Introduction

- This document provides an introduction to the hand-on of linear and non-linear causality analysis. Guidance of necessary software environment setup and installation will be provided.
- The sample code cover time series analysis techniques including unit root test, cointegration test, VAR estimation, linear Granger causality test and non-linear Granger causality test.
- We illustrate the techniques by going through a simplified bivariate analysis with real economic data.

Data

In this illustration, our data include information from India, from 1950 to 2014 obtained from the International Financial Statistics, the database of International Monetary Fund.

India is selected as one the countries in the study because it meets our criteria that the country should have a population exceeding one million in 2017 and the database has enough data in all the variables.

It is difficult to directly measure both financial development and economic growth.

Economic growth is commonly measured using real GDP per capita, see for example Gelb (1989), Roubini and Sala-i-Martin (1992), King and Levine (1993), and Demetriades and Hussein (1996).

Similar to these studies, a logarithm of real GDP per capita is used to proxy economic growth and has been denoted as Y.

For the variable to measure financial development, it is a common practice (Gelb, 1989 and King and Levine, 1993) to use a ratio of some broad measures of the money stock to the level of nominal GDP.

However, Demetriades and Hussein (1996) argue that this type of ratio may reflect more extensive use of currency rather than an increase in the volume of bank deposits.

To circumvent the limitation, they recommend excluding currency in circulation from the broad money stock.

In this illustration we follow their recommendation and use the logarithm of the ratio of bank deposit liabilities to nominal GDP as the first proxy for financial development and represented as M.

We exhibit the summary statistics of where M and Y are the logarithms of the ratio of bank deposit liabilities to nominal GDP (M), and real GDP per capita (Y), respectively, for India.

Methodology

In the literature on the subject of the relationship between financial development and economic growth, academics, such as Horng, et al. (2012) are interested in testing the following two hypotheses:

H_0^1 : financial development does not cause economic growth, and
H_0^2: economic growth does not cause financial development.

Academics and practitioners, such as Horng, et al. (2012) and Fan, et al. (2018) employ linear causality to study whether there is any unidirectional or bidirectional causality between financial development and economic growth. Thus, they set $H_0^{1\prime}$: financial development does not cause economic growth if there is no linear causality from financial development to economic growth.

However, in this illustration, we set

H_0^1: financial development does not cause economic growth if there is no linear and no nonlinear causality from financial development to economic growth.

Similarly, definitions are set for $H_0^{2\prime}$ and H_0^2.

Ascertaining whether financial development and economic growth are cointegrated is an important piece of information. If financial development and economic growth are cointegrated, we conjecture that both demand-following and supply-leading theories hold so that financial development and economic growth move positively together.

Thus, in this illustration, we examine the following hypothesis:

H_0^3: financial development and economic growth are not positively cointegrated.

In the following subsections, we will discuss cointegration and linear and nonlinear causality tests to analyze the relationship between financial development and economic growth in developing countries. We first discuss the cointegration approach in next subsection.

3.2.1 Cointegration

As mentioned in Section 3.1 *M*, *D*, and *Y* have been designated to be the logarithms of the ratio of bank deposit liabilities to nominal GDP (*M*), the ratio of claims on private sector to nominal GDP, and real GDP per capita, respectively. If all the variables (*M*, *D*, and *Y*) are integrated in degree one, academics and practitioners will be interested in examining whether there is any cointegration relationship among the variables. To analyse the issue, we employ the Johansen cointegration test proposed by Johansen (1988), Johansen and Juselius (1990)

and Johansen (1991) as some studies, for example, Gonzalo (1994), confirm that the Johansen cointegration test performs better than the other cointegration tests, namely the ADF test (Engle and Granger, 1987). In addition, when GARCH errors exist in the model, Lee and Tse (1996) conclude that the bias is not too serious when using Johansen's cointegration test if we compare its performance with other cointegration tests.

Johansen and Juselius (1990) and Johansen (1991) develop a multivariate maximum likelihood (ML) procedure for the estimation of the cointegrating vectors. According to Johansen's procedure, the p-dimensional unrestricted Vector Autoregression (VAR) model should be first specified with k lags:

$$\boldsymbol{Z_t} = \sum_{i=1}^{k} \boldsymbol{A_i Z_{t-i}} + \Psi \boldsymbol{D_t} + U_t \qquad (1)$$

where $\boldsymbol{Z_t} = [M_t, D_t, Y_t\,]'$ is a 3×1 vector of stochastic variables and M_t, D_t, and Y_t are to be the logarithms of the ratio of bank deposit liabilities to nominal GDP, the ratio of claims on private sector to nominal GDP, and real GDP per capita in period t, respectively. $\boldsymbol{D_t}$ is a vector of dummies and $\boldsymbol{A_i}$ is a vector of parameters. This VAR could be rewritten as:

$$\Delta \boldsymbol{Z_t} = \sum_{i=1}^{k-1} \Phi_i \Delta \boldsymbol{Z_{t-i}} + \boldsymbol{\Pi Z_{t-i}} + \Psi \boldsymbol{D_t} + U_t \,. \qquad (2)$$

The hypothesis of cointegration is formulated as a reduced rank of the $\boldsymbol{\Pi}$ matrix where $\boldsymbol{\Pi = \alpha \beta'}$ such that $\boldsymbol{\alpha}$ is the vector or matrix of the adjustment parameter and $\boldsymbol{\beta}$ is the vector or matrix of the cointegrating vectors. According to Engle and Granger (1987), if the rank of $\boldsymbol{\Pi}$ (r) is not equal to zero, then r cointegrating vectors exist.

The number of cointegrating vectors is less than or equal to the number of variables, which is 3 in our case. The likelihood ratio (LR) reduced the rank test for the null hypothesis of at most r cointegrating vectors is given by the following Trace statistic, and for the null hypothesis of r against the alternative of r+1 cointegrating vectors is known as the maximal eigenvalue statistic

$$3\lambda_{trace} = -\mathrm{T}\sum_{i=r}^{m} \ln(1-\lambda_{i+1}) \quad , \qquad \lambda_{max}$$

$$= -\mathrm{T}\ln(1-\lambda_{r+1}) \qquad (3)$$

where m is the maximum number of possible cointegrating vectors which is 3 in our case, in this illustration, $r = 0, 1, 2$ and $\lambda_1 > \lambda_2 > \lambda_3$ denote eigenvalues of their corresponding eigenvectors $\mathrm{v} = (v_1, v_2, v_3)$.If the null hypothesis of r cointegrating vectors is accepted, then the rank of the **Π** matrix equal to r and there is exactly r cointegrating vector.

3.2.2 Granger Causality

Since our analysis presented in next section (see Table 1) confirms that all the variables $M_t, D_t,$ and Y_t are $\mathrm{I}(1)$, academics and practitioners are interested in testing whether there is any causality relationship among the differences of the variables $M_t, D_t,$ and Y_t. We let $m_t = \Delta M_t$, $d_t = \Delta D_t$, and $y_t = \Delta Y_t$. This means that academics and practitioners are interested in testing whether there is any causality relationship among m_t, d_t, and y_t. Thus, in this illustration we will test whether there is any linear Granger causality and thereafter examine whether there is any nonlinear Granger causality among the variables m_t, d_t, and y_t.

3.2.2.1 Linear Granger Causality

To test the linear causality relationship between two vectors of stationary time series, we set $x_t = (x_{1,t}, \dots, x_{n_1,t})'$ and $y_t = (y_{1,t}, \dots, y_{n_2,t})'$ say $x_t = (m_t, d_t)'$ and $y_t = (y_t)'$, where there are 3 series in total. Under this setting, one could construct the following vector autoregressive regression (VAR) model:

$$\begin{pmatrix} x_t \\ y_t \end{pmatrix} = \begin{pmatrix} A_{x[2\times1]} \\ A_{y[1\times1]} \end{pmatrix} + \begin{pmatrix} A_{xx}(L)_{[2\times2]} & A_{xy}(L)_{[2\times1]} \\ A_{yx}(L)_{[1\times2]} & A_{yy}(L)_{[1\times1]} \end{pmatrix} \begin{pmatrix} x_{t-1} \\ y_{t-1} \end{pmatrix} + \begin{pmatrix} e_{x,t} \\ e_{y,t} \end{pmatrix} \quad (4)$$

where $A_{x[2\times1]}$ and $A_{y[1\times1]}$ are two vectors of intercept terms, $A_{xx}(L)_{[2\times2]}$, $A_{xy}(L)_{[2\times1]}$, $A_{yx}(L)_{[2\times1]}$, and

$A_{yy}(L)_{[1\times1]}$ are matrices of lag polynomials, $e_{x,t}$ and $e_{y,t}$ are the corresponding error terms.

Testing the following null hypotheses: $H_0^1: A_{xy}(L) = 0$ and $H_0^2: A_{yx}(L) = 0$ is equivalent to testing the linear causality relationship between x_t and y_t.There are four different situations for the causality relationships between x_t and y_t in (1): (a) rejecting H_0^1 but not rejecting H_0^2 implies a unidirectional causality from y_t to x_t, (b) rejecting H_0^2 but not rejecting H_0^1 implies a unidirectional causality from x_t to y_t, (c) rejecting both H_0^1 and H_0^2 implies the existence of feedback relations, and (d) not rejecting both H_0^1 and H_0^2 implies that x_t and y_t are not rejected to be independent. Readers may refer to Bai, et al. (2010) for the details of testing H_0^1 and/or H_0^2.

If the time series are cointegrated, one should impose the error-correction mechanism (ECM) on the VAR to construct a vector error correction model (VECM) in order to test Granger causality between the variables of interest. In particular, when testing the causality relationship between two vectors of non-stationary time series, we let $\Delta x_t = (\Delta M_t, \Delta D_t)'$ and $\Delta y_t = (\Delta Y_t)'$ be the corresponding stationary differencing series such that there are 3 series in total. If x_t and y_t are cointegrated, then instead of using the VAR in (1), one should adopt the following VECM model:

$$\begin{pmatrix} \Delta x_t \\ \Delta y_t \end{pmatrix} = \begin{pmatrix} A_{x[2\times1]} \\ A_{y[1\times1]} \end{pmatrix} + \begin{pmatrix} A_{xx}(L)_{[2\times2]} & A_{xy}(L)_{[2\times1]} \\ A_{yx}(L)_{[1\times2]} & A_{yy}(L)_{[1\times1]} \end{pmatrix} \begin{pmatrix} \Delta x_{t-1} \\ \Delta y_{t-1} \end{pmatrix} + \begin{pmatrix} \alpha_{x[2\times1]} \\ \alpha_{y[1\times1]} \end{pmatrix} \cdot ecm_{t-1} + \begin{pmatrix} e_{x,t} \\ e_{y,t} \end{pmatrix} \quad (5)$$

where ecm_{t-1} is lag one of the error correction term, and $\alpha_{x[2\times1]}$ and $\alpha_{y[1\times1]}$ are the coefficient vectors for the error correction term ecm_{t-1}. There are now two sources of causation of $y_t(x_t)$ by $x_t(y_t)$, either through the lagged dynamic terms $\Delta x_{t-1}(\Delta y_{t-1})$, or through the error correction term ecm_{t-1}. Thereafter, one could test the null hypothesis $H_0 : A_{xy}(L) = 0\big(H_0 : A_{yx}(L) = 0\big)$ and/or $H_0 : \alpha_x = 0\big(H_0 : \alpha_y = 0\big)$ to identify Granger causality relation using the LR test.

