RATS Programming Language

Walter Enders

Department of Economics, Finance & Legal Studies
University of Alabama
Tuscaloosa, AL 35487
wenders@cba.ua.edu

© 2003 Walter Enders. All Rights Reserved

Distributed by Estima (www.estima.com/enders).

This book is distributed free of charge, and is intended for personal, non-commercial use only. You may view, print, or copy this document for your own personal use. You may not modify, redistribute, republish, sell, or translate this material without the express permission of the copyright holder.

Table of Contents

Quick Index of Selected Functions and Instructions iv

Chapter 1: Linear and Nonlinear Estimation 1
1. The Data Set 2
2. Linear Regression and Hypothesis Testing 6
   Examples: 10
3. The LINREG Options 12
3.1 Using Switch Options 13
   Examples: 14
3.2 Using Choice Options 15
   Examples: 15
3.3 Using Switches, Choices and Internal Variables 16
4. Nonlinear Least Squares 21
4.1 Changing the Convergence Criteria 23
   Examples: 24
4.2 Examples of Nonlinear Least Squares 24
5. Maximum Likelihood Estimation 32
   Examples 34
6. GARCH Models 40
6.1 Examples of GARCH Processes 41

Chapter 2: VARs and Error-Correction Models 47
   Examples: 48
1. Hypothesis Testing and Model Selection 51
1.1 Innovation Accounting 53
   Examples: 53
2. Example: Estimation of a 3-Equation VAR 55
   Block Exogeneity: 57
   Innovation Accounting: 58
   Extensions 60
2.1 Near-VARs 61
   Example 62
3. Error-Correction Models 64
4. Structural Decompositions 69
4.1 Structural VARs with a Known G Matrix 71
   Examples 71
5. The Sims-Bernanke Decomposition 75
   Examples 75
6. The Blanchard-Quah Decomposition 82
6.1 The Technical Details 83
   Example 85
6.2 Decomposing GDP, Real M2 and the Interest Rate 86

Chapter 3: Loops Over Dates and Series 89
1. Dates as Integers 90
1.1 Omitting CALENDAR 91
   Examples: 92
2. Series as Integers 94
   Examples: 94
   Retrieving Labels and Integer Numbers 98
3. Do Loops 99

3.1 DO Loops, Switches and Choices 100
Examples: 100
3.2 Lag Length Tests 102
Modifying the Program 103
3.3 Lag Length Tests in a VAR 105
4. Loops for Dates 107
5. Loops for Series 109
6. The DOFOR Instruction 110
6.2 DOFOR and ENTRIES 113
Jazzing Up the Program 114
7. Loops with While and Until 115
Frequently Asked Questions 117

Chapter 4: IF Statements and Monte Carlo Experiments 119

Examples 120
1. If-Then-Else Blocks 121
Examples 121
1.1 Sample Program: Lag Lengths Again 123
2. The %IF(x,y,z) Function 126
Examples: 126
3. Estimating a Threshold Autoregression 128
3.1 Estimating the Threshold 130
4. Branching 133
Examples: 133
4.1 Sample Program Using BRANCH 133
5 Monte Carlo Experiments 136
Examples 137
5.1 A Simple Monte Carlo Experiment 137
5.2 Downward Bias in an AR Model 139
5.3 Power of the Dickey-Fuller Test 141
Power of the Test 143
5.4 The Enders-Granger Statistic 146
5.5 Inference in a Cointegrated System 149
The Program 151
6. Antithetic Random Variables 154
6.1 Bias in NLLS Estimates 154
7. Bootstrapping 158
7.1 Bootstrapping Regression Coefficients 160
7.2 The *AR* Coefficients of Real *GDP* Growth 164

Chapter 5: Vector and Matrix Manipulations 168

1. Creating Matrices and Vectors 168
1.1 Declare 169
Examples 170
1.2 COMPUTE 171
Examples 171
2. Matrix Operations 176
2.1 Operations on Subcomponents of a Matrix 176
2.2 Selecting ARMA Coefficients 178
2.3 Manipulating the Output of a VAR 179
3. Example: ENTER and Supplementary Cards 182
3.1 Automating Model Selection in a VAR 183
3.2 Creating a Near-VAR Using ENTER 185
Jazzing Up the Program 187
4. Example: Moving Average Representations 189

4.1 Impulse Responses in a First-Order VAR 195
5. Creating Matrices from Your Data 197
Examples 197
5.1 Estimating the Regression Coefficients 198
Exercises: 200
5.2 Hypothesis Testing in the Regression Model 200
5.3 Creating Series from a Matrix 203

Chapter 6: Writing Your Own Procedures 204

1. A Procedure to Display the AIC and SBC 206
2. Using SWITCH Options 207
2.1 Integer and Choice Options 209
3. Passing Series to a Procedure 211
4. Writing a Procedure to Test for Unit Roots 213
Notes: 214
4.1 Creating Local Variables 214
4.2 Adding Options 215
5. Retrieving START and END entry values 219
Examples: 220
5.1 Passing Information by Address 222
5.2 Optional Fields 223
Examples: 224
6. A Procedure for Computing Lag Lengths 226
7. Interacting With Procedures 229
Examples 230
8. Creating a Menu 233
Example 233
8.1 Creating a USERMENU 235
Jazzing up the Procedure 237
9. An Interactive Procedure with Menu and USERMENU 239
Jazzing up the Procedure 1 242
Jazzing up the Procedure 2 244
Cleaning up 246

Index 248

References 250

Quick Index of Selected Functions and Instructions

NAME	Description as used in test	Approx. Page
%IF	Evaluate a conditional statement	126
%L(i)	Returns the label of series i	98
%S(L)	Returns series number corresponding to L	182
BOXJENK	Estimates an ARIMA model	178
BRANCH	Program execution jumps to a new location	133
CDF	Cumulative density function	41
COMPUTE	Evaluates an expression	171
CORRELATE	Creates a correlogram	7
CVMODEL	Covariance matrix modeling	75
DECLARE	Creates a matrix	169
ENTER	Manipulates items on supplemental cards	182
ENTRIES	Number of entries to process on supplementary card	113
EQUATION	Creates an equation	151
EQV	Assigns labels to series	96
ERRORS	Creates forecast error variances and impulse responses	53
ESTIMATE	Estimates a system of equations	48
EWISE	Operates on elements of a matrix	177
FORECAST	Creates dynamic forecasts	60
GRAPH	Creates a high-resolution graph	5
GROUP	Combines equations into a system	62
IMPULSE	Creates impulse response functions	67
INFOBOX	Displays dialog box	145
INQUIRE	Selects the starting and ending values of a series	211
LABELS	Assigns labels to series	97
LINREG	Estimates a linear regression	6
LOCAL	Creates local variables in a procedure	214
MAKE	Creates a matrix from data	197
MAXIMIZE	Finds the maximum of a function	33
MENU	Displays a set of choices	233
MESSAGEBOX	Displays a message	229
NLLS	Nonlinear least squares	22
NLPAR	Select convergence criteria	23
NONLIN	Lists parameters for a nonlinear estimation	21
QUERY	Prompts user to input variables	229
RATIO	Performs a likelihood ratio test	51
RESTRICT	Test linear restrictions	10
ROBUSTERRORS	Corrects for heteroscedasticity	12
SCRATCH	Creates consecutively numbered series	97
SIMULATE	Creates a simulated series	151
SUBFORMULAS	Used in creating formulas	46
SUR	Seemingly unrelated regressions	61
UNTIL	Conditional control of program execution	115
USERMENU	Presents user with a list of choices	235
WHILE	Conditional control of program execution	115

Some instructions can perform multiple tasks. The descriptions refer to the task performed in the text. The pages are the locations containing the primary explanation or illustration.

Chapter 1:
Linear and Nonlinear Estimation*

This book is not for you if you are just getting familiar with RATS. Instead, it is designed to be helpful if you want to simplify the repetitive tasks you perform in most of your RATS sessions. Performing lag length tests, finding the best fitting ARMA model, finding the most appropriate set of regressors, and setting up and estimating a VAR can all be automated using RATS programming language. As such, you will not find a complete discussion of the RATS instruction set. It is assumed that you know how to enter your data into RATS and how to make the standard data transformations. If you are interested in learning about any particular RATS instruction, you can use RATS' *Help Menu* or refer to the *Reference Manual* and *User's Guide*. The emphasis here is on what I call RATS' programming language. These are the instructions and options that enable you to write your own advanced programs and procedures and to work with vectors and matrices. The book is intended for applied econometricians conducting the type of research that is suitable for the professional journals. As I tell my students, to do state-of-the-art research, it is often necessary to go "off the menu." I'm being a bit facetious, but by the time a procedure is on the menu of an econometric software package, it's not new. This book is especially for those of you who want to start the process of "going off the menu."

Of course, it will be impossible to illustrate even a small portion of the vast number of potential programs you can write. My intent is to give you the tools to write your own programs. Towards that end, I will discuss a number of the key instructions and options in RATS programming language and illustrate their use in some straightforward programs. I hope that the examples provided here will enable you to improve your programming technique. This book is definitely not an econometrics text. If you are like me, it is too difficult to learn econometrics and the programming tools at the same time. As such, I will try not to introduce any sophisticated econometric methods or techniques. Moreover, all of the examples will use a single data set MONEY_DEM.XLS and all examples are compatible with RATS 5.0.

This chapter begins with a quick overview of some of the basic RATS instructions and options we will be using in the later chapters. It is intended to refresh your memory and to introduce the use of switches, options, choices and internal variables. It then shows how to estimate nonlinear models using nonlinear least squares (NLLS) and maximum likelihood techniques.

---

* Thomas Doan, Thomas Maycock and Mark Wohar made many helpful comments on the earlier versions of the manuscript. All errors that remain are my own.

1. The Data Set

The file labeled MONEY_DEM.XLS contains quarterly values of seasonally adjusted U.S. nominal *GDP*, real *GDP* in 1996 dollars (*RGDP*), the money supply as measured by *M*2 and *M*3, and the 3-month and 1-year treasury bill rates for the period 1959:1 – 2001:1. Both interest rates are expressed as annual rates and the other variables are in billions of dollars. The data was obtained from the website of the Federal Reserve Bank of St. Louis (www.stls.frb.org/index.html) and saved in Excel format. If you open the file, you will see that the first eight observations are:[1]

DATE	GDP	RGDP	M2	M3	TB3mo	TB1yr
1959.1	496.10	2273.00	287.80	290.05	2.77	NA
1959.2	509.20	2332.40	292.12	294.35	3.00	NA
1959.3	510.20	2331.40	296.12	298.24	3.54	4.49
1959.4	514.20	2339.10	297.14	299.10	4.23	4.74
1960.1	527.90	2391.00	298.66	300.63	3.87	4.36
1960.2	527.10	2379.20	301.11	303.23	2.99	3.65
1960.3	529.90	2383.60	306.48	308.89	2.36	2.90
1960.4	524.60	2352.90	310.93	313.66	2.31	2.81

To help you understand the output from the sample programs, several conventions are used concerning typefaces:

Boldface — Within a sample program, a RATS instruction, set of instructions, or a procedure in **boldface** produces the subsequent sample output. Instructions and procedures not in boldface either produce no output or the output is not shown.

`Courier 10 point.` — RATS output is shown in `Courier 10 point font`. The output is either indented or contained in

a highlighted box

In addition, the `Courier 10 point font` is sometimes used to separate references to a particular program statement from the remainder of the text.

Italics — Many RATS instructions are used with parameters and options that you need to specify. The fields that you should specify are *italicized*. For example, the ALLOCATE instruction can be used to indicate the terminal date in a data set. Since you need to input

[1] See the RATS User's Guide for details about working with data sets that are not in an EXCEL format and with variables that are not quarterly.

the terminal value, the description of the instruction is written as: ALLOCATE *date*.

UPPERCASE — All text references to file names contained on the data disk are in UPPERCASE. In addition, textual references to the proper names of RATS instructions and procedures are all in upper case. RATS itself does not distinguish between UPPERCASE and lowercase characters. Within the sample programs, all names are in lower case.

If you have the file in your a:\ drive, you can read in the data set using the following four lines contained in Program 1.1 in the file CHAPTER1.PRG:

```
cal 1959 1 4
all 2001:1
open data a:\money_dem.xls          ;* Alter this line if the data set is not on drive a:\
data(org=obs,format=xls)
```

Note that only the first three letters of the CALENDAR and ALLOCATE instructions have been used—in fact, any RATS instruction can be called using only the first three letters of its name. If you use the TABLE instruction, your output should be:

table

Series	Obs	Mean	Std Error	Minimum	Maximum
DATE	169	1979.876331	12.232185	1959.100000	2001.100000
GDP	169	3572.739053	2873.158128	496.100000	10243.600000
RGDP	169	5142.364497	1950.840494	2273.000000	9439.900000
M2	169	1904.835266	1399.706717	287.800000	5043.710000
M3	169	2414.462229	1916.764710	290.053333	7260.136667
TB3MO	169	5.915148	2.590483	2.303333	15.053333
TB1YR	167	6.153872	2.393622	2.713333	14.380000

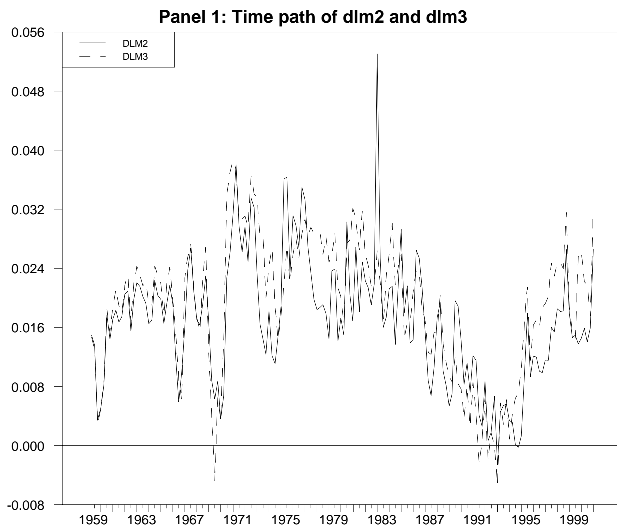
Many of the examples presented will use the growth rates of *M*2, *M*3 and real *GDP*, the first differences of the 3-month and 1-year *T*-bill rates and the rate of inflation (as measured by the growth rate of the GDP deflator). You can create these six variables using:

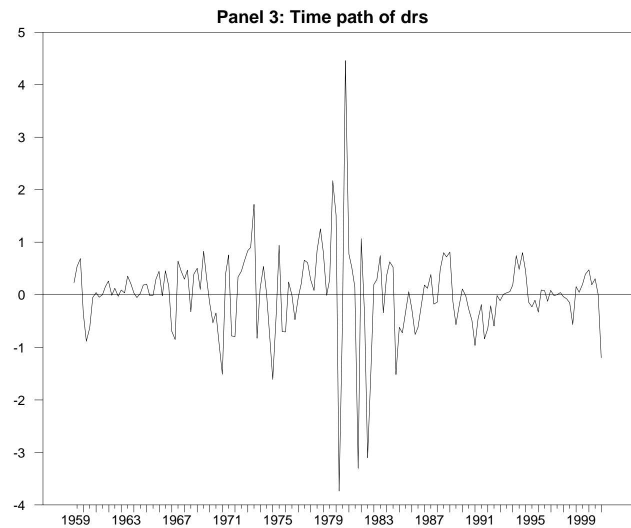
```
set dlrgdp = log(rgdp) - log(rgdp{1})
set dlm2 = log(m2) - log(m2{1})
set dlm3 = log(m3) - log(m3{1})
dif tb3mo / drs                     ;* Note that / instructs RATS to use the default range
dif tb1yr / drl                     ;*  of the data.
set price = gdp/rgdp
set dlp = log(price) - log(price{1})
```

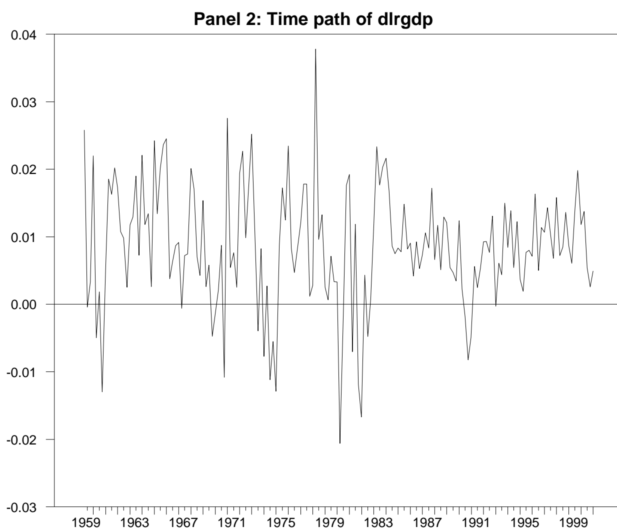
Notice that the logarithmic growth rate of a variable is preceded by *dl*, the suffixes *s* and *l* refer to the short-term and long-term interest rates, and that *price* (*i.e*. the *GDP* deflator) is computed as the ratio of nominal to real U.S. *GDP*. The logarithmic change in *price* (called *dlp*) is the quarterly inflation rate. We can create graphs of five of these series using:

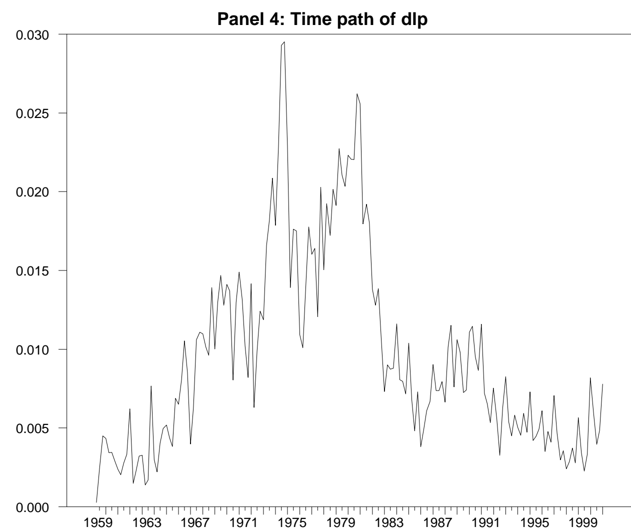
```
spgraph(hea='Graphs of the Five Principal Series',$
  hfi=2,vfi=2,key=upleft)
gra(hea='Panel 1: Time path of dlm3') 2 ; # dlm2; # dlm3
gra(hea='Panel 2: Time path of dlrgdp') 1 ; # dlrgdp
gra(hea='Panel 3: Time path of drs') 1 ; # drs
gra(hea='Panel 4: Time path of dlp') 1 ; # dlp
spgraph(done)
```

Graphs of Five Principal Series









Recall that the typical syntax of the GRAPH instruction is:

graph(options) *number*
\# *series start end*

where:

number	The number of series to graph. Here, the number 2 tells RATS that two series are to be graphed. The names of the series are indicated on the supplementary cards; there is one such card for each series.
series	The name of the series to graph. Remember, there is one supplementary card for each series.
start end	Range to plot. If omitted, RATS uses the current sample range.

The graph created here illustrates only a few of the available options. The commonly used options are:

HEADER =	A string of characters placed in quotes.
KEY =	The location of the KEY. You can use KEY = [NONE]/ UPLEFT/UPRIGHT/LOLEFT/LORIGHT/ABOVE/BELOW/ LEFT/RIGHT. The default is NONE.
DATES/ [NODATES]	RATS will label the horizontal axis unless the NODATES option is specified.
STYLE =	The default style is a line graph. The full set of styles is: STYLE= [LINE]/POLYGON/BAR/STACKEDBAR/OVERLAPBAR/ VERTICAL/STEP/SYMBOL
PATTERNS/ [NOPATTERNS]	By default, graphs use colors to distinguish series (if possible) and dashed lines and hatched patterns otherwise. PATTERNS forces the latter to be used.

The program illustrates the use of the SPGRAPH instruction to place multiple graphs on a single page. The first time SPGRAPH is encountered, RATS is told to expect a total of four graphs. The layout is such that two will go in the horizontal field (HFIELD=) and two in the vertical field (VFIELD=). The option HEADER= produces *Graphs of Five Principal Series* as the header for the full page. The individual headers on the four GRAPH instructions produce the headers on the individual panels. The instruction SPGRAPH(DONE) instructs RATS to produce the output shown on the next page.

2. Linear Regression and Hypothesis Testing

The LINREG instruction is the backbone of RATS and it is necessary to review its use. As such, suppose you want to estimate the first difference of the 3-month *T*-bill rate (i.e., *drs*) as the autoregressive process:

$$drs_t = \alpha_0 + \sum_{i=1}^{7} \alpha_i drs_{t-1} + \varepsilon_t$$

The next two lines of the program estimate the model over the entire sample period (less the seven usable observations lost as a result of the lags and the additional usable observation lost as a result of differencing) and save the residuals in a series called *resids*.

linreg drs / resids
 # constant drs{1 to 7}

```
Linear Regression - Estimation by Least Squares
Dependent Variable DRS
Quarterly Data From 1961:01 To 2001:01
Usable Observations  161   Degrees of Freedom  153
Centered R**2    0.284460   R Bar **2   0.251723
Uncentered R**2  0.284720   T x R**2   45.840
Mean of Dependent Variable   0.0155900621
Std Error of Dependent Variable 0.8194146456
Standard Error of Estimate      0.7088184693
Sum of Squared Residuals       76.870814220
Regression F(7,153)             8.6892
Significance Level of F         0.00000001
Durbin-Watson Statistic         1.956107

  Variable           Coeff      Std Error    T-Stat     Signif
**************************************************************************
*
1. Constant        0.013585280 0.055912496    0.24297  0.80835102
2. DRS{1}          0.361765228 0.079570494    4.54647  0.00001102
3. DRS{2}         -0.419337490 0.084224961   -4.97878  0.00000171
4. DRS{3}          0.393346293 0.089250834    4.40720  0.00001961
5. DRS{4}         -0.185273449 0.093546517   -1.98055  0.04943488
6. DRS{5}          0.201542364 0.089190935    2.25967  0.02525340
7. DRS{6}         -0.096578440 0.084095745   -1.14843  0.25258165
8. DRS{7}         -0.214686434 0.078991307   -2.71785  0.00732911
```

Notice that the estimation begins in 1961:1; RATS automatically adjusts for the eight observations lost due to differencing and the use of seven lags. As such, there are 161 usable observations (169 – 8 = 161); given the five parameters estimated, there are 161 - 8 = 153 degrees of freedom. Next, RATS reports four Goodness-of-Fit measures: centered R^2, R-bar square (centered R^2 adjusted for degrees of freedom), uncentered R^2, and TR^2 (number of

observations multiplied by the uncentered R^2).[2] The mean and standard error of *drs* over the 161 observations used in the regression are reported to be `0.0155900621` and `0.8194146456`, respectively. (*Note*: This will differ from the mean and standard error over all 168 observations.) These are followed by the standard error of the estimate (*i.e.*, the square root of the sum of the squared regression residuals divided by the degrees of freedom) and the sum of squared residuals. The *F*-statistic and its significance level can be used to test the hypothesis that all coefficients in the regression (other than the constant) are zero. Here, the sample value of *F* for the joint test $a_1 = a_2 = a_3 = \ldots = a_7 = 0$ is `8.6892`. Given that there are 7 restrictions and 153 degrees of freedom, this *F*-value is significant at the `0.00000001 level`. The Durbin-Watson test for first-order serial correlation in the residuals is `1.956107` (2.0 is the theoretical value of this statistic in the absence of serial correlation).

For each right-hand-side variable, the next portion of the output reports the coefficient estimate (Coeff), the standard error of the coefficient (Std Error), the *t*-statistic for the null hypothesis that the coefficient equals zero (T-Stat), and the marginal significance level of the *t*-test (Signif). For example, the coefficient of the first lag of *drs* is estimated to be `0.361765228` with a standard error equal to `0.079570494`. The associated *t*-test for the null hypothesis $\alpha_1 = 0$ is `4.54647`. If you use a *t*-table, you can verify that the significance level for this value of *t* is `0.00001102`.

It is always important to determine whether there is any serial correlation in the regression residuals. The CORRELATE instruction **c**alculates the autocorrelations (and the partial autocorrelations) of a specified series. The syntax and principal options are:

correlate(options) *series start end*

where:

series	The series used to compute the correlations.
start end	The range of entries to use. The default is the entire series.
corrs	Series used to save the autocorrelations (Optional).

The principal options are:

NUMBER=	The number of autocorrelations to compute. The default is the integer value of one-fourth the total number of observations.
PARTIAL=	Series for the partial autocorrelations. If you omit this option, the PACF will not be calculated.
QSTATS	Use this option if you want the Ljung-Box Q-statistics.
SPAN=	Use with *qstats* to set the width of the intervals tested. For example, with quarterly data, you can set *span = 4*, to obtain *Q*(4), *Q*(8), *Q*(12), and so forth.

In the example at hand, we can obtain the first twelve autocorrelations, partial autocorrelations (and the associated Q-statistics) of the residuals with:

---

[2] Let the dependent variable be denoted by *y*. Uncentered R^2 is 1 - (sum of squared regression residuals)/(sum of squared values of *y*). Centered R^2 is 1 - (sum of squared regression residuals)/(sum of squared deviations of *y* from the mean of *y*).

cor(number=12,partial=partial,qstats,span=4) resids

```
Correlations of Series RESIDS
Quarterly Data From 1959:03 To 2001:01
Autocorrelations
   1:  0.0154810 0.0029326 -0.0157832 -0.0125802 -0.0346243 0.0536726
   7: -0.0680555 0.1041120 -0.1047616  0.0499244 -0.1387784 0.0229181
Partial Autocorrelations
   1:  0.0154810 0.0026936 -0.0158741 -0.0121047 -0.0341736 0.0546344
   7: -0.0702982 0.1061879 -0.1110885  0.0575503 -0.1455796 0.0333431

Ljung-Box Q-Statistics
Q(4)  =     0.1125. Significance Level 0.99847633
Q(8)  =     3.5673. Significance Level 0.89390219
Q(12) =     9.5558. Significance Level 0.65486726
```

All of the autocorrelation and partial autocorrelations are small and the Ljung-Box Q(4), Q(8) and Q(12) statistics do not indicate the values are statistically significant. Moreover, the individual *t*-statistics suggest that only one of the autocorrelation coefficients is insignificant at conventional significance levels. However, the lags of {*drs*} are correlated with each other so that the individual *t*-statistics can be misleading. We might be concerned that the model is over-parameterized since sum of coefficients $\alpha_5 + \alpha_6 + \alpha_7$ is approximately zero. The EXCLUDE, SUMMARIZE, TEST, and RESTRICT instructions allow you to perform hypothesis tests on several coefficients at once. EXCLUDE is followed by a supplementary card listing the variables to exclude from the most recently estimated regression. RATS produces the *F*-statistic and the significance level for the null hypothesis that the coefficients of all excluded variables equal zero. Consider the following two exclusion restrictions:

exclude
 # drs{5 to 7}

```
Null Hypothesis : The Following Coefficients Are Zero
DRS        Lag(s) 5 to 7
F(3,153)=   7.69621 with Significance Level 0.00008002
```

exclude
 # constant drs{5 to 7}

```
Null Hypothesis : The Following Coefficients Are Zero
Constant
DRS        Lag(s) 5 to 7
F(4,153)=   5.78538 with Significance Level 0.00023113
```

The first exclusion restriction tests the joint hypothesis $\alpha_5 = \alpha_6 = \alpha_7 = 0$ and the second tests the joint hypothesis $\alpha_0 = \alpha_5 = \alpha_6 = \alpha_7 = 0$. The results support those of the *t*-tests; both of these null hypotheses can be rejected at conventional significance levels. SUMMARIZE has the same syntax as EXCLUDE but is used to test the null hypothesis that the sum of the coefficients indicated on the supplementary card is equal to zero. In the following example, the value of *t* for the null hypothesis $\alpha_5 + \alpha_6 + \alpha_7 = 0$ is -0.88256.

summarize
 # drs{5 to 7}

```
Summary of Linear Combination of Coefficients
DRS        Lag(s) 5 to 7
Value          -0.1097225
t-Statistic     -0.88256
Standard Error  0.1243225
Signif Level    0.3788566
```

EXCLUDE can only test whether a group of coefficients is jointly equal to zero. The TEST instruction has a great deal more flexibility; it is able to test joint restrictions on particular values of the coefficients. Suppose you have estimated a model and want to perform a significance test of the joint hypothesis restricting the values of coefficients α_i, α_j, ... , and α_k to equal r_i, r_j, ... , and r_k, respectively. Formally:

$\alpha_i = r_i$, $\alpha_j = r_j$ ... and $\alpha_k = r_k$.

