EC306: Time series Econometrics Assignment

u2235356

April 10, 2025

1 Question 1

(a) Figure 1 shows that the series presents a slow exponential growth, SACF with a linear decay and SPACF with a significant lag, suggesting unit root behaviour.

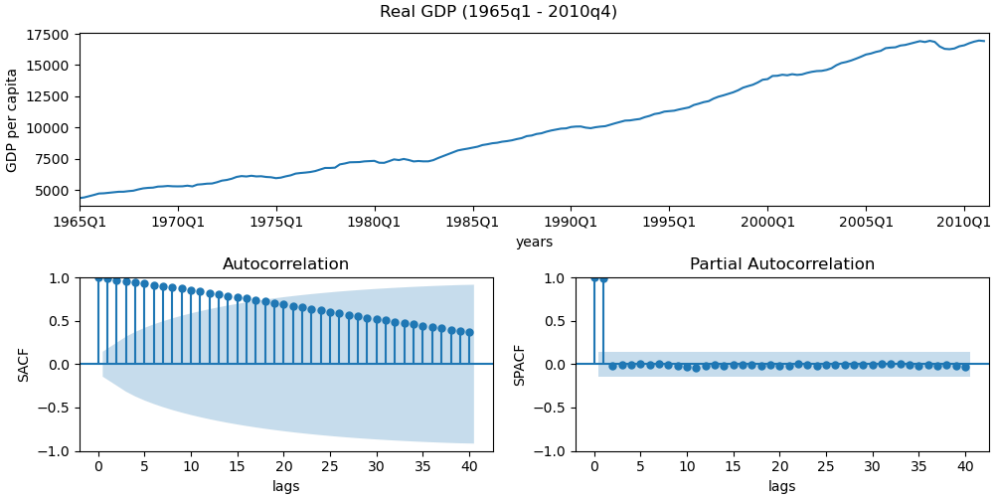

Figure 1: real GDP per capita

On conducting the following ADF test with the following auxillary equation:

$$\Delta GPD_t = \mu + \alpha t + \gamma GDP_{t-1} + \sum_{j=1}^{p} \delta_j \Delta GDP_{t-j} + \varepsilon_i$$

With the following null and alternative hypothesis:

$$H_0 : \gamma = 0 \quad \text{(non-stationary with non-zero drift)}$$
$$H_A : \gamma < 1 \quad \text{(stationary around trend with non-zero drift)}$$

yielded an insignificant test statistic of -2.02, confirming the hypothesis that the series is at least $I(1)$ [1]. Then, taking the first difference suggests a stationary process as seen in Figure 2 with the series returning to its mean frequently and generally stable SACF and SPACF.

Furthermore, another ADF test confirms this hypothesis with a test statistic of -4.23, which is significant at the 1% level. Therefore, the series is confirmed to be $I(1)$, and the chosen transformation is the first difference of the log-transformed real GDP per capita. Following the arguments of Stock & Watson (2020), the series is log-transformed as:

(a) Usually, the variance of the series is proportional to the level of the series; therefore, the variance of the log-transformed series is approximately constant, mitigating heteroskedasticity.

(b) It provides a meaningful interpretation of quarterly growth rate.

[1] Following Pesaran (2015), p is chosen automatically that minimises AIC

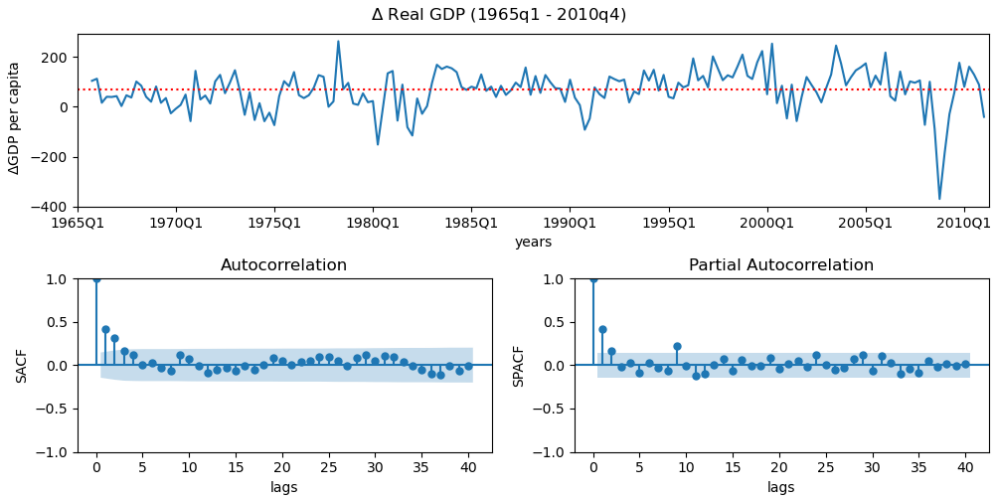

Figure 2: first-differenced real GDP per capita

(b) The results from the previous question indicate that real GDP per capita is non-stationary while growth rates are stationary. The non-stationary level of GDP per capita implies that the long-term trend is influenced by structural factors, such as capital accumulation and technological progress, which are persistent and evident from the ACF in Figure 1. On the other hand, the stationarity of its growth rates indicates short-run fluctuations consistent with the real business cycle, with transitory shocks – such as changes in investment, productivity, or policy- characterised by frequent deviations from and reversions to the mean, and insignificant ACF as seen in Figure 2.