3.2.2.2 Nonlinear Granger Causality

Bai, et al. (2010, 2011, 2018) and Chow, et al. (2018) extend the nonlinear causality test developed by Hiemstra and Jones (1994) and others to the multivariate setting. To identify any nonlinear Granger causality relationship from any two series, say $\{x_t\}$ and $\{y_t\}$ in the bivariate setting, one has to first apply the linear model to $\{x_t\}$ and $\{y_t\}$ to identify their linear causal relationships and obtain the corresponding residuals, $\{\hat{\varepsilon}_{1t}\}$ and $\{\hat{\varepsilon}_{2t}\}$. Thereafter, one has to apply a nonlinear Granger causality test to the residual series, $\{\hat{\varepsilon}_{1t}\}$ and $\{\hat{\varepsilon}_{2t}\}$, of the two variables being examined to identify the remaining nonlinear causal relationships between their residuals. This is also true if one would like to identify the existence of any nonlinear Granger causality relation between two vectors of time series, say $x_t = (x_{1,t}, \dots, x_{n1,t})'$ and $y_t = (y_{1,t}, \dots, y_{n2,t})'$ in the multivariate setting. One has to apply the VAR

model or the VECM model to the series to identify their linear causal relationships and obtain their corresponding residuals. Thereafter, one has to apply a nonlinear Granger causality test to the residual series. For simplicity, in this section we denote $X_t = (X_{1,t}, \dots, X_{n1,t})'$ and $Y_t = (Y_{1,t}, \dots, Y_{n2,t})'$ to be the corresponding residuals of any two vectors of variables being examined. We first define the lead vector and lag vector of a time series, say $X_{i,t}$, as follows:

for

$X_{i,t}$, i $= 1,2$, the m_{x_i}-length lead vector and the L_{x_i}-length lag vector of $X_{i,t}$ are:

$$X_{i,t}^{m_{x_i}} \equiv \left(X_{i,t}, X_{i,t+1}, \dots, X_{i,t+m_{x_i}-1}\right), m_{x_i} = 1,2, \dots, t = 1, 2, \dots,$$

$$X_{i,t-L_{x_i}}^{L_{x_i}} \equiv \left(X_{i,t-L_{x_i}}, X_{i,t-L_{x_i}+1}, \dots, X_{i,t-1}\right), L_{x_i} = 1, 2, \dots, \text{t} = L_{x_i} + 1, L_{x_i} + 2, \dots,$$

respectively. We denote $M_x = \left(m_{x1}, \dots, m_{x_{n_1}}\right)$, $L_x = \left(L_{x1}, \dots, L_{x_{n_1}}\right), m_x = \max\left(m_{x1}, \dots, m_{n_1}\right)$, and $l_x = \max\left(L_{x1}, \dots, L_{x_{n_1}}\right)$. The m_{y_i}-length lead vector, $Y_{i,t}^{m_{y_i}}$, the L_{y_i}-length lag vector, $Y_{i,t-L_{y_i}}^{L_{y_i}}$, of $Y_{i,t}$, and M_y, L_y, m_y, and l_y can be defined similarly.

Given m_x, m_y, L_x, L_y, and $e > 0$, we define the following four events:

$$\left\{\left\|X_t^{M_x} - X_s^{M_x}\right\| < e\right\}$$

$$\equiv \left\{\left\|X_{i,t}^{M_{x_i}} - X_{i,s}^{m_{x_i}}\right\| < e, \text{for any } i = 1, \ldots, n_1\right\};$$

$$\left\{\left\|X_{t-L_x}^{L_x} - X_{s-L_x}^{L_x}\right\| < e\right\}$$

$$\equiv \left\{\left\|X_{i,t-L_{x_i}}^{L_{x_i}} - X_{i,s-L_{x_i}}^{L_{x_i}}\right\| < e, \text{for any } i\right.$$

$$\left.= 1, \ldots, n_1\right\};$$

$$\left\{\left\|Y_t^{M_y} - Y_s^{M_y}\right\| < e\right\}$$

$$\equiv \left\{\left\|Y_{i,t}^{m_{y_i}} - Y_{i,s}^{m_{y_i}}\right\| < e, \text{for any } i\right.$$

$$\left.= 1, \ldots, n_2\right\}; and$$

$$\left\{\left\|Y_{t-L_y}^{L_y} - Y_{s-L_y}^{L_y}\right\| < e\right\}$$

$$\equiv \left\{\left\|Y_{i,t-L_{y_i}}^{L_{y_i}} - Y_{i,s-L_{y_i}}^{L_{y_i}}\right\| < e, \text{for any } i\right.$$

$$\left.= 1, \ldots, n_2\right\};$$

where $\|\cdot\|$ denotes the maximum norm which is defined as $\|X - Y\| = \max(|x_1 - y_1|, |x_2 - y_2|, \ldots, |x_n - y_n|)$ for any two vectors $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_n)$. The vector series $\{Y_t\}$ is said not to strictly Granger cause another vector series $\{X_t\}$ if

$$Pr\left(\left\|X_t^{M_x} - X_s^{M_x}\right\| < e \middle| \left\|X_{t-L_x}^{L_x} - X_{s-L_x}^{L_x}\right\| < e, \left\|Y_{t-L_y}^{L_y} - Y_{s-L_y}^{L_y}\right\| < e\right)$$

$$= Pr\left(\left\|X_t^{M_x} - X_s^{M_x}\right\| < e \middle| \left\|X_{t-L_x}^{L_x} - X_{s-L_x}^{L_x}\right\| < e\right) \quad (6)$$

where $\Pr(\cdot \mid \cdot)$ denotes conditional probability. Applying (6), one has to use the following test statistic to test for the nonlinear Granger causality:

$$\sqrt{n}\left(\frac{C_1(M_x+L_x,L_y,e,n)}{C_2(L_x,L_y,e,n)} - \frac{C_3(M_x+L_x,e,n)}{C_4(L_x,e,n)}\right) \quad (7)$$

Readers may refer to Bai, et al. (2010, 2011, 2018) and Chow, et al. (2018) for the details of the equation (7). Under this setting, Bai, et al. (2010, 2011) prove that to test the null hypothesis, H_0, that $\{Y_{1,t}, \dots, Y_{n2,t}\}$ does not strictly Granger cause $\{X_{1,t}, \dots, X_{n1,t}\}$, under the assumptions that the time series $\{X_{1,t}, \dots, X_{n1,t}\}$ and $\{Y_{1,t}, \dots, Y_{n2,t}\}$ are strictly stationary, weakly dependent, and satisfy the mixing conditions stated in Denker and Keller (1983), if the null hypothesis, H_0, is true, the test statistic defined in (7) is distributed as $N\left(0, \sigma^2\left(M_x, L_x, L_y, e\right)\right)$. When the test statistic in (7) is too far away from zero, we reject the null hypothesis. Readers may refer to Bai, et al. (2010, 2011, 2018) and Chow, et al. (2018) for the details of the consistent estimator of the covariance matrix.

The nonlinear causality test has the ability to detect a nonlinear deterministic process which originally "looks"

random. The nonlinear causality test is a complementary test for the linear causality test as linear causality tests could not detect nonlinear causal relationship while the nonparametric approach adopted in this paper can capture the nonlinear nature of the relationship among variables.

From literature we note an interest in analyzing the cross-correlation relationship. For example, Podobnik and Stanley (2008) propose a detrended cross-correlation analysis (DXA) to investigate power-law cross-correlations between different simultaneously-recorded time series in the presence of non-stationarity. Podobnik, et al. (2009) introduce a joint stochastic process to model cross-correlations. In addition, using stock market returns from two stock exchanges in China, Ruan, et al. (2018) employ the MF-DCCA to investigate the non-linear cross-correlation between individual investor sentiment and Chinese stock market return. Zhang, et al. (2018) study the cross-correlations between Chinese stock markets and the other three stock markets. Xiong, et al. (2018) use a new policy uncertainty index to investigate the time-varying correlation between economic policy uncertainty and Chinese stock market returns. On the other hand, Wan and Wong (2001) develop a model to study the contagion effect. Cerqueti, et al. (2018) develop a model based on

Mixed Poisson Processes to deal with the theme of contagion in financial markets. Wang, et al. (2018) propose a non-Markovian social contagion model in multiplex networks with inter-layer degree correlations to delineate the behavior of spreading, and develop an edge-based compartmental theory to describe the model. The nonlinear causality used in this paper could also be used to measure nonlinear cross-correlation to handle the nonlinear contagion effect. One could easily use or modify Equation (6) to deal with the nonlinear cross-correlation and the nonlinear contagion effect.

Linear Granger Causality Analysis using R

Step 1: Install R

First, download the latest version or R. R for Windows can be downloaded at https://cran.r-project.org/bin/windows/base/

R is an incredibly powerful open source program for statistics and graphics. It can run on pretty much any computer and has a very active and friendly support community online. Graphics created by R are extremely extensible and are used in high level publications like the New York Times (as explained by this former NYT infographic designer).

The latest version of R is 3.6.0. Click the link "Download R 3.6.0 for Windows" to download and install.

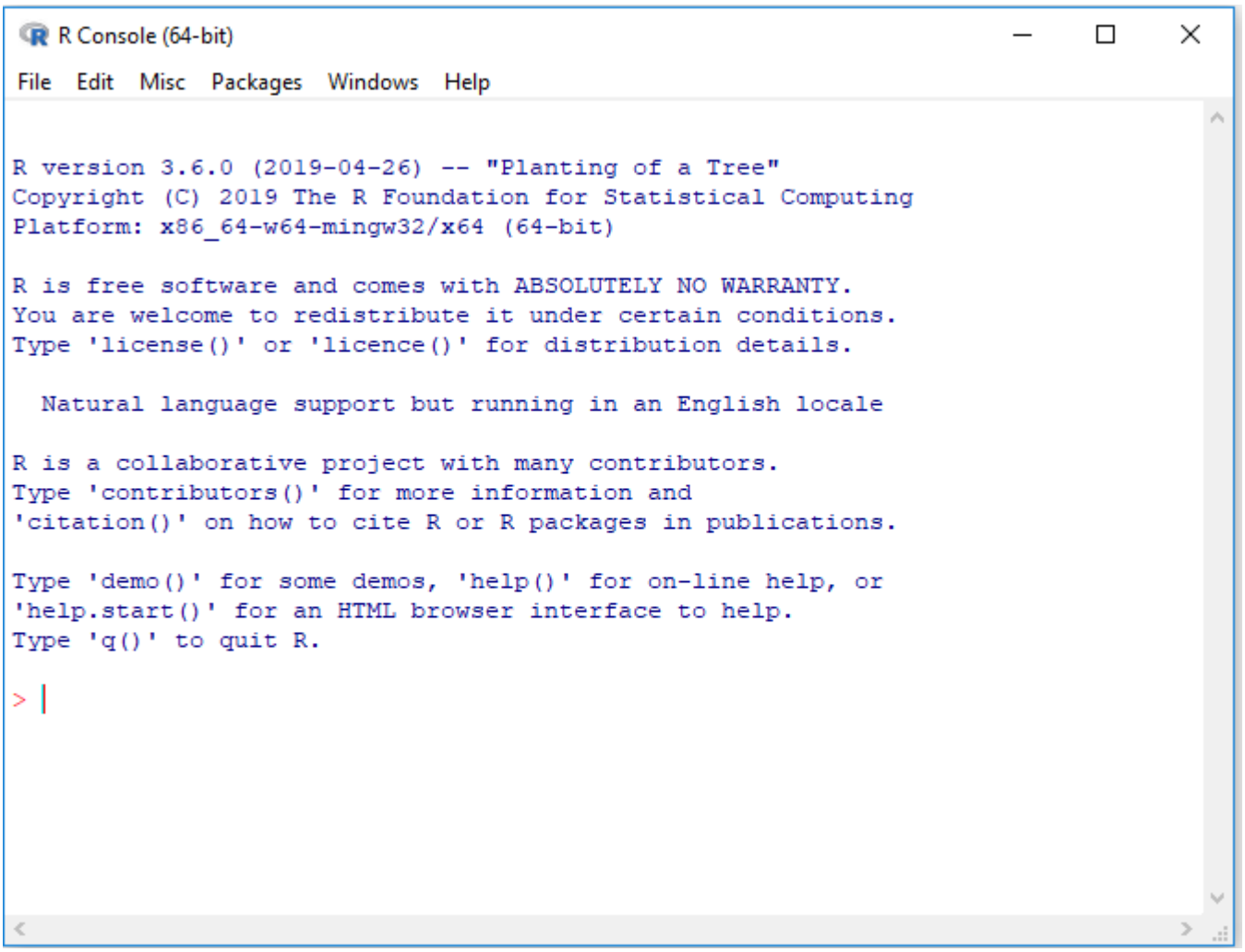


Figure 1. The R Console Screen

1. Download R from http://cran.us.r-project.org/ (click on "Download R for Windows" > "base" > "Download R 3.6.0 for Windows")
2. Install R. Leave all default settings in the installation options.

