CORE- FINANCIAL ECONOMETRICS

Program 1

(Q) 1. For the given data find out the time series component present in it.

2. Install Packages and calling installed packages related with time series in R

3. Understanding the function of ts packages in R

4. Plotting of time series data and conclude the possible analysis for the same.

AIM :

- Identify the components of a given time series: trend, seasonality, and noise.
- Install and utilize relevant time series packages in R for analysis.
- Plot the time series data for visual analysis.
- Provide a basic interpretation of the time series decomposition.

Algorithm :

Step 1: Install and Load Required Packages

Step 2: Load Time Series Data

Step 3: Decompose the Time Series

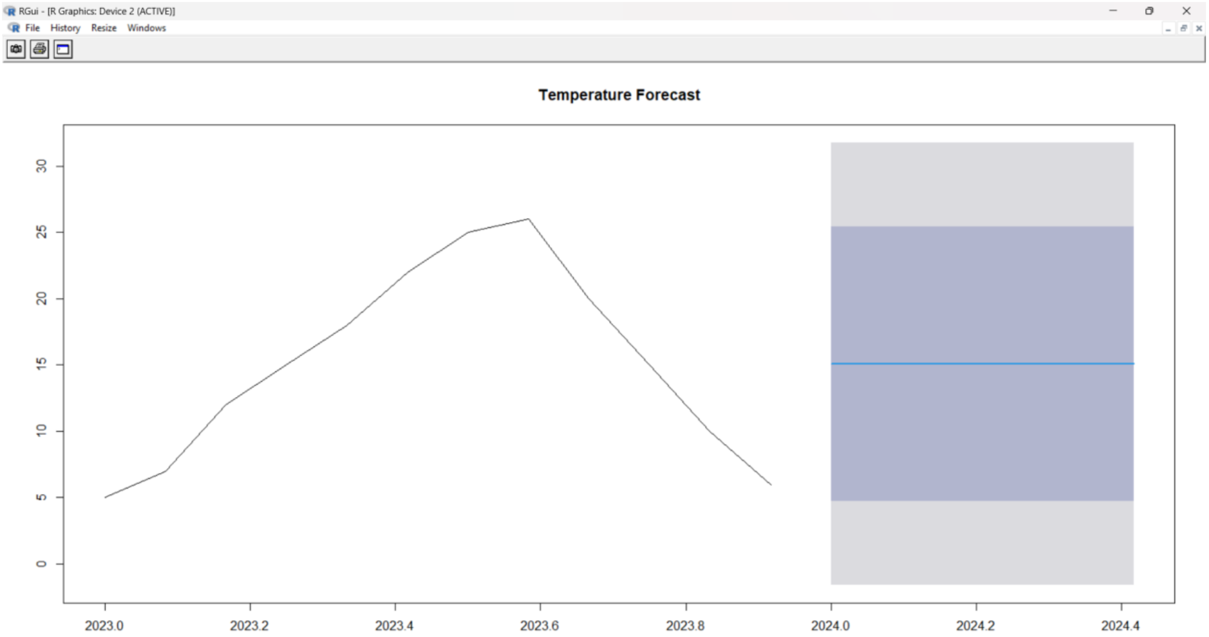
Step 4: Plot the Time Series and Components

Step 5: Analysis and Conclusion

Code:

```
# Install the forecast package if not already installed
install.packages("forecast")
library(forecast)

# Step 1: Create a vector of temperature data (e.g., average monthly temperatures)
temperature_data <- c(5, 7, 12, 15, 18, 22, 25, 26, 20, 15, 10, 6)

# Create a time series object starting from January 2023, with monthly frequency
temperature_ts <- ts(temperature_data, start=c(2023, 1), frequency=12)

# Step 2: Plot the time series
plot(temperature_ts, main="Monthly Average Temperature Data", ylab="Temperature (°C)", xlab="Month")

# Step 3: Forecast future temperatures (e.g., for the next 6 months) using meanf()
forecast_temperature <- meanf(temperature_ts, h=6)

# Plot the forecast
plot(forecast_temperature, main="Temperature Forecast")
```

OUTPUT:



RESULT:

The Above Program Is Verified And Executed Successfully.