To perform the test, you first type TEST followed by two supplementary cards. The first supplementary card lists the coefficients (by their number in the LINREG output list) that you want to restrict and the second lists the restricted value of each. Suppose you want to restrict the coefficients of the last three lags of *drs* to all be 0.1 (i.e., $\alpha_5 = 0.1$, $\alpha_6 = 0.1$, and $\alpha_7 = 0.1$). To test this restriction, use:

test
 # 6 7 8
 # 0.1 0.1 0.1

```
F(3,153)=   12.20650 with Significance Level 0.00000033
```

RATS displays the *F*-value and the significance level of the joint test $\alpha_5 = 0.1$, $\alpha_6 = 0.1$, and $\alpha_7 = 0.1$. If the restriction is binding, the value of *F* should be high and the significance level should be low. Hence, we can be quite confident in rejecting the restriction that each of the three coefficients equals 0.1. To test the restriction that the constant equals zero and that $\alpha_5 = 0.1$, $\alpha_6 = 0.1$, and $\alpha_7 = 0.1$, use:

test
 # 1 6 7 8
 # 0. 0.1 0.1 0.1

```
F(4,153)=   9.16377 with Significance Level 0.00000116
```

RESTRICT is the most powerful of the hypothesis testing instructions. RESTRICT can test multiple linear restrictions on the coefficients and estimate the restricted model. Although RESTRICT is a bit difficult to use, it can perform the tasks of SUMMARIZE, EXCLUDE, and TEST. Each restriction is entered in the form:

$$\beta_i\alpha_i + \beta_j\alpha_j + ... + \beta_k\alpha_k = r$$

where: α_i are the coefficients of the estimated model (i.e., each coefficient is referred to by its assigned number).
β_i are weights you assign to each coefficient.
and: r represents the restricted value of the sum (which may be zero).

To implement the test, you type RESTRICT followed by the number of restrictions you want to impose. Each restriction entails the use of two supplementary cards. The first lists the coefficients to be restricted (by their number) and the second lists the values of the β_i and r.

Examples:

1. To test the restriction that the constant equals zero use:

```
restrict 1
# 1
# 1 0
```

The first line instructs RATS to prepare for one restriction. The second line is the supplementary card indicating that coefficient 1 (i.e., the constant) is to be restricted. The third line imposes the restriction $1.0*\alpha_0 = 0$.

2. To test the restriction that $\alpha_1 = \alpha_2$ (i.e., $\alpha_1 - \alpha_2 = 0$), use:

```
restrict 1
# 2 3
# 1 -1  0
```

Again, the first line instructs RATS to prepare for one restriction. The second line is the supplementary card indicating that coefficients 2 and 3 are to be restricted. The third line imposes the restriction $1.0*\alpha_1 - 1.0\alpha_2 = 0$.

3. If you reexamine the regression output, it seems as if $\alpha_1 + \alpha_2 = 0$, $\alpha_3 + \alpha_4 = 0$ and $\alpha_5 + \alpha_6 = 0$. To test these three restrictions use:

```
restrict 3        ;* There are 3 restrictions and one set of supplemental cards
# 2 3             ;* for each restriction
# 1 1 0
# 4 5
# 1 1 0
# 5 6
# 1 1 0
```

Note that RESTRICT can be used with the CREATE option to test and estimate the restricted form of the regression. Moreover, whenever CREATE is used, you can save the regression residuals simply by providing RATS with the name of the series in which to store the residuals. In the example below, RATS displays the output of the restrictions from Example 3 above, and stores the regression residuals in the series *resids*. (NOTE: Only a portion of the output is shown).

```
restrict(create) 3 resids
# 2 3
# 1 1 0
# 4 5
# 1 1 0
# 5 6
# 1 1 0
```

```
F(3,153)=   2.25276 with Significance Level 0.08449735

  Variable           Coeff        Std Error     T-Stat    Signif
****************************************************************************
1. Constant         0.015301284  0.056541363    0.27062  0.78704020
2. DRS{1}           0.386719181  0.068502626    5.64532  0.00000008
3. DRS{2}          -0.386719181  0.068502626   -5.64532  0.00000008
4. DRS{3}           0.288370925  0.079736678    3.61654  0.00040262
5. DRS{4}          -0.288370925  0.079736678   -3.61654  0.00040262
6. DRS{5}           0.155835311  0.074036283    2.10485  0.03690688
7. DRS{6}          -0.155835311  0.074036283   -2.10485  0.03690688
8. DRS{7}          -0.198914927  0.078523550   -2.53319  0.01229020
```

Notice that the *F*-statistic (with three degrees of freedom in the numerator and 153 in the denominator) is `2.25276` with a significance level of `0.08449735`. Hence, at the 5% significance it is possible to reject the null hypothesis and conclude that the restriction is binding. At the 10% significance, we accept the null hypothesis.

3. The LINREG Options

LINREG has many options that will be illustrated in the following chapters. The usual syntax of LINREG is:

linreg(options) *depvar start end residuals*
\# *list*

where:

depvar	The dependent variable.
start end	The range to use in the regression. The default is the largest common range of all variables in the regression.
residuals	Series name for the residuals. Omit if you do not want to save the regression residuals.
list	The list of explanatory variables.

The most useful options for our purposes are:

DEFINE=	You can name the equation by setting DEFINE equal to the name you choose. Later, you can refer to the equation by its name.
[PRINT]/NOPRINT	Print the regression output.
VCV/[NOVCV]	Print the covariance/correlation matrix of the coefficients.
ENTRIES=	Number of entries to use from the supplementary card [all].

Note that LINGEG also contains options for correcting standard errors and *t*-statistics for hypothesis testing. The ROBUSTERRORS option computes a consistent estimate of the covariance matrix that corrects for heteroscedasticity as in White (1980). ROBUSTERRORS and LAGS= can produce various types of Newey-West estimates of the coefficient matrix. Moreover, SPREAD is used for weighted least squares and INSTRUMENTS is used for instrumental variables. The appropriate use of these options is described in Chapter 5 of the RATS User's Guide.

LINREG creates a number of variables that you can use in subsequent computations. A partial list of these variables is:

%BETA	The coefficient vector. The first coefficient estimated is %BETA(1), the second %BETA(2), and so on. For example, in the output for dr above, the constant is %BETA(1), the coefficient for dr{1} is %BETA(2), and so forth.
%XX	The $(X'X)^{-1}$ matrix. Note that %XX(i,j) contains the estimated covariance of coefficient i with coefficient j.

%TSTATS	The vector containing the t-stats for the coefficients. The first t-statistic is %TSTATS(1), the second is %TSTATS(2), and so on.
%STDERRS	Vector of coefficient standard errors.[3]
%NDF	Degrees of freedom.
%NOBS	Number of observations.
%NREG	Number of regressors.
%RSS	Residual sum of squares.
%RSQUARED	Centered R2.
%SEESQ	Standard error of estimate squared.
%DURBIN	Durbin-Watson statistic.
%QSTAT	Ljung-Box Q-statistic.
%QSIGNIF	Significance level of Q-statistic.
%RHO	First-lag correlation coefficient of the residuals.

The internal variables can be called from anywhere in a RATS program (including a procedure). You need to be a bit careful since the internal variables are recalculated every time you estimate a regression.

3.1 Using Switch Options

You can see how to work with RATS options by re-estimating the unconstrained *AR*(7) model of the change in the 3-month interest rate.

lin(robusterrors) drs
 # constant drs{1 to 7}

```
  Variable             Coeff      Std Error     T-Stat   Signif
*************************************************************************
*
1. Constant        0.013585280 0.052823866    0.25718 0.79703927
2. DRS{1}          0.361765228 0.096834548    3.73591 0.00018704
3. DRS{2}         -0.419337490 0.195669363   -2.14309 0.03210569
4. DRS{3}          0.393346293 0.123604815    3.18229 0.00146116
5. DRS{4}         -0.185273449 0.151116811   -1.22603 0.22018812
6. DRS{5}          0.201542364 0.112293295    1.79479 0.07268790
7. DRS{6}         -0.096578440 0.136399595   -0.70806 0.47891099
8. DRS{7}         -0.214686434 0.092486533   -2.32127 0.02027215
```

Notice that the LINREG instruction now uses the ROBUSTERRORS option to correct for possible heteroscedasticity. Since ROBUSTERRORS corrects only the covariance matrix, the point estimates of the coefficients are necessarily unchanged. If you compare these results to those obtained earlier, you will see that the option had very little effect on standard errors and the *t*-statistics.

[3] Note that some versions of the *Reference Manual* incorrectly refer to this as %STDERRORS.

Notice that the ROBUSTERRORS option can be ON or OFF—you can enter ROBUSTERRORS or NOROBUSTERRORS. Similarly, you can use either PRINT or NOPRINT for the regression output or VCV or NOVCV for displaying the covariance matrix. It is helpful to think of these options as a switch. The option will be executed if the switch is ON and will be bypassed if the switch is OFF. If neither choice is indicated in the options field of an instruction, RATS will resort to the default value of that particular option. **For all RATS switching options, you can also turn on the switch by equating its value to 1, and turn off the switch by equating its value to zero.** This can be very useful in writing procedures (see Chapter 6) since it is simple for the user to pass the appropriate switch value to the procedure.

Examples:

1. Since the PRINT option is the LINREG default, all of the following have equivalent effects:

```
lin drs

lin(print) drs

lin(print=1) drs

com j = 1                (where j is an integer)
lin(print=j) drs
```

2. Since NOROBUSTERRORS is the LINREG default, all of the following have equivalent effects:

```
lin(robusterrors) drs

lin(robusterrors=1) drs

com j = 1                (where j is an integer)
lin(robusterrors=j) drs
```

3. Since DATES is the GRAPH default, all of the following have the same effect:

```
gra(nodates) 1

gra(dates=0) 1

com dates = 0
graph(dates=dates)
```

3.2 Using Choice Options

Notice that GRAPH has a number of options requiring you to make choices. For example, the full set of choices for the location of the key is KEY = [none]/upleft/upright/loleft/loright/above/below/left/right. Similarly, STYLE = [line]/polygon/bar/stackbar/overlap/vertical/step/symbol. **For all RATS choice options, you can select the choice by its integer value in the choice list.**

Examples:

1. Since NONE is the default value and the first choice for KEY = and BAR is the third choice for STYLE = , the following will all produce a bar graph without a key:

```
gra(style=bar) 1

gra(sty = 3) 1

com j = 3
gra(key=1,sty=j) 1
```

2. Suppose you saved the residuals from a regression in a series called *resids*. You can obtain the first 12 autocorrelations in a series called *corrs* and the partial correlations in a series called *partial* by using:

```
cor(number=12,partial=partial) resids / cors
```

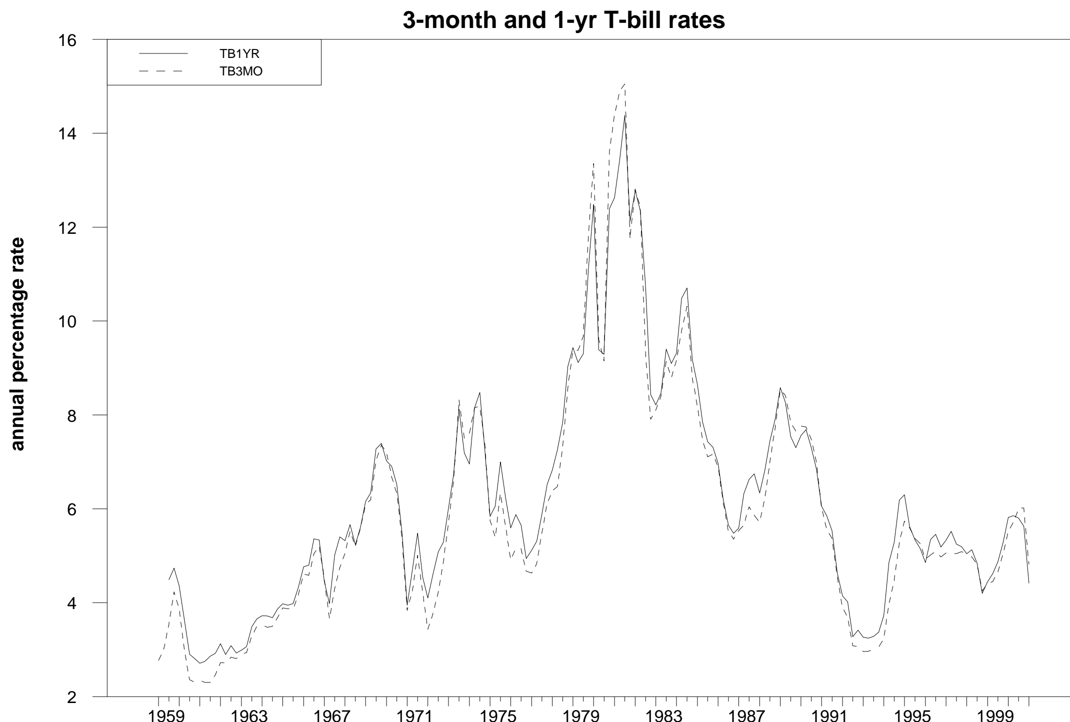
You can plot an overlapping bar graph of the ACF and PACF without dates, containing a header called ACF and PACF, and a key at the bottom of the graph either of the following:

```
gra(nodates,sty=overlap,key=below,header='ACF and PACF') 2 ; # cors ; # partial

gra(nodates,sty=5,key=7,header='ACF and PACF') 2 ; # cors ; # partial
```

3.3 Using Switches, Choices and Internal Variables

The best way to illustrate the use of switches, choices and internal variables is to work with another example. Economic theory suggests that long-term and short-term interest rates bear a long-run equilibrium relationship. Suppose that we try to estimate this relationship using the variables *tb3mo* and *tb1yr* from the file MONEY_DEM.XLS As a first step, we might want to plot the two variables. Consider:[4]

gra(header='3-month and 1-yr T-bill rates',vlabel='annual percentage rate', $
patterns,key=upleft) 2
 # tb1yr
 # tb3mo



It appears that both variables are non-stationary and there are periods of time during which they move apart. Nevertheless, they do seem to bear a strong long-run relationship with each other. We can estimate this relationship using:

lin tb1yr / resids

---

[4] Note that a $ indicates that the instruction continues on the next line. The option VLABEL= '*label*' allows you to supply a header for the vertical axis.

 # constant tb3mo

```
  Variable            Coeff          Std Error      T-Stat   Signif
**************************************************************************
*
1. Constant         0.6980794657  0.0665742554   10.48573 0.00000000
2. TB3MO            0.9167216207  0.0102654419   89.30172 0.00000000
```

We can obtain the first 12 residual autocorrelations and partial autocorrelations using:
cor(number=12,partial=partial) resids / cors

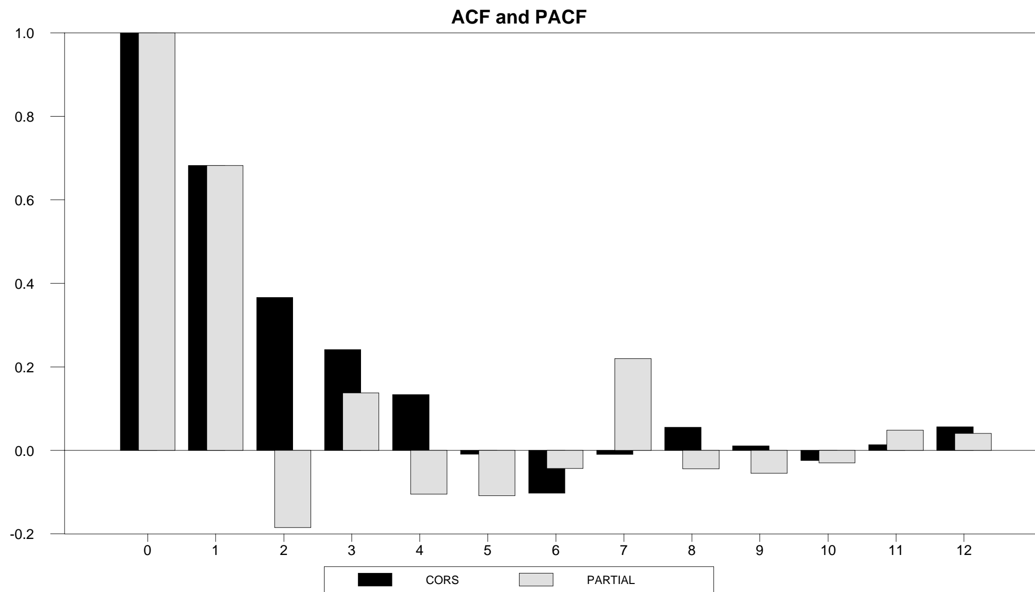
```
Correlations of Series RESIDS
Quarterly Data From 1959:03 To 2001:01
Autocorrelations
   1:  0.6820700  0.3663894  0.2412555  0.1334900 -0.0091483 -0.1024514
   7: -0.0096375  0.0551392  0.0106175 -0.0239909  0.0132915  0.0564374

Partial Autocorrelations
   1: 0.6820700 -0.1848049  0.1378831 -0.1048575 -0.1085939 -0.0432259
   7: 0.2198241 -0.0441826 -0.0550733 -0.0300141  0.0483862  0.0408635

Ljung-Box Q-Statistics
Q(12) =    118.3055. Significance Level 0.00000000
```

As expected, the residual autocorrelations seem to decay at a geometric rate. Notice that we used the QSTATS option--this option produces the Ljung-Box Q-statistic for the null hypothesis that all 12 autocorrelations are zero. Clearly, this null is rejected at any conventional significance level. If we wanted additional Q-statistics, we could have also used the SPAN= option. For example, if we wanted to produce the Q-statistics for lags, 4, 8, and 12, we could use:

cor(number=12,partial=partial,qstats,span=4) resids / cors

Since we are quite sure that the autocorrelations differ from zero, we dispense with this option. As indicated in the example above, we can graph the ACF and PACF using:

gra(nodates,number=0,sty=5,key=7,header='ACF and PACF',patterns=0) 2 $
 # cors
 # partial



Notice that we used the PATTERNS option. The default value is NOPATTERNS (*i.e*., the default is PATTERNS = 0). With this option OFF, your monitor will show the ACF in black and the PACF in blue.

Since it is clear that the residuals decay over time, we can estimate the dynamic process. Take the first difference of the *resids* and call the result *dresids*:

dif resids / dresids

Now estimate the dynamic adjustment process as:[5]

$$dresids_t = \alpha_0 resids_{t-1} + \sum_{i=1}^{p} \alpha_i dresids_{t-i} + \varepsilon_t$$

If we can conclude that α_0 is less than zero, we can conclude that the {*resids*} sequence is a convergent process. However, it is not straightforward to estimate the regression equation and then test the null hypothesis $\alpha_0 = 0$. One problem is that under the null hypothesis of no equilibrium long-run relationship (*i.e*., under the null of no cointegration between the two rates), we cannot use the usual *t*-statistics. Secondly, we do not know the appropriate lag length (*p*) to use when estimating the regression equation.

The ACF suggests that we can look at a relatively short lag lengths although the partial autocorrelation coefficient at lag 7 appears to be significant. As such, it seems prudent to estimate the regression equation using all lags through lag 8. One way to do this to enter the following program instructions:

---

[5]The Engle-Granger test for cointegration requires that $-2 < \alpha_0 < 0$. Note that an intercept is not necessary since regression residuals necessarily have a mean of zero.

```
dif resids / dresids
lin(noprint) dresids 1961:4 *
# resids{1} dresids{1 to 8}
compute aic = %nobs*log(%rss) + 2*(%nreg)
compute sbc = %nobs*log(%rss) + (%nreg)*log(%nobs)
dis 'T-stat' %tstats(1) 'The aic = ' aic ' and sbc = ' sbc
```

The first line creates the first difference of *resids* as the series *dresids.* The next two lines instruct RATS to regress $dresids_t$ on $resids_{t-1}$ and on dresids{1 to 8}. In order to ensure that all eight regressions are estimated over the sample period, the *start* date on the LINREG instruction is fixed at 1961:4. Since we are interested in only the *t*-statistic on $resids_{t-1}$, we suppress the output using the NOPRINT option. Next, we calculate the Akaike Information Criterion (*AIC*) and the Schwartz Bayesian Criterion (*SBC*) as:[6]

$AIC = T$ ln(residual sum of squares) + $2n$
$SBC = T$ ln(residual sum of squares) + n ln(T)

where: n = number of parameters estimated, including the intercept term (if any),
and T = number of usable observations.

We can use the internal variables constructed by LINREG to create *AIC* and *SBC* since:

%nobs	The number of usable observations in the previously estimated model
%rss	The residual sum of squares in the previously estimated model
%nreg	The number of regressors in the previously estimated model

Since %TSTATS(1) contains the *t*-statistic for the coefficient on $resids_{t-1}$, the last line displays this *t*-statistic, the *AIC* and the *SBC*. Now, you could go back and rerun the routine after editing the supplementary card for the LINREG instruction such that:

```
# resids{1} dresids{1 to 7}
```

However, to preview some of the material in the next chapter, it is more efficient to use RATS programming language. We can embed the routine in a DO loop:

---

[6] The formulas reported here and in the *RATS User's Manual* are easily computable monotonic transformations of the *AIC* and *SBC*. They will select the same model as the actual *AIC* and/or *SBC*.

```
do i = 1,8
  lin(noprint) dresids 1961:4 *
  # resids{1} dresids{1 to i}                     << Note the modification {1 to i}
  compute aic = %nobs*log(%rss) + 2*(%nreg)
  compute sbc = %nobs*log(%rss) + (%nreg)*log(%nobs)
  *  Note the modification to the next line:
  dis 'Lags: ' i 'T-stat' %tstats(1) 'The aic = ' aic ' and sbc = ' sbc
end do I
```

```
Lags: 1 T-stat   -5.79665 The aic =  362.90523 and sbc =  369.03042
Lags: 2 T-stat   -4.46711 The aic =  361.94474 and sbc =  371.13252
Lags: 3 T-stat   -4.64086 The aic =  362.25523 and sbc =  374.50561
Lags: 4 T-stat   -4.83109 The aic =  362.34292 and sbc =  377.65590
Lags: 5 T-stat   -4.63676 The aic =  364.11804 and sbc =  382.49361
Lags: 6 T-stat   -3.38834 The aic =  358.41748 and sbc =  379.85565
Lags: 7 T-stat   -3.40155 The aic =  360.07214 and sbc =  384.57290
Lags: 8 T-stat   -3.44382 The aic =  361.59437 and sbc =  389.15772
```

Now, each time RATS cycles through the LOOP, *i* increases from 1 to 8. Each time the supplementary card in encountered, RATS estimates the regression using a lags 1 through *i* and displays the results. Regardless of whether we use the 6-lag model selected by the *AIC* or the 1-lag model selected by the *SBC*, the *t*-statistic is sufficiently negative that we reject the null hypothesis α_0 equals zero. As such, we conclude that the two interest rates are cointegrated.

4. Nonlinear Least Squares

Given that many economic variables display asymmetric adjustment, nonlinear estimation methods have become quite popular. RATS allows you to perform nonlinear estimations in a number of ways. We will focus on nonlinear least squares and maximum likelihood estimation. Suppose that you want to estimate the following model using nonlinear least squares:

$$y_t = \alpha x_t^{\beta} + \varepsilon_t$$

Since the disturbance term is additive, you cannot simply take the log of each side and estimate the equation using OLS.[7] However, nonlinear least squares allows you to estimate α and β without transforming the variables. In RATS, you use the following structure to estimate a model using nonlinear least squares:

NONLIN *list of parameters to be estimated*
FRML *formula name the equation to be estimated*
COMPUTE *initial guesses for the parameters*
NLLS(FRML=*formula name***)** *dependent variable*

For the example at hand, you could use:

```
nonlin alpha beta
frml equation_1  y  = alpha*x**beta
com alpha = 1.0 , beta = 0.5
nlls(frml = equation_1)  y
```

The first instruction informs RATS that two parameters, named *alpha* and *beta*, are to be estimated. The FRML instruction sets up a formula named *equation_1*, the form of the equation is *y = alpha*x**beta*. The COMPUTE instruction provides the initial guesses for *alpha* and *beta*. The last line instructs RATS to use nonlinear least squares (NLLS) to estimate the series *y* using the formula previously defined as *equation_1*. In fact, all nonlinear least squares estimations use this four-step procedure. Specifically:

Step 1. Specify the parameter set to be estimated using the NONLIN instruction. The syntax for NONLIN is:

NONLIN parameter list

In most instances, the *parameter list* will be a simple list of the coefficients to be estimated. You can also include equality constraints such as *a* = = *b* or *a* + *b* = =1.0 (note the double

---

[7] If the model had the form $y_t = \alpha x_t^{\beta}\varepsilon_t$ where $\{\varepsilon_t\}$ is log-normal, it would be appropriate to estimate the regression in logs using LINREG.

equal sign) or weak inequality constraints of the form *b.ge.0* or *b.le.0*. In some instances, you might find it helpful to use a previously defined VECTOR for the parameter list.

Step 2. The formula needs to be specified. This is accomplished using the FRML instruction. The syntax for FRML is:

FRML(options) *formulaname depvar = function*(t)

where:

formula name	= the name you choose to give to the formula
depvar	= dependent variable
function(*t*)	= the function to be estimated

In the example above, *equation_1* is the name of the formula to be estimated, *y* is the dependent variable and the equation to be estimated is: *alpha*x**beta*. It would also be possible to omit the *depvar* field and use: FRML *equation_1 = alpha*x**beta*.

The most useful options for nonlinear least squares estimate are:

LASTREG/[NOLASTREG]	Converts the last regression estimated into a formula.
EQUATION=*equation to convert*	Converts the indicated linear equation to a formula. (Use only after estimating the equation).

Step 3. RATS requires initial guesses for the parameters to be estimated. This is accomplished using COMPUTE.

Step 4. Instruct RATS to estimate the FRML using the NLLS instruction. The syntax is:

NLLS(frml=*formulaname*, other options) *depvar start end residuals coeffs*

where:

depvar	Dependent variable used on the FRML instruction.
start end	Range to estimate.
residuals	Series to store the residuals (Optional).
coeffs	Series to store the estimated coefficients (Optional).

The principal options are:

METHOD =	[GAUSS]/SIMPLEX/GENETIC. GAUSS requires a twice differential function. USE SIMPLEX if you have convergence problems. It is possible to use SIMPLEX or GENETIC to refine the initial guesses before using GAUSS.
ITERATIONS=	Maximum number of iterations to use.

ROBUSTERRORS/ [NOROBUST]	As in LINREG, this option calculates a consistent estimate of the covariance matrix.

Note that NLLS defines most of the same internal variables as LINREG including %RSS, %BETA, %TSTATS and %NOBS. Moreover, NLLS defines the internal variable %CONVERGED. Note that %CONVERGED = 1 if the estimation converged and otherwise is equal to 0.

4.1 Changing the Convergence Criteria

Numerical optimization algorithms use iteration routines that cannot guarantee precise solutions for the estimated coefficients. Various types of 'hill-climbing' methods are used to find the parameter values that maximize a function or minimize the sum of squared residuals. If the partial derivatives of the function are *near* zero for a wide range of parameter values, RATS may not be able to converge to the optimum point. Moreover, you should also be cautious of results indicating that convergence takes place in one iteration; the resulting parameter values and associated *t*-statistics are often unreliable. NLPAR allows you to select the various criteria RATS uses to determine when (and if) the solution converges. As such, the NLPAR instruction allows you to control the precision of your answers. There are three principal options for NLPAR; the syntax is:

nlpar(options)

CRITERION=	In the default mode, CRITERION=COEFFICIENTS. Here, convergence is determined using the change in the numerical value of the coefficients between iterations. Setting CRITERION= VALUE means that convergence is determined using the change in the value of the function being maximized.
CVCRIT=	Converge is assumed to occur if the change in the COEFFICIENTS or VALUE is less than the number specified. The default is 0.00001.
SUBITERATIONS=	Subiteration limit [30]. Limits the number of new coefficient vectors examined once a direction has been chosen. You should increase the limit only if requested by RATS.

Note: CVCRIT and SUBITERATIONS are options for the NLLS and MAXIMIZE instructions. Unlike NLPAR, the convergence criterion is altered for this estimation only.

Examples:

1. nlpar(cvcrit=0.0001)

Setting CVCRIT=0.0001 means that RATS will continue to search for the values of the coefficients that minimize the sum of squared residuals until the change in the coefficients between iterations is not more than 0.0001.

2. nlls(frml=equation_1, cvcrit=0.0001) y

RATS will continue to search for the values of the coefficients that minimize the sum of squared residuals until the change in the coefficients between iterations is not more than 0.0001. Unlike example 1, the convergence criterion is 0.0001 for this estimation only. The instruction nlpar(cvcrit=0.0001) changes the default criterion for all subsequent estimations.

3. nlpar(criterion=value,cvcrit=0.0000001)

Setting CVCRIT=0.0000001 and CRITERION=VALUE means that RATS will continue to search for the values of the coefficients that minimize the sum of squared residuals until the change between iterations is less than 0.0000001.