(c) The models yielded the following results:

Table 1: Summary of models

model	AR(2)	ARMA(1, 1)
log-liklihood	627.042	626.588
AIC	-1246.084	-1245.176
BIC	-1233.247	-1232.338
HQIC	-1240.881	-1239.972
# observations	183	183

The lag structure was chosen for all models using the Hendry method, and in-sample information criteria were minimised to achieve the best fit. For both autoregressive and moving average components, the modelling process started with four lags due to the quarterly nature of the series and iteratively reduced based on the significance of lags and information criteria.

Furthermore, both specifications passed the Ljung-Box test, indicating no serial correlation in the errors. Based on the sample fit, the AR(2) performed best based on all information criteria and log-likelihood.

(d) Table 2 contains the out-of-sample root mean squared forecast error (RMSFE) and mean absolute forecast error (MAFE). Based on these metrics, the ARMA(1, 1) model forecasts are marginally better out-of-sample than AR(2) in both 1-step and 4-step forecasts.

Model	#-step	RMSFE	MAFE
AR(2)	1-step	0.00423	0.00339
AR(2)	4-step	0.0041	0.00322
ARMA(1, 1)	1-step	0.00422	0.00338
ARMA(1, 1)	4-step	0.00408	0.0032

Table 2: out-of-sample forecast metrics

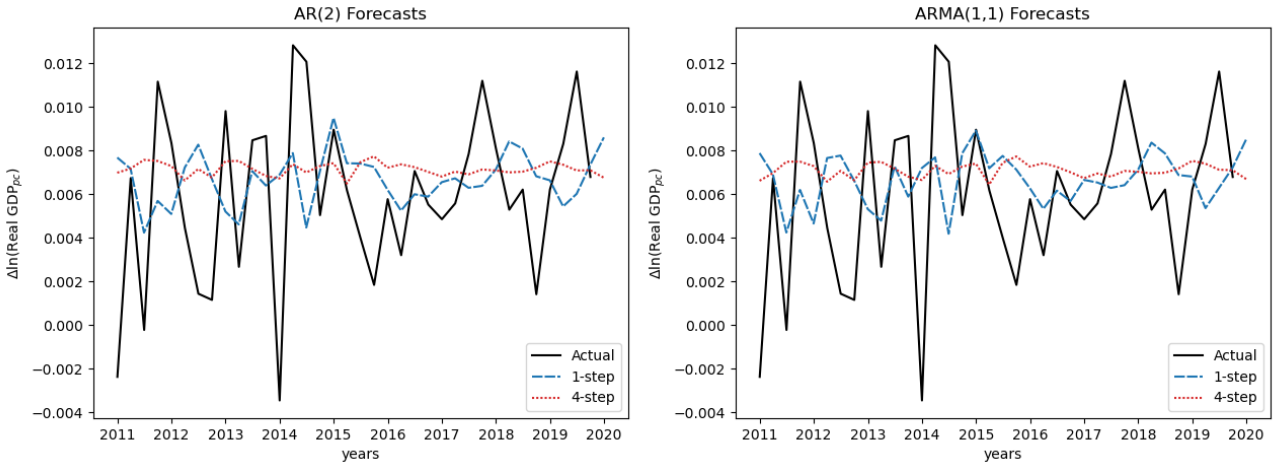

Figure 3: 1-step and 4-step forecasts

(e) The direct step forecast performed slightly worse than the AR(2) 4-step forecast with an RMSFE of 0.00417 and MAFE of 0.00332. This is unexpected as a direct-step regression is explicitly trained for the 4-step horizon. However, it might be the case because AR(2) might be a better approximation of the true data-generating process.

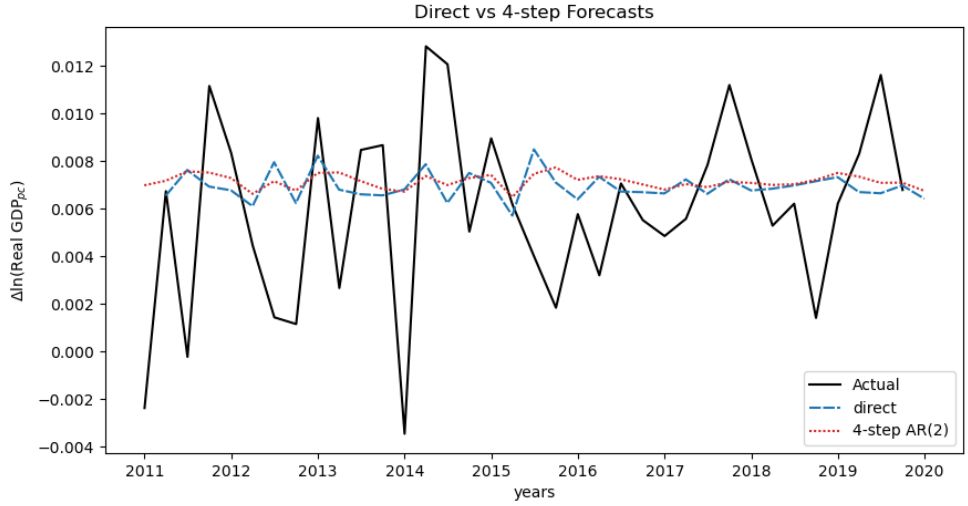

Figure 4: direct step forecasts

2 Question 2

Firstly, to model interest rate volatility, the order of integration of federal funds rates is determined using the ADF test, as seen in Table 3. Since rates are non-stationary, differenced rates are used for GARCH modelling. Additionally, differenced rates are modelled as ARMA(2, 1) based on Hendry's method, and

the residuals were tested for ARCH effects. The ARCH effects test concluded the presence of conditional heteroskedasticity with a statistic of 70.731 at the 1% significance level.