Program 2

(Q)1. Create the moving average model for the given data - Simple Average.

2. Create the moving average model for the given data - Moving Average.

3. Create the moving average model for the given data - Weighted Moving Average.

4. Fit naives forecasting model for the given data.

5. Fit Smoothing forecasting model for the given data Exponential Smoothing (Holts Method).

AIM :

- ❖ Build three types of moving average models: Simple Average, Moving Average, and Weighted Moving Average.
- ❖ Fit a Naive forecasting model.
- ❖ Fit an Exponential Smoothing model using Holt's Method.
- ❖ Visualize and interpret the results for each model to make predictions.

Algorithm :

Step 1: Install and Load Required Packages

Step 2: Load Time Series Data

Step 3: Decompose the Time Series

Simple Average Moving Model

Moving Average Model

Weighted Moving Average Model

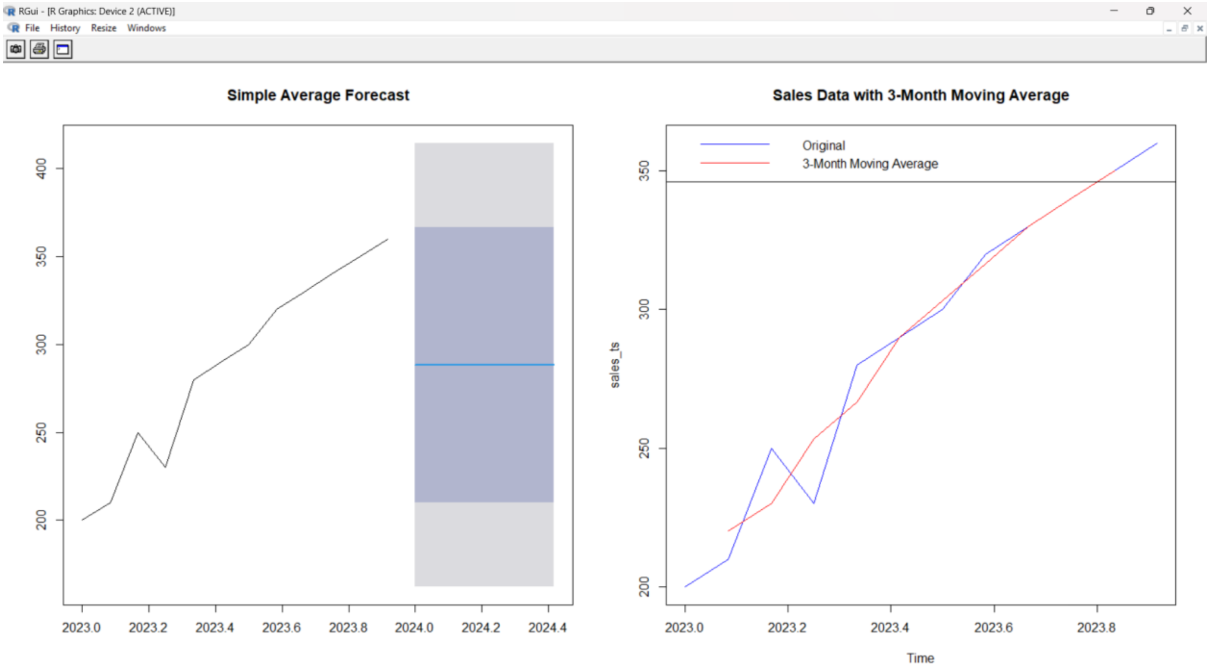
Step 4: Plot the Time Series and Components

Step 5: Analysis and Conclusion

Code:

```
# Install the forecast package if not already installed
install.packages("forecast")
library(forecast)

# Example sales data
sales_data <- c(200, 210, 250, 230, 280, 290, 300, 320, 330, 340, 350, 360)

# Convert the data into a time series
sales_ts <- ts(sales_data, start=c(2023, 1), frequency=12)

# Set up the plotting area to have 1 row and 2 columns (side by side)
par(mfrow=c(1, 2))

# Simple Average - Forecasting using the mean of all past data
simple_avg_forecast <- meanf(sales_ts, h=6)
plot(simple_avg_forecast, main="Simple Average Forecast")
# Moving Average with a window of 3 months
moving_avg_3 <- ma(sales_ts, order=3)
plot(sales_ts, main="Sales Data with 3-Month Moving Average", col="blue")
lines(moving_avg_3, col="red")
legend("topleft", legend=c("Original", "3-Month Moving Average"),
col=c("blue", "red"), lty=1)
# Reset plotting parameters back to default (optional)
par(mfrow=c(1, 1))
```

OUTPUT:



RESULT:

The Above Program Is Verified And Executed Successfully.