4.2 Examples of Nonlinear Least Squares

1. The FRML instruction can include lagged dependent variables. For example, to estimate $y_t = \alpha y_{t-1}^{\beta} + \varepsilon_t$, you can use:

```
nonlin alpha beta
frml equation_1 y = alpha*y{1}**beta
com alpha = 1.0 , beta = 0.5
nlls(frml = equation_1) y
```

2. The NONLIN instruction allows you to impose restrictions on the parameters. To estimate $y_t = \alpha x_{1t}^{\alpha_1} x_{2t}^{\alpha_2} x_{3t}^{\alpha_3} + \varepsilon_t$, your first two instructions should be:

```
nonlin alpha alpha1 alpha2 alpha3
frml y = alpha*(x1**alpha1)*(x2**alpha2)*x3**alpha3
```

To ensure $\alpha_1 = \alpha_2$ modify the NONLIN instruction such that:

```
nonlin alpha alpha1 alpha2 alpha3 alpha1==alpha2
```

To ensure that alpha2 in not negative, use:

```
nonlin alpha alpha1 alpha2 alpha3 alpha2>=0.
```

or:

```
nonlin alpha alpha1 alpha2 alpha3 alpha2.ge.0.
```

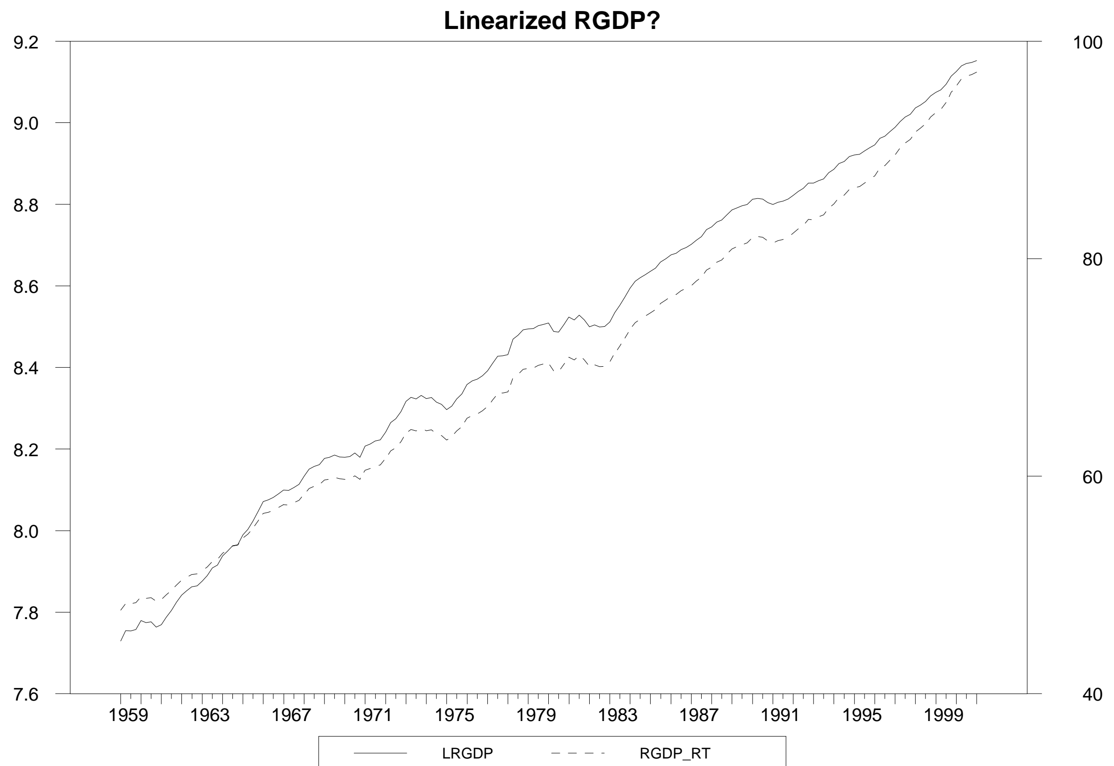
3. Suppose you are having difficulty estimating $y_t = \alpha\, y_{t-1}^{\beta} + \varepsilon_t$. If NLLS does not converge, you can increase the number of iterations from the default value of 40, provide better initial guesses, use NLPAR or use an alternative estimation method. A useful way to obtain satisfactory initial guesses is to use the simplex or genetic estimation method for a few iterations and then switch to the GAUSS method. To apply this technique to the equation from example 1, use:

```
nonlin alpha beta
frml equation_1 y = alpha*y{1}**beta
com alpha = 1.0 , beta = 0.5
nlls(frml = equation_1,method=simplex,iters=5) y
nlls(frml = equation_1) y
```

The first NLLS instruction uses the simplex method to perform the estimation and the second uses these results as initial guesses.

4. Time-series forecasters often estimate a nonlinear series using a quadratic or cubic trend. Alternatively, the series can be 'linearized' with a logarithmic or a square-root transformation. The appropriately transformed series can then be estimated with a linear time trend. The issue is to compare the 'fit' of the various models. The first four lines of Program 1.2 on the file CHAPTER1.PRG read in the data set MONEY_DEM.XLS. If you plot the series *rgdp*, you will notice that the trend appears to be nonlinear. The next four instructions of the program plot the time paths of log($rgdp_t$) and $(rgdp_t)^{0.5}$:

```
log rgdp / lrgdp            ; * Log transformation
sqrt rgdp / rgdp_rt         ; * Square root transformation
gra(overlay=line, header='Linearized RGDP?', key=below, patterns) 2
# lrgdp; # rgdp_rt
```



Each series seems to be linear; as such, the most appropriate transformation may not be clear. Hence, you might try estimating $rgdp_t$ using the following three forms:

$rgdp_t = \alpha_0 + \alpha_1 time + \alpha_2 time^2$;* Quadratic trend
$lrgdp_t = \alpha_0 + \alpha_1 time$;* Log transformation
$rgdp_rt_t = \alpha_0 + \alpha_1 time$;* Square root transformation

A problem arises in comparing the three estimates since each uses a different dependent variable. Instead, it is possible to estimate the last two equations in the form:[8]

$rgdp_t = \exp(\alpha_0 + \alpha_1 time) + e_t$;* Exponential trend

and:

$rgdp_t = (\alpha_0 + \alpha_1 time)^2 + e_t$;* Squared trend

You can estimate the quadratic specification using:

set time = t ; set t2 = t*t

[8] The squared trend model is equivalent to $rgdp_t = \alpha_0^2 + 2\alpha_0\alpha_1 time + \alpha_1^2 time^2 + \varepsilon_t$. In fact, this is a quadratic trend with only two free coefficients.

lin(noprint) rgdp; # constant time t2
compute aic = %nobs*log(%rss) + 2*(%nreg)
compute sbc = %nobs*log(%rss) + (%nreg)*log(%nobs)
dis 'The aic = ' aic ' and sbc = ' sbc

```
The aic =   2596.41409 and sbc =   2605.80379
```

To estimate the exponential specification, use the following four instructions:

nonlin a0 a1
frml model_2 rgdp = exp(a0 + a1*time)
com a0 = 1. , a1 = 0.1
nlls(frml=model_2) rgdp

```
  Variable   Coeff        Std Error      T-Stat       Signif
*************************************************************
1. A0     7.8099164840  0.0064018809  1219.94092  0.00000000
2. A1     0.0078135227  0.0000509728   153.28810  0.00000000
```

compute aic = %nobs*log(%rss) + 2*(%nreg)
compute sbc = %nobs*log(%rss) + (%nreg)*log(%nobs)
dis 'The aic = ' aic ' and sbc = ' sbc

```
The aic =   2566.27519 and sbc =   2572.53499
```

As such, the both the *AIC* and *SBC* select the exponential specification over the quadratic specification. The square root specification can be estimated using:

nonlin a0 a1
frml model_3 rgdp = (a0 + a1*time)**2
com a0 = 1. , a1 = 0.1
nlls(frml=model_3) rgdp

compute aic = %nobs*log(%rss) + 2*(%nreg)
compute sbc = %nobs*log(%rss) + (%nreg)*log(%nobs)
dis 'The aic = ' aic ' and sbc = ' sbc

```
The aic =   2649.14346 and sbc =   2655.40326
```

Note that the *AIC* and *SBC* rate this model as the one with the poorest fit.

5. ***The Logistic Smooth Transition Autoregressive (LSTAR) Model***: The *LSTAR* model generalizes the standard autoregressive model to allow for a varying degree of autoregressive decay. In its simplest form, the *LSTAR* model can be represented by:[9]

---

[9] To allow the threshold τ to be something other than zero, use: $\theta = \left[1 + \exp\left[\gamma(y_{t-1} - \tau)\right]\right]^{-1}$

$$y_t = \alpha_0 + \sum_{i=1}^{p} \alpha_i y_{t-i} + \theta(\beta_0 + \sum_{i=1}^{p} \beta_i y_{t-i}) + \varepsilon_t$$

where: $\theta = [1 + \exp(-\gamma y_{t-1})]^{-1}$ and $\gamma > 0$ is a scale parameter.

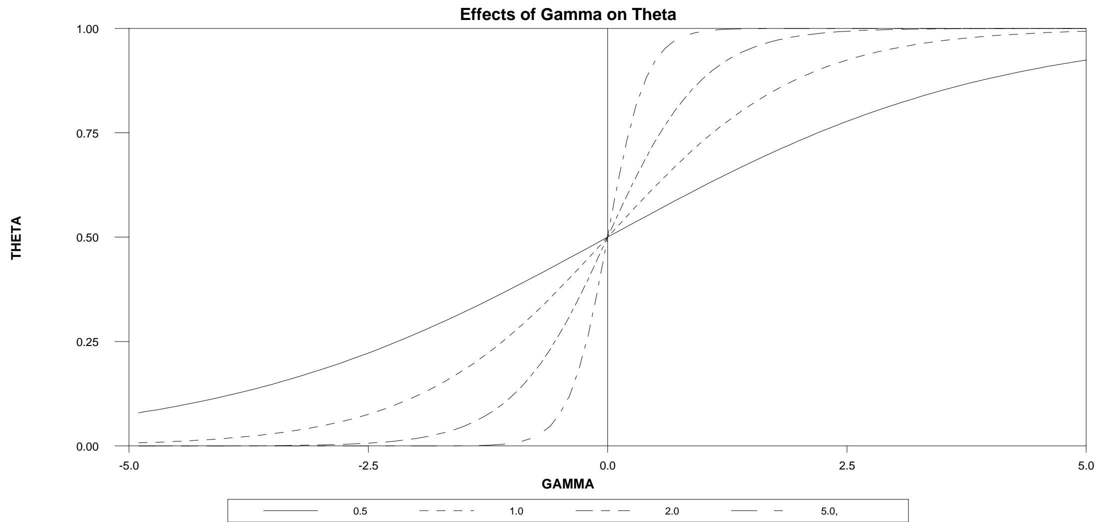
In the limit, as $\gamma \to 0$ or ∞, the *LSTAR* model becomes an *AR*(*p*) model since each value of θ is constant. As $y_{t-1} \to -\infty$, $\theta \to 0$ so that the behavior of y_t is given by $\alpha_0 + \alpha_1 y_{t-1} + \ldots + \alpha_p y_{t-p} + \varepsilon_t$. Similarly, as $y_{t-1} \to +\infty$, $\theta \to 1$ so that the behavior of y_t is given by $(\alpha_0 + \beta_0) + (\alpha_1 + \beta_1) y_{t-1} + \ldots + (\alpha_p + \beta_p) y_{t-p} + \varepsilon_t$. For intermediate values of γ, the degree of autoregressive decay depends on the value of y_{t-1}.

You can see the effects of γ using Program 1.3 on the file CHAPTER1.PRG. Since we are not interested in using calendar dates, we omit the CAL instruction. The first instruction sets the default length of a series to 101 entries. The second instruction creates the series *y*; since t runs from 1 to 101, *y* runs from –5.0 to +5.0. Each of the next four SET instructions creates a series representing θ as a function of γy_t for $\gamma = 0.5$, 1, 2.0 and 5.0.

```
all 101
set y = -5.1 + 0.1*t
set theta1 = (1 + exp(-0.5*y))**-1
set theta2 = (1 + exp(-y))**-1
set theta3 = (1 + exp(-2*y))**-1
set theta4 = (1 + exp(-5*y))**-1
```

The remainder of the program creates a scatter diagram of the various functions of θ. Note that γ acts as a smoothness parameter; for y_t near zero, movements in y_t have small effects on θ when γ is small.

```
com labels = || ' 0.5' , '1.0', '2.0', '5.0,' ||
scatter(header='Effects of Gamma on Theta',style=lines,patterns,klabels=labels,key=below, $
  vlabel='THETA',hlabel='GAMMA') 4
# y theta1 ; # y theta2 ; # y theta3 ; # y theta4
```



Now, use Program 1.4 to read in MONEY_DEM.XLS. Form the logarithmic change in *M3* and call the resulting series *dlm3*:

set dlm3 = log(m3) - log(m3{1})

Next, estimate $dlm3_t$ as an $AR(1)$ process and obtain the *AIC* and *SBC* using:

lin dlm3 / resids ; # constant dlm3{1}

```
 Variable        Coeff        Std Error     T-Stat    Signif
*************************************************************
1. Constant   0.0030875133  0.0009012005   3.42600  0.00077312
2. DLM3{1}    0.8436040010  0.0424922667  19.85312  0.00000000
```

compute aic = %nobs*log(%rss) + 2*(%nreg)
compute sbc = %nobs*log(%rss) + (%nreg)*log(%nobs)
dis 'The aic = ' aic ' and sbc = ' sbc

```
The aic =   -908.23350 and sbc =   -901.99752
```

Diagnostic checking of the residuals reveals no evidence of any significant autocorrelations. However, as correlation coefficients are measures of linear association, it is possible that money supply growth displays nonlinear adjustment. As such, we might want to estimate $dlm3_t$ as the following *LSTAR* model:

$$dlm3_t = \alpha_0 + \alpha_1 dlm3_{t-1} + \frac{\beta_0 + \beta_1 dlm3_{t-1}}{1 + \exp(-\gamma dlm3_{t-1})} + \varepsilon_t$$

Consider the following instructions:

```
nonlin a0 a1 b0 b1 gamma gamma.ge.0.
frml lstar dlm3 = a0 + a1*dlm3{1} + ( b0 + b1*dlm3{1})/(1+exp(-gamma*(dlm3{1})))
lin(noprint) dlm3 ; # constant dlm3{1}
com a0 = %beta(1), a1 = %beta(2), b0 = 1. , b1 = 1., gamma = 500.
```

The NONLIN instruction indicates that we want to estimate the five parameters *a0, a1, b0, b1* and *gamma*. Moreover, the value of *gamma* is constrained to be greater or equal to zero. The FRML instruction creates the desired formula and assigns it the name *lstar*. Next, we need the initial guesses. The LINREG instruction estimates $dlm3_t$ as an $AR(1)$ model. The COMPUTE instruction uses these estimates to obtain the initial guesses for *a0* and *a1*. The initial guesses for *b0* and *b1* were obtained by trial-and-error. The NLLS instruction below instructs RATS to estimate $dlm3_t$ using the formula named *lstar*.

nlls(frml=lstar) dlm3 / resids

```
Dependent Variable DLM3
Variable        Coeff       Std Error     T-Stat      Signif
***************************************************************
1. A0        -0.0026210    0.0031396   -0.83479  0.40506392
2. A1        -1.6729950    0.6362132   -2.62961  0.00937179
3. B0         0.0094409    0.0054145    1.74363  0.08312034
4. B1         2.3815524    0.6026591    3.95174  0.00011560
5. GAMMA    240.2373406   65.4646481    3.66973  0.00032915
```

```
compute aic = %nobs*log(%rss) + 2*(%nreg)
compute sbc = %nobs*log(%rss) + (%nreg)*log(%nobs)
```
dis 'The aic = ' aic ' and sbc = ' sbc

```
The aic =   -909.99076 and sbc =   -894.40079
```

The model appears satisfactory. The estimated values of *a1*, *b1* and *gamma* are significant at conventional levels and the estimated value of *b0* has a *prob*-value of `0.083`. If you examine the autocorrelations of the residuals using CORRELATE, you will find that they are all insignificant at conventional levels. Moreover, in a linear $AR(1)$ model, we require that the absolute value of the autoregressive coefficient be less than unity in absolute value. Here, the maximum and minimum values of $dlm3_t$ are -0.00538 and 0.03858, respectively. As such, the autoregressive coefficient $\alpha_1 + \beta_1/(1+\exp(-\gamma y_{t-1}))$ has a range of -0.90542 to 0.76053.

The *AIC* selects the *LSTAR* model while the *SBC* selects the linear $AR(1)$ model. Since *a0* is not significant at conventional levels, we can re-estimate the model without this intercept term. Hence, modify the first two lines of the estimation procedure so as to eliminate *a0*:

```
nonlin a1 b0 b1 gamma gamma.ge.0.
frml lstar dlm3 = a1*dlm3{1} + ( b0 + b1*dlm3{1})/(1+exp(-gamma*(dlm3{1})))
```

Now the NLLS estimation instruction yields:

nlls(frml=lstar) dlm3

```
  Variable       Coeff     Std Error      T-Stat   Signif
***************************************************************
1. A1        -1.3256892  0.5077631   -2.61084  0.00987429
2. B0         0.0053999  0.0014724    3.66727  0.00033150
3. B1         2.0864844  0.5009183    4.16532  0.00005029
4. GAMMA    256.7862707  61.7734749   4.15690  0.00005199
```

compute aic = %nobs*log(%rss) + 2*(%nreg)
compute sbc = %nobs*log(%rss) + (%nreg)*log(%nobs)
dis 'The aic = ' aic ' and sbc = ' sbc
The aic = -911.59844 and sbc = -899.12647

All of the coefficients are significant at conventional levels. As measured by the *AIC* and *SBC*, there is little to choose between the alternative *LSTAR* models. Moreover, the *SBC* continues to select the *AR*(1) model. As such, there is only mild evidence that $dlm3_t$ displays *LSTAR* adjustment.

5. Maximum Likelihood Estimation

Suppose you wanted to estimate:

$y_t = \beta x_t + \varepsilon_t$; $\varepsilon_t \sim N(0, \sigma^2)$

Of course, the most straightforward technique is to use OLS. However, you could obtain the maximum likelihood maximum likelihood estimate of β and σ^2 using the following instructions:

NONLIN b var
FRML l = -log(var) - (y - b*x)**2/var
COMPUTE b = initial guess, var = initial guess
MAXIMIZE L

Notice that the steps to perform maximum likelihood estimation are very similar to those for nonlinear least squares. To obtain maximum likelihood estimates:

Step 1. Specify the parameters to be estimated on a NONLIN instruction.

In the example above, the NONLIN instruction prepares RATS to estimate the parameters *b* and *var*. (Since RATS cannot process Greek characters, *b* and *var* are used to denote β and σ^2, respectively.)

Step 2. Define the likelihood (or support) for observation *t* using a FRML statement.

In the example above, if we are willing to assume that the values of $\{\varepsilon_t\}$ are assumed to be normally distributed random variables that are independent of each other, the log likelihood of observation *t* is:

$$\Lambda_t = -(1/2)\ln(2\pi) - (1/2)\ln\sigma^2 - \frac{1}{2\sigma^2}(y_t - \beta x_t)^2$$

The values of β and *var* that maximize $\sum_{t=1}^{T}\Lambda_t$ are identical to those maximizing:

$$-T\ln\sigma^2 - \frac{T}{\sigma^2}\sum_{t=1}^{T}(y_t - \beta x_t)^2$$

Hence, it is appropriate to use: FRML L = -log(var) - (y - b*x)**2/var

Step 3. Set the initial values of the parameters using the COMPUTE command.

Step 4. Use the MAXIMIZE instruction to maximize the formula defined in Step 2.

The MAXIMIZE instruction is the key to performing any maximum likelihood estimation. Suppose your data set contains T observations of the variables y_t and x_t and you have used the FRML instruction to define the function:

$L = f(y_t, x_t; \beta)$

where: x_t and β can be vectors (and x_t can represent a lagged value of y_t).

MAXIMIZE is able to find the value(s) of β that solve:

$$\max_{\beta} \sum_{t=1}^{T} f(y_t, x_t; \beta)$$

The syntax and principal options of MAXIMIZE are:

maximize(options) *frml start end funcval*

where:

frml	A previously defined formula
start end	The range of the series to use in the estimation
funcval	(Optional) The series name for the computed values of $f(y_t, x_t; \beta)$

The key options for our purposes are:

METHOD=	RATS is able to use any one of four different algorithms to find the maximum: BFGS, BHHH, GENETIC or SIMPLEX.
ITERATIONS=	The upper limit of the number of iterations used in the maximization.
ROBUSTERRORS [NOROBUST]	Computes a consistent estimate of the covariance matrix that corrects for heteroscadesticity.
CVCRIT =	Convergence limit [0.00001]. Also note that NLPAR affects the MAXIMIZE instruction.
TRACE/[NOTRACE]	Prints the intermediate results including the values of the estimated coefficients and function values. This is useful for tracking convergence problems.

MAXIMIZE produces a number of internal variables including %BETA, %TSTATS, %NOBS, %NREG, and %CONVERGED. In addition, the internal variable %FUNCVAL is equal to final value of the function being maximized.

Note: You can use TEST and RESTRICT with the BFGS and BHHH options. Coefficients are numbered by their position in the NONLIN statement.

Examples[10]

1. To estimate the model $y_t = \alpha x_t^{\beta} + \varepsilon_t$ use:

```
NONLIN a b var
FRML L = -log(var) - (y - a*x**beta)**2/var
COMPUTE a = guess, b = guess, var = guess
MAXIMIZE L
```

2. ***Maximum Likelihood Estimates of the LSTAR Model*:** In constructing your model, it is often helpful to define the log likelihood function using several FRML statements instead of one complicated expression. To illustrate the procedure, recall that in the previous section, we estimated $dlm3_t$ as *LSTAR* model:

$$dlm3_t = \alpha_0 + \alpha_1 dlm3_{t-1} + \frac{\beta_0 + \beta_1 dlm3_{t-1}}{1 + \exp(-\gamma dlm3_{t-1})} + \varepsilon_t$$

We can prepare to obtain the maximum likelihood estimates using the three instructions below. Notice that the NONLIN instruction is identical to the one we used for the NLLS estimation. As before, γ is restricted to be positive. However, there is no need to restrict *var*; the form of the log likelihood function is such that RATS will not find a negative value for *var*. The first FRML instruction creates the formula *expression*; note that *expression* is used in the second FRML instruction. In either case, you can obtain the desired formula for the log likelihood. However, breaking down a complicated formula into several smaller expressions is a useful way to prevent errors in your programs

```
nonlin a0 a1 b0 b1 gamma var gamma.ge.0.
frml expression = a0 + a1*dlm3{1} + (b0 + b1*dlm3{1})/(1+exp(-gamma*(dlm3{1})))
frml lstar = -log(var) - (dlm3 - expression)**2/var
```

Next, a linear regression is estimated. The estimated intercept and slope coefficients are used as the initial guesses for *a0* and *a1*. Unlike NLLS, we also need an initial guess for the estimated variance *var*. The LINREG instruction creates the internal variable %SEESQ equal to the standard error of estimate squared. This value is used as the initial guess for *var*. I had difficulty finding a solution using the BFGS method. As such, two different MAXIMIZE instructions are used. The first maximize instruction uses the SIMPLEX method to find the maximum likelihood estimates of *a0 a1 b0 b1 gamma var*. The second MAXIMIZE instruction uses these estimates as its initial guesses.

```
lin(noprint) dlm3 ; # constant dlm3{1}
com a0 = %beta(1), a1 = %beta(2), b0 = 1. , b1 = 1., gamma = 500. , var = %seesq
maximize(iters=20,method=simplex) lstar
maximize lstar
```

---

[10] All of the examples in the remainder of this chapter are in Program 1.4 of the file CHAPTER1.PRG.

```
MAXIMIZE - Estimation by Simplex
Quarterly Data From 1959:01 To 2001:01
Usable Observations    167
 Total Observations    169      Skipped/Missing        2
Function Value                    1603.41185675

   Variable                     Coeff
*****************************************
1.  A0                         0.00277148
2.  A1                         0.19562779
3.  B0                         0.00035734
4.  B1                         0.64739304
5.  GAMMA                    349.87711434
6.  VAR                        0.00002487

MAXIMIZE - Estimation by BFGS
Convergence in    36 Iterations. Final criterion was  0.0000000 <  0.0000100
Quarterly Data From 1959:01 To 2001:01
Usable Observations    167
 Total Observations    169      Skipped/Missing        2
Function Value                    1607.69576372

   Variable            Coeff       Std Error      T-Stat     Signif
*******************************************************************
1.  A0               -0.0026446    0.0032940     -0.80285  0.42206111
2.  A1               -1.6780205    0.6332808     -2.64973  0.00805571
3.  B0                0.0094796    0.0058559      1.61881  0.10548832
4.  B1                2.3860318    0.5841316      4.08475  0.00004412
```

The estimated coefficients and their associated *t*-statistics are very similar to those obtained from NLLS. Notice that we also have an estimate of the variance equal to `0.0000243`.

3. ***Maximum Likelihood Estimation of Moving Average Processes*****:** It is straightforward to use maximum likelihood estimation to estimate a model with unobserved components. Consider the simple *MA*(1) model:[11]

$$y_t = \varepsilon_t + \beta_1 \varepsilon_{t-1}$$

Since the $\{\varepsilon_t\}$ sequence is unobserved, it is not possible to use LINREG or NLLS to estimate the process. To estimate β_1 using maximum likelihood techniques, it is necessary to construct a formula of the form $\varepsilon_t = y_t - \beta_1 \varepsilon_{t-1}$. However, the following is an **illegal** statement because e_t is defined in terms of its own lagged value (a nonresolvable recursive expression):

```
frml e = y – b1*e{1}
```

[11] The RATS instruction BOXJENK is the most useful way to estimate standard *ARMA*(*p*, *q*) models. The aim of this example is to illustrate the use of a SUBFORMULA on the FRML instruction.

The way to circumvent this problem is to create a "placeholder" series using the SET instruction. Then, define the desired formula in terms of the placeholder series. Finally, use a SUBFORMULA to equate the placeholder and the desired series. For example, a simple way to create the formula for the *MA*(1) process is:

```
set temp = 0.0
nonlin b1 var
frml e = y – b1*temp{1}
frml L = (temp = e), -log(var) - e**2/var
```

The SET instruction generates the placeholder series *temp* containing all zeros. The first FRML instruction defines the desired relationship such that e_t is equal to y_t – b_1*temp{1}*. The second FRML statement uses the SUBFORMULA to equate *temp* with *e* (so that $e_t = y_t - b_1e_{t-1}$) and creates the log likelihood *L*. The way to conceptualize the process is to suppose you knew β and *var* and wanted to construct the log likelihood function:

$$\Lambda_t = -\sum_{t=2}^{T}\left[\log(\text{var}) + (y_t - \beta_1\varepsilon_{t-1})^2 / \text{var}\right]$$

One way to construct the sum would be to loop over the following three instructions beginning with $t = 2$:

$\varepsilon_t = y_t - \beta_1 temp_{t-1}$
$temp_t = \varepsilon_t$
$L_t = -\log(var) - (\varepsilon_t)^2/var$

The first time through the loop, $t = 2$ so that $\varepsilon_2 = y_2 - \beta_1 temp_1$, $temp_2 = \varepsilon_2$ and $L_2 = -\log(var) - (\varepsilon_2)^2/var$. The next time through the loop, $t = 3$ and $\varepsilon_3 = y_3 - \beta_1 temp_2 = y_3 - \beta_1\varepsilon_2$. Hence, $temp_3 = \varepsilon_3$ and $L_3 = -\log(var) - (\varepsilon_3)^2/var$. Continuing through $t = T$, yields the values L_2, L_3 , … L_T. The sum of these values yields the desired log likelihood function Λ_t.