series	τ	p-value	decision
federal funds rate (FFR)	-2.749	0.066	at least I(1)
Δ FFR	-5.378	0.000	I(0)
BAA - AAA	-4.330	0.000	I(0)
BAA - GS30	-4.581	0.000	I(0)
inflation rate	-3.073	0.03	I(0)
volatility (FFR volatility)	-3.142	0.024	I(0)

Table 3: ADF tests of series

Secondly, the residuals from the ARIMA(2, 1) were modelled using Threshold GARCH(1, 1) as asymmetry provided a better fit in terms of AIC and significance of terms. The central hypothesis is that the spread between riskier (BAA 30-year corporate bond) projects and safer (AAA or government 30-year bonds) projects is positively associated with higher rate volatility. This indicates investment rationing since riskier projects have a higher cost of capital as compensation for the additional risk undertaken by an investor. So, they must be selective in their investments to choose the highest payoff projects compared to other projects.

Table 4: GARCH in mean with different spreads

	BAA - AAA	BAA - GS30
constant	0.976***	1.894***
	(0.022)	(0.032)
inflation rate	-21.022***	-46.854***
	(2.748)	(4.116)
volatility$_{t-1}$	0.779***	0.899***
	(0.043)	(0.065)
Observations	563	563
R^2	0.369	0.296
Adjusted R^2	0.367	0.293
Residual Std. Error	0.347	0.520
Residual ADF test	-4.957***	-4.482***
F Statistic	163.676***	117.699***
Note:	*p<0.1; **p<0.05; ***p<0.01	

So, BAA spreads must be positively associated with rate volatility, which is confirmed in Table 4. Therefore, this evidence suggests that investment rationing occurs when there is greater interest rate volatility. Additionally, all series in the regressions are stationary, as seen in Table 3, and the residuals are stationary (illustrated by residual ADF tests in 4), suggesting that the observed association is not spurious.

3 Question 3

First, all quarterly frequency series are transformed to be stationary based on the ADF tests on the sample from 1965Q1 to 2010Q4, as shown in Table 5. The main reasoning for choosing each variable to include in the VAR to forecast GDP is:

1. **Real Investment**: Investment is a significant component of GDP and improves productivity by investing in technology or labour to create more products for consumption and exports.

2. **Federal Funds Rates**: The federal funds rate is a primary instrument used by the US Central Bank to control the cost of capital for firms. This can affect both investment and GDP alike.

series	τ	p-value	decision
$\Delta \ln \text{GDP}_{\text{pc}}$	-14.77	0.000	I(0)
$\Delta \ln$ Real Investment	-7.01	0.000	I(0)
Δ Federal Funds rate	-6.52	0.000	I(0)

Table 5: ADF tests of series

Additionally, the results from the Granger-Causality test, as seen in table 6, indicate that investment and federal funds rates improve forecasts of GDP growth rates and vice versa. The ordering of the variables is not considered, as forecasting is the primary objective.

	dlgdp_x	dlrinv_x	dffr_x
dlgdp_y	1.000	0.000	0.007
dlrinv_y	0.003	1.000	0.000
dffr_y	0.012	0.000	1.000

Table 6: P-value matrix from Granger-Causality tests

Finally, the autoregressive lags are determined by Hendry's method, starting with four lags and iteratively reducing them to 2 lags as AIC improved with the presence of significant lags. VAR(2) specification also passed the Portmanteau test, confirming no serial correlation in the errors. This specification is used to

step	RMSFE	MAFE
1-step	0.00465	0.00349
4-step	0.00429	0.00325

Table 7: VAR(2) forecast metrics

make 1-step and 4-step forecasts from 2011Q1 to 2019Q4 to compare its performance to the univariate models from question 1. From table 7, it is clear that VAR(2) performed worse than AR(2) and ARMA(1, 1) in out-of-sample performance in both 1-step and 4-step forecasts. Since VAR models are large, they are susceptible to overparameterisation, leading to overfitting, which introduces estimation variance into the forecasts. Additionally, investments are relatively more volatile than the other series. This might introduce noise from its projections to the forecasts of the GDP growth rate. This might be the main reason for the worse performance.

4 Question 4

The inflation growth rate ($\Delta^2 \ln$ CPI) is the monthly frequency variable incorporated into the quarterly frequency. This is because large changes in inflation rates require monetary policy, such as adjusting rates to ensure all other economic variables, including GDP and investments, are stable. To introduce such dynamics to improve forecasts, the following bridging procedure is used:

1. **Model inflation growth rate in monthly frequency**: The inflation growth rate is modelled as an ARIMA(1, 3) because it minimises AIC and has all significant lags and no serial correlation of errors.

2. **Aggregation of monthly data**: The monthly inflation growth rate is aggregated to quarterly by taking the mean of rates within each quarter when the data is observable and forecasts are aggregated when the data is unobservable.

3. **VAR with aggregated inflation growth rate**: The VAR model is supplied with this aggregated monthly series of observed values and forecasts as another endogenous variable in the system to forecast the quarterly GDP growth rate.

This process led to comparatively better 1-step forecast performance than the VAR without inflation growth rate. However, it performed worse on the 4-step forecast, as seen in Table 8.

step	RMSFE	MAFE
1-step	0.00460	0.00346
4-step	0.00472	0.00361

Table 8: bridge forecast metrics

This might be the case because a 4-step horizon forecast requires the ARIMA model to forecast in the 12-step horizon, leading to an aggregation of errors passed onto the VAR. To mitigate such effects, different weighting regimes and more efficient estimators, such as MIDAS, may perform better than the current bridging setup.