Program 3

Model Evaluation Techniques using - Error or Blas, MAD, MAPE, MSE

AIM :

To evaluate the performance of a predictive model using error metrics such as Bias, MAD (Mean Absolute Deviation), MAPE (Mean Absolute Percentage Error), and MSE (Mean Squared Error).

Algorithm :

Step 1: Import necessary libraries (e.g., NumPy, Pandas).

Step 2: Load the dataset into a DataFrame.

Step 3: Extract actual and predicted values.

Step 4: Calculate Bias, MAD, MAPE, and MSE using the formulas above.

Step 5: Store the results in a structured format (e.g., dictionary or DataFrame).

Step 6: Print or visualize the results for better understanding.

Code:

```
# Sample data
actual <- c(100, 150, 200, 250, 300)
predicted <- c(110, 140, 210, 240, 290)

# Error and Bias
error <- actual - predicted
bias <- mean(error)

# MAD (Mean Absolute Deviation)
MAD <- mean(abs(error))

# MAPE (Mean Absolute Percentage Error)
MAPE <- mean(abs((actual - predicted) / actual)) * 100

# MSE (Mean Squared Error)
MSE <- mean((actual - predicted)^2)

# Print the results
cat("Bias:", bias, "\n")
cat("MAD:", MAD, "\n")
cat("MAPE:", MAPE, "%\n")
cat("MSE:", MSE, "\n")
```

OUTPUT:

```
> # Print the results
> cat("Bias:", bias, "\n")
Bias: 2
> cat("MAD:", MAD, "\n")
MAD: 10
> cat("MAPE:", MAPE, "%\n")
MAPE: 5.8 %
> cat("MSE:", MSE, "\n")
MSE: 100
```

RESULT:

The Above Program Is Verified and Executed Successfully.

Program 4

Testing the stationarity of the given data

Testing the autocorrelation

AIM :

Testing the stationarity of the given data Testing the autocorrelation

Algorithm :

Step 1: Start.

Step 2: Input the time series data, data.

Step 3: Install and load the packages: tseries and forecast.

Step 4: Perform ADF Test:

Step 5: Apply adf.test(data).

Step 6: If p-value < 0.05, the series is stationary; otherwise, it is non-stationary.

Step 7: Plot Autocorrelation (ACF):

Step 8: Apply acf(data) to plot the autocorrelations at various lags.

Step 8: Plot Partial Autocorrelation (PACF):

Step 9: Apply pacf(data) to plot the partial autocorrelations.

Step 10: Output:

Step 11: ADF test result (stationarity).