To illustrate the procedure, recall that the change in the 3-month interest rate (*drs*) was estimated as an *AR*(7) process. It turns out that a more parsimonious representation of the series is:

$drs_t = \alpha_1 drs_{t-1} + \alpha_7 drs_{t-7} + \varepsilon_t + \beta_1\varepsilon_{t-1}$

This *ARMA* model can be estimated using the six instructions listed below. The SET instruction is used to create the series *temp* containing all zeros. The NONLIN instruction prepares RATS to estimate *a1, a7,* b_1 and *var*. The first FRML instruction creates the desired formula such that e_t is equal to $drs_t - a_1drs_{t-1} - a_7drs_{t-7} - b_1temp_{t-1}$. The second FRML statement uses a SUBFORMULA to equate t*emp* with *e* (so that the 'next time through the loop' $temp_t = e_t$) and creates the formula $L = -\log(var) - e^2/var$. When you use a placeholder, you need to properly specify the *start end* dates on the MAXIMIZE instruction. In the case at hand, one usable observation is lost as a result of differencing and another seven usable

observations are lost as a result of the term drs_{t-7}. Since eight usable observations are lost, the maximization must begin in period 9. If you omit the *start end* dates, or begin with a starting date less than 9, you will obtain the error message:

SR10. Missing Values And/Or SMPL Options Leave No Usable Data Points

```
set temp = 0.
nonlin a1 a7 b1 var
frml e = drs - a1*drs{1} - a7*drs{7} – b1*temp{1}
frml L = (temp=e), -log(var) - (e)**2/var
com a1 = 0.4, a7 = -.3, b1 = .5, var = 1.
max L 9 *
```

```
MAXIMIZE - Estimation by BFGS
Convergence in    14 Iterations. Final criterion was  0.0000025 <  0.0000100
Quarterly Data From 1961:01 To 2001:01
Usable Observations    161
Function Value                      -46.71000012

   Variable            Coeff       Std Error      T-Stat     Signif
*******************************************************************
1.  A1             -0.337729340  0.071464518     -4.72583  0.00000229
2.  A7             -0.352966296  0.051912480     -6.79926  0.00000000
3.  B1              0.817502905  0.044594888     18.33176  0.00000000
4.  VAR             0.491705037  0.038656437     12.71987  0.00000000
```

The same technique is used for estimating a higher-order $MA(q)$ process. Suppose you wanted to estimate the spread as:

$$drs_t = \alpha_1 drs_{t-1} + \alpha_7 drs_{t-7} + \varepsilon_t + \beta_1\varepsilon_{t-1} + \beta_2\varepsilon_{t-2}$$

Now, the NONLIN instruction contains the coefficient *b2*. The first FRML instruction uses *temp*{1} and *temp*{2} as placeholders for e_{t-1} and e_{t-2}. The second FRML instruction creates the desired log likelihood and the COMPUTE instruction provides the initial guesses. Notice that the *start* date can remain at 9 since no usable observations are lost from the *MA* terms.

```
set temp = 0.
nonlin a1 a7 b1 b2 var
frml e = drs - a1*drs{1} - a7*drs{7} - b1*temp{1} - b2*temp{2}
frml L = (temp=e), -log(var) - (e)**2/var
com a1 = 0.4, a7 = -.3, b1 = .5, b2 = 0.3, var = 1.
max L 9 *
```

4. ***A Bilinear*(1,1) *Model of the Money Supply*:** Given that $dlm3_t$ seems to exhibit nonlinear behavior, it might be desirable to estimate an alternative nonlinear specification. The bilinear model generalizes the standard $ARMA(p, q)$ model by allowing for an interaction among the *AR* and *MA* terms. Consider the following bilinear(1,1) specification for $dlm3_t$:

$dlm3_t = \alpha_1 dlm3_{t-1} + \varepsilon_t + \beta_1\varepsilon_{t-1} + c_1 dlm3_{t-1}\varepsilon_{t-1}$

The bilinear specification is a way to allow for nonlinear adjustment. In a period with $\varepsilon_{t-1} = 0$, the autoregressive coefficient is α_1 and the moving average coefficient is β_1. However, the presence of the interaction term $c_1 dlm3_{t-1}\varepsilon_{t-1}$ means that the degree of autoregressive decay and the coefficients of the moving average will change over time. To estimate the bilinear model, you can use:

```
set temp = 0.
nonlin a1 b1 c1 var
frml e = dlm3 - a1*dlm3{1} - b1*temp{1} - c1*dlm3{1}*temp{1}
frml L = (temp=e), -log(var) - (e)**2/var
```

Since the bilinear specification contains an *MA* term, it is necessary to use the placeholder *temp*. The NONLIN instruction prepares RATS to estimate the four parameters *a1, b1, c1* and *var*. The first FRML instruction creates the formula e_t as $dlm3_t - \alpha_1 dlm3_{t-1} - \beta_1 e_{t-1} - c_1 dlm3_{t-1} e_{t-1}$. The second FRML instruction uses a SUBFORMULA to equate $temp_t$ and e_t and to create the log likelihood function. The following COMPUTE instruction provides the initial guesses. For this example, the BFGS method does not work well unless the initial guesses are quite good. As such, the initial guesses are refined using the SIMPLEX method and the final estimates are reported using the default BFGS method. Notice that the maximization begins with observation three (one usable observation is lost as a result of differencing and another is lost as a result to the term $dlm3_{t-1}$).

```
com a1 = .8, b1 = 0., c1 = 0.1, var = 0.01
max(method=simplex,iters=4) L 3 *
max L 3 *
```

```
MAXIMIZE - Estimation by BFGS
Convergence in    32 Iterations. Final criterion was  0.0000095 <  0.0000100
Quarterly Data From 1959:03 To 2001:01
Usable Observations    167
Function Value                     1590.80067422

   Variable           Coeff       Std Error      T-Stat     Signif
*******************************************************************
1.  A1             0.976977877  0.013236303     73.81048  0.00000000
2.  B1            -0.273500979  0.134326080     -2.03610  0.04174058
3.  C1             9.121528605  5.860399296      1.55647  0.11959666
4.  VAR            0.000026837  0.000002077     12.92295  0.00000000
```

You can see that the bilinear coefficient *c1* is not significant at conventional levels. As such, it does ***not*** appear that the bilateral model is a satisfactory representation of the $\{dlm3_t\}$ sequence. One word of caution is in order since it appears that β_1 is significant at the 5% level. However, it does not follow that $dlm3_t$ follows an ARMA(1,1) process. Since $dlm3_{t-1}\varepsilon_{t-1}$ is correlated with ε_{t-1} the individual *t*-statistics can be misleading. In fact, if you eliminate the

bilinear coefficient c_1 and estimate $dlm3_t$ as a pure *ARMA*(1,1) model, the *MA*(1) coefficient is insignificant. To illustrate the point, consider:

```
set temp = 0.
nonlin a1 b1 var
frml e = dlm3 - a1*dlm3{1} - b1*temp{1}
frml L = (temp=e), -log(var) - (e)**2/var
com a1 = .8, b1 = 0., var = 0.01
max(method=simplex,iters=4) L 3 *
max L 3 *
```

```
   Variable            Coeff       Std Error      T-Stat     Signif
*******************************************************************
1.  A1                  0.9799       0.0125       78.12362  0.00000000
2.  B1                 -0.0965       0.0653       -1.47713  0.13964074
3.  VAR                 2.7033e-05   2.0527e-06   13.16948  0.00000000
```

6. GARCH Models

Suppose you want to estimate a simple regression model with an *ARCH*(1) error process:

$$y_t = \beta_0 + \beta_1 x_t + \varepsilon_t$$

where: $\varepsilon_t = v_t\sqrt{\alpha_0 + \alpha_1\varepsilon_{t-1}^2}$ and $v_t \sim$ WN(0, 1).

Since v_t is white-noise, $E_{t-1}\varepsilon_t = 0$ and $E_{t-1}\varepsilon_t^2 \equiv h_t = \alpha_0 + \alpha_1\varepsilon_{t-1}^2$. Hence, the desired formula for the log likelihood of y_t can be written in the form:

$$-\log(h_t) - \log(\varepsilon_t^2)/h_t$$

The autocorrelation function of the residuals is not satisfactory for detecting *ARCH* errors. Correlations measure linear association and *ARCH* errors manifest themselves in the autocorrelations of the squared residuals. The Lagrange Multiplier (*LM*) test for *ARCH* disturbances has been proposed by Engle (1982). After you have estimated the most appropriate model for y_t, save the residuals. Suppose you have estimated the model:

```
lin y / resids ; # constant x
```

Then obtain the square of the residuals and regress these squared residuals on a constant and on *n* lagged values of the squared residuals. For example, if $n = 4$:

```
set r2 = resids**2
lin r2
# constant r2{1 to 4}
```

If there are no *ARCH* or *GARCH* effects, this regression will have little explanatory power so that the coefficient of determination (i.e., the usual R^2-statistic) will be quite low. With a sample of T residuals, under the null hypothesis of no ARCH errors, the test statistic TR^2 converges to a χ^2 distribution with n degrees of freedom. If TR^2 is sufficiently large, rejection of the null hypothesis is equivalent to rejecting the null hypothesis of no GARCH errors. On the other hand, if TR^2 is sufficiently low, it is possible to conclude that there are no ARCH effects.

Since LINREG creates the internal variables %NOBS (*i.e.*, T) and %RSQUARED (R^2), you can easily compute TR^2 and the significance of *trsq* as χ^2 with 4 degrees of freedom with:

```
compute trsq = %nobs*%rsquared
cdf chisqr trsq 4
```

Here, the CDF instruction calculates the marginal significance of *trsq* using a χ^2 distribution with 4 degrees of freedom. The syntax for CDF is:

CDF *distribution statistic degree1 degree2*

where:

distribution	The desired F, t, χ^2 or normal distribution is selected using: FTEST, TTEST, CHISQ, or NORMAL.
statistic	The value of the test statistic.
degree1	Degrees of freedom for TTEST and CHISQ or numerator degrees of freedom for FTEST.
degree2	Denominator degrees of freedom for FTEST.

The six instructions below can be used to estimate a regression with *ARCH*(1) errors. The NONLIN instruction indicates that the four parameters *a0, a1, b0* and *b1* are to be estimated. The three FRML instructions create the appropriate log likelihood. The first FRML instruction defines e_t as y_t - $b0$ - $b1x_t$. The second defines the conditional variance h as an *ARCH*(1) process. The third uses the definitions of e and h to define the log likelihood function L. The MAXIMIZE command instructs RATS to find the maximum likelihood estimates of *a0, a1, b0* and *b1*.

```
nonlin a0 a1 b0 b1
frml e = y – b0 - b1*x
frml h = a0 + a1*e(t-1)**2
frml L = - log(h) - log(e**2)/h
com initial guesses
max L 2 *
```

6.1 Examples of GARCH Processes

1. ***An ARCH Model of the Spread*:** If you continue to enter the instructions on Program 1.4, you can form the difference between the 1-year rate and the 3-month rate as:

```
set spread = tb1yr - tb3mo
```

It appears that an *AR*(3) model of the spread is quite reasonable. Consider:

lin spread / resids; # constant spread{1 to 3}

```
   Variable            Coeff       Std Error      T-Stat     Signif
*************************************************************************
1.  Constant         0.047619130  0.024652597      1.93161  0.05517527
2.  SPREAD{1}        0.890042258  0.078014946     11.40861  0.00000000
3.  SPREAD{2}       -0.318602488  0.101791701     -3.12995  0.00207856
4.  SPREAD{3}        0.161545395  0.077536269      2.08348  0.03879731
```

The individual autocorrelations and Ljung-Box Q-statistics of the residuals indicate that there is no serial correlation in the residual series. The first twelve autocorrelations and the associated Q-statistics can be obtained from:

cor(number=12,span=4,qstats) resids

```
Correlations of Series RESIDS
Quarterly Data From 1959:03 To 2001:01
Autocorrelations
     1:  0.0177090 -0.0072871  0.0877679 -0.0372146  0.0197141 -0.1939013
     7: -0.0416785  0.0473398 -0.0010301 -0.0979544 -0.1078251  0.0908990

Ljung-Box Q-Statistics
Q(4)   =          1.6279.  Significance Level 0.80377726
Q(8)   =          8.9906.  Significance Level 0.34309102
Q(12)  =         14.3235.  Significance Level 0.28052925
```

Nevertheless, *ARCH* errors manifest themselves in the autocorrelations of the squared residuals. You can form the squared residuals and perform the Lagrange multiplier test using:

set r2 = resids**2
lin r2 ; # constant r2{1 to 3}

```
   Variable            Coeff         Std Error       T-Stat    Signif
*********************************************************************
1.  Constant       0.046242482    0.018049071      2.56204  0.01132890
2.  R2{1}          0.217038776    0.078030863      2.78145  0.00606210
3.  R2{2}         -0.025838287    0.079861347     -0.32354  0.74670932
4.  R2{3}          0.159631091    0.077999507      2.04657  0.04233716
```

To test the restriction that the coefficients for the 3-lagged values of *r2* all equal zero, use:

compute trsq = %nobs*%rsquared
cdf chisqr trsq 3

```
Chi-Squared(3)=     11.981860 with Significance Level 0.00744556
```

Since we reject the null hypothesis of no *ARCH* errors, we can try to estimate the spread using the following specification:

$$spread_t = \alpha_0 + \alpha_1 spread_{t-1} + \alpha_2 spread_{t-2} + \alpha_3 spread_{t-3} + \varepsilon_t$$

$$\varepsilon_t = v_t\sqrt{b_0 + b_1\varepsilon_{t-1}^2 + b_2\varepsilon_{t-2}^2 + b_3\varepsilon_{t-3}^2}$$

and $v_t \sim WN(0, 1)$.

As such $E_{t-1}\varepsilon_t = 0$ and $E_{t-1}\varepsilon_t^2 \equiv h_t = b_0 + b_1\varepsilon_{t-1}^2 + b_2\varepsilon_{t-2}^2 + b_3\varepsilon_{t-3}^2$. The NONLIN instruction prepares RATS to estimate the six parameters a0, a1, a2, a3, b0, b1, b2, and b3. Since the coefficients in h_t cannot be negative, b0, b1, b2, and b3 are constrained to be non-negative.[12] The first FRML creates e_t as $spread_t$ - a0 - a1*$spread_{t-1}$ - a2*$spread_{t-2}$ - a3*$spread_{t-3}$. The

[12] If any of these coefficients is zero, it becomes possible to estimate a negative value for the conditional variance.

second FRML creates the ARCH(3) model for the conditional variance and the third creates the log likelihood.

```
nonlin  a0 a1 a2 a3 b0 b1 b2 b3 b0.ge.0. b1.ge.0. b2.ge.0. b3.ge.0.
frml e = spread - a0 - a1*spread{1} - a2*spread{2} - a3*spread{3}
frml h = b0 + b1*e{1}**2 + b2*e{2}**2 + b3*e{3}**2
frml L =  -log(h) - (e)**2/h
```

A linear regression (without ARCH errors) is used to obtain the initial guesses for a0, a1, a2, a3 and b0. The initial guesses are refined using the SIMPLEX method and the final estimates are reported using the default BFGS method. Notice that we do not need to specify the start end dates here. The maximization begins with observation nine (three usable observations are lost as a result of the term $spread_{t-3}$, another three are lost as a result of the term e_{t-3}, and two are missing since tb1yr begins in period 3.).

```
lin(noprint) spread ; # constant spread{1 to 3}
com a0 = %beta(1), a1 = %beta(2), a2 = %beta(3), a3 = %beta(4) , $
b0 = %seesq, b1 = 0.2,  b2 = 0.2, b3 = 0.2
max(method=simplex,iters=4) L
max(iters=200) L
```

```
MAXIMIZE - Estimation by BFGS
Convergence in    26 Iterations. Final criterion was  0.0000058 <  0.0000100
Quarterly Data From 1959:01 To 2001:01
Usable Observations    161
 Total Observations    169      Skipped/Missing        8
Function Value                      317.68938352

   Variable          Coeff       Std Error      T-Stat     Signif
*******************************************************************
1.  A0            0.057038155  0.014792370      3.85592  0.00011530
2.  A1            0.793283630  0.066602420     11.91073  0.00000000
3.  A2           -0.144647372  0.085452808     -1.69272  0.09050946
4.  A3            0.100323333  0.052534251      1.90967  0.05617510
5.  B0            0.022879390  0.003589145      6.37461  0.00000000
6.  B1            0.203326143  0.119328231      1.70392  0.08839543
7.  B2            0.102314211  0.081292747      1.25859  0.20817858
8.  B3            0.446546881  0.112246500      3.97827  0.00006942
```

2. ***The ARCH-M model*****:** Engle, Lilien, and Robbins (1987) developed the *ARCH* in Mean (*ARCH-M*) model to allow the conditional variance of an asset's return to affect the expected return. The idea is that an increase in risk (as measured by an increase in the conditional volatility of the asset's returns) should increase the expected reward for holding the asset. If r_t is the one-period excess return from holding the asset and h_t the conditional volatility, they estimate a model in the form:

$$r_t = \beta_0 + \beta_1 h_t + \varepsilon_t$$

$$h_t = \alpha_0 + \alpha_1(0.4\,\varepsilon^2_{t-1} + 0.3\,\varepsilon^2_{t-2} + 0.2\,\varepsilon^2_{t-3} + 0.1\,\varepsilon^2_{t-4})$$

The appropriate FRML instructions to estimate this model are:

```
frml e = r - b0 - b1*h
frml h = α0 + α1*(0.4*ε{1}**2 + 0.3*ε{2}**2 + 0.2*ε{3}**2 + 0.1*ε{4}**2)
```

The first statement defines ε_t as r_t - β_0 - $\beta_1 h_t$. If β_1 is positive, increases in risk (as measured by h_t) increase the expected return. The second statement defines the conditional variance.

3. ***An IGARCH Model of the Spread*****:** The specification for the $spread_t$ used above required four coefficients to estimate the conditional variance h_t. A more parsimonious specification is the *GARCH*(1, 1) model:

$$spread_t = \alpha_0 + \alpha_1 spread_{t-1} + \alpha_2 spread_{t-2} + \alpha_3 spread_{t-3} + \varepsilon_t$$

$$h_t = b_0 + b_1\varepsilon^2_{t-1} + c_1 h_{t-1}$$

Notice that it is not possible to define h_t using a FRML statement. As in the case of a MA model, the following is an **illegal** statement because h_t is defined in terms of its own lagged value:

```
frml h = b0 + b1*e{1}**2 + c1*h{1}
```

The appropriate solution is to use a placeholder for h{1}. As such, the SET statement initializes the series temp to be zero. NONLIN prepares RATS to estimate the parameters a0, a1, a2, a3, b0, b1, and c1. The NONLIN instruction restricts b0, b1 and c1 to be positive. The first FRML instruction creates e_t as the AR(3) model of the spread. The second FRML instruction creates the conditional variance as: h_t = b0 + b1*e{1}**2 + c1*temp{1}. The third FRML statement uses a SUBFORMULA to equate $temp_t$ with h_t and to define the log likelihood L_t.

```
set temp = 0.
nonlin  a0 a1 a2 a3 b0 b1 c1 b0.ge.0. b1.ge.0. c1.ge.0.
frml e = spread - a0 - a1*spread{1} - a2*spread{2} - a3*spread{3}
frml h = b0 + b1*e{1}**2 + c1*temp{1}
frml L = (temp = h), -log(temp) - (e)**2/temp
```

After initializing the parameters with the LINREG and COMPUTE instructions, the first MAXIMIZE instruction refines the initial guesses with the SIMPLEX method. The second obtains the final estimates using to BFGS method:

```
lin(noprint) spread ; # constant spread{1 to 3}
com a0 = %beta(1), a1 = %beta(2) , a2 = %beta(3), a3 = %beta(4) , b0 = %seesq, $
b1 = 0.2,  c1 = 0.5
max(method=simplex,iters=5) L 7 *
```

max(iters=200) L 7 *

```
MAXIMIZE - Estimation by BFGS
Convergence in    26 Iterations. Final criterion was  0.0000017 <  0.0000100
Quarterly Data From 1960:03 To 2001:01
Usable Observations    163
Function Value                     332.16269201

   Variable           Coeff       Std Error      T-Stat     Signif
*********************************************************************
1.  A0             0.047084962  0.013313562      3.53662  0.00040529
2.  A1             0.881755754  0.055770333     15.81048  0.00000000
3.  A2            -0.242706983  0.075485128     -3.21530  0.00130310
4.  A3             0.133517900  0.055961728      2.38588  0.01703837
5.  B0             0.001382754  0.000804149      1.71952  0.08551898
6.  B1             0.204112593  0.040165933      5.08173  0.00000037
7.  C1             0.806864043  0.028403811     28.40689  0.00000000
```

Notice that the estimated values of *b1* and *c1* are such that their sum exceeds unity. It is possible to estimate an *IGARCH*(1,1) model by restricting $b1 + c1 = 1$. The only modification needed is to replace the NONLIN instruction above with:

nonlin a0 a1 a2 a3 b0 b1 c1 b0.ge.0. b1.ge.0. c1.ge.0. b1+c1.eq.1.

If you re-estimate the model you will find:

max(iters=200) L 7 *

```
MAXIMIZE - Estimation by BFGS
Convergence in    25 Iterations. Final criterion was  0.0000027 <  0.0000100
Quarterly Data From 1960:03 To 2001:01
Usable Observations    163
Function Value                     332.09657868

   Variable         Coeff       Std Error      T-Stat     Signif
*******************************************************************
1.  A0           0.047285026  0.013536155      3.49324  0.00047720
2.  A1           0.880737529  0.056049098     15.71368  0.00000000
3.  A2          -0.243572528  0.075258624     -3.23647  0.00121017
4.  A3           0.136373691  0.053798145      2.53491  0.01124748
5.  B0           0.001590978  0.000591367      2.69034  0.00713794
6.  B1           0.193993102  0.029293592      6.62237  0.00000000
7.  C1           0.806006898  0.029293592     27.51479  0.00000000
```

4. ***An ARMA*(1,1)-*IGARCH*(1,1) *Model of the Spread*:** As a final example for this section, suppose you want to estimate the *spread* as an *ARMA*(1,1) model with *IGARCH*(1,1) errors:

$$spread_t = \alpha_0 + \alpha_1 spread_{t-1} + \varepsilon_t + \beta\varepsilon_{t-1}$$

$$h_t = b_0 + b_1\varepsilon_{t-1}^2 + c_1 h_{t-1}$$

Now it is necessary to use two placeholders; one for e_{t-1} and another for h_{t-1}. In the program below, temp1 is the placeholder in the equation for ε_t = spread$_t$ - α_0 - $\alpha_1$spread$_{t-1}$ - $\beta\varepsilon_{t-1}$. As such, the first FRML instruction creates e_t as spread$_t$ - a0 - a1*spread{1} - beta*temp1{1}. Similarly, the second FRML instruction uses the placeholder temp2 to create the conditional variance as h = b0 + b1*e{1}**2 + c1*temp2{1}. The third FRML instruction uses two SUBFORMULAS; the first equates temp1$_t$ with e_t and the second equates temp2$_t$ with h_t.

```
set temp1 = 0.  ; set temp2 = 0.
nonlin  a0 a1 beta b0 b1 c1 b0.ge.0. b1.ge.0. c1.ge.0.  b1+c1 == 1.
frml e = spread - a0 - a1*spread{1} - beta*temp1{1}
frml h = b0 + b1*e{1}**2 + c1*temp2{1}
frml L = (temp1 = e), (temp2 = h), -log(temp2) - (temp1)**2/h
```

The next two instructions are used to initialize the parameters. For now, it is sufficient to note that the BOX(constant,ar=1,ma=1,noprint) spread instruction estimates an ARMA(1,1) model of the spread without GARCH errors. The initial guesses for a0, a1, beta and b0 are taken from this ARMA(1,1) model. Finally, the MAXIMIZE instructions are used to obtain the maximum likelihood estimates.

```
box(constant,ar=1,ma=1,noprint) spread
com a0 = %beta(1), a1 = %beta(2) , beta = %beta(3),  b0 = %seesq, b1 = 0.2,  c1 = 0.5
max(method=simplex,iters=5) L 7 *
max(iters=200) L  7 *
```

```
MAXIMIZE - Estimation by BFGS
Convergence in    31 Iterations. Final criterion was  0.0000050 <  0.0000100
Quarterly Data From 1960:03 To 2001:01
Usable Observations    163
Function Value                     330.04737868

   Variable          Coeff        Std Error      T-Stat     Signif
*******************************************************************
1.  A0           0.0673437939 0.0205030899      3.28457  0.00102139
2.  A1           0.6930927974 0.0663996430     10.43820  0.00000000
3.  BETA         0.2087641734 0.0987429689      2.11422  0.03449664
4.  B0           0.0015160176 0.0006060467      2.50149  0.01236732
5.  B1           0.1927189124 0.0318551475      6.04985  0.00000000
6.  C1           0.8072810876 0.0318551475     25.34225  0.00000000
```

Chapter 2:
VARs and Error-Correction Models

A vector autoregression (VAR) is a multivariate generalization of the single-equation autoregressive model. In the two-variable case, we can let the time path of the $\{y_t\}$ be affected by current and past realizations of the $\{z_t\}$ sequence **and** let the time path of the $\{z_t\}$ sequence be affected by current and past realizations of the $\{y_t\}$ sequence. Consider the following 2-variable 1-lag VAR in standard form:

$$y_t = a_{10} + a_{11}y_{t-1} + a_{12}z_{t-1} + e_{1t}$$

$$z_t = a_{20} + a_{21}y_{t-1} + a_{22}z_{t-1} + e_{2t}$$

It is assumed that e_{1t} and e_{2t} are serially uncorrelated but the covariance $Ee_{t1}e_{2t}$ need not be zero. If the variances and covariance are time-invariant, we can write the variance/covariance matrix as:

$$\Sigma = \begin{bmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{21} & \sigma_{22} \end{bmatrix}$$

where: $\text{Var}(e_{it}) = \sigma_{ii}$ and $\text{Cov}(e_{1t},e_{2t}) = \sigma_{12} = \sigma_{21}$.

Note that the right-hand-sides of the VAR equations contain only pre-determined variables. Since the error terms are serially uncorrelated with constant variances, *each equation in the system can be estimated using OLS*. Moreover, OLS estimates are consistent and asymptotically efficient. Even though the errors are correlated across equations, estimation using seemingly unrelated regressions (SUR) does not add to the efficiency of the estimation procedure since both regressions have identical right-hand-side variables.

If one or more of the equations is constrained so as to have different right-hand-side variables than the others (including the possibility of differing lag lengths), the system is called a **near-VAR**. A near-VAR can be estimated using RATS SUR instruction. In this case, SUR improves the efficiency of the estimates.

Preparing RATS to perform a VAR analysis consists of the following two steps:[13]

Step 1: After making the necessary data transformations, you must define the equations to use in the VAR. Typically, you will use the following five instructions to set up a VAR:

[13] Don't let the saying 'You can't teach an old dog new tricks' apply to you. Users of RATS 4.3 and earlier will find that all of their VAR programs are compatible with version 5.0. However, to take advantage of some of the new features in RATS, you must use the MODEL=*modelname* OPTION with the SYSTEM instruction.

```
SYSTEM(MODEL=modelname)
VARIABLES list of dependent variables
LAGS 1 to lag length
DETERMINISTIC list of deterministic (constant, seasonals) and exogenous variables
END(SYSTEM)
```

Step 2: You instruct RATS to estimate the system using ESTIMATE. The typical form of the ESTIMATE instruction is:

ESTIMATE(OUTSIGMA=*V*,residuals=*resids*,other options) *start end*

For an *n*-equation VAR, the OPTION RESIDUALS = *resids* creates *n* series of residuals. The residuals from the first equation are stored in the series called *resids (1),* the residuals from the second equation are stored in the series called *resids(2),* and so forth.

Other Options:

OUTSIGMA=*matrix*	Computes and saves the covariance matrix of the residuals.
NOPRINT	Suppresses printing of the OLS estimation of each equation.
NOFTESTS	Suppresses printing the results of all Granger causality tests.
SIGMA	Displays (but does not save) the covariance matrix of the residuals. Use both OUTSIGMA= and SIGMA if you want to compute, save, and print the variance/covariance matrix.
COEFFICIENTS=*coef*	Creates a matrix of the coefficients. Column *i* contains the coefficients of the *i-th* equation.

Note that RATS 5.0 recognizes the older form of the ESTIMATE instruction. As such, you can still use:

ESTIMATE(OUTSIGMA=*V*, *other options*) *start end residuals*

where: *residuals* is the first series in a block of series used to store the residuals.

Examples:

It is straightforward to set up and estimate the following 2-variable 1-lagVAR as a MODEL called *example1*:

$$y_t = a_{10} + a_{11}y_{t-1} + a_{12}z_{t-1} + e_{1t}$$

$$z_t = a_{20} + a_{21}y_{t-1} + a_{22}z_{t-1} + e_{2t}$$

The first instruction below prepares RATS to create a system of equations with the name *example1*. The VARIABLES instruction names the two variables in the system and the LAGS instruction indicates that one lag of each is to be included in the model. The DETERMINISTIC instruction informs RATS to include a constant term in each regression equation. END closes the system and ESTIMATE produces the coefficient estimates and the *F*-statistics for the Granger-causality tests. The option RESIDUALS=*resids* instructs RATS to save the residuals from the y_t

equation in a series called *resids(1)* and the residuals from the z_t equation in a series called *resids(2)*.

```
system(model=example1)
var y z
lags 1
det constant
end(system)
estimate(residuals=resids)
```

Modification of the SYSTEM-END(SYSTEM) block is straightforward. If you want to:

1. Include 4 lags of y_t and z_t in each equation, replace the LAGS instruction with:

```
lags 1 to 4
```

2. Include lags 1, 2, 3, 4, and 8 of y_t and z_t in each equation, replace the LAGS instruction with:

```
lags 1 to 4  8
```

3. Include an exogenous variable w such that the VAR is:

$$y_t = a_{10} + a_{11}y_{t-1} + a_{12}z_{t-1} + a_{13}w_t + a_{14}w_{t-1} + e_{1t}$$
$$z_t = a_{20} + a_{21}y_{t-1} + a_{22}z_{t-1} + a_{23}w_t + a_{24}w_{t-1} + e_{2t}$$

replace line 4 with:

```
det constant w{0 to 1}
```

4. Estimate a 3 variable VAR using y, z and w, replace the VARIABLES instructions with:

```
var y z w
```

5. Include quarterly seasonal dummy variables so that the system becomes:

$$y_t = a_{10} + b_{11}D_1 + b_{12}D_2 + b_{13}D_3 + a_{11}y_{t-1} + a_{12}z_{t-1} + e_{1t}$$

$$z_t = a_{20} + b_{21}D_1 + b_{22}D_2 + b_{23}D_3 + a_{21}y_{t-1} + a_{22}z_{t-1} + e_{2t}$$

First, create the seasonal dummy variables using:

```
seasonal dummy
```

Next, modify the DET instruction such that:

```
det constant dummy{-1 to –3}
```

In some instances, you might want to create centered seasonal dummy variables. Centered seasonal dummies are normalized to have a mean of zero. For example, instead of taking on the values 0, 0 , 0 and 1, a centered seasonal dummy variable for quarterly data has the values -0.25, -0.25, -0.25 and 0.75. Centered seasonal dummy variables are useful in situations, such as unit root tests, where you do not want to shift the magnitude of the intercept term. To estimate the VAR with centered seasonal dummy variables use:

```
seasonal(centered) dummy
det constant dummy{-1 to –3}
```

1. Hypothesis Testing and Model Selection

Most hypothesis tests in a VAR involve cross-equation restrictions. The RATIO instruction can easily perform such tests. Let Σ_u and Σ_r be the variance/covariance matrices of the unrestricted and restricted systems, respectively. Form the test statistic L:

$$L = (T\text{-}c)(\log |\Sigma_r|) - \log |\Sigma_u|)$$

where: $|\Sigma_r|$ and $|\Sigma_u|$ are the determinants of Σ_u and Σ_r, c is the maximum number of regressors contained in the longest equation of either VAR system and T is the number of usable observations.