We note while R installation is required, since RStudio is used in this illustration, opening R Console is not necessary.

Step 2: Install RStudio

RStudio is the most popular open-source IDE (Integrated Development Environment) for R.

It's basically a nice front-end for R, giving you a console, a scripting window, a graphics window, and an R workspace, among other options.

It provides more features from the user interface as well as being more user-friendly than the raw development environment comes with the base installation of R.

https://www.andrewheiss.com/blog/2012/04/17/install-r-rstudio-r-commander-windows-osx/

RStudio can be downloaded at https://www.rstudio.com/products/rstudio/download/#download.

Click the Download button below the “RStudio Desktop”. Choose the right version of installer for your operating system.

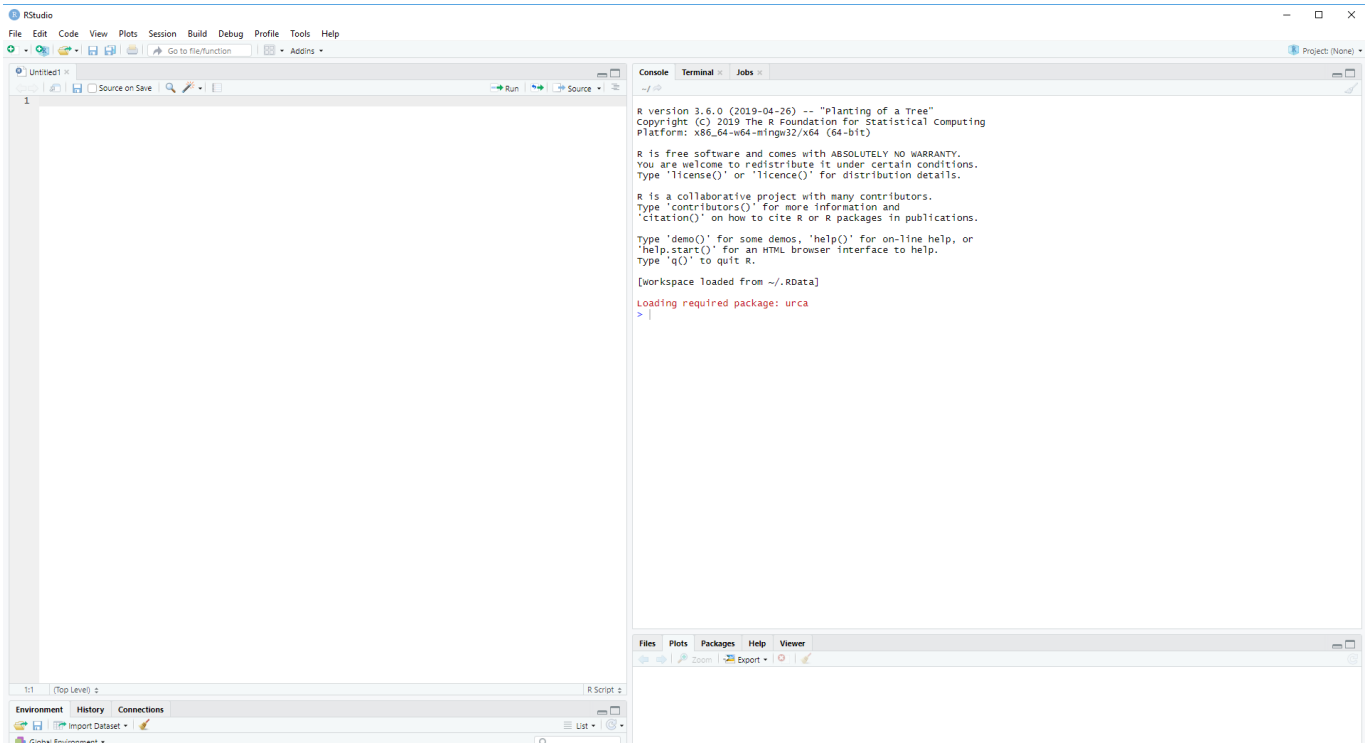


Figure 2. RStudio Screen

1. Download RStudio from http://rstudio.org/download/desktop and install it. Leave all default settings in the installation options.

RStudio
Desktop
Open Source
License

RStudio 1.2.1335 - Windows 7+ (64-bit)

Step 3: Open the R File in RStudio

To open the R script used in this illustration, click the File button in the top-left corner and then click Open File.

Then browse to the folder where the R program is stored, select the file and click Open.

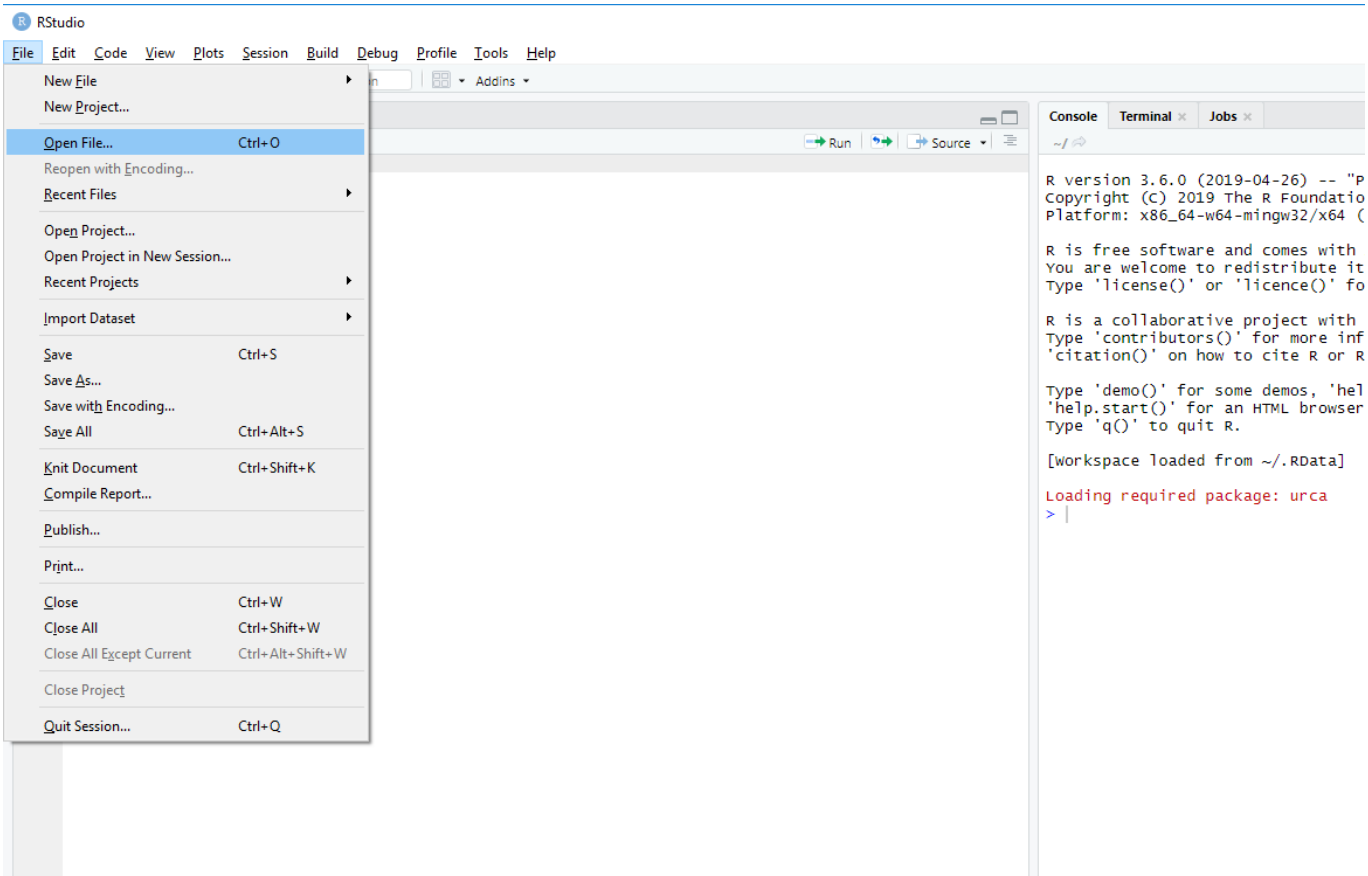


Figure 3. Open R File in RStudio

Step 4: Prepare the Data

The R Program consumes a data file which should have at least 2 variables in CSV format (with header). Save the data file to the same folder as the R file. In this illustration, the data for Indian as mentioned in the data section named ly (logarithms of real GDP per capita) and lm (logarithms of the ratio of bank deposit liabilities to nominal GDP) are used are included in the data file.

```
ld,lm,ly
20.62405919,20.77381812,-17.98541486
20.79094126,20.67734409,-17.96693208
20.69304131,20.60832464,-18.00374521
20.61918585,20.53929548,-17.95811052
20.68974314,20.59621709,-17.92596942
20.78932066,20.63942229,-17.90868502
20.96381105,20.64238345,-17.87270571
21.1044225,20.71603515,-17.8918594
21.05589561,20.63491128,-17.8391267
21.14300763,20.64016836,-17.83123793
21.2656769,20.6577318,-17.80049648
```

Figure 4. Sample Data File for the R Script

Step 5: Run the R Script

There are different ways in RStudio you can run R code. The first choice is to “source” the file, i.e. to evaluate all the R code in the opened file. Please click the Source button in the top-right side in the code editor. Alternatively, in Windows, you may press Ctrl + Shift + S to source a file.

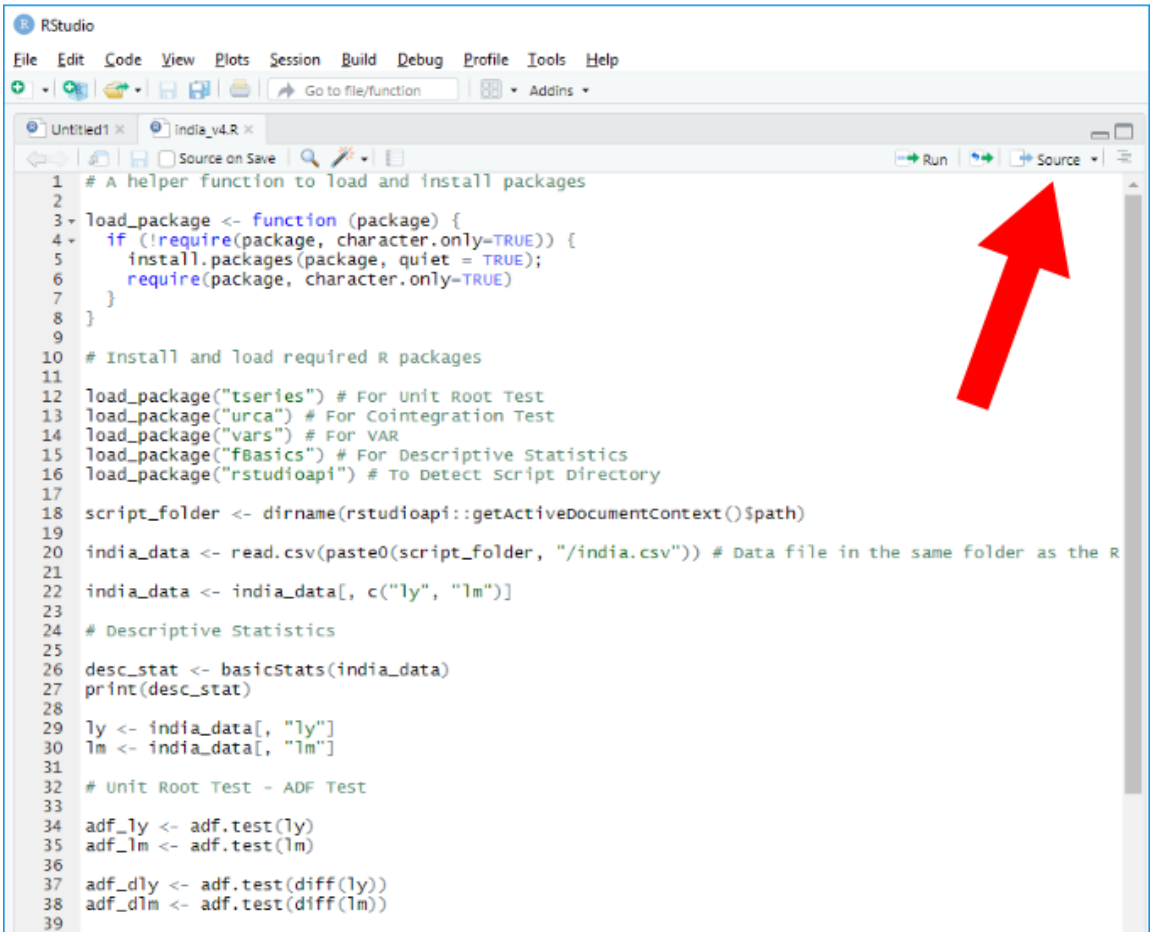


Figure 5. Source R File in RStudio

R, being a scripting language, can be run line by line or evaluate only selected area in your code. When only part of the code is to be evaluate, select the required part of the code, and then click R. Alternatively, in Windows, you may press Ctrl + Enter to run selected code.