5 Appendix

This section contains all the code used for each section.

5.1 Question 1

```python
# importing required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import adfuller, kpss
from statsmodels.tsa.arima.model import ARIMA

####################################
##### INITIAL PRE-PROCESSING #####
####################################
DIRECTORY = "D:/Academics/Year 3/EC306 Time Series Econometrics/Assignment"

# loading dataset
data = pd.read_excel(f"{DIRECTORY}/AssignmentQDATA.xlsx")
# re-formatting date variables
data['Date'] = pd.to_datetime(data['Date'], format = "%Y-%m-%d")
data['Date'] = data['Date'].dt.to_period('Q')

# setting date as the index
data.set_index('Date', inplace=True)
data = data.asfreq('Q')

####################################
####### SERIES EVALUATION ##########
####################################

class SeriesEvaluation:
    """
    This class takes in a time-series and contains
    many auxiliary functions to compute ADF test
    and plotting functions
    """
    def __init__(self, series: pd.Series, model: str, dropna = False) -> None:
        if dropna:
            series = series.dropna()
        self.series = series
```

```
        self.model = model

    def adf_test(self) -> None:
        print(f"Results of Dickey-Fuller Test for {self.series.name}:")
        dftest = adfuller(self.series, autolag="AIC", regression = self.model)
        self.adf_output = pd.Series(dftest[0:4], index=["Test Statistic", "p-value",
                                                        "# Lags Used", "Number of Observations Used"])
        for key, value in dftest[4].items():
            self.adf_output["Critical Value (%s)" % key] = value
        print(self.adf_output)

    def plotting(self) -> None:
        fig, axd = plt.subplot_mosaic([['series', 'series'],
                                ['sacf', 'spacf']],
                               figsize=(10, 5),
                               layout="constrained")
        try:
            datetime_index = self.series.index.to_timestamp()
        except:
            datetime_index = self.series.index

        start_period = min(self.series.index)
        end_period = max(self.series.index)
        yearly_ticks_periods = [pd.Period(f'{year}Q1') for year in range(start_period.year,
        ↪  end_period.year + 1, 5)]
        yearly_ticks = [period.to_timestamp() for period in yearly_ticks_periods]

        # plotting series
        axd['series'].plot(datetime_index, self.series)
        axd['series'].set_xlabel("years")
        axd['series'].set_xlim(start_period, end_period)
        axd['series'].set_xticks(yearly_ticks)
        axd['series'].set_xticklabels([f"{period}" for period in yearly_ticks_periods], rotation=0)
        axd['series'].set_ylabel(f"{self.series.name}")

        # plotting SACF
        plot_acf(self.series, ax = axd['sacf'], lags = 40)
        axd['sacf'].set_xlabel("lags")
        axd['sacf'].set_ylabel("SACF")

        # plotting SPACF
        plot_pacf(self.series, ax = axd['spacf'], lags = 40)
        axd['spacf'].set_xlabel("lags")
        axd['spacf'].set_ylabel("SPACF")
        fig.suptitle(f'{self.series.name} ({start_period} - {end_period})')

####################################
######## CODE FOR FIGURE 1 ########
####################################
gdp = SeriesEvaluation(series = data['GDPC1_65']['1965Q1':'2010Q4'],
                       model = "ct")
gdp.adf_test()
gdp.plotting()

####################################
########  CODE FOR FIGURE 2 ########
####################################

# taking the first difference of real GDP per capita
dgdp = SeriesEvaluation(series = data['GDPC1_65']['1965Q1':'2010Q4'].diff(),
                       model = "c", dropna=True)
dgdp.adf_test()
dgdp.plotting()

##########################################
##### MODELS AR(2), ARMA(1, 1) - 1(C) #####
##########################################
```

```python
# first-differenced log-transforming real GDP per capita
data['dlgdp'] = np.log(data['GDPC1_65']).diff()
data.dropna(inplace=True)
train = data.loc[:"2010Q4", 'dlgdp']
test = data.loc["2011Q1":"2019Q4", 'dlgdp']

# defining the ARIMA model
model_1 = ARIMA(endog = train, order = (1, 0, 1)).fit()

# defining the AR models
model_3 = ARIMA(endog = train, order = (2, 0, 0)).fit()

# formatting the results in a table
results = pd.DataFrame({
    'model': ['AR(2)', 'ARMA(1, 1)'],
    'log-liklihood':[model_3.llf, model_1.llf],
    'AIC': [model_3.aic,  model_1.aic],
    'BIC':[model_3.bic,  model_1.bic],
    'HQIC':[model_3.hqic,  model_1.hqic],