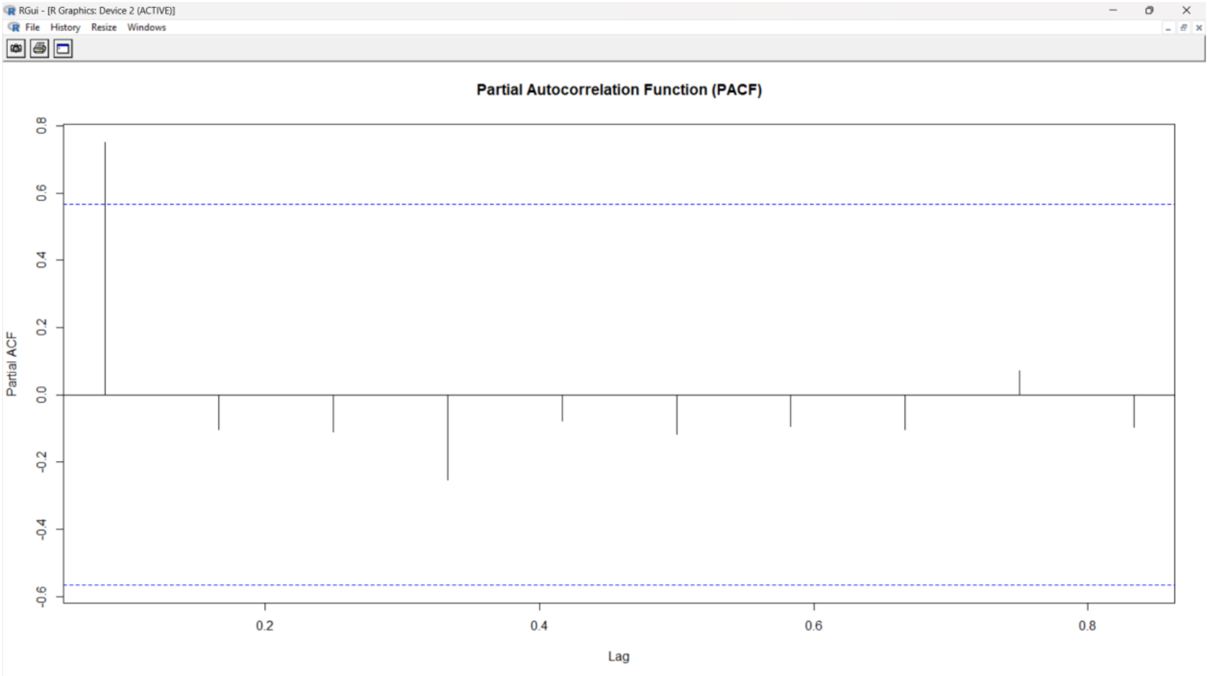
Step 12: ACF and PACF plots (autocorrelation behavior).

Step 12: End.

Code:

```
# Step 1: Install necessary packages if not already installed

if (!require(tseries)) install.packages("tseries", dependencies=TRUE)

if (!require(forecast)) install.packages("forecast", dependencies=TRUE)


# Step 2: Load the libraries

library(tseries)

library(forecast)


# Step 3: Load your time series data

# Example time series data, replace with actual data

# Ensure you input a valid numeric time series

data <- ts(c(100, 102, 104, 103, 106, 108, 110, 112, 115, 117, 119, 121), start=c(2020, 1), frequency=12)


# Check if 'data' is properly formatted as a time series

if (!is.ts(data)) {

  stop("The input data is not a time series object. Please convert your data into a time series format using the ts() function.")

}


# Step 4: Test for Stationarity using the ADF Test

adf_test <- adf.test(data)

print(adf_test)

# Step 5: Plot Autocorrelation Function (ACF)

acf(data, main="Autocorrelation Function (ACF)")

# Step 6: Plot Partial Autocorrelation Function (PACF)

pacf(data, main="Partial Autocorrelation Function (PACF)")
```

OUTPUT:



Program 5

1. ACF and PACF

2. Correlogram

AIM :

The Partial Auto-Correlation Function (PACF) is used to determine the direct correlation between a time series and its lagged values after removing the effects of shorter lags. This helps in identifying the appropriate number of lags to include in autoregressive models, particularly ARIMA.

Algorithm :

Step1 : Input: A time series dataset.

Step 2 : Base R package that provides the pacf() function.

Step 3 :A time series dataset (e.g., X). Maximum lag

step 4: p (the number of lags to consider).

