavily influenced by nuisance parameters of the model. The test however is asymptotically pivotal; our results will not depend on the parameters of the model for a large enough sample and that is where we stand to improve our inferences by running a Monte-Carlo simulation.

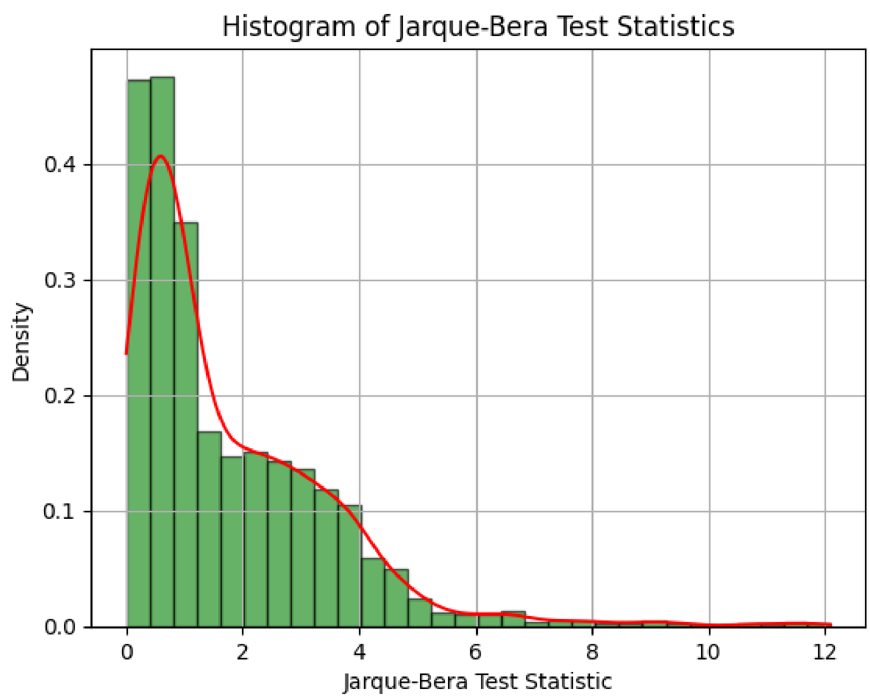
Under the null hypothesis, our model has errors that are normally distributed. We run our simulations such that random normal errors are added to the fitted values from the model and then the VAR model is run and the normality test is performed to the data over and over again to obtain an empirical chi-square distribution for which we can use to make a comparison with the chi-square statistic obtained from the conventional test.

The table below shows the p-value results of our simulation.

Simulated Critical Test Statistic	P-value
4.7009	0.0000
4.6482	0.0000

The simulation confirms our earlier held notion of non-normality of the errors in the model. The probability of observing such a chi-square statistic for data that was simulated under the assumption of normality is very unlikely so we are able to proceed with confidence that our errors are not normally distributed.





AUTOCORRELATION TEST

We run the Portmanteau test for autocorrelation (Ljun-Box test) to check whether the errors of the test are serially correlated. The table below shows our conventional test results.

Test Statistic	Critical Test Statistic	P-Value
60.394	50.998	0.007

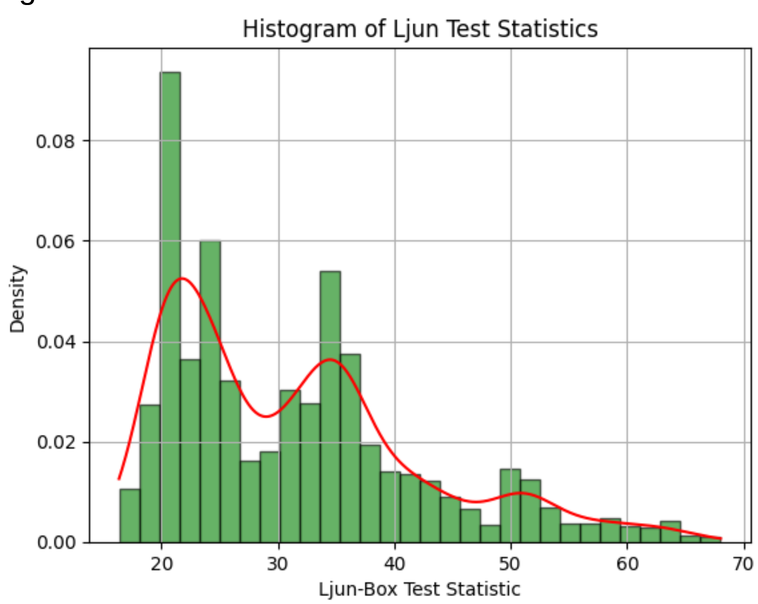
The null hypothesis of the test posits that the errors of the VAR model are random and we reject that at the 5% significance level and conclude that the residuals of the model are autocorrelated.

PARAMETRIC BOOTSTRAPPING FOR LJUN-BOX TEST

This test also not a pivotal test. We obtain the correlation rho of the errors based on the data at hand which is a random estimate. The test is however asymptotically pivotal and we can improve inference by running a simulation. The basis for resorting to simulations is that by replicating the test for a number of times for errors that share similarities with those of our VAR model, we are able to assess whether the chi-squared we computed is likely to occur. We will do this using a parametric bootstrap. We run the simulation for random normal errors that share the same mean and standard deviation as those of our VAR model and iteratively test for autocorrelation for all such simulations and compare our conventional test statistic to the simulated statistics.

Simulated Critical Test Statistic	P-value
52.9898	0.0185

Based on our simulation, we reject the null hypothesis of no autocorrelation at the 5% significance and conclude that the VAR model contains errors that are serially correlated.



GRANGER CAUSALITY TEST

We run the Granger causality test to check whether we can make predictions of another variable based on past values of another variable. The table below shows the results of our conventional test.