L can be compared to a χ^2 distribution with degrees of freedom equal to the number of restrictions in the system. If L exceeds this critical value, reject the null hypothesis that the restriction is not binding (*i.e.*, conclude that the restriction is binding). The usual form of the RATIO instruction is:

ratio(*degrees*=df, *mcorr*=c, other options) *start end*
\# *series containing the residuals from the unrestricted system*
\# *series containing the residuals from the restricted system*

The first supplemental card lists the series containing the residuals from the unrestricted system and the second supplemental card lists the series containing the residuals from the restricted system. RATS uses these lists to construct $|\Sigma_r|$ and $|\Sigma_u|$.

where:

start end	The range over which the test is to be performed.
degrees=dfc	The number of degrees of freedom (equal to the number of restrictions in the system).
mcorr=c	Sims' small sample correction for likelihood ratio tests (*i.e.*, the value of *c*). Set *mcorr* equal to the largest number of parameters estimated in any one of the equations (usually equal to the number of parameters estimated in each of the unrestricted equations).

The other principal option, NOPRINT, suppresses the printing of the covariance matrices and the marginal significance level of the test. It is possible to obtain the marginal significance level with the instruction:

```
display  %signif
```

The likelihood ratio test is based on asymptotic theory that may not be very useful in the small samples available to time-series econometricians. Moreover, the likelihood ratio test is only

applicable when one model is a restricted version of the other. Alternative test criteria are the multivariate generalizations of the *AIC* and *SBC*:

$AIC = T \log|\Sigma| + 2N$
$SBC = T \log|\Sigma| + N \log(T)$

where $|\Sigma|$ = determinant of the variance/covariance matrix of the residuals and N = total number of parameters estimated *in all equations*. Thus, if each equation in an n-variable VAR has p lags and an intercept, $N = n^2p+n$ (each of the n equations has np lagged regressors and an intercept).

Note that for a VAR, ESTIMATE creates the following variables:

%NOBS	Number of usable observations
%LOGDET	Log determinant of the estimate of Σ
%SIGMA	Covariance matrix of residuals

When you use the OUTSIGMA= option on the ESTIMATE statement, RATS computes the covariance matrix of the residuals. You can fetch the logarithmic determinant of this covariance matrix using %LOGDET. The following three statements will compute and display the multivariate versions of the *AIC* and *SBC*, respectively:

```
compute aic =  %nobs*%logdet + 2*N
compute sbc =  %nobs*%logdet + N*log(%nobs)
dis 'aic = '  aic 'sbc = ' sbc
```

where: you must set *N* to equal the number of parameters estimated in the system.

1.1 Innovation Accounting

The variance decomposition and impulse response functions are easily obtained using the ERRORS instruction. To obtain the impulse responses and variance decompositions using a Choleski decomposition use:

errors(IMPULSES,MODEL=*modelname*) *equations steps name*

where:

modelname	The name of the model, as defined on the SYSTEM instruction
equations	Number of equations in the VAR.
steps	The forecast horizon and the number of impulse responses.
name	The name of the covariance matrix used on the ESTIMATE instruction.

If you exclude IMPULSES, RATS calculates and prints only the variance decompositions. Note that the IMPULSE instruction (discussed below) gives you more control over calculation and display of the impulse response functions.

Examples:

1. The sample program used in Section 2 below estimates a 3-equation 12-lag VAR using the variables *dlrgdp*, *dlrm2* and *drs*. The first two instructions create the growth rate of the real value of the money supply as measured by *M2*. The SYSTEM instruction creates a MODEL called *chap2*. The VARIABLES command instructs RATS to create a 3-variable VAR using *dlrgdp, dlrm2* and *drs*. Note that 12 lags of each variable and a constant are to be included in each equation.

```
set lrm2 = log(m2/price)    ;* Creates the log of the ‘real’ value of m2
dif lrm2 / dlrm2            ;* Creates the first-difference of lrm2

system(model=chap2)
var dlrgdp dlrm2 drs
lags 1 to 12
det constant
end(system)
```

ESTIMATE produces the estimates of the three equations and the *F*-statistics for the Granger-causality tests. The variance/covariance of the residuals is saved as the matrix *v*. The ERRORS instruction produces forecast error variances (from 1-step ahead through 24-step ahead horizons) and impulse responses for each of the three variables in the system. The ordering of the Choleski decomposition is that used on the VARIABLES instruction. Hence, the *errors* statement below uses the ordering *dlrgdp* → *dlrm2* → *drs.*

```
estimate(outsigma=v)
errors(impulses,model=chap2) 3 24 v
```

2. Suppose that the VARIABLES instruction in the example above was replaced with:

```
var drs dlrm2 dlrgdp
```

The forecast error variances (from 1-step ahead through 12-step ahead horizons) will be displayed for each of the three variables in the system. Now the *errors(model=chap2) 3 12 V* statement uses the ordering *drs* → *dlrm2* → *dlrgdp*. (Note that IMPULSE allows you to experiment with different orderings without having to re-estimate the model).

2. Example: Estimation of a 3-Equation VAR

Program 2.1 in the file CHAPTER2.PRG contains all of the instructions in the sample program discussed below. Suppose you want to estimate a VAR using the three variables *dlpgdp, dlrm2* and *drs*. After reading in the data set MONEY_DEM.XLS and constructing the variables, set up the VAR system with 12 lags of each variable using:

```
system(model=chap2)
var dlrgdp dlrm2 drs
lags 1 to 12
det constant
end(system)
```

Next, estimate the system using:

```
estimate(noprint,residuals=resids12)
```

Notice that we used the NOPRINT option—a three variable VAR with twelve lags produces a substantial amount of output. Since we are not sure if we actually want the 12-lag model, we suppress the output. The residuals are saved in the vector of series *resids12*; *resids12(1)* contains the residuals from the first regression, *resids12(2)* contains the residuals from the second regression and *resids12(3)* contains the residuals from the third regression.

The next three lines are used to compute and display the multivariate *AIC* and *SBC*. Notice that $N = 37*3$ since there are thirty-seven estimated coefficients in each of the three equations of the system.[14]

```
compute aic =  %nobs*%logdet + 2*(37*3)
compute sbc =  %nobs*%logdet + 37*3*log(%nobs)
dis 'aic = '  aic 'sbc = ' sbc

   aic =     -3198.00556 sbc =     -2859.47155
```

In order to perform a lag-length test, we re-estimate the system using only 8 lags. Notice that we restrict the estimation to begin in 1962:2 so that both systems are estimated over the same sample period. The residuals are saved in the vector *resids8*. Hence:

---

[14] The function %EQNSIZE(*number*) returns the number of regressors in equation *number* and %EQNSIZE(0) returns the number of regressors the most recently estimated equation. Since equations 1, 2 and 3 each contain the same number of regressions, an alternative is to use: compute aic = %nobs*%logdet + 2*3*%eqnsize(1).

```
system(model=chap2)
var dlrgdp dlrm2 drs
lags 1 to 8
det constant
end(system)
estimate(noprint,residuals=resids8) 1962:2 *

compute aic =  %nobs*%logdet + 2*(25*3)
compute sbc =  %nobs*%logdet + 25*3*log(%nobs)
```

dis 'aic = ' aic 'sbc = ' sbc

```
aic =    -3197.47118 sbc =    -2968.73198
```

The *AIC* selects the 12-lag model whereas the *SBC* selects the 8-lag model. We can also determine lag-length using a likelihood ratio test. Under the null hypothesis, we can restrict lags 9 - 12 of all coefficients in all three equations to be zero. If this restriction is binding, we reject the null hypothesis. Consider the following set of instructions:[15]

ratio(degrees=4*3*3,mcorr=37) 1962:2 *
 # resids12
 # resids8

```
Covariance\Correlation Matrices
               RESIDS12(1)    RESIDS12(2)    RESIDS12(3)
RESIDS12(1)  0.00004258409   0.2427449755    0.2104634745
RESIDS12(2)  0.00000843189   0.00002833363  -0.2239885437
RESIDS12(3)  0.00075612185  -0.00065639951   0.30309775570


              RESIDS8(1)     RESIDS8(2)     RESIDS8(3)
RESIDS8(1)  0.00004674920   0.1932339871    0.2567983642
RESIDS8(2)  0.00000726876   0.00003026766  -0.2174072492
RESIDS8(3)  0.00112342275  -0.00076529225   0.40938101386

Log Determinants are -21.923113 -21.458149
Chi-Squared(36)=     55.330714 with Significance Level 0.02068921
```

The RATIO instruction uses DEGREES=4*3*3 since the 8-lag model eliminates four lags of three variables in each of the three equations. MCORR = 37 since there are 37 coefficients in the unrestricted equations of the system. The elements along the principal diagonal of the

[15] Note that *resids12* contains the series *resids12(1), resids12(2)* and *resids12(3)* and that *resids8* contains the series *resids8(1), resids8(2)* and *resids8(3).* Thus, it is sufficient to use *resids12* on the first supplementary card and *resids8* on the second supplementary card. The identical output is obtained using:

```
ratio(degrees=4*3*3,mcorr=37) 1962:2 *
# resids12(1) resids12(2) resids12(3)
# resids8(1) resids8(2)  resids8(3)
```

Covariance\Correlation Matrices are the autocovariances of the residuals. For example, in the 12-lag model, the variance of the residuals from the *drs* equation is `0.30309775570`. The residual covariances are in the lower portion of the matrices (*i.e.*, below the diagonal) and the residual correlations are in the upper portion of the matrices. The log determinants of the unrestricted and restricted models are `-21.923113` and `-21.458149`, respectively. Given the calculated value of χ^2 = `55.330714`, the restriction is binding at the 5% (but not the 1%) significance level. Thus, the *AIC* and likelihood ratio test both select the 12-lag model.

Block Exogeneity:

We can perform a block exogeneity test to determine whether lags of *drs* enter the equations for *dlrgdp* and *dlrm2*. The name is a bit misleading; I prefer to use the term 'block exclusion' test. If lags of *drs* can be excluded from both the *dlrgdp* and *dlrm2* equations, we can model these two variables using a simple 2-variable VAR. The way to perform the test is to estimate a VAR with the lags of *drs* and a second without the lags. Consider:

```
system(model=unrestricted)
var dlrgdp dlrm2
lags 1 to 12
det constant drs{1 to 12}
end(system)
estimate(noprint,residuals=unrest)

system(model=restricted)
var dlrgdp dlrm2
lags 1 to 12
det constant
end(system)
estimate(noprint,residuals=rest)
```

The first block of instructions estimates a VAR for *dlrgdp* and *dlrm2* that includes the 12 lags of *drs*. Even though *drs* is not deterministic, the DETERMINISTIC instruction allows you include deterministic regressors *and* variables that are not estimated within the system. The residuals of this unrestricted VAR are saved in *unrest*. The second block of instructions estimates a 2-variable VAR without the lags of *drs* and saves the residuals in *rest*. The likelihood ratio test has 24 degrees of freedom (12 lags of *drs* are excluded from each equation) and mcorr = 37 (each regression in the unrestricted model has 37 regressors).

ratio(degrees=24,mcorr=37)
 # unrest ; # rest

```
Log Determinants are -20.596224 -20.059977
Chi-Squared(24)=     63.813364 with Significance Level 0.00001814
```

The restriction is clearly binding. Since the lags of *drs* should be included in the *dlrgdp* and *dlrm2* equations (so that *drs* is not block exogenous), we need to return to the 3-variable VAR. You can confirm that the multivariate *AIC* and *SBC* also indicate that *drs* is not block exogenous.

Innovation Accounting:

To obtain the variance decompositions and impulse responses, it is necessary to re-estimate the system in order to save the variance/covariance matrix. Use the OUTSIGMA= option on the ESTIMATE instruction to save the covariance matrix as *V*.

system(model=chap2)
var dlrgdp dlrm2 drs
lags 1 to 12
det constant
end(system)
estimate(outsigma=v)

```
F-Tests, Dependent Variable DLRGDP
Variable            F-Statistic       Signif
DLRGDP                    1.3277     0.2078223
DLRM2                     2.1065     0.0193254
DRS                       3.3135     0.0002667

F-Tests, Dependent Variable DLRM2
Variable            F-Statistic       Signif
DLRGDP                    0.8990     0.5494694
DLRM2                     8.5958     0.0000000
DRS                       3.9210     0.0000285

F-Tests, Dependent Variable DRS
Variable            F-Statistic       Signif
DLRGDP                    5.4620     0.0000001
DLRM2                     2.3067     0.0097938
DRS                       8.0504     0.0000000
```

The ESTIMATE instruction will produce the equivalent output of three OLS regression estimates for each of the three equations in the system. To save a considerable amount of space, the output box above reports only the Granger-causality tests. At conventional significance levels, *dlrm2* and *drs* Granger-cause *dlrgdp*, *dlrgdp* does not Granger-cause *dlrm2* and all variables Granger-cause *drs*.

It is straightforward to obtain the impulse responses using the ERRORS instruction. As indicated above, we can obtain the impulse responses and variance decompositions using the ordering *dlrgdp* → *dlrm2* → *drs* with the following set of instructions:[16]

---

[16] Although RATS produces the impulse responses for periods 1 through 24, only the first three impulses are shown here. RATS displays the variance decompositions for all forecast horizons through 24; we display only the 1-step, 8-step, 12-step and 24-step ahead forecast error variances.

errors(impulses,model=chap2) 3 24 v

As shown on the next page, a one standard deviation shock to *dlrgdp* (approximately equal to `0.00653` units) induces a contemporaneous increase in *dlrm2* by `0.00129` units and a contemporaneous increase in *drs* by `0.11587` units. After one period, *dlrgdp* is still `0.00093` units above its initial value, while *dlrm2* and *drs* are `0.00058` and `0.21130` units away from their initial values. On the other hand, a one standard deviation shock to *dlrm2* (equal to `0.00516` units) has no contemporaneous effect on *dlrgdp* but induces a contemporaneous decrease of `-0.15611` units on *drs*. After one period, *dlrgdp* = `0.00069`, *dlrm2* = `0.00430` and *drs* = `-0.10718`. Given the ordering of the Choleski decomposition, a one standard deviation *drs* shock (equal to `0.51507`) has no contemporaneous effect on the other variables in the system. After one period, *dlrgdp* = `0.00144`, *dlrm2* = `-0.00221` and *drs* = `0.20055`.

```
Responses to Shock in DLRGDP
Entry            DLRGDP          DLRM2            DRS
           1   0.006525648724  0.001292115488  0.115869223380
           2   0.000928019134  0.000576615555  0.211296040285
           3   0.000473985466 -0.000059458005  0.088298485503

Responses to Shock in DLRM2
Entry            DLRGDP          DLRM2            DRS
           1   0.000000000000  0.005163725798 -0.156111296557
           2   0.000686946710  0.004308977026 -0.107180122895
           3   0.001613299752  0.002441287281  0.047996790185

Responses to Shock in DRS
Entry            DLRGDP          DLRM2            DRS
           1   0.000000000000  0.000000000000  0.515074112975
           2   0.001437493942 -0.002210813500  0.200554826822
           3  -0.001143540867 -0.001410062827 -0.132047370809

Decomposition of Variance for Series DLRGDP
Step  Std Error      DLRGDP     DLRM2      DRS
   1  0.006525649   100.000     0.000     0.000
   8  0.008207901    69.424    15.408    15.168
  12  0.008465552    66.607    16.607    16.785
  24  0.008657572    65.424    16.476    18.100

Decomposition of Variance for Series DLRM2
Step  Std Error      DLRGDP     DLRM2      DRS
   1  0.005322934     5.893    94.107     0.000
   8  0.008874702     2.823    82.859    14.318
  12  0.009251525     5.115    78.899    15.986
  24  0.009627022     6.524    74.715    18.761

Decomposition of Variance for Series DRS
Step  Std Error      DLRGDP     DLRM2      DRS
   1  0.550543146     4.429     8.041    87.530
   8  0.692450601    15.569     9.827    74.604
  12  0.793836231    28.501    11.750    59.749
  24  0.824410351    31.368    12.064    56.567
```

The variance decompositions suggest a rich interaction among the variables, particularly at the longer forecast horizons. For example, *dlrgdp* explains all of its 1-step ahead forecast error variance, but *dlrm2* and *drs* explain 15.408 and 15.168 percent of the 8-step ahead forecast error variance in *dlrgdp*, respectively.

Extensions

1. If you want to reverse the ordering of the variables such that *drs* → *dlrm2* → *dlrgdp* use:

```
system(model=chap2)
var drs dlrm2 dlrgdp
lags 1 to 12
det constant
end(system)
estimate(outsigma=v)
errors(impulses,model=chap2) 3 24 v
```

2. You can produce multivariate forecasts using the FORECAST instruction. The most useful form of the instruction is:

FORECAST(model=*modelname*, results=forecasts) * *steps start*

where:

modelname	The model name used on the SYSTEM instruction.
results=forecasts	Creates the series forecasts(1), … , forecasts(n) which contain the forecasts of the n variables in the system
steps	Number of periods to forecast.
start	First period to forecast.

The following FORECAST instruction uses the VAR model chap2 to produce 12 out-of-sample forecasts beginning with 2001:2.

forecast(model=chap2,results=fores) * 12 2001:2
pri / fores

```
ENTRY        FORES(1)         FORES(2)         FORES(3)
 2001:02   0.007005058389  0.018494401728 -0.194920330700
 2001:03   0.012225055977  0.015287643122  0.242193105882
 2001:04   0.011144421777  0.015551352314 -0.624105358190
 2002:01   0.010444864257  0.015556006874 -0.017371770409
 2002:02   0.016462341748  0.014234534497  0.382157540684
 2002:03   0.013156260294  0.012041121844 -0.184462465867
 2002:04   0.011033652617  0.013449326547  0.446338917263
 etc.
```

The 12 forecasts for $dlrgdp_t$, $dlrm2_t$ and drs_t are contained in the series *fores(1), fores(2)* and *fores(3)*, respectively.

2.1 Near-VARs

In a near-VAR, the right-hand sides of the equations in the system are not identical. Examples include:

i. Different lag lengths:

$$y_t = a_{11}(1)y_{t-1} + a_{11}(2)y_{t-2} + a_{12}z_{t-1} + e_{1t}$$
$$z_t = a_{21}y_{t-1} + a_{22}z_{t-1} + e_{2t}$$

ii. The $\{z_t\}$ series does not Granger-cause $\{y_t\}$:

$$y_t = a_{11}y_{t-1} + e_{1t}$$
$$z_t = a_{21}y_{t-1} + a_{22}z_{t-1} + e_{2t}$$

iii. A third variable $\{w_t\}$ affects only $\{z_t\}$:

$$y_t = a_{11}y_{t-1} + a_{12}z_{t-1} + e_{1t}$$
$$z_t = a_{21}y_{t-1} + a_{22}z_{t-1} + a_{23}w_t + e_{2t}$$

Since the equations have different right-hand-side variables, the efficiency of the estimates can be improved using Seemingly Unrelated Regressions. Use the following method to estimate a near-VAR using RATS' SUR instruction.

Step 1: You must define the equations to use in the near-VAR. The simplest way to set up your equations is using the DEFINE= option of the LINREG instruction.

Examples:

1. To set up the first near-VAR system above, use:

```
linreg(define=equation1) y
# y{1 to 2} z{1}
linreg(define=equation2) z
# y{1} z{1}
```

2. To set up the third near-VAR system above, use:

```
linreg(define=equation1) y
# y{1} z{1}
linreg(define=equation2) z
# y{1} z{1} w
```

Step 2: Use the SUR instruction to estimate the system. The typical syntax of SUR is:

SUR(OUTSIGMA=V) *equations start end*
equation

where:

equations	The number of equations in the system you want to estimate.
start end	The range of entries to use.

There is one supplementary card for each equation in the system. The information on each supplementary card contains the equation name used for the DEFINE= option on LINREG instruction.

SUR creates the variables: %XX = covariance matrix of coefficients, %NOBS = number of observations, %NREG = number of regressors, %LOGDET = log determinant of the estimate of Σ and %SIGMA = final estimate of Σ.

Step 3: To create a model, GROUP the equations and provide a *modelname*:

GROUP *modelname equation1 equation2 …*

Step 4: As in a VAR, obtain the impulse responses and variance decompositions with:

ERRORS(IMPULSES,MODEL=*modelname*) *equations steps name*

Similarly, the forecasts can be obtained with:

FORECAST(MODEL=*modelname*, RESULTS=*forecasts*) * *steps start*

Example

The 12-lag VAR for *dlrgdp, dlrm2* and *drs* indicated that it was possible to eliminate *dlrgdp{1 to 12}* from the *dlrgdp* and *dlrm2* equations. To impose these restrictions, set up the following three equations. Note that *eq1* regresses $dlrgdp_t$ on 12 lags of *dlrm2* and *drs*, *eq2* regresses $dlrm2_t$ on 12 lags of dlrm2 and drs, and eq3 regresses drs_t on 12 lags of all three variables.

```
lin(define=eq1) dlrgdp ; # constant dlrm2{1 to 12} drs{1 to 12}
lin(define=eq2) dlrm2 ; # constant dlrm2{1 to 12} drs{1 to 12}
lin(define=eq3) drs ; # constant dlrgdp{1 to 12} dlrm2{1 to 12} drs{1 to 12}
```

Next, estimate the system of equations (saving the covariance matrix of the residuals) using:

```
sur(outsigma=v) 3
# eq1 ; # eq2 ; # eq3
```

Since SUR creates %NOBS and %LOGDET we can display the multivariate *AIC* and *SBC* using:

```
compute aic =  %nobs*%logdet + 2*(2*25+37)
compute sbc =  %nobs*%logdet + (2*25+37)*log(%nobs)
dis 'aic = '  aic 'sbc = ' sbc
```

```
aic =    -3217.51381 sbc =    -2952.17634
```

Note that there are 25 regressors in the first two equations and 37 regressors in the third equation. If you compare these values of multivariate *AIC* and *SBC* to those from the 3-variable VAR, you will find that the near-VAR has the better fit.

Now GROUP these three equations into a model called *chap2_sur*. You can obtain 24 impulse responses and 1-step to 24-step ahead forecast error variances from a Choleski decomposition of *v* using (For brevity, only a partial list of the impulses is shown):

group chap2_sur eq1 eq2 eq3
errors(impulses,model=chap2_sur) 3 24 v

```
Responses to Shock in DLRGDP
Entry             DLRGDP           DLRM2            DRS
            1   0.006850793364  0.001122502049  0.126303901431
            2   0.000558731002  0.000268039828  0.222089855433
            3   0.000366973202 -0.000599073730  0.089690963848

Responses to Shock in DLRM2
Entry             DLRGDP           DLRM2            DRS
            1   0.000000000000  0.005388227762 -0.160730936792
            2   0.000713637578  0.004584256661 -0.113798715256
            3   0.001465609436  0.002529968390  0.045146339746

Responses to Shock in DRS
Entry             DLRGDP           DLRM2            DRS
            1   0.000000000000  0.000000000000  0.515144919333
            2   0.001322010498 -0.002214755409  0.198307071178
            3  -0.001325937131 -0.001076247253 -0.148405838091
```

At this point, you could produce 12 out-of-sample forecasts beginning with 2001:2 with:

FORECAST(model=chap2_sur,results=fores) * 12 2001:2

3. Error-Correction Models

RATS works a bit differently if you want to estimate an error-correction model. In Chapter 1, we established a long-run relationship between the 1-year and 3-month *T*-bill rates. Recall that the estimated long-run relationship is:

$$tb1yr_t = 0.6980794657 + 0.9167216207\ tb3mo_t$$

As such, we might want to estimate an error-correction model of the form:

$$drl_t = \alpha_{10}[tb1yr_{t-1} - 0.6980794657 - 0.9167216207\ tb3mo_{t-1}] + A_{11}(L)drl_{t-1} + A_{12}(L)drs_{t-1} + e_{1t}$$

$$drs_t = \alpha_{20}[tb1yr_{t-1} - 0.6980794657 - 0.9167216207\ tb3mo_{t-1}] + A_{21}(L)drl_{t-1} + A_{22}(L)drs_{t-1} + e_{2t}$$

where: $A_{ij}(L)$ are polynomials in the lag operator L.

The steps in setting up the VAR including the error-correction term are a bit different.

Step 1: Estimate the long-run equilibrium relationship, using the DEFINE= option on the LINREG instruction. This step allows you to pass the estimated coefficients from LINREG to the VAR system. Thus, in the interest rate example, we can use:

```
lin(define=spread) tb1yr / resids
# constant tb3mo
```

Step 2: Set up the VAR system using the MODEL= option on the SYSTEM instruction. Moreover, in the SYSTEM-END(SYSTEM) block, include the instruction:

ECT *name*

where: *name* comes from the LINREG(DEFINE=*name*) instruction used to estimate the long-run equilibrium relationship.

Thus, in the interest-rate example, if you want to estimate a model with 5-lagged changes in each series, use:[17]

[17] If you do your own hypothesis testing, you will find that a 6-lag specification seems quite reasonable.

```
system(model=term)
var tb1yr tb3mo
lags 1 to 6
det constant
ect spread          << NOTE: We used DEFINE = spread on the LINREG instruction.
end(system)
```

NOTICE THAT WE SET UP THE MODEL IN LEVELS, NOT IN FIRST DIFFERENCES. RATS will report the results in first differences along with the error-correction term. Since we want 5 lags in the first differences, we use 6 lags of the level.

Step 3: Enter the appropriate ESTIMATE instruction. For the interest rate example, we can use:

estimate(outsigma=s,residuals=resid)

```
Dependent Variable TB1YR
Variable              Coeff        Std Error        T-Stat     Signif
***************************************************************************
1.  D_TB1YR(1)       -0.003371434  0.256767615     -0.01313  0.98954030
2.  D_TB1YR(2)       -0.544498028  0.251691768     -2.16335  0.03200757
3.  D_TB1YR(3)       -0.243383900  0.244713790     -0.99457  0.32145878
4.  D_TB1YR(4)        0.076416728  0.223004947      0.34267  0.73230059
5.  D_TB1YR(5)       -0.446724622  0.220182611     -2.02888  0.04413868
6.  D_TB3MO(1)        0.294864692  0.231514842      1.27363  0.20465261
7.  D_TB3MO(2)        0.144076253  0.227888340      0.63222  0.52814921
8.  D_TB3MO(3)        0.492870264  0.222328127      2.21686  0.02805167
9.  D_TB3MO(4)       -0.159091114  0.201743992     -0.78858  0.43153218
10. D_TB3MO(5)        0.512723191  0.198144931      2.58762  0.01055885
11. Constant          0.002972876  0.051042727      0.05824  0.95362828
12. EC1{1}           -0.098335589  0.230405391     -0.42679  0.67010686
```

```
Dependent Variable TB3MO
Variable              Coeff        Std Error        T-Stat     Signif
***************************************************************************
1.  D_TB1YR(1)       -0.290217816  0.280102775     -1.03611  0.30172315
2.  D_TB1YR(2)       -0.578900068  0.274565632     -2.10842  0.03656203
3.  D_TB1YR(3)       -0.409504186  0.266953492     -1.53399  0.12701934
4.  D_TB1YR(4)       -0.084036114  0.243271740     -0.34544  0.73021920
5.  D_TB1YR(5)       -0.467464611  0.240192909     -1.94620  0.05339312
6.  D_TB3MO(1)        0.637950862  0.252555018      2.52599  0.01251551
7.  D_TB3MO(2)        0.073016809  0.248598938      0.29371  0.76936005
8.  D_TB3MO(3)        0.742308339  0.242533410      3.06064  0.00259335
9.  D_TB3MO(4)       -0.085007689  0.220078579     -0.38626  0.69982004
10. D_TB3MO(5)        0.653078711  0.216152434      3.02138  0.00293320
11. Constant          0.006499189  0.055681513      0.11672  0.90722864
12. EC1{1}           -0.566413670  0.251344739     -2.25353  0.02559250
```

Step 4: To obtain the impulse responses and variance decompositions, use the instruction ERRORS(IMPULSES,MODEL=*model*).