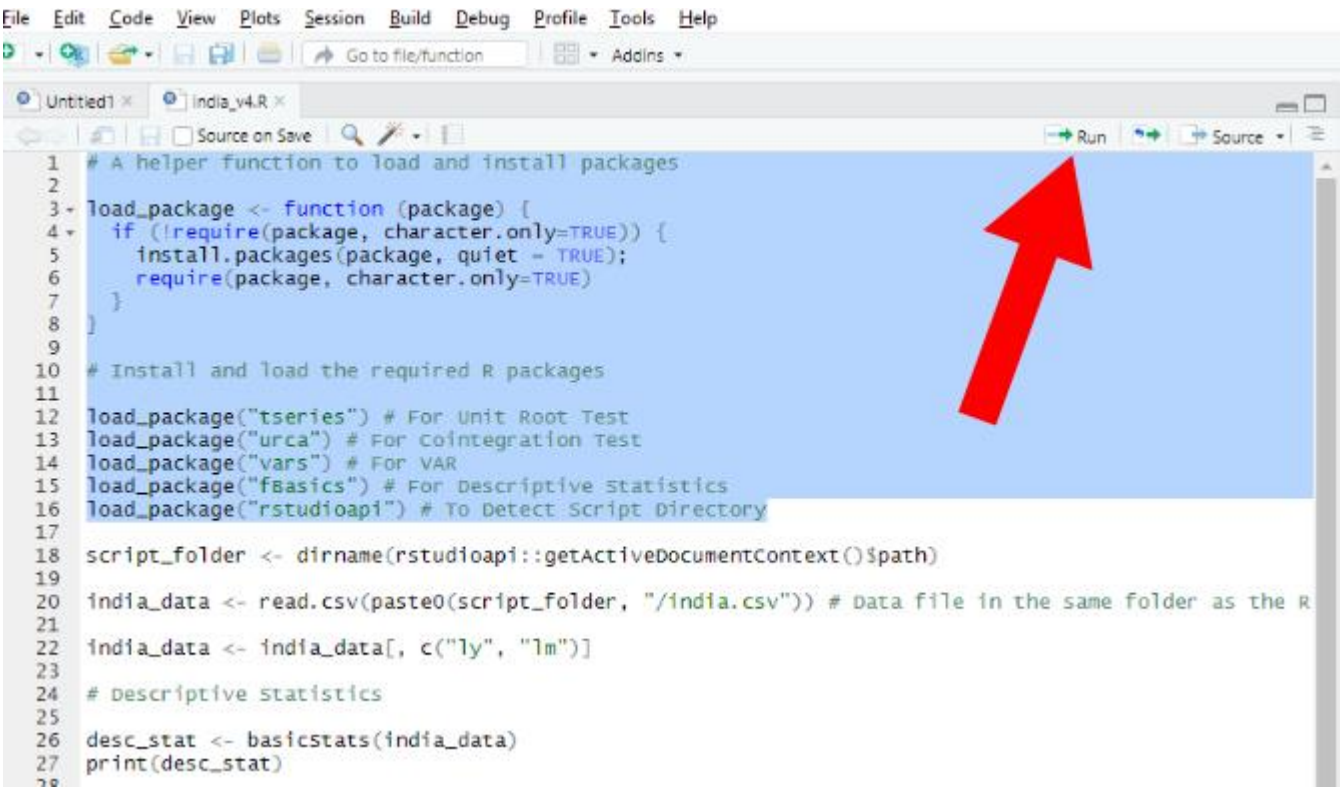


Figure 6. Evaluate Selected R Code

Execution result will be shown in the Console Pane. You are now ready to analyze the data interactively using R.

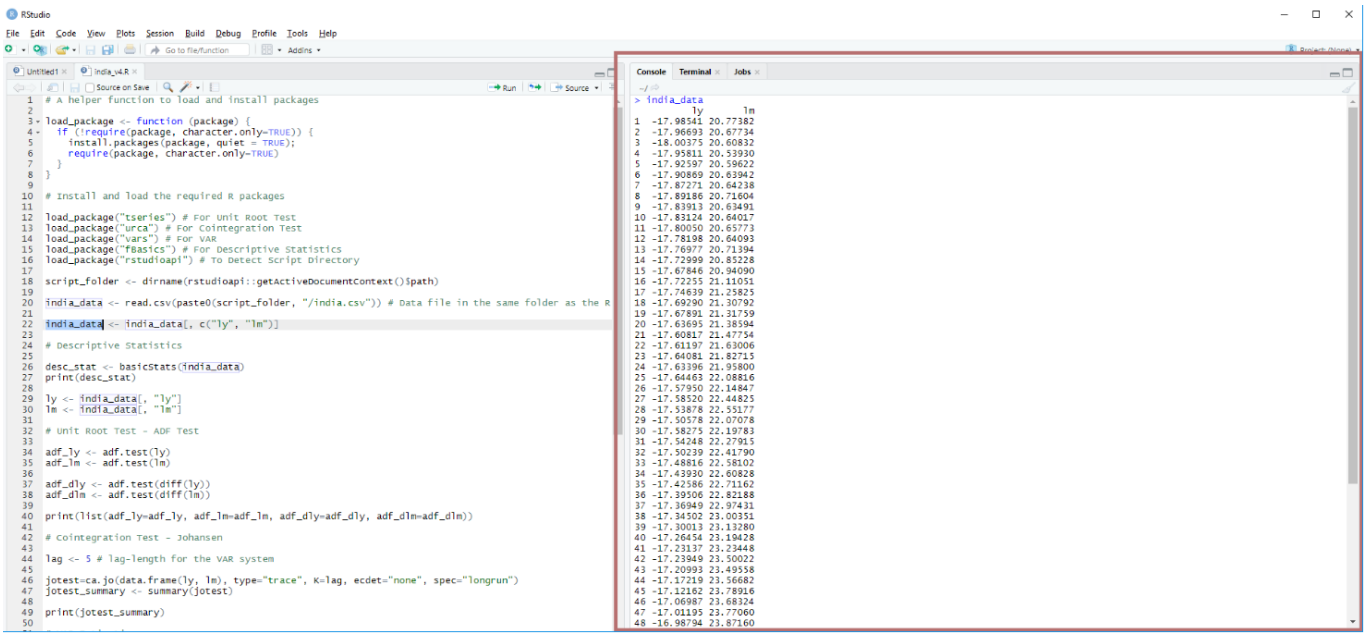


Figure 7. The Console Pane in RStudio

Step 6: Data Analysis

Install Required Libraries

Before the analysis, it is required to first install the required libraries in R.

R Code

```
# A helper function to load and install packages

load_package <- function (package) {
  if (!require(package, character.only=TRUE)) {
    install.packages(package, quiet = TRUE);
    require(package, character.only=TRUE)
  }
}
```

- A helper function is defined here to simplify the package installation process.
- The function accepts an argument of the package name and tries to load the package.
- If the package isn't installed yet, it installs and loads the package into the environment.

R Code

```
# Install and load the required R packages

load_package("tseries") # For Unit Root Test
load_package("urca") # For Cointegration Test
load_package("vars") # For VAR
load_package("fBasics") # For Descriptive Statistics
load_package("rstudioapi") # To Detect Script Directory
```

After that 5 packages are installed using the function.
These packages are

- tseries (for the unit root test),
- urca (for cointegration test),
- vars (for VAR estimation),
- fBasics (for descriptive statistics),
- rstudioapi (for a function detecting the script path).

Load Data

R Code

```
script_folder <-
dirname(rstudioapi::getActiveDocumentContext()$path)

india_data <- read.csv(paste0(script_folder, "/india.csv"))

india_data <- india_data[, c("ly", "lm")]
```

This part of the code loads the data in CSV format named india.csv into the environment.

Descriptive Statistics

R Code

```
# Descriptive Statistics

desc_stat <- basicStats(india_data)
print(desc_stat)

ly <- india_data[, "ly"]
lm <- india_data[, "lm"]
```

	ly	lm
nobs	64.000000	64.000000
NAs	0.000000	0.000000
Minimum	-18.003745	20.539295
Maximum	-16.148911	24.900501
1. Quartile	-17.700314	21.221314
3. Quartile	-16.976296	23.882365
Mean	-17.321299	22.591099
Median	-17.495277	22.566395
Sum	-1108.563121	1445.830368
SE Mean	0.064328	0.184869
LCL Mean	-17.449849	22.221669
UCL Mean	-17.192749	22.960530
Variance	0.264841	2.187292
Stdev	0.514627	1.478950
Skewness	0.729537	0.061804
Kurtosis	-0.575911	-1.421720

Figure 8. Descriptive Statistics Output

Unit Root Test

R Code

```
ly <- india_data[, "ly"]
lm <- india_data[, "lm"]

# Unit Root Test - ADF Test

adf_ly <- adf.test(ly)
adf_lm <- adf.test(lm)

adf_dly <- adf.test(diff(ly))
adf_dlm <- adf.test(diff(lm))

print(list(adf_ly=adf_ly, adf_lm=adf_lm, adf_dly=adf_dly,
adf_dlm=adf_dlm))
```

This part of the code first explicitly defines 2 variables from the dataset for better readability. ADF unit root test are then run to test for unit root in these variables.

```
$adf_ly

        Augmented Dickey-Fuller Test

data:  ly
Dickey-Fuller = 1.0744, Lag order = 3, p-value = 0.99
alternative hypothesis: stationary


$adf_lm

        Augmented Dickey-Fuller Test

data:  lm
Dickey-Fuller = -2.8639, Lag order = 3, p-value = 0.2249
alternative hypothesis: stationary


$adf_dly

        Augmented Dickey-Fuller Test

data:  diff(ly)
Dickey-Fuller = -4.595, Lag order = 3, p-value = 0.01
alternative hypothesis: stationary


$adf_dlm

        Augmented Dickey-Fuller Test

data:  diff(lm)
Dickey-Fuller = -4.0218, Lag order = 3, p-value = 0.01449
alternative hypothesis: stationary
```

Figure 9. ADF Unit Root Test Output

For each variable in the testing, the ADF unit root test using the default configuration cannot reject the null hypothesis that the series contains a unit root. The unit root tests on the first difference of these variables show that both the first-differenced time series are stationary.

The results show that both the variables ly and lm are I(1) at 5% level of significance.

For more information about the adf.test function, please see the document available at https://cran.r-project.org/web/packages/tseries/tseries.pdf.