Test Statistic	P-value
4.1302	0.0426

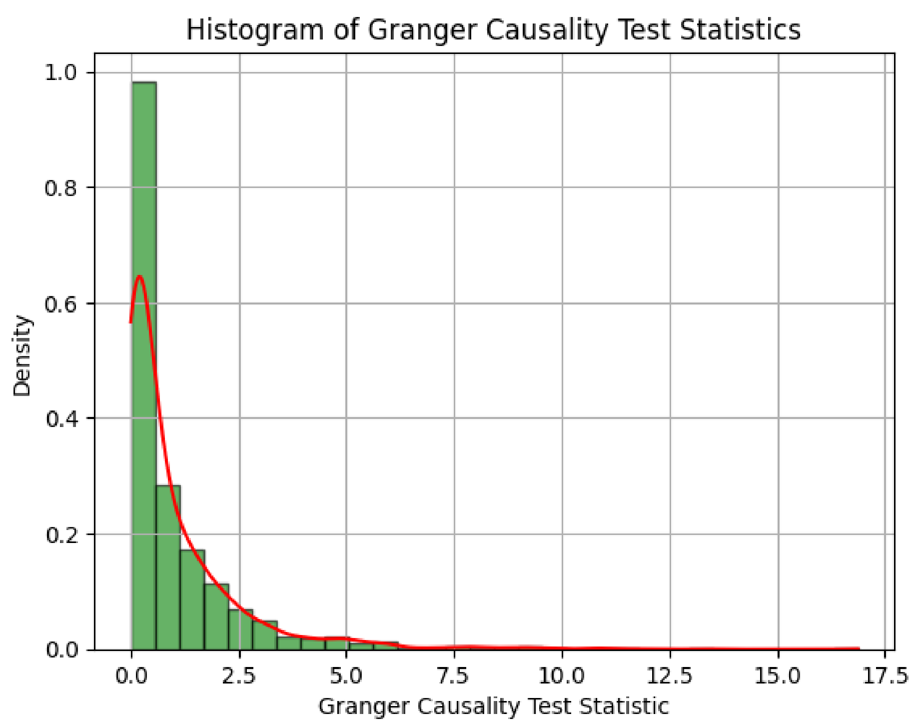
Our conventional test statistic led us to the conclusion that the US/Canada exchange rate does not Granger cause total commodity prices.

RESAMPLING FOR GRANGER CAUSALITY TEST

The Granger causality test might not perform well in finite samples. It is an asymptotically pivotal test. As a result, we stand to gain an improvement in our inferences by running simulations. For this, we use the resampling bootstrapping technique. We will randomly select different time observations of both variables and iteratively test for Granger causality and use our empirically generated distribution to make inference. The table below shows the results of our test.

Simulated Critical Test Statistic	P-value
3.8845	0.0435

Our simulation further cements our belief that the US/Canada exchange rate does not Granger cause total commodity prices since the p-value is less than the 0.05 significance level.



CONCLUSION

In this project, we aimed to enhance the reliability of conventional tests by incorporating simulation techniques. Our focus was on the Vector Autoregressive (VAR) model applied to the logged first differences of the Canadian exchange rate and the real total commodity price index. We explored improvements to diagnostic tests conducted on the VAR model, for normality, autocorrelation, and Granger causality. We used three different techniques to arrive at our conclusions: Monte-Carlo simulations, parametric bootstrapping and resampling.

For the normality test, the Jarque-Bera test rejected the null hypothesis of normally distributed errors for both equations of the VAR model. Monte Carlo simulations confirmed non-normality, supporting our initial findings.

Regarding autocorrelation, the Portmanteau test suggested serial correlation among model residuals, which was corroborated by parametric bootstrapping simulations.

For the Granger causality test, the conventional analysis indicated that the US/Canada exchange rate does not Granger cause total commodity prices. Resampling using bootstrapping reinforced this conclusion, providing further evidence against the presence of a causal relationship.

appendix1

March 19, 2024

[106]:
```python
# Importing relevant libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.api as sm
from statsmodels.tsa.stattools import grangercausalitytests
from statsmodels.tsa.api import VAR, VECM
from statsmodels.tsa.ar_model import AutoReg
from statsmodels.stats.diagnostic import het_white
from statsmodels.stats.stattools import jarque_bera
import statsmodels.tsa.vector_ar
from scipy.stats import gaussian_kde
```

[107]:
```python
# Loading the first dataset
exchange_rates = pd.read_csv('/content/drive/MyDrive/Data sets/
 ↪StatsCanExchangeRates.csv')
exchange_rates.head()
```

[107]:
```
  REF_DATE    GEO  DGUID                               Type of currency  \
0  1950-10  Canada    NaN   United States dollar, noon spot rate, average
1  1950-10  Canada    NaN  United States dollar, 90-day forward noon rate
2  1950-10  Canada    NaN         Belgian franc, noon spot rate, average
3  1950-10  Canada    NaN          Danish krone, noon spot rate, average
4  1950-10  Canada    NaN          French franc, noon spot rate, average

       UOM  UOM_ID SCALAR_FACTOR  SCALAR_ID  VECTOR  COORDINATE     VALUE  \
0  Dollars      81         units          0  v37426        1.10  1.053333
1  Dollars      81         units          0  v37437        1.22  1.047313
2  Dollars      81         units          0  v37448        1.20  0.020928
3  Dollars      81         units          0  v37452        1.30  0.152562
4  Dollars      81         units          0  v37453        1.40  0.003014

   STATUS  SYMBOL TERMINATED  DECIMALS
0     NaN     NaN        NaN         8
1     NaN     NaN        NaN         8
2     NaN     NaN          t         8
3     NaN     NaN        NaN         8
```

```
4     NaN     NaN          t         8
```

[108]:
```
# Filtering for only US/CAD related data
exchange_rates = exchange_rates[exchange_rates['Type of currency'] == 'United␣
↪States dollar, noon spot rate, average']
exchange_rates.head()
```

[108]:
```
   REF_DATE     GEO  DGUID                                Type of currency  \
0   1950-10  Canada    NaN  United States dollar, noon spot rate, average
13  1950-11  Canada    NaN  United States dollar, noon spot rate, average
26  1950-12  Canada    NaN  United States dollar, noon spot rate, average
39  1951-01  Canada    NaN  United States dollar, noon spot rate, average
55  1951-02  Canada    NaN  United States dollar, noon spot rate, average

        UOM  UOM_ID SCALAR_FACTOR  SCALAR_ID  VECTOR  COORDINATE     VALUE  \
0   Dollars      81         units          0  v37426         1.1  1.053333
13  Dollars      81         units          0  v37426         1.1  1.040312
26  Dollars      81         units          0  v37426         1.1  1.053078
39  Dollars      81         units          0  v37426         1.1  1.051875
55  Dollars      81         units          0  v37426         1.1  1.049125

    STATUS  SYMBOL TERMINATED  DECIMALS
0      NaN     NaN        NaN         8
13     NaN     NaN        NaN         8
26     NaN     NaN        NaN         8
39     NaN     NaN        NaN         8
55     NaN     NaN        NaN         8
```