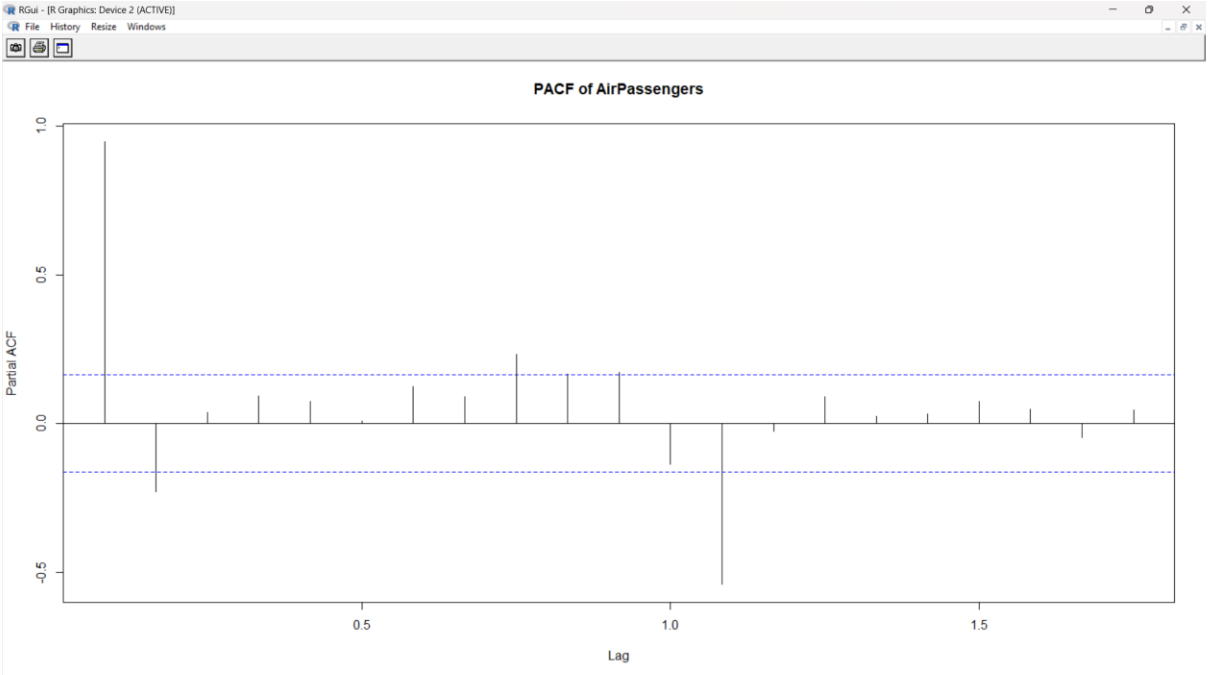
Step 5: Store the results in a structured format (e.g., dictionary).

Step 6: Print or visualize the results for better understanding

Code:

```r
# Load necessary libraries
library(stats)
library(ggplot2)

# Load the AirPassengers dataset
data("AirPassengers")

# Plot the time series
plot(AirPassengers, main="AirPassengers Time Series",
ylab="Number of Passengers", xlab="Year")

# Calculate and plot the ACF
acf(AirPassengers, main="ACF of AirPassengers")

# Calculate and plot the PACF
pacf(AirPassengers, main="PACF of AirPassengers")
summary(AirPassengers)
```

OUTPUT:



Program 6

1. Auto Regressive Model

2. Moving Average Model

AIM :

Auto Regressive Model And Moving Average Model

Algorithm :

Step 1: Fitting an Auto Regressive (AR) model

Step 2: AR(p) model where p is the order of the autoregressive model.

Step 3: Here, we'll use an AR(1) modelar_model <- arima(time_series_data, order = c(1, 0, 0))