Cointegration Test

R Code

```
# Cointegration Test - Johansen

lag <- 5 # lag-length for the VAR system

jotest=ca.jo(data.frame(ly, lm), type="trace", K=lag,
ecdet="none", spec="longrun")
jotest_summary <- summary(jotest)

print(jotest_summary)
```

The Johansen cointegration test results shows that these time series are not cointegrated at 5% level of significance. For more information about the ca.jo function,
please see the document available at
https://www.rdocumentation.org/packages/urca/versions/1.2-9/topics/ca.jo.

```
######################
# Johansen-Procedure #
######################

Test type: trace statistic , with linear trend

Eigenvalues (lambda):
[1] 0.235817473 0.003345041

Values of teststatistic and critical values of test:

          test 10pct  5pct  1pct
r <= 1 |  0.20  6.50  8.18 11.65
r = 0  | 16.07 15.66 17.95 23.52
```

Figure 10. Johansen Cointegration Test Output

VAR Estimation

R Code

```
# VAR Estimation

var_data = data.frame(dly=diff(ly), dlm=diff(lm)) # First
difference the data: I(1) and no cointegration at 5%
significance level
obs <- nrow(var_data)

var_ldy_ldm <-VAR(var_data, p=lag, type="const") #
Sample VAR Model Estimation
print(var_ldy_ldm)
```

Before we can conduct the Granger causality test, an VAR system is estimated.
For more information about the VAR function, please see the document available at https://cran.r-project.org/web/packages/vars/vars.pdf

```
VAR Estimation Results:
=======================

Estimated coefficients for equation dly:
========================================
Call:
dly = dly.l1 + dlm.l1 + dly.l2 + dlm.l2 + dly.l3 + dlm.l3 + dly.l4 + dlm.l4 + dly.l5 + dlm.l5 + const

      dly.l1       dlm.l1       dly.l2       dlm.l2       dly.l3       dlm.l3       dly.l4       dlm.l4       dly.l5       dlm.l5
 0.238345094  0.103037302  0.154311318 -0.017283784  0.092061345 -0.035855633 -0.006708777  0.044017979  0.228922624  0.022232389
       const
 0.001760934


Estimated coefficients for equation dlm:
========================================
Call:
dlm = dly.l1 + dlm.l1 + dly.l2 + dlm.l2 + dly.l3 + dlm.l3 + dly.l4 + dlm.l4 + dly.l5 + dlm.l5 + const

     dly.l1      dlm.l1      dly.l2      dlm.l2      dly.l3      dlm.l3      dly.l4      dlm.l4      dly.l5      dlm.l5       const
 0.18946899  0.04299033 -0.37711831 -0.23527690 -0.62410003  0.14063597  0.14642836 -0.11428841 -0.30789598  0.08814892  0.10610363
```

Figure 11. VAR Estimation Result

Granger Causality Test

R Code

```
granger.mpl <- function(data, restriction, causality, lag,
Nobs, df, Nos)
{
  var.u<-VAR(data,p=lag,type="const")
  unrestricted<-det(cov(as.matrix(resid(var.u))))

  var.r<-restrict(var.u,method="man",resmat=restriction)
  restricted<-det(cov(as.matrix(resid(var.r))))

  value.test<-(Nobs-lag-(1+lag*Nos))*(log(restricted)-
log(unrestricted))
  p<-pchisq(value.test, df, lower.tail=FALSE)

  return(matrix(c(df,value.test,p),nrow=1,
                byrow=TRUE,dimnames=list(c(causality),c("df",
"chi^2", "p"))))
}
```

```
# Linear Granger-causality Test

## For M causes Y
res1 <- matrix(c(rep(c(1,0),5), 1, rep(c(1,1) ,5) ,1), nrow=2,
byrow=TRUE)
granger.mpl(var_data, res1, "Y <- M", lag, obs, lag, 2)

## For Y causes M
res2 <- matrix(c(rep(c(1,1), 5), 1, rep(c(0, 1), 5), 1),
nrow=2, byrow=TRUE)
granger.mpl(var_data, res2, "M <- Y", lag, obs, lag, 2)
```

The Granger causality test results show that, the variable lm (M) Granger cause ly (Y) and ly (Y) does not Granger cause lm (M), at 5% level of significance.

```
        df    chi^2          p
Y <- M  5 12.15263 0.03275519

        df    chi^2         p
M <- Y  5 2.618145 0.7586064
```

Figure 12. Granger Causality Test Result

Export the Residuals of the VAR System for Further Analysis

R Code

```
var_resid <- resid(var_ldy_ldm) # Get the residuals from the VAR system

# Export residuals for further processing

write.table(var_resid[, 'dly'], file = paste0(script_folder, "/var_dly_resid.csv"), col.names = FALSE, row.names = FALSE)
write.table(var_resid[, 'dlm'], file = paste0(script_folder, "/var_dlm_resid.csv"), col.names = FALSE, row.names = FALSE)
```

Finally, the residuals are exported into 2 files separately to be used in the next part of the analysis.

Non-linear Granger Causality Test

The sample program is written in C. The C/C++ languages are generally implemented as a compiled language, i.e. the source code of the program is compiled into machine code before executing. A C/C++ compiler is therefore required in order to execute the program. In this illustration, CodeBlocks, a user-friendly free and open-source IDE (Integrated Development Environment) is used to run the program to simplify the code execution process. The selected CodeBlocks installer comes with the GNU Compiler Collection (GCC)[1], one of the most popular C/C++ compilers, as included in the bundled MinGW[2] development environment for Windows.

[1] https://gcc.gnu.org/
[2] http://www.mingw.org/

Step 1: Download and Install CodeBlocks

Go to the official website of CodeBlocks: http://www.codeblocks.org

Click the Binaries page link under Downloads in the main menu on the left hand side. Alternatively, direct go to this URL http://www.codeblocks.org/downloads/binaries and select the CodeBlocks version for your operating system.

In this illustration, the version *codeblocks-17.12mingw-setup.exe* is used. This setup file includes the G++ compiler, which is one of the most popular C++ compilers.

The selected version of CodeBlocks can be downloaded directly form this URL at

http://sourceforge.net/projects/codeblocks/files/Binaries/17.12/Windows/codeblocks-17.12mingw-setup.exe

After downloading the setup file, open it to install CodeBlocks.

Step 2: Prepare the Source Code and Data Files

Save the source code file to the same folder as the output files from the R code. There are three files involved, namely, the file of residuals generated from previous steps, i.e. *var_dlm_resi.csv*, and *var_dly_resid.csv*, the source code file of the C/C++ program, *z.cpp.*

z.cpp

This is the source code of the program.

var_dlm_resid.csv

This is the data file to be loaded by the program as the data for variable x

var_dly_resid.csv

This is the data file to be loaded by the program as the data for variable y

Configurations in the Source Code File (optional)

There are some configurable parameters in the source code.

Excerpts from Source Code:

```
#define Nobs    58
#define infile1  "var_dlm_resid.csv "
#define infile2  "var_dly_resid.csv"
#define outfile  "output.txt"

double epsilon=1.5;
int m=5;
```

- ***Nobs***: the number of observations in the input files.

- ***infile1***: the name of the input file for the time series variable x. When running the program, the program expects there is a file named exactly as specified here to be used as the values of the time series variable x.

- ***infile2***: similar to infile1 - the name of the input file for the time series variable y.

- ***outfile***: the name of the output file. After successful program execution, an output file with the execution result will be generated. This parameter specifies the output file name.

- ***epsilon***: the epsilon parameter used in the estimation.
- ***m***: the number of lags

Replace these parameters with the desired values should there be any required changes in the names of the files, the number of observations or the epsilon parameter.

Data Format of the Data Files (var_dly_resid.csv and var_dlm_resid.csv):

```
0.142970455
-6.34E-05
0.019360068
-0.108781912
-0.082939218
0.063785793
0.119865115
0.01925455
-0.125646739
0.059877199
-0.054504784
0.083151366
-0.013925159
0.012672463
-0.049037041
```

Figure 8. Sample Data Format for the Input Files

In the input file is a single column of the time series. Each value in a row will be loaded as a data point for the corresponding time series.