[109]:
```
# Filtering for only relevant columns
filtered_ex_rate = exchange_rates[['REF_DATE', 'VALUE']]
filtered_ex_rate.columns = ['Date','US/CAD']

# Converting the date to a date type
filtered_ex_rate['Date'] = pd.to_datetime(filtered_ex_rate['Date'],␣
↪format='%Y-%m')

filtered_ex_rate.head()
```

```
<ipython-input-109-fe1ea1171c26>:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  filtered_ex_rate['Date'] = pd.to_datetime(filtered_ex_rate['Date'],
format='%Y-%m')
```

[109]:
```
          Date    US/CAD
0   1950-10-01  1.053333
13  1950-11-01  1.040312
26  1950-12-01  1.053078
39  1951-01-01  1.051875
55  1951-02-01  1.049125
```

[110]:
```
# Loading second dataset
price_indices = pd.read_csv('/content/drive/MyDrive/Data sets/
 ↪StatsCanPriceIndices.csv')
price_indices
```

[110]:
```
     REF_DATE     GEO          DGUID                Commodity  \
0     1972-01  Canada  2016A000011124  Total, all commodities
1     1972-01  Canada  2016A000011124  Total excluding energy
2     1972-01  Canada  2016A000011124                  Energy
3     1972-01  Canada  2016A000011124     Metals and Minerals
4     1972-01  Canada  2016A000011124             Agriculture
…         …       …              …                       …
4363  2023-12  Canada  2016A000011124                  Energy
4364  2023-12  Canada  2016A000011124     Metals and Minerals
4365  2023-12  Canada  2016A000011124             Agriculture
4366  2023-12  Canada  2016A000011124                    Fish
4367  2023-12  Canada  2016A000011124                Forestry

                  UOM  UOM_ID SCALAR_FACTOR  SCALAR_ID    VECTOR  COORDINATE  \
0     Index, 1972=100     166         units          0  v52673496         1.1
1     Index, 1972=100     166         units          0  v52673497         1.2
2     Index, 1972=100     166         units          0  v52673498         1.3
3     Index, 1972=100     166         units          0  v52673499         1.4
4     Index, 1972=100     166         units          0  v52673500         1.5
…                   …      …             …          …         …
4363  Index, 1972=100     166         units          0  v52673498         1.3
4364  Index, 1972=100     166         units          0  v52673499         1.4
4365  Index, 1972=100     166         units          0  v52673500         1.5
4366  Index, 1972=100     166         units          0  v52673501         1.6
4367  Index, 1972=100     166         units          0  v52673502         1.7

       VALUE  STATUS  SYMBOL  TERMINATED  DECIMALS
0      100.0     NaN     NaN         NaN         1
1      100.0     NaN     NaN         NaN         1
2      100.0     NaN     NaN         NaN         1
3      100.0     NaN     NaN         NaN         1
4      100.0     NaN     NaN         NaN         1
…          …      …       …           …         …
4363  1285.1     NaN     NaN         NaN         1
4364   696.5     NaN     NaN         NaN         1
```

```
4365   285.6      NaN      NaN          NaN         1
4366  1634.8      NaN      NaN          NaN         1
4367   453.8      NaN      NaN          NaN         1

[4368 rows x 15 columns]
```

[111]:

```python
# Filtering data for relevant variables and placing them in different columns
filtered_price_indices = pd.DataFrame()
filtered_price_indices['Date'] = price_indices.loc[price_indices['Commodity']␣
 ↪== 'Total, all commodities', 'REF_DATE'].values
filtered_price_indices['Total Index'] = price_indices.
 ↪loc[price_indices['Commodity']=='Total, all commodities','VALUE'].values
filtered_price_indices['Tot. Index (Ex. Energy)'] = price_indices.
 ↪loc[price_indices['Commodity']=='Total excluding energy','VALUE'].values
filtered_price_indices['Energy Index'] = price_indices.
 ↪loc[price_indices['Commodity']=='Energy','VALUE'].values
filtered_price_indices['Metals & Minerals Index'] = price_indices.
 ↪loc[price_indices['Commodity']=='Metals and Minerals','VALUE'].values
filtered_price_indices['Agriculture Index'] = price_indices.
 ↪loc[price_indices['Commodity']=='Agriculture','VALUE'].values
filtered_price_indices['Fish Index'] = price_indices.
 ↪loc[price_indices['Commodity']=='Fish','VALUE'].values
filtered_price_indices['Forestry Index'] = price_indices.
 ↪loc[price_indices['Commodity']=='Forestry','VALUE'].values

# Converting the date to a date type
filtered_price_indices['Date'] = pd.to_datetime(filtered_price_indices['Date'],␣
 ↪format='%Y-%m')

filtered_price_indices
```

[111]:

```
           Date  Total Index  Tot. Index (Ex. Energy)  Energy Index  \
0    1972-01-01        100.0                    100.0         100.0
1    1972-02-01        100.4                    100.5          99.8
2    1972-03-01        101.1                    101.3         100.1
3    1972-04-01        101.2                    101.5          99.8
4    1972-05-01        101.9                    102.3         100.0
..          …          …                        …          …
619  2023-08-01        625.8                    436.4        1483.6
620  2023-09-01        649.5                    425.5        1611.3
621  2023-10-01        620.5                    416.1        1513.3
622  2023-11-01        578.3                    418.5        1334.6
623  2023-12-01        565.4                    417.7        1285.1

     Metals & Minerals Index  Agriculture Index  Fish Index  Forestry Index
0                      100.0              100.0       100.0           100.0
```

```
1                    100.7            101.2       88.9          100.1
2                    101.4            102.5       99.0          100.2
3                    101.2            102.1      103.1          100.9
4                    101.3            103.5       86.3          102.3
..                     …                …         …             …
619                  713.6            322.2     1595.0          436.2
620                  712.7            304.4     1603.6          424.9
621                  700.3            291.6     1628.2          424.7
622                  702.1            288.7     1591.1          443.3
623                  696.5            285.6     1634.8          453.8

[624 rows x 8 columns]
```

[112]:
```python
# Loading third dataset
cpi_data = pd.read_excel('/content/drive/MyDrive/Data sets/cpidata.xlsx')
cpi_data
```