For the interest rate example, we can use:

errors(impulses,model=term) 2 24 s

```
Responses to Shock in TB1YR
   Entry            TB1YR          TB3MO
            1  0.6427986990792 0.6577227963055
            2  0.8384894480559 0.9133375666545
            3  0.6580803688211 0.6962100440500

Responses to Shock in TB3MO
Entry            TB1YR          TB3MO
            1   0.000000000000  0.243115941739
            2   0.049770290069  0.271975555534
            3   0.073515992403  0.180662886845

Decomposition of Variance for Series TB1YR
Step  Std Error      TB1YR     TB3MO
   1  0.642798699   100.000     0.000
   8  2.185202713    98.828     1.172
  12  2.799388183    99.125     0.875
  24  4.042565038    99.100     0.900

Decomposition of Variance for Series TB3MO
Step  Std Error      TB1YR     TB3MO
   1  0.701216541    87.979    12.021
   8  2.463471405    95.601     4.399
  12  3.139985944    97.004     2.996
  24  4.463361225    98.009     1.991
```

As in the previous example, RATS produces the impulse responses for periods 1 through 24, only the first three impulses are shown here. RATS displays the variance decompositions for all forecast horizons through 24; we display only the 1-step, 8-step, 12-step and 24-step ahead forecast error variances.

To create graphs of the impulse responses, it is helpful to know a bit about matrices. Chapter 5 considers the construction and manipulation of matrices in great detail. For now it is sufficient to know that it is necessary to create a 2 x 2 matrix of series to hold the response functions (there are two sets of responses for each of the two variables). This is accomplished by using the DECLARE instruction to create the 2 x 2 rectangular matrix *impulses*.

declare rectangular[series] impulses(2,2)

In most circumstances, it is also necessary to create a matrix of labels. A graph that labels the variables IMPULSES(2,1) or IMPULSES(2,2) is not very helpful.

com implabels = || '1 year','3 month'||

The impulse responses are created by the IMPULSE instruction. If we use the MODEL= option, the form of the IMPULSE instruction the we need is:

impulse(MODEL=*modelname,*RESULTS=*matrix*) *equations steps shock_to name*

where:

modelname=	The model name used on the STSTEM instruction.
equations	Number of equations in the system. Use * with the MODEL= option.
matrix =	Name of the matrix used to store the impulses.
steps	The forecast horizon and the number of impulse responses.
shock_to	The component to be shocked. Use * with the MODEL= option.
name	The name of the covariance matrix used on the ESTIMATE instruction.

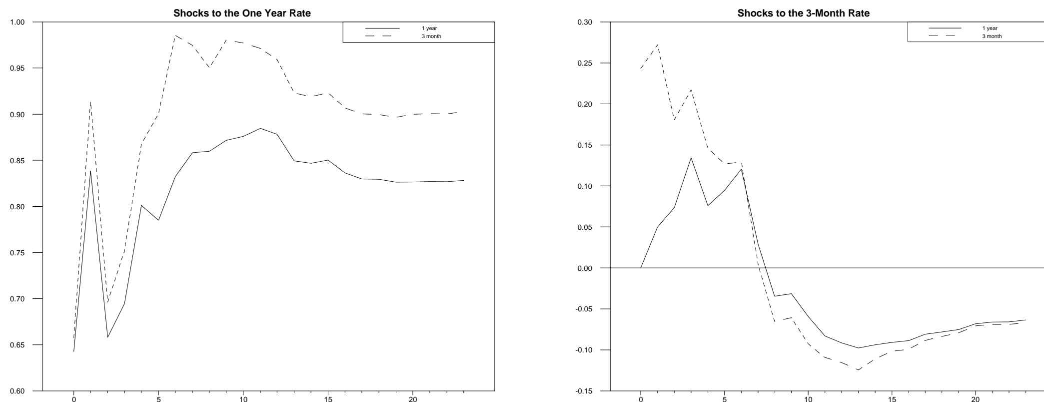
To create 24 impulses from the model *term* that are stored in the matrix *impulses*, use:

```
impulse(model=term,result=impulses,noprint) * 24 * s
```

The first * is a placeholder for the number of equations and the second tells RATS to shock all equations. The responses of *tb1yr* and *tb3mo* to innovations in *tb1yr* are stored in *impulses(1,1)* and *impulses(2,1)*, respectively. The response of *tb1yr* and *tb3mo* to innovations in *tb3mo* are stored in *impulses(1,2)* and *impulses(2,2)*, respectively. The following two graphs were created using:[18]

```
spgraph(hfields=2,vfi=1,header='Impulse Response Functions')
   graph(header='Shocks to the One Year Rate',key=upright, number =0, $
      patterns,klabels=implabels) 2
   # impulses(1,1) ; # impulses(2,1)
   graph(header='Shocks to the 3-month  Rate',key=upright,number=0, $
      patterns, klabels =implabels) 2
   # impulses(1,2) ; # impulses(2,2)
spgraph(done)
```

[18] Note that the impulses are not scaled since all are in the same units.



Notice that both graphs use the KLABELS = option. This option instructs RATS to use the vector *implabels* to label each of the series. The first element in *implabels* is *1 year* and the second is *3 month*. As such, the series on the first supplemental card is labeled *1 year* and the series on the second supplemental card is labeled *tb3mo*.

The program MONTEVAR.PRG allows you to place confidence bands around your impulse response functions. The program is distributed with RATS so that it should in the same directory as RATS itself. Now that you know how to work with VARs, it should be trivial for you to modify MONTEVAR.PRG so as obtain confidence intervals for the impulse responses.

4. Structural Decompositions

A Choleski decomposition is not the only way to obtain the impulse responses. In fact, it is straightforward to show that the impulse response function is not identified unless additional restrictions are imposed on the VAR system. The Choleski decomposition is only one way to impose the necessary number of identifying restrictions. Consider a 2-variable model:

$$y_t = \sum_{i=1}^{n} a_{11}(i) y_{t-i} + \sum_{i=1}^{n} a_{12}(i) z_{t-i} + e_{1t}$$
$$z_t = \sum_{i=1}^{n} a_{21}(i) y_{t-i} + \sum_{i=1}^{n} a_{22}(i) z_{t-i} + e_{2t}$$

The impulse response function is obtained using the moving average representation:

$$y_t = \sum_{i=1}^{n} b_{11}(i) e_{1t-i} + \sum_{i=1}^{n} b_{12}(i) e_{2t-i} + e_{1t}$$
$$z_t = \sum_{i=1}^{n} b_{21}(i) e_{1t-i} + \sum_{i=1}^{n} b_{22}(i) e_{2t-i} + e_{2t}$$

The issue is that the regression residuals $\{e_{1t}\}$ and $\{e_{2t}\}$ are linear combinations of the pure innovations in y_t and z_t. If we call these pure innovations ε_{1t} and ε_{2t}, we have:

$$e_{1t} = g_{11}\varepsilon_{1t} + g_{12}\varepsilon_{2t}$$
$$e_{2t} = g_{21}\varepsilon_{1t} + g_{22}\varepsilon_{2t}$$

or:

$$e_t = G\varepsilon_t$$

The nature of the system is such that the pure innovations are serially uncorrelated and orthogonal to each other. Nevertheless, a pure innovation in y_t will have a contemporaneous effect on z_t if $g_{21} \neq 0$ and a pure innovation in z_t will have a contemporaneous effect on y_t if $g_{12} \neq 0$. Even though ε_{1t} and ε_{2t} are serially uncorrelated, their effects have some persistence since the values of $a_{jk}(i)$ are not all equal to zero. If we let $var(\varepsilon_{1t}) = \sigma_1^2$ and $var(\varepsilon_{2t}) = \sigma_2^2$, it follows that:

$$E\varepsilon_{1t}\varepsilon_{2t} \equiv \Sigma_\varepsilon = \begin{pmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{pmatrix}$$

The problem is to identify the unobserved values of ε_{1t} and ε_{2t} from the regression residuals e_{1t} and e_{2t}. If we knew the four values g_{11}, g_{12} g_{13} and g_{14} we could obtain all of the structural shocks for the regression residuals. Of course, we do have some information about the values of the g_{ij}. Consider the variance/covariance matrix of the regression residuals:

$$Eee' = \Sigma$$

We know the four elements of this matrix—in fact, you can display the elements of the matrix using DISPLAY %SIGMA. As in the earlier sections of this chapter, denote the elements of Σ as σ_{ij}:

$$\Sigma = \begin{pmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{21} & \sigma_{22} \end{pmatrix}$$

Although the values of the g_{ij} are unknown and ε_{1t} and ε_{2t} are unobserved, we know that $e_t = G\varepsilon_t$. Hence, it must be the case that:

$$Ee_te_t' = EG\varepsilon_t\varepsilon_t'G'$$

Since $Ee_te_t' = \Sigma$ and $E\varepsilon_t\varepsilon_t' = \Sigma_\varepsilon$, it follows that:

$$\begin{pmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{21} & \sigma_{22} \end{pmatrix} = G\Sigma_\varepsilon G'$$

where: Σ_ε is the diagonal matrix (defined above) consisting of $var(\varepsilon_{1t}) = \sigma_1^2$ and $var(\varepsilon_{2t}) = \sigma_2^2$. If it is assumed that $\sigma_1^2 = \sigma_2^2 = 1$, we can write:[19]

$$\begin{pmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{21} & \sigma_{22} \end{pmatrix} = \begin{pmatrix} g_{11}^{\ 2} + g_{12}^{\ 2} & g_{11}g_{21} + g_{12}g_{22} \\ g_{11}g_{21} + g_{12}g_{22} & g_{21}^{\ 2} + g_{22}^{\ 2} \end{pmatrix}$$

Since the four values of σ_{ij} are known, it would appear that there are four equations to determine the four unknown values g_{11}, g_{12}, g_{21} and g_{22}. However, the symmetry of the system is such that $\sigma_{21} = \sigma_{12}$ so that there are only three independent equations to determine the four elements of G. The Choleski decomposition adds an additional restriction. If the pure shock to z_t is to have no contemporaneous effect on y_t, it must be the case that $g_{12} = 0$. Similarly, if the pure shock to y_t is to have no contemporaneous effect on z_t, it must be the case that $g_{21} = 0$. In either case, there is a fourth equation that can be used to solve for the other three values of the G matrix.

To generalize the argument to an n-th order VAR systems, we have:

$$\Sigma = GG'$$

where: Σ and G are n x n matrices. Using the same logic, it is possible to show that it is necessary to impose $(n^2 - n)/2$ additional restrictions on G to identify completely identify the

[19] This normalization assumption is innocuous because it simply scales the magnitudes of g_{11}, g_{12}, g_{13} and g_{14}.

system. Regardless of the size of the system, the Choleski decomposition is recursive in that it sets:

$$G = \begin{bmatrix} g_{11} & 0 & \dots & 0 \\ g_{21} & g_{22} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ g_{n1} & g_{n2} & \dots & g_{nn} \end{bmatrix}$$

Since each element above the principle diagonal is zero, there is exactly the number of restrictions needed to identify all of the remaining elements of G. However, many other possibilities exist.

RATS allows you to select the form of the G matrix so that you can impose a far richer set of restrictions on the G matrix. Moreover, it is possible to impose overidentifying restrictions so that you can test hypotheses concerning the restrictions. Suppose we normalize the elements on the principle diagonal to be unity. To keep the notation simple, suppose we let the G matrix be:

$$G = \begin{bmatrix} 1 & g_{12} & \dots & g_{1n} \\ g_{21} & 1 & \dots & g_{2n} \\ \dots & \dots & \dots & \dots \\ g_{n1} & g_{n2} & \dots & 1 \end{bmatrix}$$

There are two cases to consider. In the first case, you completely specify all of the numerical values of the g_{ij}. In such circumstances, you simply create the G matrix and enter the appropriate values for all of the g_{ij}. In the second case, you fix at least $(n^2 - n)/2$ elements of G. However, since the remaining values of the g_{ij} are free parameters, they need to be estimated from the data. This second case is the most typical and forms the basis of the Sims-Bernanke and Blanchard-Quah decompositions. Nevertheless, we begin with simple case where G is known.

4.1 Structural VARs with a Known G Matrix

It is straightforward to perform a structural decomposition when G is known. Chapter 5 describes how to work with matrices in RATS. For now, it is sufficient to know that you can use the COMPUTE to construct the matrix G and enter the desired numerical values for the g_{ij}. Then use the DECOMP=G option on the ERRORS or IMPULSE instruction. Since you are not performing a decomposition using the covariance matrix from the ESTIMATE instruction, do not specify the covariance matrix in the *name* field of the ERRORS of IMPULSE instruction.

Examples

1. In the error-correcting model of the term-structure relationship, variance decomposition and impulse responses were obtained using a Choleski decomposition. Moreover, 24 impulse responses were obtained and stored in the matrix *impulses* using:

impulse(model=term,result=impulses) * 24 * s

Recall that the first few impulses from the model are:

```
Responses to Shock in TB1YR
   Entry            TB1YR           TB3MO
           1  0.6427986990792 0.6577227963055
           2  0.8384894480559 0.9133375666545
           3  0.6580803688211 0.6962100440500

Responses to Shock in TB3MO
   Entry            TB1YR           TB3MO
           1   0.000000000000  0.243115941739
           2   0.049770290069  0.271975555534
           3   0.073515992403  0.180662886845
```

Instead, suppose you want to force the regression residual from the *tb1yr* equation to be identical to the pure innovation in *tb1yr* and the regression residual from the *tb3mo* equation to be identical to the pure innovation in *tb3mo*. This is equivalent to setting $g_{12} = g_{21} = 0$. Consider:

$$\begin{bmatrix} e_{1t} \\ e_{2t} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \varepsilon_{1t} \\ \varepsilon_{2t} \end{bmatrix}$$

To equate e_{1t} with ε_{1t} and e_{2t} with ε_{2t} use:

com g = || 1. , 0. | 0. , 1. ||

To obtain 24 impulses using g as opposed to the Choleski decomposition use:

impulse(model=term,results=impulses,decomp=g) * 24 *

```
Entry            TB1YR           TB3MO
           1  1.0000000000000 0.0000000000000
           2  1.0949641548808 0.2761958544234
           3  0.7143622610318 0.3227246002156

Responses to Shock in TB3MO
   Entry            TB1YR           TB3MO
           1   0.000000000000  1.000000000000
           2   0.204718331973  1.118707204423
           3   0.302390669558  0.743114110710
```

For comparison purposes, only the first three impulse responses are shown. Now, an innovation to *tb1yr* has no contemporaneous effect on *tb3mo*. Also note that an innovation in *tb3mo* has a 1-unit effect on *tb3mo*. This follows since G is normalized such that σ_1^2 and σ_2^2 both equal unity.

2. Recall that the long-run equilibrium relationship between the two interest rates is such that:

$tb1yr_t = 0.6980794657 + 0.9167216207\ tb3mo_t$

so that:

$\Delta\ tb1yr_t = 0.9167216207\ \Delta tb3mo_t$

or: $drl_t = 0.9167216207\ drs_t$

For illustration purposes, suppose we wanted to impose a similar restriction on the innovations. In particular, suppose we wanted to let innovations in $tb3mo_t$ be unaffected by innovations in $tb1yr_t$ but we wanted innovations in $tb3mo_t$ to change the contemporaneous value of $tb1yr_t$ by 0.9167216207 units. Since the residuals from the $tb1yr_t$ and $tb3mo_t$ equations are the $\{e_{1t}\}$ and $\{e_{2t}\}$ sequence respectively:

$$\begin{bmatrix} e_{1t} \\ e_{2t} \end{bmatrix} = \begin{bmatrix} 1 & 0.91... \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \varepsilon_{1t} \\ \varepsilon_{2t} \end{bmatrix}$$

Now a pure shock to $tb1yr_t$ (*i.e.*, an ε_{1t} shock) affects the contemporaneous value of $tb1yr_t$ but not $tb3mo_t$. A pure shock to $tb3mo_t$ (*i.e.*, an ε_{2t} shock) has a 1-unit effect on $tb3mo_t$ and a 0.9167216207-unit effect on $tb1yr_t$. To obtain the impulse responses using this G matrix, recall that we estimated the long-run relationship using:

```
lin(define=spread) tb1yr / resids
# constant tb3mo
```

Immediately after this LINREG instruction insert the line:

```
com x = %beta(2)
```

Now the variable x contains the desired slope coefficient. After estimating the error-correction model, construct G using:

```
com g = || 1.0 , x | 0. , 1.0 ||
```

To obtain 24 impulses using this G matrix use:

impulse(model=term,results=impulses,decomp=g) * 24 *

```
Responses to Shock in TB1YR
   Entry            TB1YR           TB3MO
           1  1.0000000000000 0.0000000000000
           2  1.0949641548808 0.2761958544234
           3  0.7143622610318 0.3227246002156

Responses to Shock in TB3MO
   Entry            TB1YR           TB3MO
           1  0.9167216207369 1.0000000000000
           2  1.2084956466838 1.3719019157307
           3  0.9572619992845 1.0389627292717
```

5. The Sims-Bernanke Decomposition

Suppose that you want to fix at least $(n^2 - n)/2$, but not all, of the elements of G. RATS allows you to estimate an exactly identified or an over-identified structural VAR. The procedure to obtain a Sims-Bernanke decomposition consists of the following four steps:

Step 1: Use NONLIN to enumerate the elements of G that you want to estimate. This list of free parameters informs RATS of the names of the parameters that will be estimated.

Step 2: DECLARE a FORMULA containing a rectangular matrix and use the formula to construct the G matrix. Note that you need to provide RATS with an initial guess for each free parameter to be estimated.

Step 3: Use the CVMODEL instruction to estimate the G matrix. The standard syntax you will use is:

CVMODEL(factor=*output matrix*, *other options*) %sigma *frml*

where:

output matrix	is the name of the matrix used to store the estimate of G. This will be the matrix you use to obtain the impulse responses on the IMPULSES instruction.
%sigma	is the variance/covariance matrix obtained from estimating the VAR (i.e., %sigma = Σ)
frml	is the FORMULA you created in Step 2.

and other options include:

iters=	maximum number of iterations to use in the nonlinear estimation
method=	[BFGS]/SIMPLEX/GENETIC. The BFGS algorithm can be quite sensitive to the initial guess. However, only the BFGS estimation method can display standard errors and t-statistics.

Step 4: Now that G has been created, you can use the ERRORS or IMPULSES instruction to obtain the impulse responses.

Examples

1. In the estimation of the term-structure relationship, we used the IMPULSES instruction to obtain the impulse responses from a Choleski decomposition of the variance/covariance matrix *s*. It is instructive to use the methodology described in Steps 1 to 4 above to perform the same decomposition. Hence, we want G to have the form:

$$G = \begin{bmatrix} 1 & 0 \\ g_{21} & 1 \end{bmatrix}$$

where: g_{21} is a free parameter to be estimated from the data.

Consider the following instructions:

nonlin g21
dec frml[rect] g_form
frml g_form = || 1., 0. | g21 , 1. ||
com g21 = 0.01

The NONLIN instruction informs RATS that we want to estimate a single parameter called g21. The next instruction DECLARES the FORMULA *g_form*.

The third instruction is used to specify the form of *g_form*. Unlike our previous examples, there is a free parameter. Since this parameter is estimated using non-linear estimation methods, we need to provide RATS with an initial value. In the example, the value 0.01 is used. Next, we use CVMODEL to perform the estimation:

cvmodel(factor=g) %sigma g_form

```
Covariance Model - Estimation by BFGS
Convergence in     5 Iterations. Final criterion was  0.0000039 <  0.0000100
Observations                    167
Log Likelihood                     309.97546215
Log Likelihood Unrestricted        309.97546215

 Variable         Coeff       Std Error      T-Stat     Signif
*************************************************************************
1.  G21        -1.023217373  0.029267200    -34.96123  0.00000000
```

Notice that the estimation converged in only five iterations. If an estimation does not converge, you can increase the number of iterations from the default value of 40, provide better initial guesses, or use an alternative estimation method. A useful way to obtain satisfactory initial guesses is to use the simplex or genetic estimation method for a few iterations and then switch to the BFGS method. Consider:

cvmodel(factor=g,iters=4,method=simplex) %sigma g_form
cvmodel(factor=g) %sigma g_form

Also note that the reported value of *g21* is estimated to be more than 34 standard deviations from zero. To interpret this estimate, recall that RATS estimates the matrix *G* such that $G*G'$ = %sigma. You can display *G* using:

dis g

```
0.64280       0.00000
0.65772       0.24312
```

For comparison purposes, you can display the variance/covariance matrix using:

dis %sigma:

```
0.41319
0.42278       0.49170
```

Standardizing *G* such that the diagonal elements are unity yields:

$$\begin{bmatrix} 1 & 0 \\ 1.023... & 1 \end{bmatrix}$$

Finally, you can obtain the impulse responses with the instruction:

impulse(model=term,results=impulses,decomp=g) * 24 *

```
Responses to Shock in TB1YR
   Entry            TB1YR           TB3MO
            1  0.6427986990792 0.6577227963170
            2  0.8384894480582 0.9133375666674
            3  0.6580803688245 0.6962100440585

Responses to Shock in TB3MO
   Entry            TB1YR           TB3MO
            1   0.000000000000  0.243115941753
            2   0.049770290072  0.271975555549
            3   0.073515992407  0.180662886855
```

Hence, the impulse responses are identical to those obtained using the instruction:

impulse(model=term,result=impulses) * 24 * s

2. In the 3-variable VAR with *dlrgdp, dlrm2* and *drs*, we obtained impulse responses using:

```
system(model=chap2)
var dlrgdp dlrm2 drs
lags 1 to 12
det constant
end(system)
estimate(outsigma=v)

errors(impulses,model=chap2) 3 24 v
```

Now suppose we want the contemporaneous relationships among the variables to be:

$$e_{yt} = \varepsilon_{yt}$$
$$e_{mt} = g_{21}\varepsilon_{yt} + g_{31}\varepsilon_{rt} + \varepsilon_{mt}$$
$$e_{rt} = \varepsilon_{rt}$$

where: e_y, e_{mt} and e_{rt} are the regression residual from the $dlrgdp_t$, $dlrm2_t$ and drs_t equations, and ε_{yt}, ε_{mt} and ε_{rt} are the pure shocks (*i.e.*, the structural innovations) to $dlrgdp_t$, $dlrm2_t$ and drs_t, respectively.

The economic interpretation is that the 'unforecastable' change in the log of real *M2* (*i.e.*, e_{mt}) is due to the pure shocks in $dlrgdp_t$, drs_t and $dlrm2_t$. Hence, we have imposed a standard money demand function on the contemporaneous relationship among the three variables. Moreover, the 'unforecastable' portions of $dlrgdp_t$ and drs_t (*i.e.*, e_{yt} and e_{rt}) are due only to their own pure shocks. We can model these contemporaneous relationships as:

$$\begin{bmatrix} e_{mt} \\ e_{yt} \\ e_{rt} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ g_{21} & 1 & g_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \varepsilon_{yt} \\ \varepsilon_{mt} \\ \varepsilon_{rt} \end{bmatrix}$$

Notice that we have an over-identified system in that we have restricted four elements of *G* to be zero.[20] To perform this alternative decomposition, use the following instructions:

```
nonlin g21 g23
dec frml[rect] g_form
frml g_form = || 1., 0. , 0. | g21, 1. , g23 | 0., 0. ,1. ||
com g21 = -.2 , g23 = 0.3
cvmodel(factor=g) %sigma g_form
```

The NONLIN instruction informs RATS that we want to estimate the parameters g_{21} and g_{23}. The next instruction DECLARES the FORMULA *g_form*. The third instruction is used to specify the form of *g_form*. The fourth instruction provides initial guesses for the two parameters. Next, we use CVMODEL to perform the estimation:

cvmodel(factor=g) %sigma g_form

---

[20] Since $n = 3$, exact identification entails $(n^2 - n)/2 = 3$ restrictions.

```
Covariance Model - Estimation by BFGS
Convergence in    12 Iterations. Final criterion was  0.0000000 <  0.0000100
Observations                      156
Log Likelihood                      1706.46892511
Log Likelihood Unrestricted         1710.00278213
Chi-Squared(1)                         7.06771404
Significance Level                0.00784853

Variable         Coeff         Std Error        T-Stat       Signif
*************************************************************************
1.  G21        -0.247417925    0.062562114     -3.95476    0.00007661
2.  G23         0.002782858    0.000749196      3.71446    0.00020364
```

Notice that RATS displays the log likelihood of the restricted and the unrestricted models. Hence, we might want to relax one of the four restrictions since the difference between the log likelihoods is significant at the `0.00784853` level. Nevertheless, we can obtain the impulse responses using:

impulse(model=chap2,decomp=g) * 24 *

```
Responses to Shock in DLRGDP
   Entry            DLRGDP          DLRM2            DRS
            1   0.006525648724  0.001614562466  0.000000000000
            2   0.000674747862  0.001301182781  0.163282831262
            3   0.000810331530  0.000383503057  0.118501396316

Responses to Shock in DLRM2
   Entry            DLRGDP          DLRM2            DRS
            1   0.000000000000  0.004941736423  0.000000000000
            2   0.001074367700  0.003482474955 -0.044400414535
            3   0.001212253497  0.001927339570  0.007632342684

Responses to Shock in DRS
   Entry            DLRGDP          DLRM2            DRS
            1   0.000000000000 -0.001532083485  0.550543146080
            2   0.001203397095 -0.003442724093  0.228130866164
            3  -0.001598121579 -0.002104694466 -0.143506667217
```

To graph the impulse responses, we need to create a matrix to hold the nine series (there are three responses to each of the three shocks). The next two instructions creates a 3 x 3 matrix called *impulses*. Be aware that each element of impulses is a series. The RESULT= option informs RATS to store the nine impulse responses in *impulses*. Note that *impulses(1,1)* contains the responses of $dlrgdp_t$ to an ε_{yt} shock and *impulses(3,1)* contains the responses of $dlrgdp_t$ to an ε_{rt} shock.

```
declare rectangular[series] impulses(3,3)
impulse(model=chap2,result=impulses,decomp=g) * 24 *
```

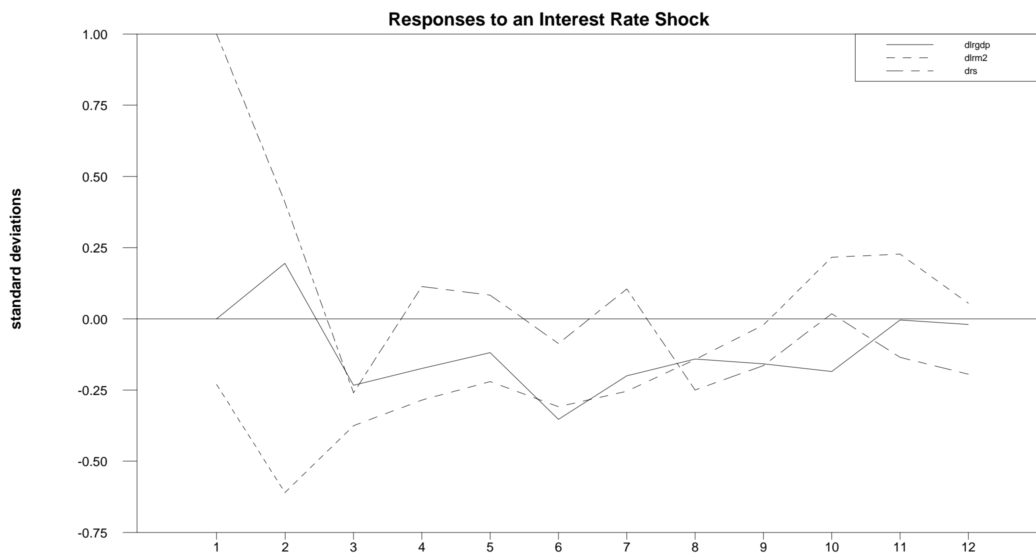
Since the variables have different units, it is useful to plot the standardized responses. Note that the first entry of *impulses(1,1)* contains the standard deviation of $dlrgdp_t$, first entry of

impulses(2,2) contains the standard deviation of $dlrm2_t$ and the first entry of *impulses(3,3)* contains the standard deviation of drs_t. Hence, we can standardize the responses of each variable to an ε_{rt} shock using:

```
set r1 1 12 = impulses(1,3)/impulses(1,1)(1)
set r2 1 12 = impulses(2,3)/ impulses(2,2)(1)
set r3 1 12 = impulses(3,3)/impulses(3,3)(1)
```

We can graph the three series using:

```
com implabels = || 'dlrgdp','dlrm2', 'drs'||
GRAPH(HEADER='Responses to an Interest Rate Shock',KEY=upright,patterns, $
number=1, klabels=implabels,vlabel='standard deviations') 3
# r1 ; # r2 ;  # r3
```



Hence, a one-standard deviation innovation in the 3-month *T*-bill rate is predicted to reduce real money balances and (after the second period) real *GDP*. Even though these results seem plausible, one caution is in order. Nonlinear estimations of a likelihood function may find a local not a global maximum. It is always wise to repeat the estimations using various initial guesses of the parameters to be estimated. In working through this example, I tried a number of initial guesses. It turns out that initial guesses very near zero lead to an unsatisfactory result. Consider the output from using initial guesses g_{21} = 0.02 and g_{23} = 0.01:

```
com g21 = .02 , g23 = 0.1
cvmodel(factor=g) %sigma g_form
```

```
Covariance Model - Estimation by BFGS
Convergence in     3 Iterations. Final criterion was  0.0000000 <  0.0000100
Observations                    156
Log Likelihood                    1696.18382404
Log Likelihood Unrestricted       1710.00278213
Chi-Squared(1)                      27.63791617
Significance Level              1.46283041e-07

Variable                     Coeff       Std Error      T-Stat     Signif
*****************************************************************************
1.  G21                   0.0431096354 0.0000000000      0.00000  0.00000000
2.  G23                   0.0022092447 0.0007865452      2.80880  0.00497272
```

Notice that the routine did converge. However, the standard error and *t*-statistic of g_{31} are both shown to be zero. Moreover, the log likelihood of the restricted model is smaller than that for initial guesses g_{21} = -.2 , g_{23} = 0.3. I experimented with a wide range of initial guesses, and usually obtained the first set of estimations. Hence, it seems reasonable to conclude that initial guesses near zero lead to a local, not a global, maximum.