    '# observations': [model_3.nobs,  model_1.nobs]
})

# printing the results as a table
print(results.T.to_latex(float_format="%.3f", header = False, label = "fig:tab_1",
                         caption = "Summary of models", column_format = "cccc"))
results

##################################
########### 1(D) ###############
##################################

# creating functions for evaluation
class ForecastEvaluation:
    def __init__(self, actual, forecasts):
        self.actual = actual
        self.forecasts = forecasts
        self.combined = pd.DataFrame({'forecast': self.forecasts, 'actual': self.actual}).dropna()

    def rmsfe(self):
        result = np.sqrt(np.mean((self.combined['actual'] - self.combined['forecast'])**2))
        return result

    def mae(self):
        result = np.mean(np.abs(self.combined['actual'] - self.combined['forecast']))
        return result

    def result_table(self):
        print(f"RMSFE: {round(self.rmsfe(), 5)}, MAE: {round(self.mae(), 5)}")

# creating a function for getting rolling forecast
def rolling_forecast(train_index, end_index, order, step, data):

    train_index = data.index.get_loc(train_index)
    end_index = data.index.get_loc(end_index)

    # setting the train data and test data
    train = data.iloc[:train_index - step]
    test = data.iloc[train_index - step + 1 :end_index+1]
    # Initialize storage for 4-step-ahead predictions
    predictions = pd.DataFrame(index=test.index, columns=['forecast', "upper_conf", "lower_conf"])

    for i in range(len(test)-step+1):  # Ensure we stop where a h-step forecast is possible
        # Fit ARIMA model on the current train set
        model = ARIMA(endog=train, order=order).fit()

        # Forecast 4 steps ahead
```

```
        forecasts = model.get_forecast(steps=step)

        # Store the 4th step-ahead forecast along with the confidence intervals
        predictions.loc[test.index[i+step-1]] = [forecasts.predicted_mean.iloc[step-1],
                                                 forecasts.conf_int().iloc[step-1, 0],
                                                forecasts.conf_int().iloc[step-1, 1]]

        # Append the actual observed value to training set with correct index
        train = pd.concat([train, pd.Series(test.iloc[i], index=[test.index[i]])])

    # predictions.dropna(inplace=True)
    return predictions

dlgdp = np.log(data['GDPC1_65']).diff().dropna()
train_index = "2010Q4"
end_index = "2019Q4"

# getting the forecasts for the AR(2) model
ar2_1step = rolling_forecast(train_index, end_index, order = (2, 0, 0), step = 1, data = dlgdp)
ar2_4step = rolling_forecast(train_index, end_index, order = (2, 0, 0), step = 4, data = dlgdp)

# getting the forecasts for the ARMA(1, 1) model
arma11_1step = rolling_forecast(train_index, end_index, order = (1, 0, 1), step = 1, data = dlgdp)
arma11_4step = rolling_forecast(train_index, end_index, order = (1, 0, 1), step = 4, data = dlgdp)



# Plot the results
fig, axes = plt.subplots(1, 2, figsize=(15, 5))
axes = axes.flatten()

#  AR(2)
axes[0].plot(test.index.to_timestamp(),  test, label='Actual', color='black')
axes[0].plot(ar2_1step.index, ar2_1step['forecast'], label='1-step', color='tab:blue', linestyle=(0, (5,
↪  1)))
axes[0].plot(ar2_4step.index, ar2_4step['forecast'], label='4-step', color='tab:red', linestyle=(0, (1,
↪  1)))
axes[0].set_title('AR(2) Forecasts')
axes[0].set_ylabel('$\Delta$ln(Real GDP$_{pc}$)')
axes[0].set_xlabel('years')
axes[0].legend()

# ARMA(1, 1)
axes[1].plot(test.index.to_timestamp(), test, label='Actual', color='black')
axes[1].plot(arma11_1step.index, arma11_1step['forecast'], label='1-step', color='tab:blue',
↪  linestyle=(0, (5, 1)))
axes[1].plot(arma11_4step.index, arma11_4step['forecast'], label='4-step', color='tab:red',
↪  linestyle=(0, (1, 1)))
axes[1].set_title('ARMA(1,1) Forecasts')
axes[1].set_ylabel('$\Delta$ln(Real GDP$_{pc}$)')
axes[1].set_xlabel('years')
axes[1].legend()


# printing evaluations
print("AR(2)- 1 step evaluation")
ForecastEvaluation(actual = dlgdp, forecasts = ar2_1step['forecast']).result_table()
print("AR(2)- 4 step evaluation")
ForecastEvaluation(actual = dlgdp, forecasts = ar2_4step['forecast']).result_table()

print("ARMA(1, 1)- 1 step evaluation")
ForecastEvaluation(actual = dlgdp, forecasts = arma11_1step['forecast']).result_table()
print("ARMA(1, 1)- 4 step evaluation")
ForecastEvaluation(actual = dlgdp, forecasts = arma11_4step['forecast']).result_table()

##################################
########### 1(E) ###############
```