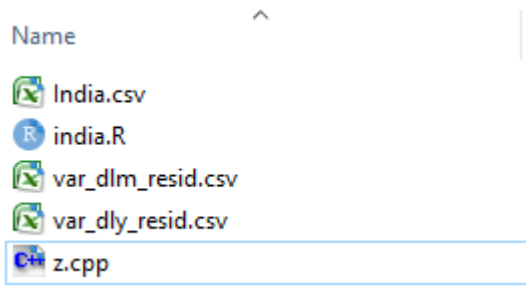


Figure 10. Source Code and Data Files

Step 3: Open the z.cpp source code file with CodeBlocks

The required files and software are ready. Open CodeBlocks

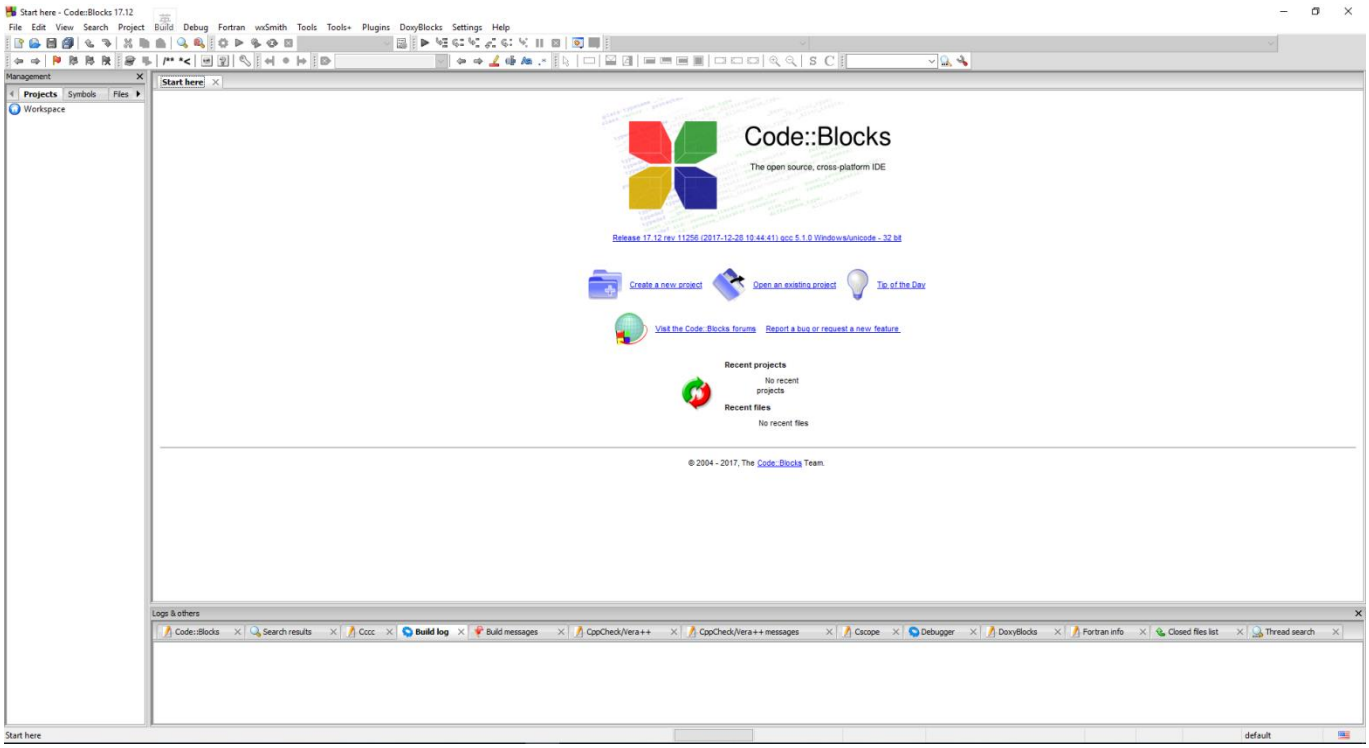


Figure 11. Welcome Screen of CodeBlocks

In the upper-left corner, click File then click Open

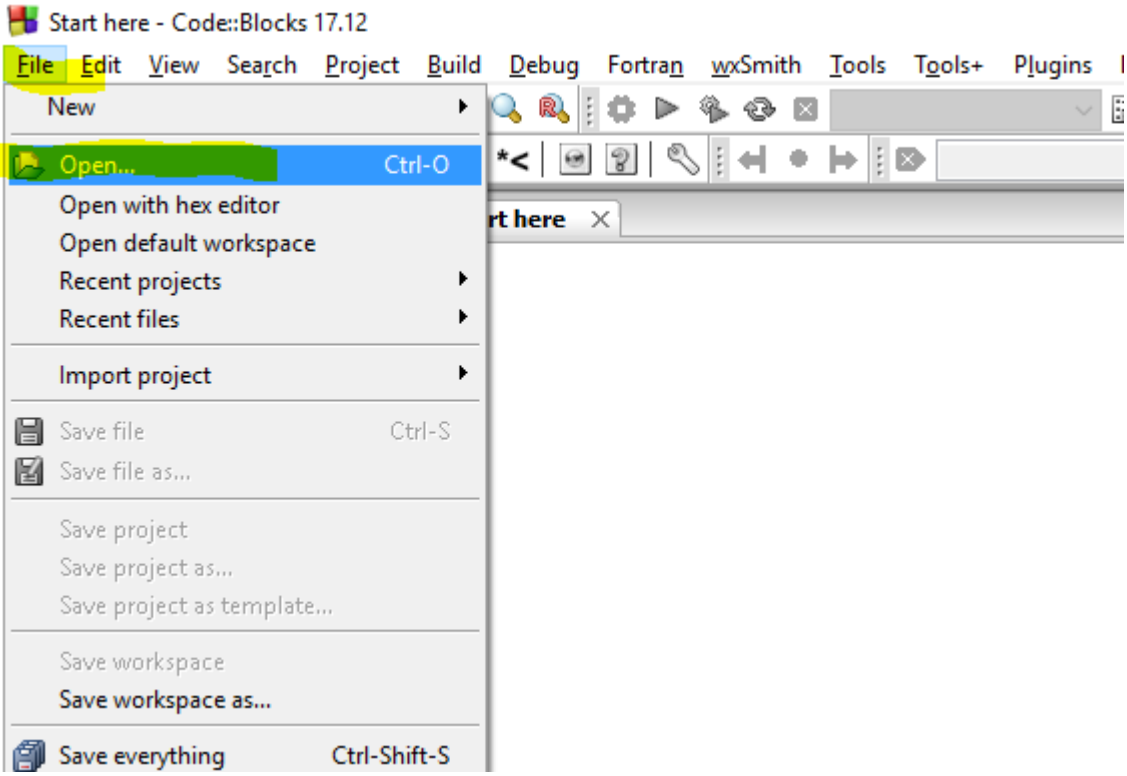


Figure 12. Open File in CodeBlocks

Browse to the folder where the 3 downloaded files are saved. Then, select the z.cpp file and click Open.

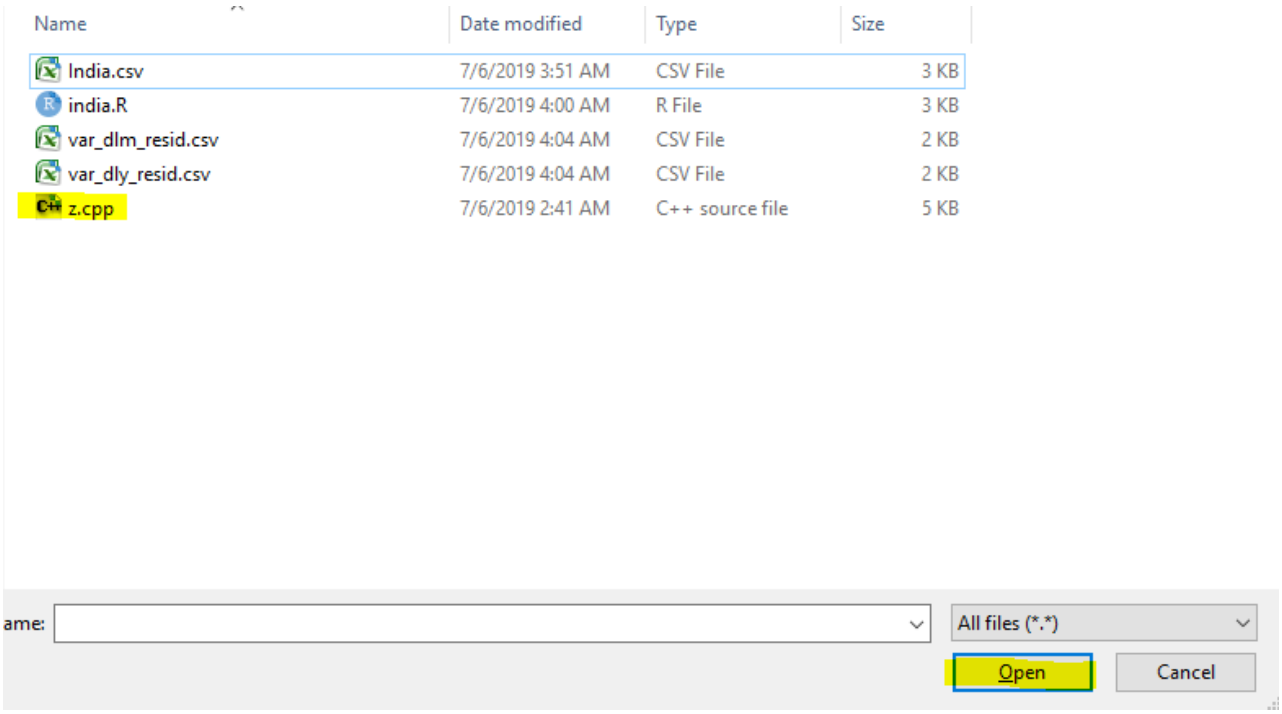


Figure 13. Choose the Source Code File

The source code of the program is displayed inside the CodeBlocks editor. In case of any required change, for example, change of the sample size or names of the input files, please modify the parameter values accordingly and save.

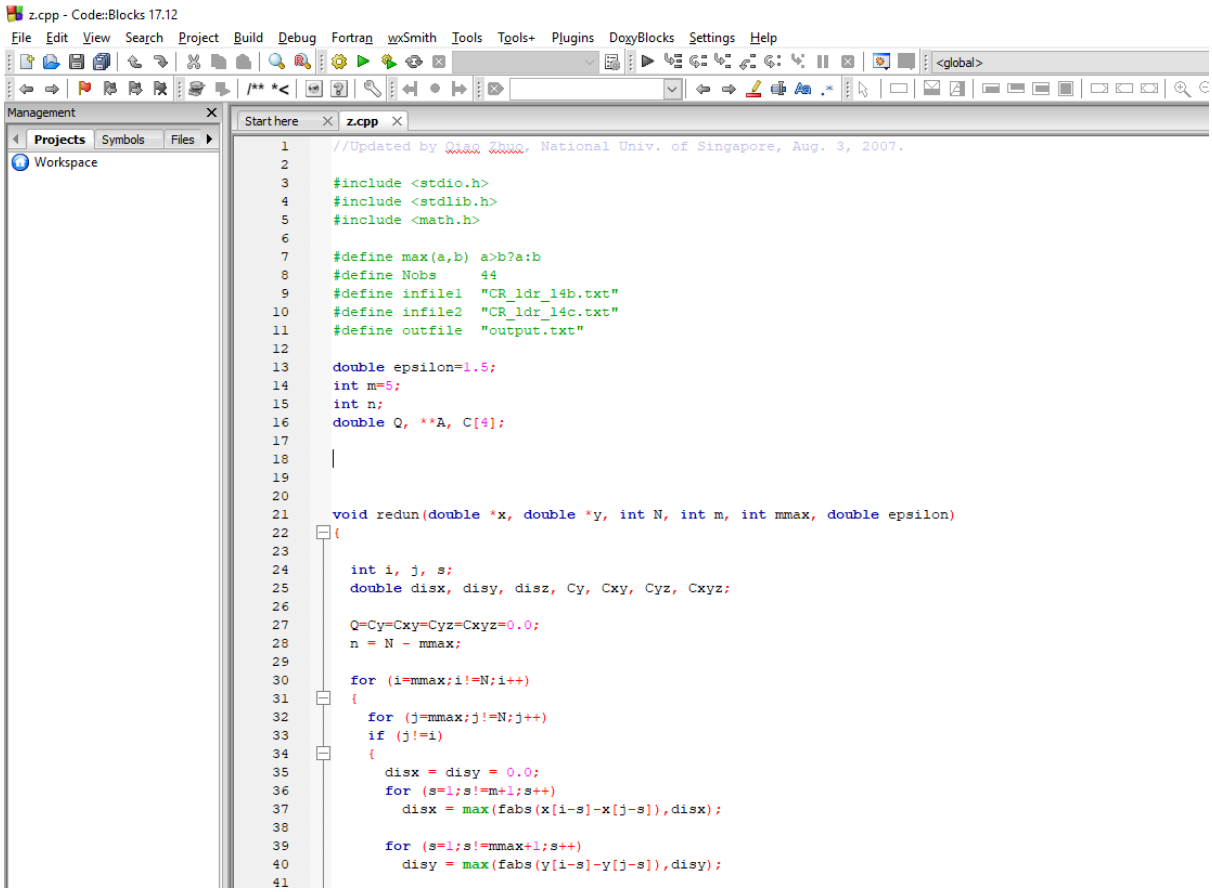


Figure 14. Code Editor in CodeBlocks

Step 4: Build the Code

A build process is required in order to execute the C++ program. After the build process, an executable file, in our case z.exe will be created.

Click the Build button

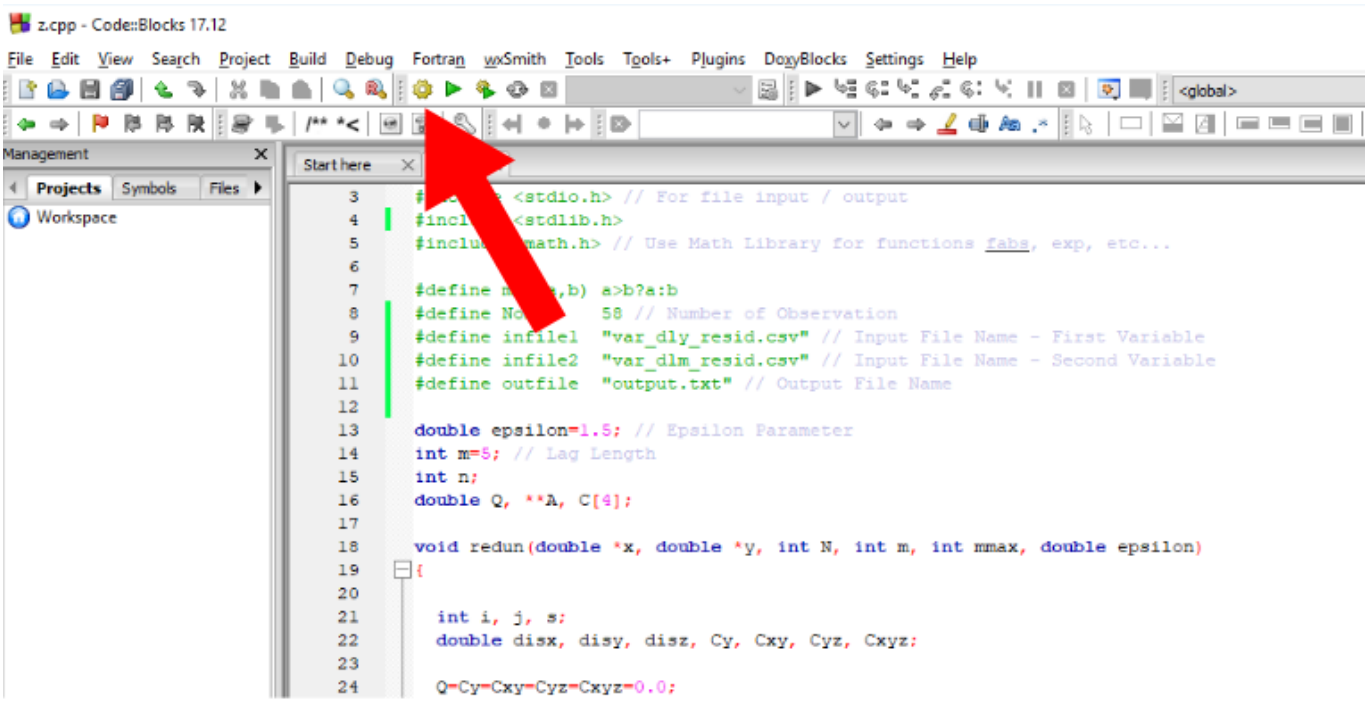


Figure 15. Build Button in CodeBlocks

After the build completed, there are 2 new files in the folder, namely z.exe and z.o.