6. The Blanchard-Quah Decomposition

Blanchard and Quah (1989) provide an alternative way to obtain a structural identification. Let $\{y_t\}$ be a difference-stationary series and let $\{z_t\}$ be stationary. Ignoring any deterministic regressors, we can estimate a 2-variable VAR of the form:

$$\Delta y_t = \sum_{i=1}^{p} a_{11}(i)\Delta y_{t-i} + \sum_{i=1}^{p} a_{12}(i) z_{t-i} + e_{1t}$$
$$z_t = \sum_{i=1}^{p} a_{21}(i)\Delta y_{t-i} + \sum_{i=1}^{p} a_{22}(i) z_{t-i} + e_{2t}$$

In order to use the Blanchard-Quah technique, both variables must be in a stationary form. Since $\{y_t\}$ is $I(1)$, we use the first-difference of the series. If, in your own work, you find $\{z_t\}$ is also $I(1)$, use its first-difference in the VAR.

In contrast to the Sims-Bernanke procedure, Blanchard and Quah do not directly associate the structural variables $\{\varepsilon_{1t}\}$ and $\{\varepsilon_{2t}\}$ with pure shocks to $\{y_t\}$ and $\{z_t\}$. Instead, the $\{y_t\}$ and $\{z_t\}$ sequences are the endogenous variables and the $\{\varepsilon_{1t}\}$ and $\{\varepsilon_{2t}\}$ sequences represent what an economic theorist would call the exogenous variables. The structural variables are assumed to be uncorrelated with each other and to have unit variances.

Although the structural variables are unobserved, they are related to the regression residuals by:

$$\begin{bmatrix} e_{1t} \\ e_{2t} \end{bmatrix} = \begin{bmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \end{bmatrix} \begin{bmatrix} \varepsilon_{1t} \\ \varepsilon_{2t} \end{bmatrix}$$

Changes in ε_{2t} will have no long-run effect on the $\{y_t\}$ sequence if:

$$g_{12}\left[1 - \sum_{k=0}^{p} a_{22}(k)\right] + g_{22}\sum_{k=0}^{p} a_{12}(k) = 0$$

This long-run restriction provides the extra piece of information that allows us to identify the four elements G matrix. Given the relationship between the regression residuals and the structural variables, it follows that:

$$\text{var}(e_1) = (g_{11})^2 + (g_{12})^2$$
$$\text{var}(e_2) = (g_{21})^2 + (g_{22})^2$$
$$\text{cov}(e_1 e_2) = g_{11}g_{21} + g_{21}g_{22}$$

where: the time subscripts have been omitted since Δy_t and z_t are assumed to be covariance stationary.

Estimation of the VAR provides you with var(e_1), var(e_2), cov(e_1e_2) and the coefficient sums 1 - $\Sigma a_{11}(k)$ and $\Sigma a_{12}(k)$. Hence there are four equations that allow you to solve for the four unknowns g_{11}, g_{12}, g_{21} and g_{22}. Once the G matrix is identified, it is possible to obtain the impulse responses and variance decompositions using the ERRORS and IMPULSES instructions.

6.1 The Technical Details

In the n-variable case, $(n^2 - n)/2$ restrictions are needed for the exact identification of G. RATS contains a very simple mechanism that allows you to impose $(n^2 - n)/2$ long-run restrictions. Let $\mathbf{x}_t$, $\boldsymbol{e}_t$ and $\boldsymbol{\varepsilon}_t$ be the n x 1 vectors of variables, regression residuals and structural shocks, respectively. The estimated VAR has the form:

$$\mathbf{x}_t = A(L)\,\mathbf{x}_{t-1} + \boldsymbol{e}_t$$

or:

$$(1 - A(L)\,L)\,\mathbf{x}_t = \boldsymbol{e}_t$$

where: $A(L) = n$ x n matrix with elements $A_{ij}(L)$ and $A_{ij}(L)$ is p-th order polynomial in the lag operator L.

Given that the variables in $\mathbf{x}_t$ are stationary, we know there exists a moving average representation of the form:

$$\begin{aligned}\mathbf{x}_t &= C(L)^{-1}\,\boldsymbol{e}_t \\ &= C(L)^{-1}\,G\boldsymbol{\varepsilon}_t\end{aligned}$$

where: $C(L)$ is $(1 - A(L)\,L)$ and G is the n x n matrix relating the regression residuals and the structural shocks.

Now let the variables in $\mathbf{x}_t$ be arranged such that only ε_{1t} has a long-run effect on x_{1t}, only ε_{1t} and ε_{2t} have a long-run effect on x_{2t}, only ε_{1t}, ε_{2t} and ε_{3t} have a long-run effect on x_{3t} and so on. Notice that there are exactly $(n^2 - n)/2$ such restrictions. Since the coefficient sums are obtained from $C(1)^{-1}\,G$, these restrictions translate into the assumption that each element above the principle diagonal in $C(1)^{-1}\,G$ be zero.[21] The key point to note is that we can impose these restrictions on $C(1)^{-1}\,G$ from a Choleski decomposition of $C(1)^{-1}\,GG'\,(C(1)^{-1})'$.

---

[21] If we evaluate $A_{11}(L) = a_{11}(0) + a_{11}(1)L + a_{11}(2)L^2 + a_{11}(3)L^3 + \ldots$ at $L = 1$, we obtain the coefficient sum $\Sigma a_{11}(k)$.

Given the relationship between the regression residuals and the structural variables, it follows that:

$$E\boldsymbol{e}_t\boldsymbol{e}_t' = GG'$$

Yet, $E\boldsymbol{e}_t\boldsymbol{e}_t'$ is precisely the variance/covariance matrix of the regression residuals that we have called Σ. Thus, to obtain $C(1)^{-1}G$, we need only to obtain the Choleski decomposition of $C(1)^{-1}\Sigma\,(C(1)^{-1})'$.

Once the VAR has been estimated, you can obtain the desired matrix using:

```
COMPUTE C = %VARLAGSUMS
COMPUTE S1 = %MQFORM(%SIGMA,TR(INV(C)))
COMPUTE S2 = %DECOMP(S1)
COMPUTE G = C*S2
```

To explain, when you use the ESTIMATE instruction RATS creates the matrix %VARLAGSUMS containing the n x n matrix of the appropriate sums of the lag coefficients. Hence, the first COMPUTE instruction creates a matrix C corresponding to the matrix $C(1)$ in the discussion above. The function %MQFORM(X, Y) creates a matrix equal to $Y'XY$ for $Y_{n\ x\ n}$ and $X_{n\ x\ m}$. INV(C) creates C^{-1} and TR(INV(C)) creates the transpose of C^{-1}. Hence, the second instruction creates $(C^{-1})'\Sigma(C^{-1})'$. The function %DECOMP(S1) creates the matrix $S2$ equal to Choleski decomposition of $(C^{-1})'\Sigma C^{-1} = C^{-1}G$. Multiplication by C yields the factorization containing the desired form of G. At this point it is possible to obtain the impulse responses and variance decompositions using the DECOMP=G option on an ERRORS or IMPULSES instruction.

Example

The 3-variable VAR with $dlrgdp_t$, $dlrm2_t$ and drs_t is in the appropriate form since all of the variables appear to be difference stationary. Although there is strong evidence that a 12-lag model is appropriate, it is instructive to estimate the system using only 1-lag. The goal here is to illustrate the creation of the desired matrices and coefficient sums; these sums are trivial to calculate in a 1-lag model. As such, re-estimate the 3-variable VAR using the following instructions:

```
system(model=chap2)
var dlrgdp dlrm2 drs
lags 1
det constant
end(system)
estimate(outsigma=v,noftests)
```

```
VAR/System - Estimation by Least Squares
Dependent Variable DLRGDP
   Variable           Coeff        Std Error         T-Stat    Signif
*****************************************************************************
1.  DLRGDP{1}      0.1006658743   0.0783973356      1.28405  0.20092557
2.  DLRM2{1}       0.3655350277   0.0719395585      5.08114  0.00000101
3.  DRS{1}         0.0015984145   0.0008299146      1.92600  0.05582371
4.  Constant       0.0047686768   0.0009110037      5.23453  0.00000050

Dependent Variable DLRM2
    Variable          Coeff        Std Error       T-Stat      Signif
*****************************************************************************
1.  DLRGDP{1}      0.060434793    0.061966497      0.97528    0.33084820
2.  DLRM2{1}       0.629182216    0.056862167     11.06504    0.00000000
3.  DRS{1}        -0.003657927    0.000655978     -5.57630    0.00000010
4.  Constant       0.002338028    0.000720072      3.24694    0.00141302

Dependent Variable DRS
   Variable           Coeff        Std Error       T-Stat      Signif
******* *********************************************************************
1.  DLRGDP{1}     12.09953383     7.74975243       1.56128    0.12037364
2.  DLRM2{1}       7.98709105     7.11138667       1.12314    0.26300908
3.  DRS{1}         0.15367238     0.08203892       1.87316    0.06281438
4.  Constant      -0.15426829     0.09005476      -1.71305    0.08858167
```

The ESTIMATE instruction creates the 3 x 3 matrix %VARLAGSUMS. We can display this matrix using:

dis %varlagsums

```
     0.89933      -0.36554      -0.00160
    -0.06043       0.37082       0.00366
   -12.09953      -7.98709       0.84633
```

Notice that the coefficient on *dlrgdp*$_{t-1}$ in the first equation [*i.e.*, $a_{11}(1)$] equals `0.1006658743` and the first element of %varlagsums is `0.89933`. Hence, %varlagsums(1, 1) = $1 - a_{11}(1)$. Similarly %varlagsums(2, 2) and %varlagsums(3, 3) correspond to $1 - a_{22}(1)$ and $1 - a_{33}(1)$, respectively. The off-diagonal elements %varlagsums(*i* , *j*) equal $-a_{ij}(1)$.

Next, create the matrices *C*, *s1* and *s2* using the following three instructions:

```
com c = %VARLAGSUMS
com s1 = %MQFORM(%SIGMA,TR(INV(c)))
com s2 = %DECOMP(S1)
```

Notice that *s2* corresponds to $C(1)^{-1}G$—we want each element above the principle diagonal to be zero. We can display this matrix using:

dis s2

```
  0.01222       0.00000       0.00000
  0.01210       0.01518       0.00000
  0.18724      -0.51647       0.63922
```

Next, we can compute and display G using:

com g = C*S2 ; dis g

```
   0.00626      -0.00472      -0.00102
   0.00443       0.00374       0.00234
  -0.08599      -0.55838       0.54099
```

The impulses responses can be obtained using:

```
impulses(decomp=g,model=chap2) * 24 *
```

6.2 Decomposing GDP, Real M2 and the Interest Rate

The neoclassical macroeconomic model suggests that aggregate demand shocks can have short-run, but not long-run, effects on economic real variables. As such, the Blanchard-Quah decomposition is ideally suited for analyzing the effects of various shocks on key macroeconomic variables. Let ε_{ft}, ε_{mt} and ε_{pt} represent a fiscal policy shock, a monetary policy shock and a productivity shock, respectively. In terms of our 3-variable VAR, we might suppose that fiscal shocks and monetary shocks have no long-run effects on real *GDP*. Thus, we have two of the requisite three restrictions. To obtain a third restriction, it might be argued that fiscal shocks have no long-run effect on real money balances.

The relationship among the regression residuals and the structural shocks is:

$$\begin{bmatrix} e_{1t} \\ e_{2t} \\ e_{3t} \end{bmatrix} = \begin{bmatrix} g_{11} & g_{12} & g_{13} \\ g_{21} & g_{22} & g_{23} \\ g_{31} & g_{32} & g_{33} \end{bmatrix} \begin{bmatrix} \varepsilon_{pt} \\ \varepsilon_{mt} \\ \varepsilon_{ft} \end{bmatrix}$$

Now, restricting the elements of G such that $C(1)^{-1}G$ has all elements above the principal diagonal equal to zero, is identical to assuming that monetary and fiscal policy shocks have no long-run effect on the $\{dlrgdp_t\}$ sequence and that fiscal shocks have no long-run effect on the $\{dlrm2_t\}$ sequence. Estimate the 3-variable VAR using all 12 lags.

You can create the appropriately restricted G matrix using:

```
compute g=%varlagsums*%decomp(%mqform(%sigma,tr(inv(%varlagsums))))
```

As in earlier programs, the impulse responses will be saved in a 3 x 3 matrix called *impulses*. The next two lines instruct RATS to create this matrix and to obtain the impulse responses using the decomposition of G:

```
declare rectangular[series] impulses(3,3)
impulses(model=chap2,result=impulses,decomp=g) * 24 *
```

```
Responses to Shock in DLRGDP
   Entry           DLRGDP          DLRM2            DRS
         1   0.003010662081  0.004756093316 -0.263743931599
         2   0.000447294073  0.004559080262 -0.063402066434
         3   0.001943388494  0.002463374844  0.128481679716

Responses to Shock in DLRM2
   Entry           DLRGDP          DLRM2            DRS
         1  -0.003711849678  0.002312008411  0.137086248729
         2   0.000701092521  0.000947945412 -0.068523721029
         3   0.000027167701  0.000666465186 -0.097559696296

Responses to Shock in DRS
   Entry           DLRGDP          DLRM2            DRS
         1   0.004443216973  0.000606481672  0.463405065412
         2   0.001645570241 -0.001450397016  0.296041401790
         3  -0.000597985872 -0.001199710195 -0.038876550562
```

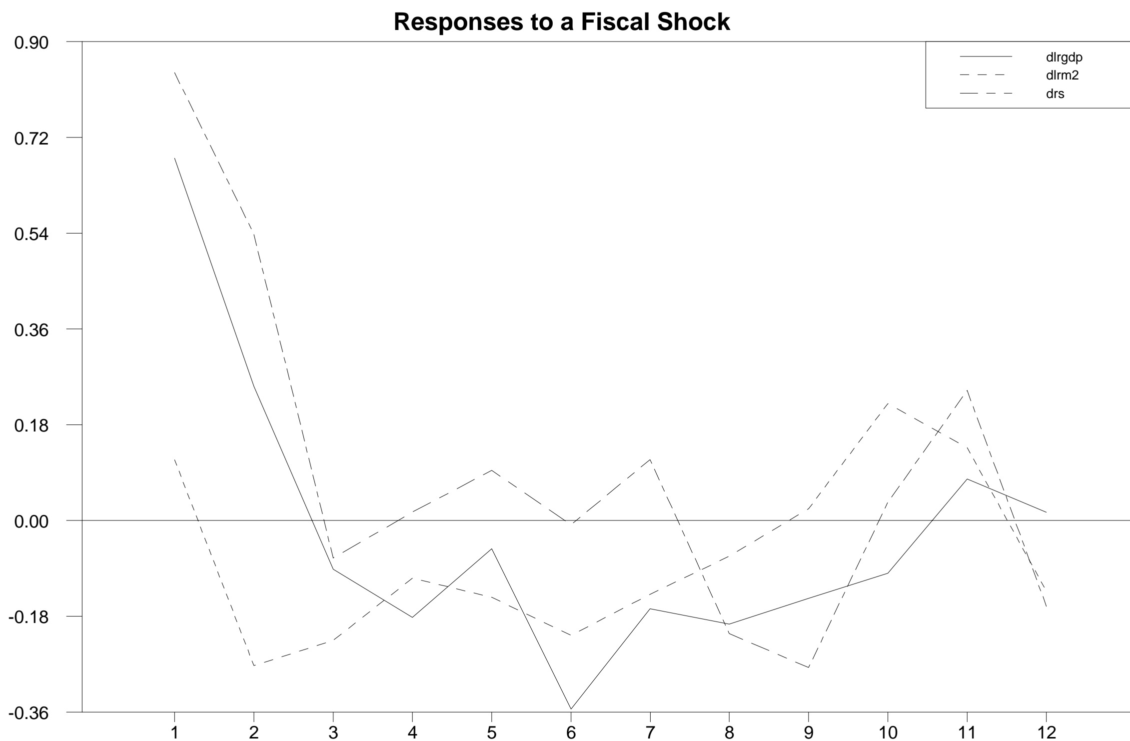
Since we normalized the shocks to have unit-variances, the interpretation of the absolute magnitudes of the impulse responses is unclear. We can scale the responses of each variable in terms of standard deviations. The scaled responses to a fiscal policy shock are obtained using:

```
set r7 1 12 = impulses(1,3)/%sigma(1,1)**.5
set r8 1 12 = impulses(2,3)/%sigma(2,2)**.5
set r9 1 12 = impulses(3,3)/%sigma(3,3)**.5
```

A graph of the scaled responses is obtained from:

```
com implabels = || 'dlrgdp','dlrm2', 'drs'||
graph(header ='Responses to a Fiscal Shock', key=upright, number=1, $
  klabels=implabels,patterns) 3
# r7 ; # r8 ;  # r9
```

Initially, the fiscal shock acts to increase $dlrgdp_t$, however, by the third quarter $dlrgdp_t$ is negative. By construction, the cumulated change in $dlrgdp_t$ zero. Similarly, the fiscal shock is estimated to create a sharp increase in the short-term interest rate. Note that drs_t is positive for the first two quarters. Thereafter, drs_t seems to fluctuate around zero so that the cumulated change in the 3-month *t*-bill rate is positive. This is possible since we did not impose any restriction concerning the effect of the fiscal shock on drs_t.



Chapter 3:
Loops Over Dates and Series

Try this little program:[22]

```
all 10
scratch 1
set 1 = 5
pri / 1
```

Notice that the program does not use the RATS' CALENDAR Instruction. Instead, line 1 instructs RATS to set the default length of a series to 10—by default, any series will have 10 observations. The second instruction instructs to create one series. The third line instructs RATS to set 1 equal to 5. This may seem nonsensical at first, and I will explain the meaning in more detail below. However, since RATS does not display an error message, it must somehow set 1 equal to 5. Perhaps, you can figure out the dilemma by entering line 4 (line 4 instructs RATS to PRINT over the default range). Your output should look like:

```
ENTRY      No Label(1)
   1           5
   2           5
   3           5
   4           5
   5           5
   6           5
   etc.
```

Now you can make sense of the program. There is a series—called 1—that has a length of 10. Each value of the series (i.e., entries 1 through 10) is equal to the number 5. The point is that the series called 1 does not have a label like y, gdp or inflation. It is series number "1" because it is the first series that has been created. Similarly, the entries of the series are not dates like 99:2 or 2001:3. Instead of using dates, RATS numbers each value 1 through 10.

You might want to think of series 1 as a 10 x 1 vector.[23] Each element in the vector has the value of 5. The point is not that RATS can represent a series name by a number and a calendar date by a number. Instead, RATS always represents a series by a number. This is true regardless of whether or not you attach a name or label to the series. You are allowed to attach a label to a series for convenience. Similarly, RATS always represents each calendar date by a number. You are allowed to attach a date label like 99:2 or 2001:3 for your own convenience. This is true regardless of whether or not you use the CALENDAR instruction.

[22] For your convenience, the programs illustrated in this section can all be found on the file labeled CHAPTER3_1.PRG.
[23] As explained when we discuss matrices, within RATS Programming Language, there is a difference between a series and a declared vector.

1. Dates as Integers

In case you skipped over the last two chapters, note that the file labeled MONEY_DEM.XLS contains a number of variables in Excel format. If you open the file labeled CHAPTER3_1.PRG, you will find Program 3.2; open the file and read in the data set by entering the following four lines:

```
cal 1959 1 4
all 2001:1
open data a:\money_dem.xls            ;* Modify this line if your data is not on drive a:\
data(org=obs,format=xls)
```

Next, print out the four values of real GDP (rgdp) from 1959:1 through 1959:4 using:

```
pri 1959:1 1959:4 rgdp
       ENTRY     RGDP
       1959:01  2273.0
       1959:02  2332.4
       1959:03  2331.4
       1959:04  2339.1
```

Now try using:

```
pri(nodates) / rgdp
       ENTRY      RGDP
           1      2273.0
           2      2332.4
           3      2331.4
           4      2339.1
         ...       ......
         165      9191.8
         166      9318.9
         167      9369.5
         168      9393.7
         169      9439.9
```

Thus, rdgp is one-dimensional array containing 169 observations or entries. As you can see, entry 1 is equivalent to 1959:1, entry 2 is equivalent to 1959:2, … and 2001:1 is entry 169. In fact, you can substitute the integers for the date labels whenever you find it convenient. For example, you can obtain first four values using:

```
pri 1 4 rgdp
       ENTRY     RGDP
       1959:01  2273.0
       1959:02  2332.4
       1959:03  2331.4
       1959:04  2339.1
```

If you do not want the date labels use:

```
pri(nodates) 1 4 rgdp
      ENTRY   RGDP
        1       2273.0
        2       2332.4
        3       2331.4
        4       2339.1
```

What about the series *tb1yr*? Recall from Chapter 1, that the first two observations for this series were *NA* (or missing). In the language of econometricians, *tb1yr* contains only 167 observations. Nevertheless, in RATS Programming Language, *tb1yr* contains the same number of **entries** (*i.e.*, 169) as all of the other series in the data set. Entries 1 and 2 are simply recorded as *NA*. If you PRINT entries 1 to 4 and entries 165 to 169 you will obtain:

print(nodates) 1 4 tb1yr

```
     ENTRY        TB1YR
        1           NA
        2           NA
        3      4.493333333333
        4      4.740000000000
```

print(nodates) 165 169 tb1yr

```
      ENTRY        TB1YR
       165      5.816666666667
       166      5.856666666667
       167      5.803333333333
       168      5.630000000000
       169      4.416666666667
```

Thus, you can refer to value of the 1-year T-bill rate in 2001:1 using either tb1yr(2000:1) or tb1yr(169). You can DISPLAY the equivalent specifications if you enter:

```
dis tb1yr(2001:1)  tb1yr(169)
    4.41667       4.41667
```

1.1 Omitting CALENDAR

Given that all of the series in MONEY_DEM.XLS all have 169 entries, you could read in the data set using:

```
all 169
open data a:money_dem.xls
data(org=obs,format=xls)
```

In fact, if you want to refer to entries only by number, rather than by date label, you never use the CALENDAR instruction. However, you would not be able to use date labels. Thus, using CALENDAR gives you the choice of using date labels or entry values. You cannot use date labels if you omit CALENDAR.

Examples:

1. **Date arithmetic:** Since 2001:1 = 169, it follows that 2001:1-8 =161 is equivalent to 1999:1. Hence, to print the last two years of the *rgdp* series you can use:

   ```
   pri 2001:1-8 * rgdp
   ```

   Note that RATS will perform the date arithmetic 2001:1-8 if you do not use spaces adjacent to the minus sign.

2. Estimate an *AR*(1) autoregression of the logarithmic change in *rgdp* using the first 100 observations:

   ```
   set dlrgdp = log(rgdp) – log(rgdp{1})
   lin dlrgdp * 100
   # constant dlrgdp{1}
   ```

   The first line creates the logarithmic change of *rgdp*. The second line prepares RATS to estimate a regression with *dlrgdp* as the dependent variable such that the last sample point is observation 100 (Note: A second observation will be lost as a result of the lagged change). The asterisk instructs RATS to use the default value for the *start* entry.

3. Estimate an *AR*(1) autoregression of the logarithmic change in *rgdp* using the last 100 observations and save the residuals in the series *resids*:

   ```
   lin dlrgdp 2001:1-99 * resids
   # constant dlrgdp{1}
   ```

   Now line 2 instructs RATS to estimate the regression using the start date beginning at 100 observations from the end of the data set. Note that if you want RATS to perform the date arithmetic 2001:1–100, you cannot put a space on either side of the minus sign. Also note that you will get precisely the same output using:

   ```
   lin dlrgdp 69 * resids
   # constant dlrgdp{1}
   ```

   The reason that the two are equivalent is that there are 169 observations for *rgdp*. Observation 69 is identical to 2001:1–100. If you take the time to count, you can verify that both of these entries are equivalent to 1976:1. Fortunately, you never have to actually count the number that is equivalent to a particular date label. In fact, RATS provides a number of instructions that are helpful for using date manipulation. The two most useful ones are:

%CAL(YEAR,PERIOD) = The entry number PERIOD of YEAR.
%DATELABEL(T) = The date string (*e.g.*, 1991:3) corresponding to the entry value.

4. To find the date label of the observation 100 periods before 2001:1 use:

```
dis %DATELABEL(2001:1-100)
```

Your output will be: `1976:01`. (Note: In performing date arithmetic, RATS does not allow you use spaces inside the %DATELABEL function; hence, in the instruction above, you cannot place a space on either side of the minus sign). You can also use %DATELABEL with any integer value. For example, if you insist on putting spaces next to the minus sign, you can use:

```
com i = 2001:1 - 100
dis %DATELABEL(i)
```

Similarly, **dis %cal(1976,1)** yields:

```
69
```

2. Series as Integers

Just as each calendar date has an associated entry value, each series has its own sequence number. If continue to use Program 3.2 and type TABLE, you will see:

```
Series   Obs       Mean        Std Error       Minimum        Maximum
DATE     169   1979.876331     12.232185   1959.100000    2001.100000
GDP      169   3572.739053   2873.158128    496.100000   10243.600000
RGDP     169   5142.364497   1950.840494   2273.000000    9439.900000
M2       169   1904.835266   1399.706717    287.800000    5043.710000
M3       169   2414.462229   1916.764710    290.053333    7260.136667
TB3MO    169      5.915148      2.590483      2.303333      15.053333
TB1YR    167      6.153872      2.393622      2.713333      14.380000
DLRGDP   168      0.008475      0.008960     -0.020598       0.037804
RESIDS   101      0.000000      0.007737     -0.027190       0.031366
```

Notice that the series in MONEY_DEM.XLS are in order, followed by the dlrgdp series you created. The series numbers are such that date is series 1, gdp is series 2, rgdp is series 3. The series dlrgdp created with the SET instruction, is series 8. As you create additional series, RATS stores each consecutively. Thus, the series resids created by the LINREG instruction is series 9.

As discussed above, you can just use the series number instead of its label. Anywhere RATS expects a series name, you can simply use the sequence number. You do need to make sure that you reference sequence numbers as integers and not floating point numbers.
Thus, you can print out the first values of rgdp using:

pri 1 4 3

Recall that the syntax for the PRINT instruction is PRINT start end series list. Thus, pri 1 4 3 instructs RATS to print, from entry 1 through 4, the values of series 3. If you follow the logic, you know that it is possible to print the first four values of rgdp and dlrgdp using:

pri 1 4 3 8

```
ENTRY          RGDP            DLRGDP
1959:01        2273.0            NA
1959:02        2332.4       0.025797235495
1959:03        2331.4      -0.000428834862
1959:04        2339.1       0.003297294498
```

Care must be taken if a series is on the right-hand side of a FRML, SET or COM instruction since RATS will interpret the integer as a scalar. In fact, whenever it is ambiguous, you can force RATS to use the series instead of an integer if you use: [series]number.

Examples:

1. To print the last two years of the *rgdp* series you can use:

pri 2001-7 * 3

2. Estimate an *AR*(1) autoregression of the logarithmic change in *rgdp* using the first 100 observations:

```
set 8 = log(rgdp) – log(rgdp{1})
lin 8 * 100
# constant 8{1}
```

As before, the first line creates the logarithmic change of *rgdp*. However, there is no label attached to the series—it is just referred to as "8". The second line prepares RATS to estimate a regression with series 8 as the dependent variable such that the last sample point is observation 100. The third line instructs RATS to include a constant and the lagged value of series 8 in the regression.

3. set y = log(2) versus set y = log([series]2)

The first instruction sets each entry of *y* equal to the natural log of 2; hence, all values of *y* are 0.69315. The second statement sets each entry of *y* equal to the natural log of the corresponding entry of series 2. Suppose that the first four entries of series 2 are 1, 4, 2 and 6. The second statement sets the first four values of *y* to be: 0, 1.38629, 0.69315 and 1.79176.