[112]:
```
     Year      Jan      Feb      Mar      Apr      May      Jun      Jul  \
0    1913    9.800    9.800    9.800    9.800    9.700    9.800    9.900
1    1914   10.000    9.900    9.900    9.800    9.900    9.900   10.000
2    1915   10.100   10.000    9.900   10.000   10.100   10.100   10.100
3    1916   10.400   10.400   10.500   10.600   10.700   10.800   10.800
4    1917   11.700   12.000   12.000   12.600   12.800   13.000   12.800
..    …       …        …        …        …        …        …        …
107  2020  257.971  258.678  258.115  256.389  256.394  257.797  259.101
108  2021  261.582  263.014  264.877  267.054  269.195  271.696  273.003
109  2022  281.148  283.716  287.504  289.109  292.296  296.311  296.276
110  2023  299.170  300.840  301.836  303.363  304.127  305.109  305.691
111  2024  308.417      NaN      NaN      NaN      NaN      NaN      NaN

         Aug      Sep      Oct      Nov      Dec
0      9.900   10.000   10.000   10.100   10.000
1     10.200   10.200   10.100   10.200   10.100
2     10.100   10.100   10.200   10.300   10.300
3     10.900   11.100   11.300   11.500   11.600
4     13.000   13.300   13.500   13.500   13.700
..       …        …        …        …        …
107  259.918  260.280  260.388  260.229  260.474
108  273.567  274.310  276.589  277.948  278.802
109  296.171  296.808  298.012  297.711  296.797
110  307.026  307.789  307.671  307.051  306.746
111      NaN      NaN      NaN      NaN      NaN

[112 rows x 13 columns]
```

[113]:
```python
# Retaining only relevant columns
cpi_data = cpi_data.iloc[:,1:]
```

```
# Re-arranging cpi values into a single column in a new data frame
cpi_data_new = pd.DataFrame()
data_list = []
for i in range(len(cpi_data)):
    x = list(cpi_data.iloc[i])
    data_list += x

cpi_data_new['CPI']=data_list

# Adding a date column
start_date = pd.to_datetime('1913-01')
cpi_data_new['Date'] = pd.
 ↪date_range(start=start_date,freq='MS',periods=len(cpi_data_new))

cpi_data_new
```

[113]:
```
      CPI       Date
0     9.8 1913-01-01
1     9.8 1913-02-01
2     9.8 1913-03-01
3     9.8 1913-04-01
4     9.7 1913-05-01
…    …         …
1339  NaN 2024-08-01
1340  NaN 2024-09-01
1341  NaN 2024-10-01
1342  NaN 2024-11-01
1343  NaN 2024-12-01

[1344 rows x 2 columns]
```

[114]:
```
# Merging all three data sets
merged_data = pd.merge(filtered_ex_rate, filtered_price_indices,on='Date').
 ↪dropna()
merged_data = pd.merge(cpi_data_new, merged_data, on='Date').dropna()
merged_data.set_index('Date',inplace=True)
merged_data.tail()
```

[114]:
```
                CPI    US/CAD  Total Index  Tot. Index (Ex. Energy)  \
Date
2016-12-01  241.432  1.332935        388.8                    304.7
2017-01-01  242.839  1.319090        398.4                    310.0
2017-02-01  243.603  1.310989        409.9                    328.2
2017-03-01  243.801  1.338752        393.5                    321.2
2017-04-01  244.524  1.344395        410.0                    326.4
```

```
            Energy Index  Metals & Minerals Index  Agriculture Index  \
Date
2016-12-01         919.9                    494.9              207.4
2017-01-01         953.6                    500.9              212.4
2017-02-01         953.8                    540.1              217.7
2017-03-01         898.4                    516.3              213.7
2017-04-01         959.9                    524.1              213.7

            Fish Index  Forestry Index
Date
2016-12-01      1239.7           357.1
2017-01-01      1329.9           360.5
2017-02-01      1361.2           389.7
2017-03-01      1413.0           393.5
2017-04-01      1424.7           411.0
```

[115]:
```python
# Deflating the data by the US CPI
deflated_data = merged_data.iloc[:,1:5].copy()
for col in deflated_data.columns:
  if col != 'US/CAD':
    deflated_data[col]= deflated_data[col]/merged_data['CPI']

deflated_data.columns= [f'Deflated {col}' if col!='US/CAD' else col for col in␣
 ↪deflated_data.columns]
deflated_data
```

[115]:
```
              US/CAD  Deflated Total Index  Deflated Tot. Index (Ex. Energy)  \
Date
1972-01-01  1.005922              2.433090                          2.433090
1972-02-01  1.004583              2.430993                          2.433414
1972-03-01  0.998395              2.442029                          2.446860
1972-04-01  0.995594              2.438554                          2.445783
1972-05-01  0.988665              2.449519                          2.459135
…             …                   …                                 …
2016-12-01  1.332935              1.610391                          1.262053
2017-01-01  1.319090              1.640593                          1.276566
2017-02-01  1.310989              1.682656                          1.347274
2017-03-01  1.338752              1.614021                          1.317468
2017-04-01  1.344395              1.676727                          1.334838

            Deflated Energy Index
Date
1972-01-01               2.433090
1972-02-01               2.416465
1972-03-01               2.417874
1972-04-01               2.404819
1972-05-01               2.403846
```

```
…                           …
2016-12-01                  3.810183
2017-01-01                  3.926882
2017-02-01                  3.915387
2017-03-01                  3.684973
2017-04-01                  3.925586

[544 rows x 4 columns]
```

[116]:
```
logged_deflated_data = np.log(deflated_data.copy())
logged_deflated_data.columns = [f'LN {col}' for col in logged_deflated_data.
 ↪columns]
logged_deflated_data
```

[116]:
```
            LN US/CAD  LN Deflated Total Index  \
Date
1972-01-01   0.005904                 0.889162
1972-02-01   0.004573                 0.888300
1972-03-01  -0.001606                 0.892829
1972-04-01  -0.004416                 0.891405
1972-05-01  -0.011400                 0.895892
…                 …                       …
2016-12-01   0.287383                 0.476477
2017-01-01   0.276942                 0.495058
2017-02-01   0.270782                 0.520373
2017-03-01   0.291738                 0.478729
2017-04-01   0.295944                 0.516844

            LN Deflated Tot. Index (Ex. Energy)  LN Deflated Energy Index
Date
1972-01-01                             0.889162                  0.889162
1972-02-01                             0.889295                  0.882306
1972-03-01                             0.894806                  0.882889
1972-04-01                             0.894365                  0.877475
1972-05-01                             0.899810                  0.877070
…                                           …                        …
2016-12-01                             0.232740                  1.337677
2017-01-01                             0.244174                  1.367846
2017-02-01                             0.298083                  1.364914
2017-03-01                             0.275712                  1.304263
2017-04-01                             0.288810                  1.367516

[544 rows x 4 columns]
```

[117]:
```
# Creating a dataframe for variables in first differences
differenced_data = logged_deflated_data.copy().diff().dropna()
differenced_data.columns = [f'\u0394 {col}' for col in differenced_data.columns]
```

```
differenced_data
```