The z.o file is an intermediate file produced during the build process. It is not directly executable but is used by the CodeBlocks IDE to produce the final executable program, i.e. the z.exe file.

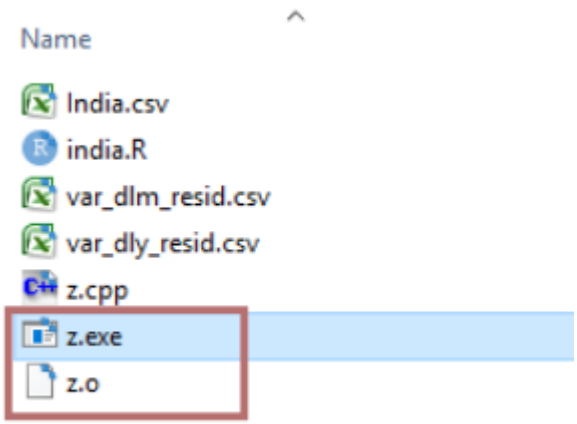


Figure 16. Generated Files after the Build Process

Step 5: Run the Executable

The executable program z.exe has been created from source code and is ready to be run. It can be run

by just double clicking the z.exe in the folder. Alternatively, using the CodeBlocks IDE, users can run the program by clicking the Run button in the user interface, as shown below.

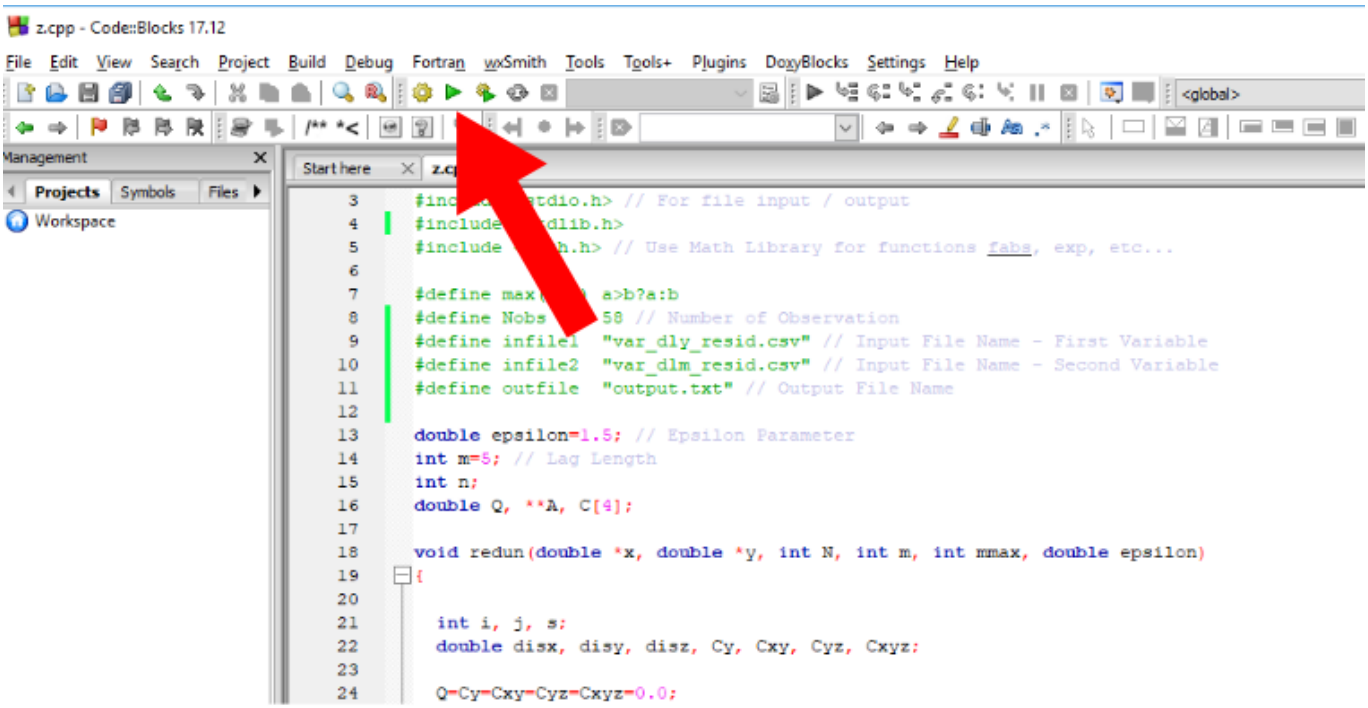


Figure 17. The Run Button in CodeBlocks

After the program execution, the execution time is reported. The return value 0 signals the success of the program execution.

Press any key to close the prompt.

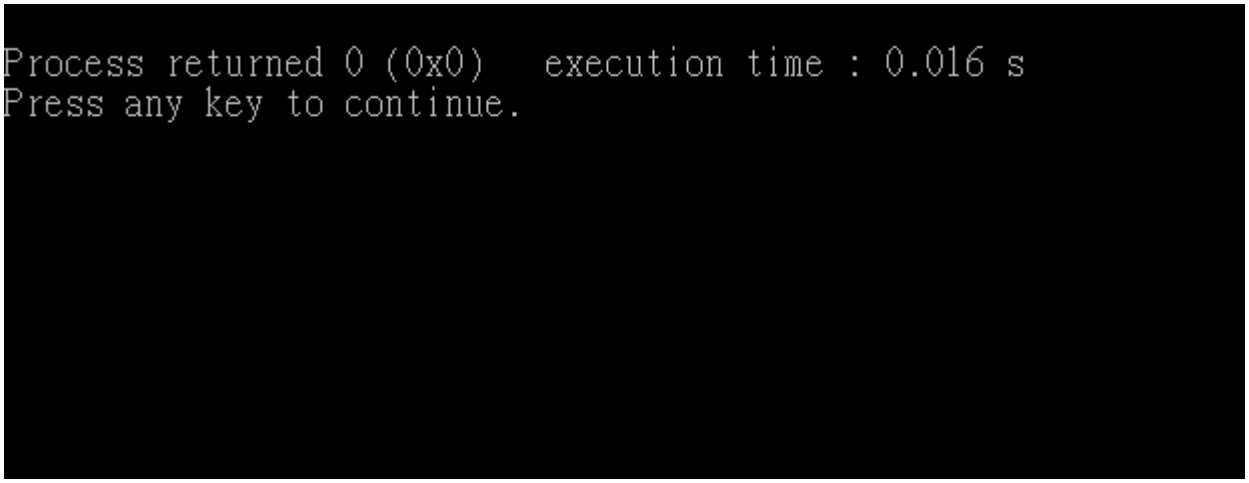
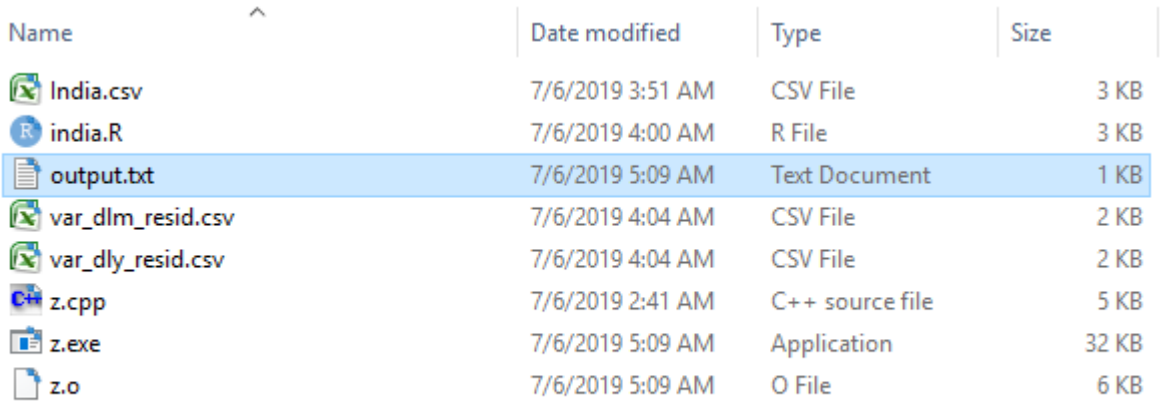


Figure 18. Screen after Successful Run

Step 6: Review the Output

Finally, to view the execution result, go to the program folder. A plain text output file output.txt has been generated after the execution. (as described in the configuration section in Step 2, the output file

name can be configured as needed. The default file name is output.txt)



Name	Date modified	Type	Size
India.csv	7/6/2019 3:51 AM	CSV File	3 KB
india.R	7/6/2019 4:00 AM	R File	3 KB
output.txt	7/6/2019 5:09 AM	Text Document	1 KB
var_dlm_resid.csv	7/6/2019 4:04 AM	CSV File	2 KB
var_dly_resid.csv	7/6/2019 4:04 AM	CSV File	2 KB
z.cpp	7/6/2019 2:41 AM	C++ source file	5 KB
z.exe	7/6/2019 5:09 AM	Application	32 KB
z.o	7/6/2019 5:09 AM	O File	6 KB

Figure 19. Output File Generated from the Program

Open the output.txt to view the result. The output file reports the Sample Size, the Epsilon parameter value, the HJ Statistic and the P-value in a CSV (comma-separated values) with header format.