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

import statsmodels.api as sm
import pandas as pd
import numpy as np

# creating the dataset:
q1_e = pd.DataFrame({
    "dlgdp": dlgdp,
    "L4.dlgdp": dlgdp.shift(4),
    "L5.dlgdp": dlgdp.shift(5)
})
q1_e.dropna(inplace=True)

# creating a function for getting direct forecast
def direct_forecast(train_index, end_index, data):
    step = 1
    train_index = data.index.get_loc(train_index)
    end_index = data.index.get_loc(end_index)

    # setting the train data and test data
    train = data.iloc[:train_index - step].copy()
    test = data.iloc[train_index - step + 1:end_index+1].copy()

    # Initialize storage for 1-step-ahead predictions
    predictions = pd.DataFrame(index=test.index, columns=['forecast'])

    for i in range(len(test)-step+1):  # Ensure we stop where a h-step forecast is possible
        # separate the train into Y and X
        Y_train = train['dlgdp']
        X_train = train[['L4.dlgdp', 'L5.dlgdp']]
        X_train = sm.add_constant(X_train)

        # Fit OLS model
        model = sm.OLS(Y_train, X_train).fit()

        # define X_test & forecast 4-periods ahead
        X_test = np.asarray([1, test.iloc[i+step-1, 1], test.iloc[i+step-1, 2]])
        forecast = model.predict(exog=X_test.reshape(1, -1))
        # Store the forecast
        predictions.loc[test.index[i+step-1]] = forecast[0]
        # Append the actual observed value to training set with correct index
        train = pd.concat([train, test.iloc[i:i+1]])

    return predictions

# Call the function
direct_forecasts = direct_forecast(train_index="2011Q1", end_index="2019Q4", data=q1_e)

# Plot the results
fig, axes = plt.subplots(1, 1, figsize=(10, 5))
axes.plot(test.index.to_timestamp(),  test, label='Actual', color='black')
axes.plot(direct_forecasts.index, direct_forecasts['forecast'], label='direct', color='tab:blue',
↪  linestyle=(0, (5, 1)))
axes.plot(ar2_4step.index, ar2_4step['forecast'], label='4-step AR(2)', color='tab:red', linestyle=(0,
↪  (1, 1)))
axes.set_title('Direct vs 4-step Forecasts')
axes.set_ylabel('$\Delta$ln(Real GDP$_{pc}$)')
axes.set_xlabel('years')
axes.legend()
plt.savefig("D:/Academics/Year 3/EC306 Time Series
↪  Econometrics/Assignment/plots/q1e_direct_forecast.png")

# Conducting evaluation
print("Direct-step evaluation")
ForecastEvaluation(actual = dlgdp, forecasts = direct_forecasts['forecast']).result_table()
```

5.2 Question 2

```
import pandas as pd
from statsmodels.stats.diagnostic import het_arch
from statsmodels.tsa.arima.model import ARIMA
import statsmodels.api as sm
from arch import arch_model
from stargazer.stargazer import Stargazer

# reading the dataset
monthly_data = pd.read_excel("D:/Academics/Year 3/EC306 Time Series
↪  Econometrics/Assignment/InterestRates.xlsx")
monthly_data.set_index('DATE', inplace=True)
monthly_data = monthly_data.asfreq("MS")

# extracting the delta fed funds rate and spreads
dffr = monthly_data['FEDFUNDS'].diff()
dffr.dropna(inplace = True)

BAA_AAA = monthly_data['BAA'][min(dffr.index):max(dffr.index)] -
↪  monthly_data['AAA'][min(dffr.index):max(dffr.index)]
BAA_AAA.name = "BAA - AAA"

BAA_GS30 = monthly_data['BAA']["1977-02-01":] - monthly_data['GS30']["1977-02-01":]
BAA_GS30.name = "BAA - GS30"



# conducting ADF test
SeriesEvaluation(monthly_data['FEDFUNDS'], model = "c").adf_test()
SeriesEvaluation(dffr, model = "c").adf_test()
SeriesEvaluation(BAA_AAA, model = "c").adf_test()
SeriesEvaluation(BAA_GS30, model = "c").adf_test()
SeriesEvaluation(np.log(monthly_data['CPIAUCSL']).diff(2), model = "c", dropna = True).adf_test()

# ARIMA modelling
ffr_arima = ARIMA(endog = dffr, order = (2, 0, 1)).fit()
print(ffr_arima.summary())
# conducting ARCH Effects test
arch_test=het_arch(ffr_arima.resid, nlags = 6)
print(f"ARCH Effects Test: \nTest Statistic: {arch_test[0]}\nP-Value: {arch_test[1]}")
# GARCH modelling
garch_model = arch_model(ffr_arima.resid, vol='GARCH', p=1, o=1, q=1).fit(disp="off")
print(garch_model.summary())


# generating OLS dataset
OLS_data = pd.DataFrame({"vol": garch_model.conditional_volatility,
                         "cons": [1 for j in range(len(garch_model.conditional_volatility))],
                         "L1.vol": garch_model.conditional_volatility.shift(),
                         "BAA - AAA": BAA_AAA,
                         "BAA - GS30": BAA_GS30,
                         "inf_rate": np.log(monthly_data['CPIAUCSL']).diff(2)},
                 index = garch_model.conditional_volatility.index)
OLS_data.dropna(inplace = True)

# running OLS
model_1 = sm.OLS(OLS_data['BAA - AAA'], OLS_data[["cons", "L1.vol", "inf_rate"]]).fit()
model_2 = sm.OLS(OLS_data['BAA - GS30'], OLS_data[["cons", "L1.vol", "inf_rate"]]).fit()


# checking for spurious correlation
SeriesEvaluation(model_1.resid, model = "c").adf_test()
SeriesEvaluation(model_2.resid, model = "c").adf_test()
SeriesEvaluation(garch_model.conditional_volatility, model = "c").adf_test()
```

```
# printing output into latex using Stargazer
stargazer = Stargazer([model_1, model_2])
stargazer.custom_columns(['BAA - AAA', 'BAA - GS30'], [1, 1])
stargazer.rename_covariates({'cons': 'constant', 'L1.vol': 'L1 rate volatility', 'inf_rate': 'inflation
↪  rate'})
stargazer.title("GARCH in mean with different spreads")
stargazer.show_degrees_of_freedom(False)
print(stargazer.render_latex())
```

5.3 Question 3

```
# Conducting ADF tests on the original series
SeriesEvaluation(series = data['ffr'], model = "ct").adf_test()
SeriesEvaluation(series = data['GDP1_65'], model = "ct").adf_test()
SeriesEvaluation(series = data['RINV'], model = "ct").adf_test()

# differencing/log differencing series to induce stationarity
data['dlgdp'] = np.log(data['GDPC1_65']).diff()
data['dlrinv'] = np.log(data['RINV']).diff()
data['dffr'] = data['ffr'].diff()
data.dropna(inplace= True)

# Conducting ADF tests on transformed series
SeriesEvaluation(data['dlgdp'], model = "c").adf_test()
SeriesEvaluation(data['dlrinv'], model = "c").adf_test()
SeriesEvaluation(data['dffr'], model = "c").adf_test()