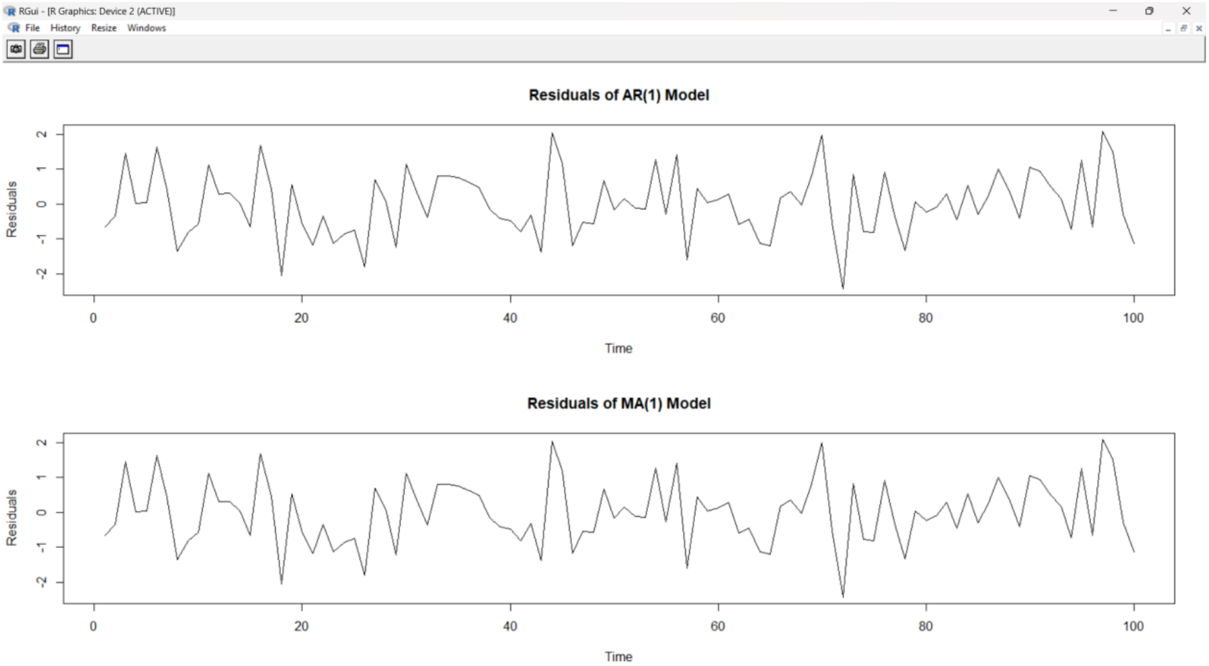
Step 4: MA(q) model where q is the order of the moving average model.

Step 5: Here, we'll use an MA () modelma_model <- arima(time_series_data,order = c(0, 0, 1

Code:

```
library(stats)

set.seed(123)  # For reproducibility
n <- 100
time_series_data <- ts(rnorm(n), frequency = 1)

plot(time_series_data, main = "Simulated Time Series Data", ylab =
"Values", xlab = "Time")

ar_model <- arima(time_series_data, order = c(1, 0, 0))  # AR(1), no
differencing, no MA component
summary(ar_model)

ma_model <- arima(time_series_data, order = c(0, 0, 1))  # MA(1), no
differencing, no AR component
summary(ma_model)

par(mfrow = c(2, 1))  # Set up 2 plots on one page
plot(residuals(ar_model), main = "Residuals of AR(1) Model", ylab =
"Residuals")
plot(residuals(ma_model), main = "Residuals of MA(1) Model", ylab =
"Residuals")
par(mfrow = c(1, 1))  # Reset plotting layout
```

OUTPUT:



Program 7

1. Fit ARMA for the given data and forecast the same for the next time period
2. Fit ARIMA for the given data and forecast the same for the next time period

AIM :

R program that fits an ARMA model to a given dataset, forecasts the next time period, and provides a summary of the model

Algorithm :

Step 1: Open R or RStudio.

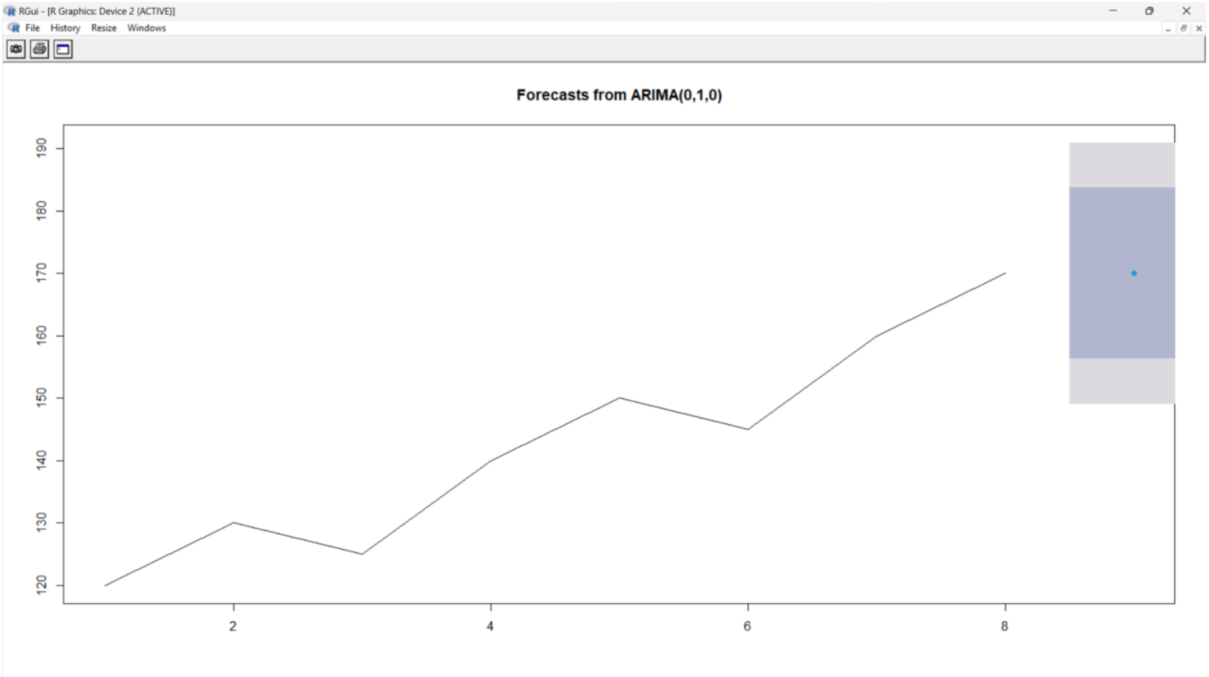
Step 2: Adjust the data variable to include your actual data points.