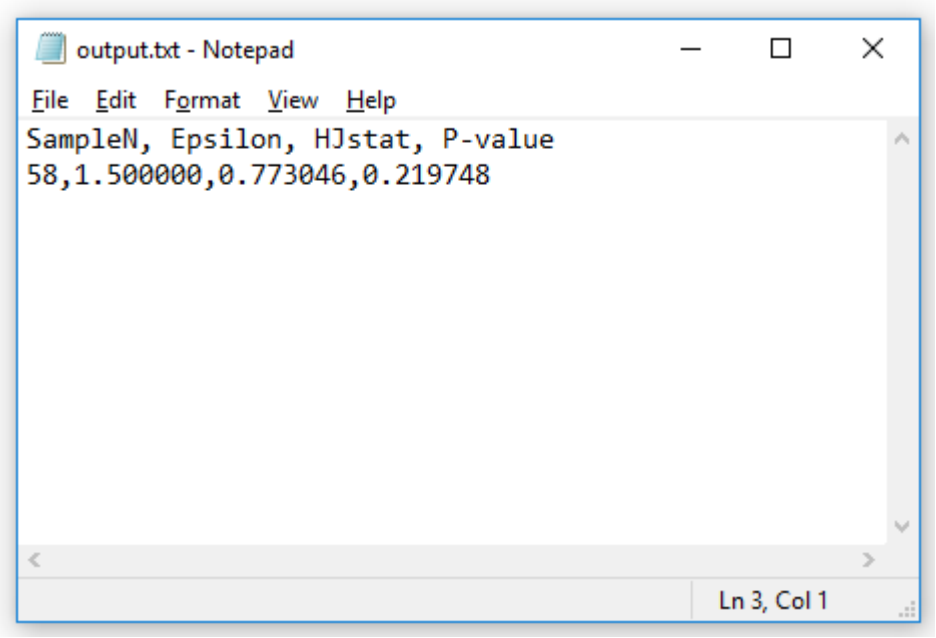


Figure 20. Result from the Program

Appendix

Source Code of the R Program

```
# A helper function to load and install packages

load_package <- function (package) {
  if (!require(package, character.only=TRUE)) {
    install.packages(package, quiet = TRUE);
    require(package, character.only=TRUE)
  }
}

# Install and load required R packages

load_package("tseries") # For Unit Root Test
load_package("urca") # For Cointegration Test
load_package("vars") # For VAR
```

```
load_package("fBasics") # For Descriptive Statistics
load_package("rstudioapi") # To Detect Script Directory

script_folder <- dirname(rstudioapi::getActiveDocumentContext()$path)

india_data <- read.csv(paste0(script_folder, "/india.csv")) # Data file in the same folder as the R program

india_data <- india_data[, c("ly", "lm")]

# Descriptive Statistics

desc_stat <- basicStats(india_data)
print(desc_stat)
```

```
ly <- india_data[, "ly"]
lm <- india_data[, "lm"]

# Unit Root Test - ADF Test

adf_ly <- adf.test(ly)
adf_lm <- adf.test(lm)

adf_dly <- adf.test(diff(ly))
adf_dlm <- adf.test(diff(lm))

print(list(adf_ly=adf_ly,                adf_lm=adf_lm,
adf_dly=adf_dly, adf_dlm=adf_dlm))

# Cointegration Test - Johansen

lag <- 5 # lag-length for the VAR system
```

```
jotest=ca.jo(data.frame(ly, lm), type="trace", K=lag, ecdet="none", spec="longrun")
jotest_summary <- summary(jotest)

print(jotest_summary)

# VAR Estimation

var_data = data.frame(dly=diff(ly), dlm=diff(lm)) # First difference the data: I(1) and no cointegration at 5% significance level
obs <- nrow(var_data)

var_ldy_ldm <-VAR(var_data, p=lag, type="const") # Sample VAR Model Estimation
print(var_ldy_ldm)

granger.mpl <- function(data, restriction, causality, lag, Nobs, df, Nos)
```

```
{
  var.u<-VAR(data,p=lag,type="const")
  unrestricted<-det(cov(as.matrix(resid(var.u))))

  var.r<-
restrict(var.u,method="man",resmat=restriction)
  restricted<-det(cov(as.matrix(resid(var.r))))

  value.test<-(Nobs-lag-
(1+lag*Nos))*(log(restricted)-log(unrestricted))
  p<-pchisq(value.test, df, lower.tail=FALSE)

  return(matrix(c(df,value.test,p),nrow=1,

byrow=TRUE,dimnames=list(c(causality),c("df",
"chi^2", "p"))))
}

# Linear Granger-causality Test
```

```
## For M causes Y
res1 <- matrix(c(rep(c(1,0),5), 1, rep(c(1,1) ,5) ,1), nrow=2, byrow=TRUE)
granger.mpl(var_data, res1, "Y <- M", lag, obs, lag, 2)

## For Y causes M
res2 <- matrix(c(rep(c(1,1), 5), 1, rep(c(0, 1), 5), 1), nrow=2, byrow=TRUE)
granger.mpl(var_data, res2, "M <- Y", lag, obs, lag, 2)

var_resid <- resid(var_ldy_ldm) # Get the residuals from the VAR system

# Export residuals for further processing
```

```
write.table(var_resid[, 'dly'], file = paste0(script_folder, "/var_dly_resid.csv"), col.names = FALSE, row.names = FALSE)
write.table(var_resid[, 'dlm'], file = paste0(script_folder, "/var_dlm_resid.csv"), col.names = FALSE, row.names = FALSE)
```

Source Code of the C/C++ Program - z.cpp

```
#include <stdio.h> // For file input / output
#include <stdlib.h>
#include <math.h> // Use Math Library for functions fabs, exp, etc...

#define max(a,b) a>b?a:b
#define Nobs     58 // Number of Observation
#define infile1  "var_dly_resid.csv" // Input File Name - First Variable
#define infile2  "var_dlm_resid.csv" // Input File Name - Second Variable
#define outfile  "output.txt" // Output File Name

double epsilon=1.5; // Epsilon Parameter
int m=5; // Lag Length
int n;
double Q, **A, C[4];
```

```
void redun(double *x, double *y, int N, int m, int
mmax, double epsilon)
{

  int i, j, s;
  double disx, disy, disz, Cy, Cxy, Cyz, Cxyz;

  Q=Cy=Cxy=Cyz=Cxyz=0.0;
  n = N - mmax;

  for (i=mmax;i!=N;i++)
  {
    for (j=mmax;j!=N;j++)
    if (j!=i)
    {
      disx = disy = 0.0;
      for (s=1;s!=m+1;s++)
        disx = max(fabs(x[i-s]-x[j-s]),disx);
```

```
for (s=1;s!=mmax+1;s++)
  disy = max(fabs(y[i-s]-y[j-s]),disy);

if (disy <= epsilon)
{
  Cy++;
  A[3][i]++;

  if (disx <= epsilon)
  {
    Cxy++;
    A[1][i]++;
  }

  disz = max(fabs(y[i]-y[j]),disy);
  if (disz <= epsilon)
  {
    Cyz++;
```

```
        A[2][i]++;

        if (disx <= epsilon)
        {
          Cxyz++;
          A[0][i]++;
        }
      }
    }   // end condition |Yi - Yj| < epsilon
  }   // end loop over j
} // end loop over i

Q = (double) Cxyz/Cxy - (double) Cyz/Cy;

C[0] = Cxyz/(double)(n*(n-1));
C[1] = Cxy/(double)(n*(n-1));
C[2] = Cyz/(double)(n*(n-1));
C[3] = Cy/(double)(n*(n-1));
```

```
  for (i=0;i!=4;i++)
   for (j=mmax;j!=N;j++)
   {
     A[i][j] /= (double)(n-1);
     A[i][j] -= C[i];  // C to A
   }
}

/* normalise the time series to unit std. dev. */

void normalise(double *x, int N)
{
  int i;
  double mean=0.0, var=0.0;

  for (i=0;i!=N;i++)
  {
    mean += x[i];
    var += x[i]*x[i];
```

```
  }

  mean /= (double)(N);
  var /= (double)(N);
  var -= mean*mean;

  for (i=0;i!=N;i++)
    x[i] = (x[i]-mean)/sqrt(var);

  return;
}

// erf function (Error Function)

extern double erf( double x ) {

    double t, z, retval;

    z = fabs( x );
```

```
    t = 1.0 / ( 1.0 + 0.5 * z );
    retval = t * exp( -z * z - 1.26551223 + t *
                  ( 1.00002368 + t *
                  ( 0.37409196 + t *
                    ( 0.09678418 + t *
                      ( -0.18628806 + t *
                        ( 0.27886807 + t *
                        ( -1.13520398 + t *
                          ( 1.48851587 + t *
                            ( -0.82215223 + t *
                              0.1708727 ) ) ) ) ) ) ) ) );
    if( x < 0.0 )
        return retval - 1.0;

    return 1.0 - retval;
}

int main()
{
```

```
double x[Nobs], y[Nobs], *ohm, S2, \
  HJ_TVAL, HJ_Pval, d[4], sigma[4][4]; //hv;
int i, j, l, k, K, mmax;  //ieps, nn;
FILE *fil;

A = (double **) malloc(4*sizeof(double *));

for (i=0;i!=4;i++)
 A[i] = (double *) malloc(Nobs*sizeof(double));

K = (int)(sqrt(sqrt(Nobs-m)));

ohm = (double *) malloc(K*sizeof(double));

ohm[0] = 1.0;
for (k=1;k<K;k++)
  ohm[k] = 2.0*(1.0-k/(double)(K));

//get external data
```

```
fil=fopen(infile1,"r");

for (i=0;i<Nobs;i++)
  fscanf(fil,"%lf",&x[i]);
fclose(fil);

fil=fopen(infile2,"r");

for (i=0;i<Nobs;i++)
   fscanf(fil,"%lf",&y[i]);
fclose(fil);

for (j=0;j!=4;j++)
{
  C[j] = 0.0;
  for (i=0;i!=Nobs;i++)
    A[j][i] = 0.0;
}
```

```
  normalise(x, Nobs);
  normalise(y, Nobs);

  mmax=m;

  redun(x,y,Nobs,m,mmax,epsilon); // call the
redun function defined above

  for (i=0;i!=4;i++)
    for (j=0;j!=4;j++)
    {
      sigma[i][j] = 0.0;
      for (k=0;k!=K;k++)
        for (l=mmax+k;l!=Nobs;l++)
          sigma[i][j]     +=     4.0*ohm[k]*(A[i][l]*A[j][l-
k]+A[i][l-k]*A[j][l])/(double)(2*(n-k));
    }
```

```
d[0] = 1.0/C[1];
d[1] = -C[0]/(C[1]*C[1]);
d[2] = -1.0/C[3];
d[3] = C[2]/(C[3]*C[3]);

S2=0.0;
for (i=0;i!=4;i++)
  for (j=0;j!=4;j++)
     S2 += d[i]*sigma[i][j]*d[j];

HJ_TVAL = Q*sqrt(n)/sqrt(S2);

// CDF
if (HJ_TVAL>0)
  HJ_Pval = 0.5 - .5*erf(HJ_TVAL/sqrt(2.0));
else
  HJ_Pval = 0.5 + .5*erf(HJ_TVAL/sqrt(2.0));

fil=fopen(outfile,"w");
```

```
  fprintf(fil, "SampleN, Epsilon, HJstat, P-value\n");
  fprintf(fil, "%i,%f,%f,%f\n", Nobs, epsilon,
HJ_TVAL, HJ_Pval);
  fclose(fil);

  return(0);
}
```

#include

The #include directive is a preprocessor command (which tells the compiler to do something before the actual compilation process) to include the file as specified to the current point). For example, it is the line “#include <math.h>” at near the top of the source code that make necessary math functions such as exp (exponential) and sqrt (square root) available in the program.

#define

A preprocessor command to define a macro. It can be used to define constants to be

substituted by the specified value in the source code.

For example the line “#define Nobs 58” in the source code instructs the preprocessor to substitube “Nobs” with 58 in the program. A macro can also be parameterized as in the line “#define max(a,b) a>b?a:b”

Comment

Use “//” for Single line comment. Any string in the same line after “//” is regarded as comments and will be ignored by the compiler.

Use “/* …. */” for multiple line comments. Any string between “/*” and “*/ is regarded as comments and will be ignored by the compiler.

Entry Point of a Program

The function main() is the entry point of the program.

If-statement

If (*condition*) *statement*

// the statement will be executed if the *condition* is true

If (*condition*) {

// statements to be executed if the *condition* is true

}

If (*condition*) {

// statements to be executed if the *condition* is true

} else {

// statements to be executed if the *condition* is not true

}

If (*condition_1*) {

// statements to be executed if the *condition* is not true

} else if (*condition_2*) {

// statements to be executed if the *condition_1* is not true

// and the *condition_2* is true

} else {

// statements to be executed if all the *conditions above*

// *i.e. condition_1 and condition_2* are false

}

For-loop

for (*initialization_step*; *condition*; *increment*) {

// statements to be executed if the *condition* is true

}

In a for loop, the initialization step is first run to initialize the counter. If the condition is true, the statements inside the for loop are executed, and then the increment step is run to update the counter.

For example,

```
#include <stdio.h>

int main()
{

    for (int i = 0; i < 5; i++) {
        printf("%d\n", i);
    }

    return 0;
}
```

Output:

0

1

2

3

4

The initialization step declare the counter i and set it as 0 (the initialization step *int i = 0*). The loop run as long as the counter i is less than 5 (the condition *i < 5*). After each loop the counter is increased by 1 (the increment step *i*++). The loop terminates as the i is increased to 5 and no longer prints the digit. In the above example, the “\n” in the first argument to the printf function is a new line character, and therefore in each loop the

counter i is outputted to a new line. The result in this example is the same if the condition is *i != 5*.

Assignment Operations

Expression	Explanation
a = b	assign the value of b to the variable a, not to be confused with the expression of the equality between two variables.
a = b = 0	assign the value 0 to variables a and b
a += b	equivalent to a = a + b
a -= b	equivalent to a = a - b
a *= b	equivalent to a = a * b
a /= b	equivalent to a = a / b

a++	return the value of a and then increase the value of a by 1

Comparison Operators

A logical comparison returns a boolean value *true* or *false* depending on the truth value of the expression.

Expression	Explanation
a == b	a is equal to b
a != b	a is no equal to b
a < b	a is less than b
a > b	a is greater than b
a <= b	a is less than or equal to b

a >= b	a is great than and equal to b

Useful Math Functions as Defined in math.h

Function	Explanation
exp	return the number of the constant e raised to the power to a floating-point number
sqrt	return the square root of a floating-point number
fabs	return the absolute value of the given floating-point number argument