# conducting Granger-Causality tests

from statsmodels.tsa.stattools import grangercausalitytests

# defining function to create granger-causality matrix
def granger_causation_matrix(data, variables, p, test = 'ssr_chi2test'):
    df = pd.DataFrame(np.zeros((len(variables), len(variables))), columns=variables, index=variables)
    for c in df.columns:
        for r in df.index:
            test_result = grangercausalitytests(data[[r, c]], p)
            p_values = [round(test_result[i+1][0][test][1],4) for i in range(p)]
            min_p_value = np.min(p_values)
            df.loc[r, c] = min_p_value
    df.columns = [var + '_x' for var in variables]
    df.index = [var + '_y' for var in variables]
    return df

result = granger_causation_matrix(data[['dlgdp', 'dlrinv', 'dffr']], ['dlgdp', 'dlrinv', 'dffr'], p = 4)
print(result.to_latex())

# VAR modelling
from statsmodels.tsa.api import VAR

# splitting the dataset into train and test
train = data["1965Q4":"2010Q4"][['dlgdp', 'dlrinv', 'dbaa', 'dffr']]
test = data["2010Q4":"2019Q4"][['dlgdp', 'dlrinv', 'dbaa', 'dffr']]

# Fit the VAR model
model = VAR(train)
results = model.fit(maxlags=2, ic='aic')  # using AIC to choose optimal lag length

# Summary of the regression equation for dlgdp
print(results.summary())

# Conducting portmanteau test for serial correlation
q_test = results.test_whiteness(nlags=4)
print(q_test)
```

```
#######################################
# ROLLING VAR FORECASTS for comparison
#######################################
def rolling_forecast_var(train_index, end_index, step, data):
    train_index = data.index.get_loc(train_index)
    end_index = data.index.get_loc(end_index)

    # Set initial train and test data
    train = data.iloc[:train_index - step]
    test = data.iloc[train_index - step + 1 : end_index + 1]

    # Prepare storage for forecasts
    predictions = pd.DataFrame(index=test.index, columns=['forecast', "upper_conf", "lower_conf"])

    for i in range(len(test) - step + 1):  # Loop through rolling windows
        # Fit VAR model
        model = VAR(train)
        results = model.fit(maxlags=3, ic='aic')

        # Forecast
        forecast_mean, lower, upper = results.forecast_interval(train.values[-3:], steps=step)

        # Get index of forecasted date (step-ahead)
        forecast_date = test.index[i + step - 1]

        # Get column index for dlgdp
        dlgdp_idx = data.columns.get_loc('dlgdp')

        # Store step-ahead forecast
        predictions.loc[forecast_date] = [
            forecast_mean[step - 1][dlgdp_idx],
            upper[step - 1][dlgdp_idx],
            lower[step - 1][dlgdp_idx]
        ]

        # Add the current observation to the train set
        train = pd.concat([train, test.iloc[[i]]])

    predictions.dropna(inplace = True)
    return predictions

q3 = data[['dlgdp', 'dlrinv', 'dffr']]
var2_1step = rolling_forecast_var(train_index="2010Q4", end_index = "2019Q4", step = 1, data = q3)
var2_4step = rolling_forecast_var(train_index="2010Q4", end_index = "2019Q4", step = 4, data = q3)