Step 3: Run the code to fit the ARMA model, view the summary, and see the forecast.

Code:

```r
# Load the required package
install.packages("forecast")  # Uncomment if not already installed
library(forecast)

# Example data: replace this with your actual data
data <- c(120, 130, 125, 140, 150, 145, 160, 170)

# Convert the data to a time series object
ts_data <- ts(data)

# Fit the ARMA model
fit <- auto.arima(ts_data)

# Print the model summary
summary(fit)

# Forecast the next time period
forecasted_values <- forecast(fit, h = 1)  # Forecast for 1 period

# Print the forecasted values
print(forecasted_values)

# Plot the forecast
plot(forecasted_values)
```

OUTPUT:



Program 8

1. Testing of spurious(non Sense Regression)
2. Unit root test
3. Heteroscedasticity
4. Granger causality

The aim of this R program is to simulate two time series, test for stationarity using the Augmented Dickey-Fuller (ADF) test, perform a simple linear regression to investigate relationships between the two series, check for heteroscedasticity in the regression model using the Breusch-Pagan test, and finally conduct a Granger Causality test to determine if one time series can predict the other.

ALGORITHM:

Load Required Libraries:

Simulate Time Series Data:

Plot Time Series:

Unit Root Test (Stationarity Check):

Null Hypothesis

Alternative Hypothesis

Perform Linear Regression:

Granger Causality Test:

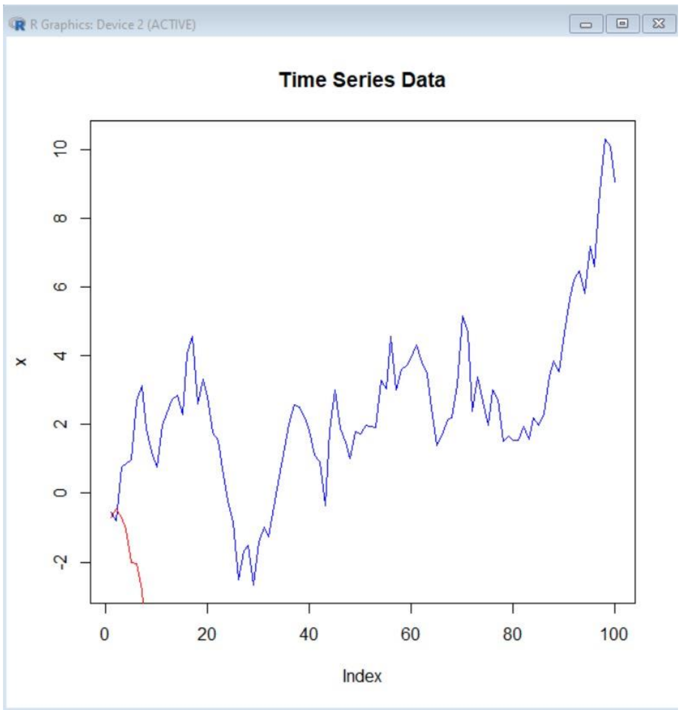
Display Results:

Print the results of the ADF tests, regression analysis, Breusch-Pagan test, and Granger causality tes

Code:

```
install.packages(c("tseries", "lmtest", "urca", "vars"))

library(tseries)

library(lmtest)

library(urca)

library(vars)

set.seed(123)

n <- 100

x <- cumsum(rnorm(n))

y <- cumsum(rnorm(n))

plot(x, type = 'l', col = 'blue', main = "Time Series Data")

lines(y, col = 'red')

adf_test_x <- adf.test(x)

adf_test_y <- adf.test(y)

cat("ADF Test for x:\n")

print(adf_test_x)

cat("\nADF Test for y:\n")

print(adf_test_y)

model <- lm(y ~ x)

summary(model)

bp_test <- bptest(model)

cat("\nBreusch-Pagan Test:\n")

print(bp_test)

max_lag <- 5  # Define maximum lag

granger_test <- grangertest(y ~ x, order = max_lag)

cat("\nGranger Causality Test:\n")

print(granger_test)
```

OUTPUT:



Program 9

VAR

AIM:

The aim of this R program is to simulate two time series, fit a Vector Autoregressive (VAR) model with one lag to analyze the relationships between the two series, and forecast the next 5 periods using the fitted VAR model.

ALGORITHM:

Install and Load Necessary Packages:

Simulate Two Time Series:

Create Data Frame:

Fit the VAR Model:

View the Summary of the VAR Model:

significance levels.

Forecast Future Values:

Display Forecasted Values:

CODE

```r
# Install and load necessary packages
install.packages("vars")
library(vars)

# Simulating two time series
set.seed(123)  # For reproducibility
n <- 100
time_series1 <- cumsum(rnorm(n))
time_series2 <- cumsum(rnorm(n))

# Combine into a data frame
data <- data.frame(time_series1, time_series2)

# Fit a VAR model
var_model <- VAR(data, p = 1)  # p = 1 for one lag

# View the summary
summary(var_model)

# Forecast the next 5 periods
forecasted_values <- predict(var_model, n.ahead = 5)

# View the forecasted values
forecasted_values
```

OUTPUT

```
$time_series1
         fcst    lower    upper       CI
[1,] 8.817411 7.020063 10.61476 1.797348
[2,] 8.600765 6.111650 11.08988 2.489115
[3,] 8.390821 5.405087 11.37655 2.985734
[4,] 8.187692 4.810561 11.56482 3.377131
[5,] 7.991445 4.292300 11.69059 3.699146

$time_series2
          fcst     lower     upper       CI
[1,] -10.90880 -12.75299 -9.064613 1.844186
[2,] -11.03768 -13.52774 -8.547611 2.490065
[3,] -11.14404 -14.06185 -8.226235 2.917809
[4,] -11.23037 -14.46053 -8.000219 3.230155
[5,] -11.29891 -14.76830 -7.829525 3.469387
```

Program 10

ARCH model fit

GARCH MODEL FIT

The aim of this R program is to simulate financial returns data, fit both an ARCH (Autoregressive Conditional Heteroskedasticity) model and a GARCH (Generalized Autoregressive Conditional Heteroskedasticity) model to analyze the volatility of the returns, and plot the fitted volatility of the GARCH model.

ALGORITHM:

Install and Load Necessary Packages:

Simulate Returns Data:

Fit an ARCH Model:

Fit a GARCH Model:

View Model Summaries:

Plot the Fitted Volatility:

CODE

```
# Install and load necessary packages
install.packages("fGarch")
library(fGarch)
# Simulate some returns data
set.seed(123)  # For reproducibility
n <- 1000
returns <- rnorm(n, mean = 0, sd = 1)
# Fit an ARCH model
arch_model <- garchFit(~ garch(1, 0), data = returns,
trace = FALSE)
summary(arch_model)
# Fit a GARCH model
garch_model <- garchFit(~ garch(1, 1), data = returns,
trace = FALSE)
summary(garch_model)
# Plotting the fitted volatility
plot(garch_model)
```