[117]:

```
            Δ LN US/CAD  Δ LN Deflated Total Index  \
Date
1972-02-01    -0.001332                  -0.000862
1972-03-01    -0.006179                   0.004530
1972-04-01    -0.002810                  -0.001424
1972-05-01    -0.006984                   0.004486
1972-06-01    -0.009441                  -0.000440
…                   …                         …
2016-12-01    -0.008118                   0.071923
2017-01-01    -0.010441                   0.018581
2017-02-01    -0.006160                   0.025316
2017-03-01     0.020956                  -0.041645
2017-04-01     0.004206                   0.038115

            Δ LN Deflated Tot. Index (Ex. Energy)  Δ LN Deflated Energy Index
Date
1972-02-01                               0.000133                   -0.006856
1972-03-01                               0.005510                    0.000583
1972-04-01                              -0.000440                   -0.005414
1972-05-01                               0.005444                   -0.000405
1972-06-01                              -0.000448                   -0.000403
…                                           …                          …
2016-12-01                               0.006919                    0.148871
2017-01-01                               0.011434                    0.030169
2017-02-01                               0.053910                   -0.002931
2017-03-01                              -0.022372                   -0.060651
2017-04-01                               0.013098                    0.063253

[543 rows x 4 columns]
```

[118]:

```python
# VAR model1 optimal lags
model1 = VAR(differenced_data[['Δ LN US/CAD','Δ LN Deflated Total Index']])
x1 = model1.select_order(maxlags=5)
x1.summary()
```

```
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
```

[118]:

	AIC	BIC	FPE	HQIC
0	-15.25	-15.24	2.376e-07	-15.25
1	-15.38*	-15.33*	2.098e-07*	-15.36*
2	-15.38	-15.30	2.100e-07	-15.35
3	-15.37	-15.26	2.120e-07	-15.32
4	-15.37	-15.23	2.116e-07	-15.31
5	-15.36	-15.19	2.130e-07	-15.29

[119]:
```
# Regression results for VAR model1
fitted_model1 = model1.fit(1)
fitted_model1.summary()
```

[119]:
```
  Summary of Regression Results
==================================
Model:                         VAR
Method:                        OLS
Date:           Tue, 19, Mar, 2024
Time:                     05:38:09
--------------------------------------------------------------------
No. of Equations:         2.00000    BIC:                   -15.3430
Nobs:                     542.000    HQIC:                  -15.3720
Log likelihood:           2638.71    FPE:                2.06996e-07
AIC:                     -15.3906    Det(Omega_mle):     2.04724e-07
--------------------------------------------------------------------
Results for equation Δ LN US/CAD
================================================================================
===============
                                  coefficient       std. error           t-stat
prob
--------------------------------------------------------------------------------
---------------
const                                0.000381         0.000598            0.638
0.524
L1.Δ LN US/CAD                       0.243684         0.045912            5.308
0.000
L1.Δ LN Deflated Total Index        -0.036608         0.018013           -2.032
0.042
================================================================================
===============

Results for equation Δ LN Deflated Total Index
================================================================================
===============
                                  coefficient       std. error           t-stat
prob
--------------------------------------------------------------------------------
---------------
```

```
const                                   -0.000414         0.001533           -0.270
0.787
L1.Δ LN US/CAD                          -0.157715         0.117741           -1.340
0.180
L1.Δ LN Deflated Total Index             0.248064         0.046194            5.370
0.000
================================================================================
===============

Correlation matrix of residuals
                             Δ LN US/CAD  Δ LN Deflated Total Index
Δ LN US/CAD                     1.000000                  -0.410141
Δ LN Deflated Total Index      -0.410141                   1.000000
```

[120]:

```
# Tests for model1
jb_test = jarque_bera(fitted_model1.resid)
print("Jarque-Bera test results:")
print("Statistic:", jb_test[0])
print("p-value:", jb_test[1])

granger_test = grangercausalitytests(differenced_data[['Δ LN US/CAD','Δ LN␣
↪Deflated Total Index']],1)
print('\nGranger Causality Test')
print(granger_test)

print('\nModel stability test')
print(fitted_model1.is_stable(verbose=True))

ljun_box = fitted_model1.test_whiteness()
print('\n',ljun_box)
```

```
Jarque-Bera test results:
Statistic: [1226.17371851  292.57850503]
p-value: [5.49237395e-267 2.93349175e-064]

Granger Causality
number of lags (no zero) 1
ssr based F test:         F=4.1302  , p=0.0426  , df_denom=539, df_num=1
ssr based chi2 test:   chi2=4.1532  , p=0.0416  , df=1
likelihood ratio test: chi2=4.1374  , p=0.0419  , df=1
parameter F test:         F=4.1302  , p=0.0426  , df_denom=539, df_num=1

Granger Causality Test
{1: ({'ssr_ftest': (4.130224392021003, 0.04261353422801978, 539.0, 1),
'ssr_chi2test': (4.153212653943198, 0.0415556094973795, 1), 'lrtest':
```

```
(4.13738095539793, 0.0419460203454895, 1), 'params_ftest': (4.1302243920209305,
0.04261353422801978, 539.0, 1.0)},
[<statsmodels.regression.linear_model.RegressionResultsWrapper object at
0x7d15261c0f40>, <statsmodels.regression.linear_model.RegressionResultsWrapper
object at 0x7d1526190f70>, array([[0., 1., 0.]])])}

Model stability test
Eigenvalues of VAR(1) rep
0.16985788026909565
0.321889722760335
True

 <statsmodels.tsa.vector_ar.hypothesis_test_results.WhitenessTestResults object.
H_0: residual autocorrelation up to lag 10 is zero: reject at 5% significance
level. Test statistic: 60.394, critical value: 50.998>, p-value: 0.007>
```

[121]:

```
# Normality test
# Jarque-Bera test simulation using Monte-Carlo technique
conventional_jb_test_statistic = sm.stats.jarque_bera(fitted_model1.resid)
conventional_jb_test_statistic[0][0]

np.random.seed(73)

jb1 = []
jb1stats = []
jb2 = []
jb2stats = []
y_jb = fitted_model1.fittedvalues.copy()

for i in range(1999):
  u1 = np.random.normal(0,1,size=y.shape[0])
  u2 = np.random.normal(0,1,size=y.shape[0])

  y_jb['Δ LN US/CAD'] = y_jb['Δ LN US/CAD'] + u1
  y_jb['Δ LN Deflated Total Index'] = y_jb['Δ LN Deflated Total Index'] + u2

  test_model = VAR(y_jb[['Δ LN US/CAD','Δ LN Deflated Total Index']]).fit(1)
  jb_test_statistic = sm.stats.jarque_bera(test_model.resid)
  jb1stats.append(jb_test_statistic[0][0])
  jb2stats.append(jb_test_statistic[0][1])

  if jb_test_statistic[0][0] > conventional_jb_test_statistic[0][0]:
    jb1.append(1)
  else:
    jb1.append(0)

  if jb_test_statistic[0][1] > conventional_jb_test_statistic[0][1]:
```

```
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
```

```
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
```