print("VAR(4)- 1 step evaluation")
ForecastEvaluation(actual = series, forecasts = var2_1step['forecast']).result_table()
print("VAR(4)- 4 step evaluation")
ForecastEvaluation(actual = series, forecasts = var2_4step['forecast']).result_table()
```

5.4 Question 4

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.api import VAR
from statsmodels.tsa.arima.model import ARIMA
from pandas.tseries.offsets import MonthEnd

# Function to aggregate monthly data to quarterly
def monthly_to_quarterly(monthly_data, aggregation='mean'):
    if aggregation == 'mean':
        return monthly_data.resample('Q').mean()
```

```
    elif aggregation == 'sum':
        return monthly_data.resample('Q').sum()
    elif aggregation == 'last':
        return monthly_data.resample('Q').last()

# Function to forecast monthly inflation using ARIMA(1,0,3)
def forecast_monthly_inflation(inflation_data, steps=3):
    model = ARIMA(inflation_data, order=(1, 0, 3))
    results = model.fit()
    forecast = results.forecast(steps=steps)
    return forecast

# Function to implement bridging equations for quarterly forecasts
def bridging_equation_forecast(quarterly_data, monthly_data, start_date, end_date, h_step,
↪   aggregation='mean'):
    # Make a copy of the data to avoid modifying the original
    quarterly_data = quarterly_data.copy()

    # Ensure the quarterly data has a PeriodIndex with quarterly frequency
    if not isinstance(quarterly_data.index, pd.PeriodIndex):
        quarterly_data.index = pd.PeriodIndex(quarterly_data.index, freq='Q')

    # Ensure monthly data has a DatetimeIndex with monthly frequency
    if isinstance(monthly_data, pd.Series):
        monthly_df = monthly_data.to_frame(name='inf_rate')
    else:
        monthly_df = monthly_data.copy()

    # Convert monthly index to DatetimeIndex if it's not already
    if not isinstance(monthly_df.index, pd.DatetimeIndex):
        monthly_df.index = pd.DatetimeIndex(monthly_df.index)

    # Make sure the index is sorted
    monthly_df = monthly_df.sort_index()

    # Convert string dates to period indices if needed
    if isinstance(start_date, str):
        start_date = pd.Period(start_date, freq='Q')
    if isinstance(end_date, str):
        end_date = pd.Period(end_date, freq='Q')

    # Get dataframe indices
    start_idx = quarterly_data.index.get_loc(start_date)
    end_idx = quarterly_data.index.get_loc(end_date)

    # Initialize results dataframe
    predictions = pd.DataFrame(index=quarterly_data.loc[start_date:end_date].index,
                               columns=['forecast', 'upper_conf', 'lower_conf'])

    # Start rolling window forecasts
    for i in range(start_idx - h_step + 1, end_idx + 1):
        # Current quarter for forecast
        current_quarter = quarterly_data.index[i]
        target_quarter = quarterly_data.index[i + h_step - 1]

        # Prepare training data up to current quarter
        train_quarterly = quarterly_data.iloc[:i]

        # Find the last month of the current quarter as datetime
        last_month_dt = current_quarter.to_timestamp(how='E')

        # Get available monthly data up to last_month
        available_monthly = monthly_df[monthly_df.index <= last_month_dt]

        # Forecast monthly inflation for future months needed
        forecast_horizon = 3 * h_step  # Number of months to forecast
```

```
        # Generate monthly forecasts
        monthly_forecasts = forecast_monthly_inflation(available_monthly['inf_rate'],
        ↪  steps=forecast_horizon)

        # Create future monthly dates
        future_months = pd.date_range(start=last_month_dt + pd.DateOffset(months=1),
                                      periods=forecast_horizon, freq='M')

        # Combine available and forecasted monthly data
        complete_monthly = pd.concat([
            available_monthly,
            pd.DataFrame(monthly_forecasts, index=future_months, columns=['inf_rate'])
        ])

        # Aggregate to quarterly
        quarterly_inf = monthly_to_quarterly(complete_monthly, aggregation=aggregation)

        # Convert quarterly_inf index to PeriodIndex to match train_quarterly
        quarterly_inf.index = pd.PeriodIndex(quarterly_inf.index, freq='Q')

        # Merge with quarterly training data
        enhanced_quarterly = train_quarterly.copy()
        enhanced_quarterly['inf_rate'] = np.nan  # Initialize column

        # Fill in the inflation values for the quarters we have
        for idx in quarterly_inf.index:
            if idx in enhanced_quarterly.index:
                enhanced_quarterly.loc[idx, 'inf_rate'] = quarterly_inf.loc[idx, 'inf_rate']

        # Forward fill any missing values (should be minimal)
        enhanced_quarterly.fillna(method='ffill', inplace=True)

        # Fit VAR model with enhanced data
        try:
            model = VAR(enhanced_quarterly)
            results = model.fit(maxlags=2, ic='aic')

            # Forecast
            forecast_mean, lower, upper = results.forecast_interval(enhanced_quarterly.values[-2:],
            ↪  steps=h_step)

            # Store forecast for the target quarter
            dlgdp_idx = enhanced_quarterly.columns.get_loc('dlgdp')
            predictions.loc[target_quarter] = [
                forecast_mean[h_step - 1][dlgdp_idx],
                upper[h_step - 1][dlgdp_idx],
                lower[h_step - 1][dlgdp_idx]
            ]
        except Exception as e:
            print(f"Error for quarter {current_quarter}: {e}")
            continue

    predictions.dropna(inplace=True)
    return predictions


# Main function to run the forecasts
def run_bridging_forecast(data, inf_rate):
    # Ensure data has PeriodIndex
    if not isinstance(data.index, pd.PeriodIndex):
        data.index = pd.PeriodIndex(data.index, freq='Q')

    # Ensure inf_rate is a DataFrame with DatetimeIndex
    if isinstance(inf_rate, pd.Series):
        inf_rate_df = inf_rate.to_frame(name='inf_rate')
    else:
        inf_rate_df = inf_rate.copy()
```

```
    if not isinstance(inf_rate_df.index, pd.DatetimeIndex):
        inf_rate_df.index = pd.DatetimeIndex(inf_rate_df.index)

    # Original VAR forecasts (for comparison)
    q4 = data[['dlgdp', 'dlrinv', 'dffr']]

    # Bridging equation forecasts
    bridge_1step = bridging_equation_forecast(q4, inf_rate_df, "2010Q4", "2019Q4", 1,
    ↪  aggregation='mean')
    bridge_4step = bridging_equation_forecast(q4, inf_rate_df, "2010Q4", "2019Q4", 4,
    ↪  aggregation='mean')

    # Display evaluation metrics
    print("\nBridging Equation - 1 step evaluation")
    bridge1_eval = ForecastEvaluation(actual=series, forecasts=bridge_1step['forecast']).result_table()

    print("\nBridging Equation - 4 step evaluation")
    bridge4_eval = ForecastEvaluation(actual=series, forecasts=bridge_4step['forecast']).result_table()

    return {
        'bridge_1step': bridge_1step,
        'bridge_4step': bridge_4step,
        'evaluations': {
            'bridge_1step': bridge1_eval,
            'bridge_4step': bridge4_eval
        }
    }

# Modelling inflation growth rate
inf_rate = np.log(monthly_data['CPIAUCSL']).diff(2)
inf_rate.dropna(inplace = True)
inf_rate_arma = ARIMA(endog = inf_rate, order = (1, 0, 3)).fit()

# running bridge forecasts
results = run_bridging_forecast(data, inf_rate)
```

References

Pesaran, M. H. (2015), Unit Root Processes, *in* M. H. Pesaran, ed., 'Time Series and Panel Data Econometrics', Oxford University Press, p. 0.
URL: *https://doi.org/10.1093/acprof:oso/9780198736912.003.0015*

Stock, J. H. & Watson, M. W. (2020), *Introduction to econometrics*, The Pearson series in economics, fourth edition, global edition edn, Pearson, London.