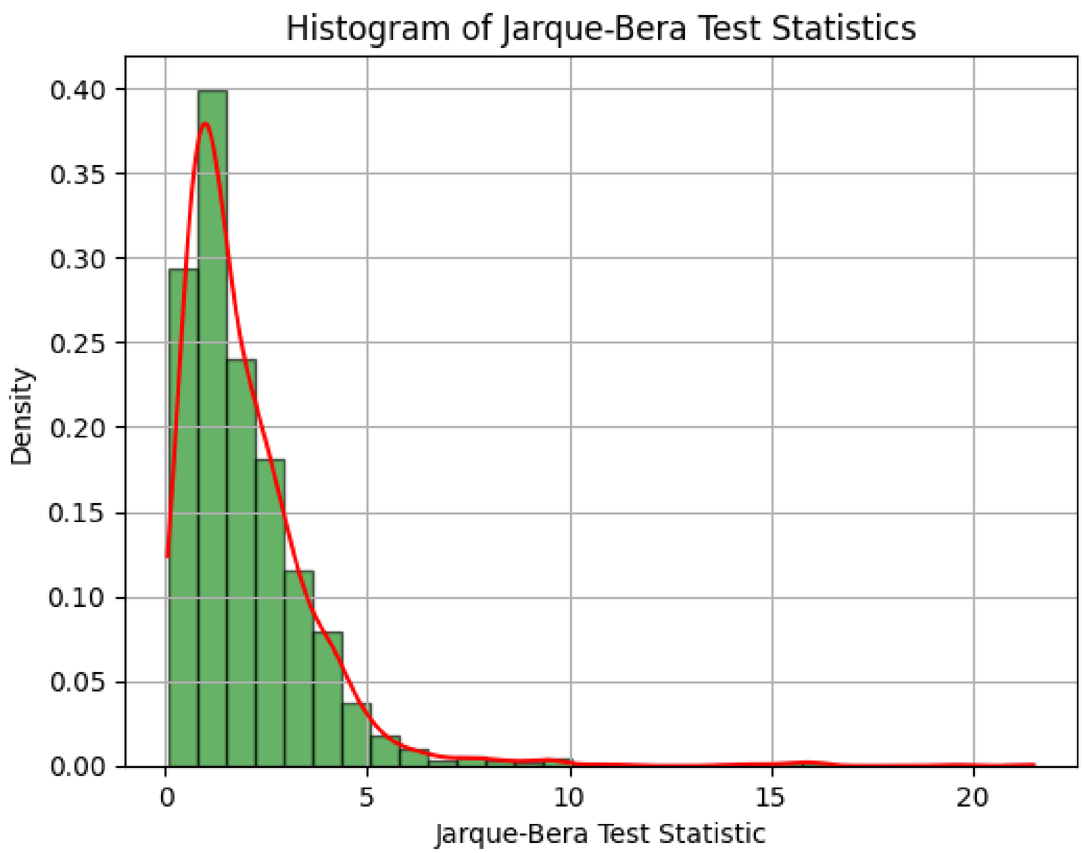
```
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
```

[122]:

```python
print(format(emp_pvalue1, '.4f'))
print(format(emp_pvalue2, '.4f'))
print(np.percentile(jb1stats,95))
print(np.percentile(jb2stats,95))
```

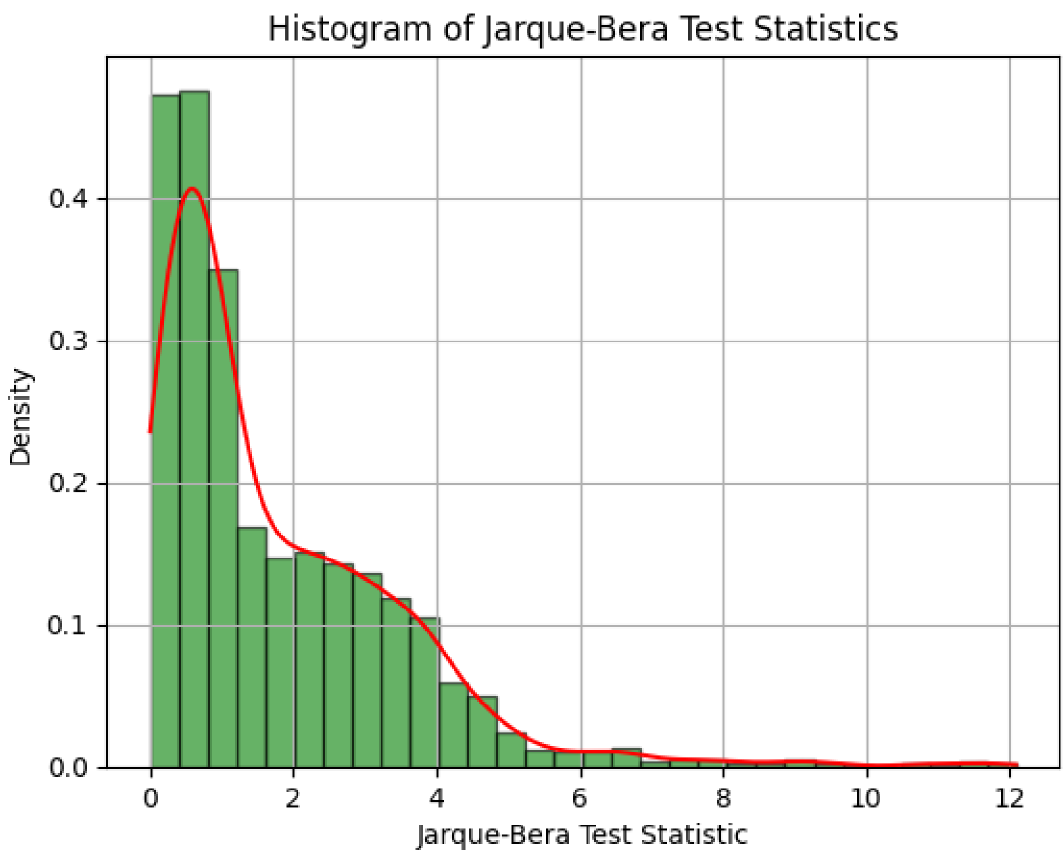
```
0.0000
0.0000
4.700858874650152
4.648232931573259
```

[123]:
```
kde = gaussian_kde(jb1stats)
x_vals = np.linspace(min(jb1stats), max(jb1stats), 1000)
y_vals = kde(x_vals)
plt.plot(x_vals, y_vals, color='red', label='KDE')

plt.hist(jb1stats, bins=30, density=True, alpha=0.6, color='g',␣
  ↪edgecolor='black')
plt.xlabel('Jarque-Bera Test Statistic')
plt.ylabel('Density')
plt.title('Histogram of Jarque-Bera Test Statistics')
plt.grid(True)
plt.show()
```



[124]:
```
kde = gaussian_kde(jb2stats)
x_vals = np.linspace(min(jb2stats), max(jb2stats), 1000)
y_vals = kde(x_vals)
plt.plot(x_vals, y_vals, color='red', label='KDE')

plt.hist(jb2stats, bins=30, density=True, alpha=0.6, color='g',␣
  ↪edgecolor='black')
```

```python
plt.xlabel('Jarque-Bera Test Statistic')
plt.ylabel('Density')
plt.title('Histogram of Jarque-Bera Test Statistics')
plt.grid(True)
plt.show()
```



[125]:
```python
# Autocorrelation test
# Simulation for Ljun-Box test using parametric bootstrapping
conventional_ljun_stat = ljun_box.test_statistic

np.random.seed(78)

lb = []
lbstats = []

y_lb = fitted_model1.fittedvalues.copy()
e1_mean = np.mean(fitted_model1.resid)[0]
e2_mean = np.mean(fitted_model1.resid)[1]
e1_std = np.std(fitted_model1.resid)[0]
e2_std = np.std(fitted_model1.resid)[1]
```

```
for i in range(1999):
  e1 = np.random.normal(e1_mean,e1_std,size=y_lb.shape[0])
  e2 = np.random.normal(e2_mean,e2_std,size=y_lb.shape[0])

  y_lb['Δ LN US/CAD'] = y_lb['Δ LN US/CAD'] + e1
  y_lb['Δ LN Deflated Total Index'] = y_lb['Δ LN Deflated Total Index'] + e2

  lb_test_model = VAR(y_lb[['Δ LN US/CAD','Δ LN Deflated Total Index']]).fit(1)
  ljun_box_statistic = lb_test_model.test_whiteness()
  sim_lb = ljun_box_statistic.test_statistic
  lbstats.append(sim_lb)

  if sim_lb > conventional_ljun_stat:
    lb.append(1)
  else:
    lb.append(0)

lb_emp_crit_stat = np.percentile(lb,95)
lb_emp_pvalue = np.mean(lb)
```

```
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeric.py:3502:
FutureWarning: In a future version, DataFrame.mean(axis=None) will return a
scalar mean over the entire DataFrame. To retain the old behavior, use
'frame.mean(axis=0)' or just 'frame.mean()'
  return mean(axis=axis, dtype=dtype, out=out, **kwargs)
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeric.py:3502:
FutureWarning: In a future version, DataFrame.mean(axis=None) will return a
scalar mean over the entire DataFrame. To retain the old behavior, use
'frame.mean(axis=0)' or just 'frame.mean()'
  return mean(axis=axis, dtype=dtype, out=out, **kwargs)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
```

```
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency MS
will be used.
  self._init_dates(dates, freq)
```

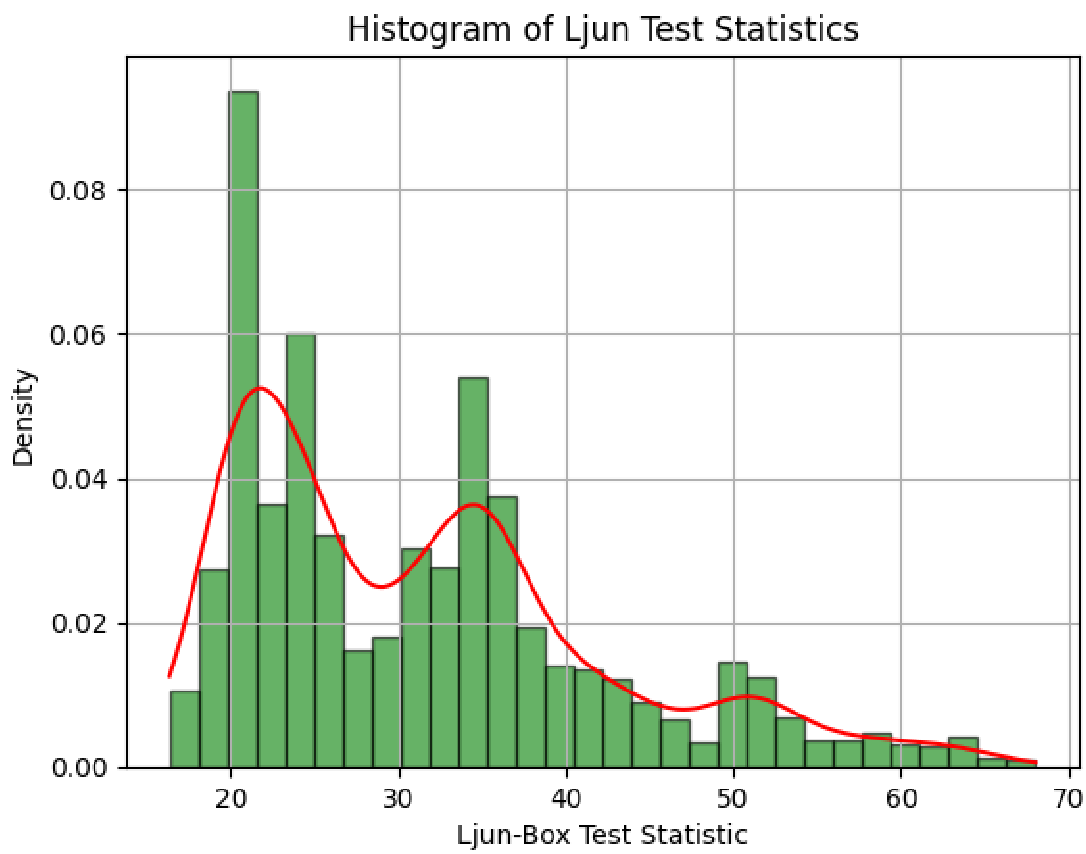
[126]:

```
print(lb_emp_pvalue)
lb_emp_crit_stat = np.percentile(lbstats,95)
print(lb_emp_crit_stat)
```

```
0.018509254627313655
52.98975423780811
```

[127]:

```
kde = gaussian_kde(lbstats)
x_vals = np.linspace(min(lbstats), max(lbstats), 1000)
y_vals = kde(x_vals)
plt.plot(x_vals, y_vals, color='red', label='KDE')

plt.hist(lbstats, bins=30, density=True, alpha=0.6, color='g',␣
 ↪edgecolor='black')
plt.xlabel('Ljun-Box Test Statistic')
plt.ylabel('Density')
plt.title('Histogram of Ljun Test Statistics')
plt.grid(True)
plt.show()
```



[128]:
```
# Simulation for Granger causality test using resampling
conventional_granger_stat = granger_test[1][0]['params_ftest'][0]

np.random.seed(78)

gc = []
gcstats = []

n = len(differenced_data)
y_gc = differenced_data.copy()

for i in range(1999):
  bootstrap_indices = np.random.choice(n, n, replace=True)
  bootstrap_data = y_gc.iloc[bootstrap_indices, :]
  granger_test_result = grangercausalitytests(bootstrap_data[['Δ LN US/CAD','Δ␣
↪LN Deflated Total Index']], 1, verbose=False)

  sim_gc_statistic = granger_test_result[1][0]['params_ftest'][0]
  gcstats.append(sim_gc_statistic)
```

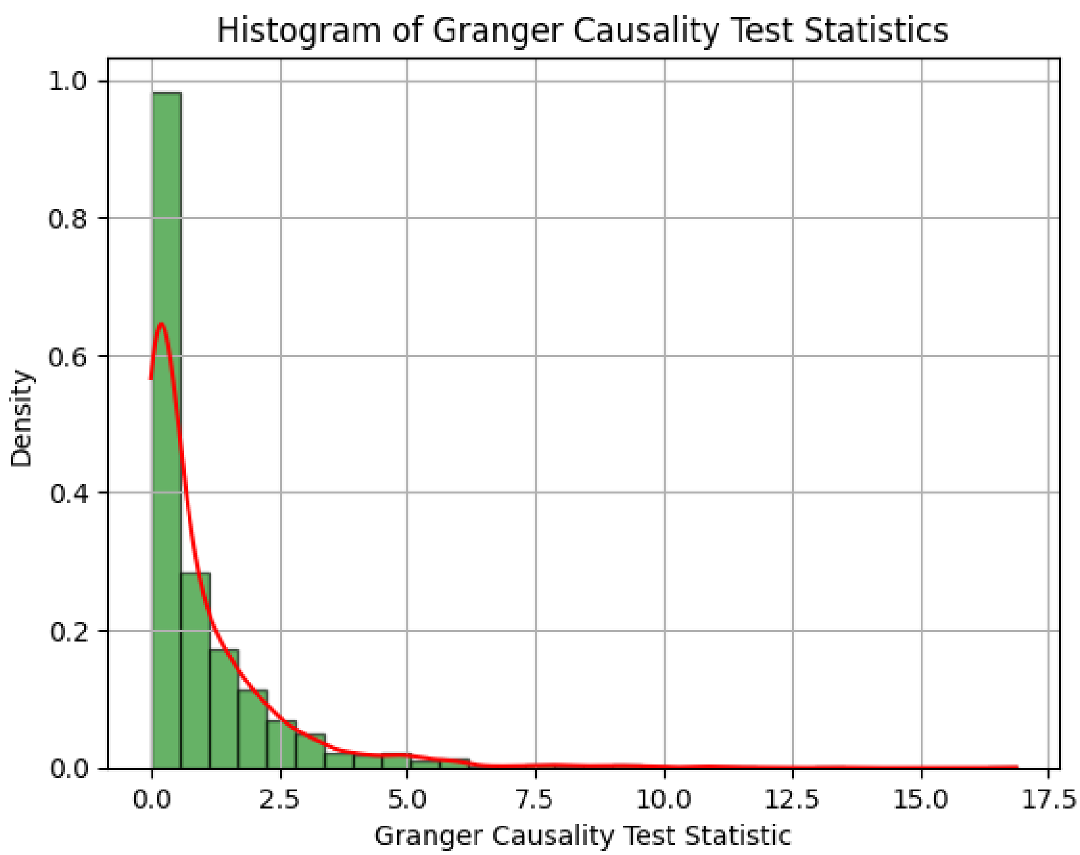
```python
  if sim_gc_statistic > conventional_granger_stat:
    gc.append(1)
  else:
    gc.append(0)


gc_emp_pvalue = np.mean(gc)
```

```
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/stattools.py:1545:
FutureWarning: verbose is deprecated since functions should not print results
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/stattools.py:1545:
FutureWarning: verbose is deprecated since functions should not print results
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/stattools.py:1545:
FutureWarning: verbose is deprecated since functions should not print results
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/stattools.py:1545:
FutureWarning: verbose is deprecated since functions should not print results
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/stattools.py:1545:
FutureWarning: verbose is deprecated since functions should not print results
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/stattools.py:1545:
FutureWarning: verbose is deprecated since functions should not print results
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/stattools.py:1545:
FutureWarning: verbose is deprecated since functions should not print results
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/stattools.py:1545:
FutureWarning: verbose is deprecated since functions should not print results
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/stattools.py:1545:
FutureWarning: verbose is deprecated since functions should not print results
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/stattools.py:1545:
FutureWarning: verbose is deprecated since functions should not print results
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/stattools.py:1545:
FutureWarning: verbose is deprecated since functions should not print results
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/stattools.py:1545:
FutureWarning: verbose is deprecated since functions should not print results
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/stattools.py:1545:
FutureWarning: verbose is deprecated since functions should not print results
  warnings.warn(
```

```
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/stattools.py:1545:
FutureWarning: verbose is deprecated since functions should not print results
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/stattools.py:1545:
FutureWarning: verbose is deprecated since functions should not print results
  warnings.warn(
```

[129]:

```
kde = gaussian_kde(gcstats)
x_vals = np.linspace(min(gcstats), max(gcstats), 1000)
y_vals = kde(x_vals)
plt.plot(x_vals, y_vals, color='red', label='KDE')

plt.hist(gcstats, bins=30, density=True, alpha=0.6, color='g',␣
 ↪edgecolor='black')
plt.xlabel('Granger Causality Test Statistic')
plt.ylabel('Density')
plt.title('Histogram of Granger Causality Test Statistics')
plt.grid(True)
plt.show()
```



[130]:
```
print(gc_emp_pvalue)
print(np.percentile(gcstats,95))
print(conventional_granger_stat)
```

```
0.043521760880440220
3.884474383383811
4.1302243920209305
```