4. Suppose that the series *y* is the second series in RATS' memory. All of the following create the growth rate of y:

```
set gy = log(y) - log(y{1})
set gy = log([series]2) - log(([series]2){1})
set gy = log(2{0}) - log(2{1})
```

The first instruction creates *gy* as the log of the current value of *y* less the lag of the previous period's log of *y*. The second instruction uses square brackets to distinguish between the number 2 and series number 2. Notice that it is necessary to use the construction log(([series]2){1}); log([series]2{1}) creates an error message. The third instruction is correct because the use of lag notation ensures that there is no ambiguity in the meaning of 2{0} and 2{1}. However, log(2) - log(2{1}) would not produce the desired effect.

5. The very first program in this chapter contained the line set 1 = 5. This set all entries of series 1 equal to the number 5.

2.1 Creating Numbered Series and Labels

Since RATS allows you work with a series using its label or its integer value, you will want to become familiar with creating numbered series, assigning a label to a series, fetching the integer value of a series from its label, and fetching the label of a series from its integer value.

In principal, you do not need to assign a label to a series. However, labels make it easier to remember recall the various steps in your program and to interpret your output. One way to create series is to use ALLOCATE instruction with the optional *series* field. Consider the first five lines of Program 3.3 on the file labeled CHAPTER3_1.PRG:

```
cal 1959 1 4
all 4 2001:1                    ;* <<< Note that line 2 is modified
open data a:\money_dem.xls
data(org=obs,format=xls)
```
tab

```
Series          Obs       Mean        Std Error       Minimum        Maximum
No Label(1)       0
No Label(2)       0
No Label(3)       0
No Label(4)       0
DATE            169  1979.8763314    12.2321846  1959.1000000   2001.1000000
GDP             169  3572.7390533  2873.1581276   496.1000000  10243.6000000
RGDP            169  5142.3644970  1950.8404937  2273.0000000   9439.9000000
. . .
TB3MO           169     5.9151479     2.5904835     2.3033333     15.0533333
TB1YR           167     6.1538723     2.3936222     2.7133333     14.3800000
```

As in all the other programs considered, line 2 sets the default series length to 2001:1. However, the value *4* in the *series* field instructs RATS to create a block of series numbered 1 to 4. Notice that the integer values assigned to the seven series contained on MONEY_DEM.XLS now begin with 5 and end with 11. At this point in the program, series 1 through 4 have no label. You can assign a name to each using the EQV instruction. The syntax of EQV (for EquiValance) is:

EQV *integer values of series*
list of names for series

For example, we can assign series 1, 2, 3 and 4 the names resids1, resids2, resids3, and resids4 using:[24]

```
eqv 1 to 4
resids1 resids2 resids3 resids4
```

After entering these two lines, a table instruction produces the names of the four series:

tab / 1 to 4

[24] Notice there are no commas between the labels and there is no # symbol on the list of labels.

```
Series         Obs       Mean        Std Error      Minimum        Maximum
RESIDS1          0
RESIDS2          0
RESIDS3          0
RESIDS4          0
```

You can use the names created with EQV for input and for output. An alternative is to assign a label to a series, using the LABELS instruction. The syntax for LABELS is:

labels *list of series numbers*
'labels' for the series (each label in quotation marks)

Hence, to assign the labels resids1, resids2, resids3, and resids4 to series 1 through 4 use:

labels 1 to 4
‘resids1’ ‘resids2’ ‘resids3’ ‘resids4’

Notice that each label is enclosed in single or double quotation marks and that you use the # symbol to begin the supplementary card for LABELS. There is an important distinction between EQV and LABELS. EQV produces a name that can be used for manipulations within a program. However, EQV cannot be used within a compiled section of a program. LABELS attaches an output label to a series that RATS displays when printing a series or writing a series to a data file. However, you cannot manipulate the series using its output label. The main reason to use an output label is to display strings that cannot be created with the SET or EQV instructions (e.g., spaces or a mix of upper case and lower case letters).

The SCRATCH instruction provides an alternative way to create consecutively numbered series. Unlike ALLOCATE, the SCRATCH instruction assigns series numbers beginning with the highest unused integer value. Since *tb1yr* is assigned the integer value of 11, any new series created by SCRATCH will begin with the integer value of 12. The simplest way to create series from SCRATCH is to use:

scratch *number start end scr_no*

where:

number	The number of series to create
start end	The range of entries to allocate to the series
scr_no	An integer variable equal to the number of existing series prior to the execution of SCRATCH. Hence, *scr_no* + 1 contains the integer value of the first series created by SCRATCH.

The next line of Program 3.3 creates two additional series and uses b to hold the value of scr_no. The TABLE instruction shows the order of the series in RATS’ memory. Since b = 11, the summary statistics for tb1yr and the two newly created series are displayed. At this point, the EQV or LABELS instructions can be used to assign labels to the two new series. Note that any additional series created will begin with integer number 14.

scratch 2 / b
tab / b to b+2

```
Series         Obs        Mean       Std Error       Minimum        Maximum
TB1YR          167  6.1538722555  2.3936221961  2.7133333333 14.3800000000
No Label(12)   0
No Label(13)   0
```

Retrieving Labels and Integer Numbers

The simplest way to retrieve the integer value assigned to a series is to use the COMPUTE instruction.[25] You COMPUTE an integer value to be equal to the name of a series:

com integer = series name.

For example, to display the integer value assigned to rgdp use:

com num = rgdp
dis num

At first, this instruction makes no sense since num is a number and rgdp is a series (recall that COMPUTE is not used to set entire series). Since RATS expects a number on the right hand side, it will equate num with the series number assigned to rgdp. Thus, after entering dis num in Program 3.3, you will see the integer 7 displayed on the screen.

The function %L(number) returns the LABEL attached to the specified variable. Hence:

dis %l(7)
RGDP

dis %l(rdgp)
RGDP

Also note that you can use com a$ = %l(7) to assign the string ‘RGDP’ to the string variable a$. Once a$ has been computed, it can be manipulated using the various string handling instruction provided with RATS.

[25] The function %s(label) also returns the series number whose name is label. Be sure to include the label in single or double quotation marks. To display the series number associated with rdgp use: com ii = %s(‘rgdp’) ; dis ii

3. Do Loops

The DO loop is a very simple way to automate many of your repetitive programming tasks. The usual structure of a DO loop is:

```
do i = 1,n
  program statements
end do i
```

The structure is such that RATS will perform all program statements within the DO loop exactly n times. The first time the program statements are executed, the counter i contains the integer value 1. On reaching the end of the loop, the counter i is incremented by 1 (i.e., i = 2). If i is less than or equal to n, the block of program statements is executed again. On reaching the end of the loop, the value of i is incremented by 1, compared to n, and if i ≤ n, the loop is executed once more. After the n-th loop i = n+1 and the instruction following the loop is executed. Consider the example:

```
do i = 1,5
  dis i
end do i
```

```
      1
2
3
4
5
```

More generally, you can use:

```
do integer = n1, n2, increment
  program statements
end do integer
```

where: n1, n2 and increment are integers. To understand how the loop performs consider

```
do i = 1,5,2
```

Here, i begins at 1 and is incremented by 2 every time the loop is completed. After three loops, i = 6 so that the loop is not performed a fourth time.

```
do i = 6 , -3, -2
```

Here i begins at 6 and is decreased by 2 every time the loop is completed. At the end of the first loop, i = 4 and the loop is completed a second time. At the end of the 5th loop, i = -4 so the loop is not completed a sixth time.

3.1 DO Loops, Switches and Choices

Since all Switch and Choice options have an integer representation, they can be selected by the index of a DO loop. Consider the following:

Examples:

1.
```
do i = 0,1
  lin(robusterrors=i) drs
  # constant drs{1 to 7}
end do i
```

In Chapter 1, the change in the short-term interest rate (*drs*) was estimated as an *AR*(7) with and without the ROBUSTERRORS option. The program segment estimates an *AR*(7) model with and without ROBUSTERRORS. Recall that ROBUSTERRORS is OFF when its value equals 0 and is ON when its value equals 1.

2. Any RATS instruction containing supplementary cards in a regression format allows you to use the ENTRIES option. The syntax for the option is:

entries = *number of entries to process*

For example, the instructions below will produce three regression equations. The first will regress *y* on a constant, the second will regress *y* on a constant and *x*, and the third will regress *y* on a constant *x* and *z*.[26]

```
do i = 1,3
  lin(entries=i) y
  # constant x z
end do i
```

3. The RATS procedure BJIDENT.SRC will display the autocorrelation and partial autocorrelation functions for the series you specify. After compiling the procedure, the syntax to EXECUTE the procedure is:

execute bjident(options) *series start end*

or, if you use @ as a shortcut for EXE,

---

[26] You need to be a bit careful since any object you include on the supplementary card (except #, $ or a space) is counted as an entry. Thus, dlrgdp{1} is counted as four entries: dlrgdp, {, 1, and }. Similarly, dlrgdp{1 to p} is counted as 6 entries: dlrgdp, {, 1, to, and p}.

@bjident(options) *series start end*

where:

start end	The range of the series to use for constructing the autocorrelations and partial autocorrelations

The most useful options are:

DIFF=	Maximum regular differencings[0]
SDIFFS=	Maximum seasonal differencings[0]
TRANS=[NONE]/LOG/ROOT	Transformation to apply to data
[GRAPH]/NOGRAPH	Do High-resolution graphs?
SPAN=	Seasonal span

Since TRANS has three choices, you can obtain the *ACF* and *PACF* of a series *y*, the log of *y* and the square root of *y* using:

```
do i = 1,3
  @bjident(trans=i) y
end do i
```

Similarly, you can obtain the *ACF* and *PACF* of *y* and its first and second differences using:

```
do i = 0, 2 ; @bjident(diff = i) ; end do i
```

You can also manipulate the *start* or *end* entry. For example, to obtain the *ACF* and *PACF* of *y* using observations 1 though 100, 150 and 200, use:

```
do i = 100,200,50 ; @bjident y * i ; end do i
```

3.2 Lag Length Tests

Now scroll down Program 3.3, and form the growth rate of real GDP using:

```
set dlrgdp = log(rgdp) – log(rgdp{1})
```

Instead of estimating the series as an AR(1), suppose that we want to ascertain the number of lags to use in an AR(p) representation. Suppose that you believe that the maximum possible lag length is 12.

```
do p = 1,12
  lin dlrgdp
  # constant dlrgdp{1 to p}
end do p
```

RATS will estimate the regression exactly 12 times. The first time through the loop, p = 1 so that RATS estimates an AR(1) model (i.e., lags{1 to p} is simply lag 1). The second time through the loop, p = 2 so that RATS includes lags 1 and 2 in the autoregression. Each time through the loop, p is increased by 1; hence, the number of lags used in the autoregression is increased by 1.

As it stands, the routine has a number of flaws. First, if we want to perform lag length tests, we need to estimate each autoregression over the same sample period. Since one usable observation is lost for each lag included in the model, we can estimate all of the autoregressions over the same sample period by modifying the LINREG instruction such that:

```
lin dlrgdp 14 *
```

Now all autoregressions begin with observation 14 (since one usable observation is lost by taking first-differences and 12 are lost when estimating the AR(12) regression). The second flaw is that we get too much output. Oftentimes, you will want to calculate the aic or sbc from each equation and then to examine the output of the model with the smallest aic and/or sbc. For each regression, you can compute and display the aic and sbc using:

```
compute aic = %nobs*log(%rss) + 2*(%nreg)
compute sbc = %nobs*log(%rss) + (%nreg)*log(%nobs)
```

It is also common to determine a lag length based on the outcome of t-tests. This methodology picks the lag length such that the t-statistic for the last lag is significant at some pre-specified level. Given the presence of an intercept, each regression will have p+1 coefficients and the t-statistic for the last lag can be obtained from %TSTATS(p+1).

```
dis 'Lags:' p 'AIC =' aic 'SBC =' sbc 't =' %tstats(p+1)
```

Hence, we can put these instructions in the loop and use the NOPRINT option in LINREG to obtain:

```
do p = 1,12
  lin(noprint) dlrgdp 14 *
  # constant dlrgdp{1 to p}
  compute aic = %nobs*log(%rss) + 2*(%nreg)
  compute sbc = %nobs*log(%rss) + (%nreg)*log(%nobs)
  dis 'Lags: ' p 'AIC = ' aic ' and SBC = ' sbc  't =' ###.### %tstats(p+1)
end do p
```

```
Lags: 1  AIC =    -703.21209 SBC =    -697.11238 t =       3.79455
Lags: 2  AIC =    -704.98313 SBC =    -695.83356 t =       1.93483
Lags: 3  AIC =    -703.03417 SBC =    -690.83475 t =      -0.22304
...
Lags: 11 AIC =    -693.50814 SBC =    -656.90987 t =      -0.38318
Lags: 12 AIC =    -695.48673 SBC =    -655.83860 t =      -1.92196
```

Here the AIC selects the model with two lags while the SBC selects the model with one lag. In this example, the choice is unclear since t-statistic on lag 2 has a prob-value that is slightly greater than 0.05. At this point, a careful researcher would subject the models to additional diagnostic checks.

Modifying the Program

The idea of looping over lags is easily extended to selecting the order p and q of an ARMA model. The remaining portion of Program 3.3 illustrates the process of fitting an ARMA(p, q) model to the change in the 1-year T-bill rate. The first line creates drl as the first difference of tb1yr. Then two DO loops are created. The program loops over all values of p and q from 0 to 4 so that a total of 25 ARMA models are estimated. Inside the DO loops, the BOXJENK instruction estimates an ARMA model (without an intercept) using the current values of p and q. The start date for all models is fixed at 1960:4 in order to ensure that all equations are estimated over the same sample period (Note: The first two observations of tb1yr are NA , one observation is lost be differencing and four more are lost due to the four autoregressive lags). For each model, the value of p and q the AIC and SBC are displayed.

```
dif tb1yr / drl
do p = 0,4
   do q = 0,4
    box(ar=p,ma=q,noprint) drl 1960:4 *
    compute aic = %nobs*log(%rss) + 2*(%nreg)
    compute sbc = %nobs*log(%rss) + (%nreg)*log(%nobs)
    dis  'Order ' p q 'The aic = ' aic ' and sbc = ' sbc
   end do q
end do p
```

```
Order  0 0 The aic =      718.76274  and sbc =      718.76274
Order  0 1 The aic =      708.79596  and sbc =      711.88356
Order  0 2 The aic =      700.19711  and sbc =      706.37230
Order  0 3 The aic =      702.19589  and sbc =      711.45868
Order  0 4 The aic =      696.65415  and sbc =      709.00454
Order  1 0 The aic =      716.09728  and sbc =      719.18488
Order  1 1 The aic =      703.94238  and sbc =      710.11758
Order  1 2 The aic =      702.19348  and sbc =      711.45627
Order  1 3 The aic =      703.75461  and sbc =      716.10500
Order  1 4 The aic =      696.61767  and sbc =      712.05565
Order  2 0 The aic =      705.59455  and sbc =      711.76974
Order  2 1 The aic =      699.89149  and sbc =      709.15428
Order  2 2 The aic =      701.02566  and sbc =      713.37604
Order  2 3 The aic =      701.05749  and sbc =      716.49547
Order  2 4 The aic =      697.58720  and sbc =      716.11277
Order  3 0 The aic =      698.58881  and sbc =      707.85160
Order  3 1 The aic =      700.04901  and sbc =      712.39940
Order  3 2 The aic =      701.86138  and sbc =      717.29936
Order  3 3 The aic =      701.71606  and sbc =      720.24163
Order  3 4 The aic =      699.19451  and sbc =      720.80769
Order  4 0 The aic =      700.27063  and sbc =      712.62102
Order  4 1 The aic =      702.03769  and sbc =      717.47568
Order  4 2 The aic =      703.86137  and sbc =      722.38695
Order  4 3 The aic =      703.53608  and sbc =      725.14926
Order  4 4 The aic =      699.36336  and sbc =      724.06413
```

The AIC selects an ARMA(1, 4) model and the SBC selects an ARMA(0, 2) [i.e., the SBC selects an MA(2) specification].[27] The final instruction in the program estimates an MA(2) model over the full sample period.

box(ma=2) drl

```
Box-Jenkins - Estimation by Gauss-Newton
Convergence  in       16  Iterations.  Final  criterion  was    0.0000062  <
0.0000100
Dependent Variable DRL
Quarterly Data From 1959:04 To 2001:01

   Variable           Coeff       Std Error      T-Stat     Signif
********************************************************************
1.  MA{1}          0.293888856  0.076814701      3.82595  0.00018490
2.  MA{2}         -0.220139206  0.076854845     -2.86435  0.00472551
```

[27] RATS uses the Gauss-Newton algorithm to estimate coefficients of an ARMA model. The default number of iterations for the BOXJENK instruction is 40. Thus, if you use this type of automated procedure to select the order of an *ARMA* model, you must check to ensure that the estimation process converged. Line 2 of the printed output for the MA(2) indicates that the process converged in 16 iterations.

3.3 Lag Length Tests in a VAR

We can use a similar procedure to perform lag length tests in a VAR. In Chapter 2, we performed some likelihood ratio tests to determine the lag length in a VAR using the variables dlrgdp, dlrm2 and drs. It is possible to automate the procedure using a DO loop. Since the data are quarterly, it seems plausible to perform leg length tests for lags 16 versus 12, 12 versus 8, and 8 versus 4. The following instruction can be found in Program 3.4 on the file labeled CHAPTER3_2.PRG. The program reads in the data set and creates the three variables dlrgdp, dlrm2 and drs. The next instruction creates a DO loop such that the variable lags runs from 16 to 8 in steps of minus 4 (i.e., lags will equal 16, 12 and 8):

```
do lags = 16,8,-4
```

Next, the SYSTEM-END(SYSTEM) block sets up the three variable VAR using a lag length of 1 to lags.

```
system(model=chap3)
vars dlrgdp dlrm2 drs
lags 1 to lags
det constant
end(system)
```

The ESTIMATE instruction below is the key to the program. The system is estimated beginning with observation 1959:2+lags since one observation is lost as a result of differencing and lags observations are lost by incorporating the lagged variables. The residuals are stored in unrestrict—unrestrict(1) contains the residuals from the dlrgdp equation, unrestrict(2) contains the residuals from the dlrm2 equation and unrestrict(3) contains the residuals from the drs equation. The following two statements calculate the AIC and SBC for the unrestricted model. Note that there are 3*lags + 1 coefficients in each of the three equations.

```
estimate(resids=unrestrict,noprint) 1959:2+lags *
com aic_u = %nobs*%logdet + 2*(3*lags+1)*3
com sbc_u = %nobs*%logdet + log(%nobs)*(3*lags+1)*3
```

The second SYSTEM-END(SYSTEM) block shown below estimates the VAR using four fewer lags (i.e., lags-4). You should be careful not to leave any spaces on either side of the plus sign. This restricted system is estimated over the same sample period as the unrestricted system (1959:2+lags *) and the residual series are saved in restrict. The AIC and the SBC for the restricted model are calculated and denoted by aic_r and sbc_r, respectively. Notice that there are (3*(lags-4)+1)*3 estimated coefficients in the system (each of the three equations contains lags-4 lags of each variable plus an intercept).

```
system(model=chap3)
vars dlrgdp dlrm2 drs
lags 1 to lags-4
det constant
end(system)
estimate(resids=restrict,noprint) 1959:2+lags *
com aic_r = %nobs*%logdet + 2*(3*(lags-4)+1)*3
com sbc_r = %nobs*%logdet + log(%nobs)*(3*(lags-4)+1)*3
```

We display the AIC and SBC for the unrestricted and restricted systems using:

```
dis 'Lags = '  lags 'aic_u = ' aic_u 'sbc_u = ' sbc_u
dis 'Lags = '  lags-4 'aic_r = ' aic_r 'sbc_r = ' sbc_r
```

Notice that there are 36 (3*4*3) restrictions—four lags of three variables in three equations and each unrestricted equation contains 3*lags + 1 parameters. Hence, the next four instructions calculate and display the significance level for the test. The final line ends the DO loop, the counter lags is decreased by four and the process continues for the next value of lags.

```
ratio(degrees=3*4*3,mcorr=3*lags+1,noprint) 1959:2+lags *
# unrestrict
# restrict
dis 'Significance level = ' %signif   ; dis ' '
end do
```

```
Lags =  16 aic_r =     -3080.45335 sbc =     -2635.94291
Lags =  12 aic =       -3108.63138 sbc =     -2772.98064
Significance level =         0.76164

Lags =  12 aic_r =     -3198.00556 sbc =     -2859.47155
Lags =   8 aic =       -3197.47118 sbc =     -2968.73198
Significance level =         0.02069

Lags =  8 aic_r =     -3289.27609 sbc =     -3058.63806
Lags =  4 aic =       -3276.50688 sbc =     -3156.57511
Significance level =     3.87731e-04
```

Thus, at conventional significance levels, the restriction from 16 to 12 lags is not binding. However, restricting the system from 12 to 8 lags has a prob-value of `0.02069` and further restriction the system from 8 to 4 lags has a prob-value of `3.87731e-04`. The multivariate AIC and SBC both select the 12-lag model over the 16-lag model. However, in the other two cases the AIC selects the model with the long lag and the SBC selects the model with the short lag.

4. Loops for Dates

As suggested by the last program, one important use of a DO loop is to perform a set of RATS instructions such that the start and/or end date is altered each time through the loop. Since dates have an equivalent integer representation, it is simple to use date manipulations in the loop.

Suppose that you wanted to compare the 1-step ahead mean square prediction errors for an AR(1) and an AR(2) model of dlrgdp. One way to do this is to estimate each model over the first 100 observations and then obtain the 1-step ahead forecast for each. Since the value of dlrgdp for period 101 is known, it is possible to obtain the squared prediction error for the AR(1) and the AR(2) specifications. Next, estimate each model over the first 101 observations and obtain the squared 1-step ahead prediction error for period 102. Repeat the entire process up through observations 168 (so that you have the 1-step ahead prediction error for period 169) and compare the means of the sum of the squared prediction errors. Continue to scroll down Program 3.4 and enter the following instructions:

```
set f_ar1 = 0.
set f_ar2 = 0.
do i = 100,168
  lin(noprint,define=ar1) dlrgdp 3 i; # constant dlrgdp{1}
  forecast 1 1
  # ar1 f_ar1
  lin(noprint,define=ar2) dlrgdp 3 i; # constant dlrgdp{1 to 2}
  forecast 1 1
  # ar2 f_ar2
end do i
```

Lines 1 and 2 initialize the two series f_ar1 and f_ar2; these series are used to store the forecasts from the two models. The third line instructs RATS to execute the DO loop 69 times beginning with i = 100 and terminating after i = 168. Within each loop, RATS estimates an AR(1) model of dlrgdp over the sample period 3 through i. Lines 5 and 6 instruct RATS to make a 1-step ahead forecast for period i+1 and store the forecast in f_ar1. Lines 7 - 9 repeat the procedure for the AR(2) specification and stores the forecast in f_ar2. After the 69 loops are completed, f_ar1 and f_ar2 each contain the 69 1-step ahead forecasts.

The remainder of the program is straightforward. The first two lines shown below calculate the squared prediction errors from the AR(1) and the AR(2)—each prediction error is the difference between the actual and forecasted values of dlrgdp. Lines 3 and 4 calculate and display the mean square prediction error for the AR(1) and lines 5 and 6 calculate and display the mean square prediction error for the AR(2).

```
set pe_1 101 169 = (dlrgdp – f_ar1)**2
set pe_2 101 169 = (dlrgdp – f_ar2)**2
sta(noprint)  pe_1
dis ‘The MSPE from the AR(1) is: ‘ %mean
```

```
sta(noprint) pe_2
dis 'The MSPE from the AR(2) is: ' %mean
```

```
The MSPE from the AR(1) is:    2.44821e-05
The MSPE from the AR(2) is:    2.31782e-05
```

A Small Quiz: Now take a little quiz. Some might recommend eliminating an initial observation every time through the loop. In this way, you would estimate a model with 98 observations each time through the loop.

How would you rewrite the DO loop to perform this task?

Answer: Rewrite the LINREG instructions such that the initial observation increases by 1 each time through the loop. The two LINREG instructions should be:

```
lin(noprint,define=ar1) dlrgdp i-97 i; # constant dlrgdp{1}
lin(noprint,define=ar2) dlrgdp i-97 i; # constant dlrgdp{1 to 2}
```

Now, the first time through the loop, the initial observation is 3 (i.e., 3 = 100 – 97) and the last observation is 100. The second time through the loop, the initial observation is 4 and the last observation is 101. Continuing through i = 168 yields an entry value of 71 for the initial observation and 168 for the last observation. Thus, each regression has 97 usable observations.

Question two of the quiz concerns the use of date notation in a DO loop. Since the use of date notation is equivalent to the use of integers, it is possible to use date notation for the indices of the DO loop. In MONEY_DEM.XLS, observation 100 is equivalent to date 1988:1 and observation 168 is 2004:1. How could you rewrite lines 3 – 10 of the program above using date labels?

Answer:

```
do i = 1988:1,2004:1
  lin(noprint,define=ar1) dlrgdp 1959:3 i; # constant dlrgdp{1}
  forecast 1 1
  # ar1 f_ar1
  lin(noprint,define=ar2) dlrgdp 1959:3 i; # constant dlrgdp{1 to 2}
  forecast 1 1
  # ar2 f_ar2
end do i
set pe_1 1988:2 2001:1 = (dlrgdp – f_ar1)**2
set pe_2 1988:2 2001:1 = (dlrgdp – f_ar2)**2
```

5. Loops for Series

One of the most powerful features of RATS is that it allows you to perform a DO loop such that the index refers to a series. Here is a routine to create the growth rates of the variables in MONEY_DEM.XLS. Recall that gdp is [series]2, rgdp is [series]3, m2 is [series]4, m3 is [series]5, TB3mo is [series]6 and TB1yr is [series]7. The first part of Program 3.5 reads in the data set MONEY_DEM.XLS, but does not create the variable dlrgdp. Instead, we can create the growth rate of each series using a DO loop. Consider:

```
scratch 6 / scr_no
do i = 1,6
   set scr_no+i  = log((i+1){0}) - log((i+1){1})
end do i
```

The first line of the routine creates six new series. The series numbers begin at 8 (since a total of 7 series reside in memory) and run through 13. The variable scr_no contains the integer value 7. Notice that the indices of the DO loop range from 1 through 6. The first time through the loop i = 1 so that series 8 (8 = scr_no + 1) is set equal to the log of series 2 minus the log of series 2 lagged one period. The next time through the loop, i = 2 so that series 9 is set equal to the log of series 3 minus the log of series 3 lagged one period. In this fashion, series 8 – 13 contain the logarithmic changes of gdp, rgdp, m2, m3, tb3mo and tb1yr, respectively.

Now we can jazz up the program a bit. We know that we can read the label assigned to a series using: %L(variable). We can also assign a label to a variable. In fact, we can make our routine more user-friendly by assigning labels to series 8 through 13. Each label will begin with 'dl' and end with the name of the original series being changed (e.g., dlrgdp for the change in the log of rdgp). As discussed in Section 2.1, we cannot use EQV in a loop, so we will want to use LABELS here. Now consider the following modification of our program:

```
do i = 1,6
   set scr_no+i  = log((i+1){0}) - log((i+1){1})
   labels scr_no+i ; # 'DL'+%l(i+1)
end do i
tab / scr_no+1 to scr_no+6
```

```
Series   Obs      Mean         Std Error     Minimum       Maximum
DLGDP     168      0.018022     0.009419     -0.010052      0.057029
DLRGDP    168      0.008475     0.008960     -0.020598      0.037804
DLRM2     168      0.017045     0.008553     -0.002633      0.053028
DLM3      168      0.019167     0.009285     -0.005379      0.038578
DLTB3MO   168      0.003286     0.109786     -0.332247      0.396932
DLTB1YR   166     -0.000104     0.102851     -0.329450      0.288489
```

Note that LABELS does not allow you to refer to a series by its label. For example, you will obtain an error message if you type TABLE / dlrgdp.

6. The DOFOR Instruction

The DO instruction forces a particular relationship between subsequent values of the index. The index of a DO instruction must be an integer and the index is increased by the same amount from one loop to the next. However, there are many instances in which we do not want one value of the index to bear any precise relationship to the adjacent values. In such circumstances, DOFOR is particularly helpful.

Program 3.6 illustrates a simple way to create multiple graphs on a page. After reading in MONEY_DEM.XLS, the program creates the logarithmic changes in real gdp, the gdp deflator and m3 and the difference in the 3-month T-bill rate (drs) using:

```
set dlrgdp = log(rgdp) - log(rgdp{1})
set dlm3 = log(m3) - log(m3{1})
dif tb3mo / drs
set price = gdp/rgdp
set dlp = log(price) - log(price{1})
```

The SPGRAPH instruction below indicates that we want to create a Special Graph with the header Graphs of Four Principal Series. The graphs are to be arranged into two horizontal fields and two vertical fields.

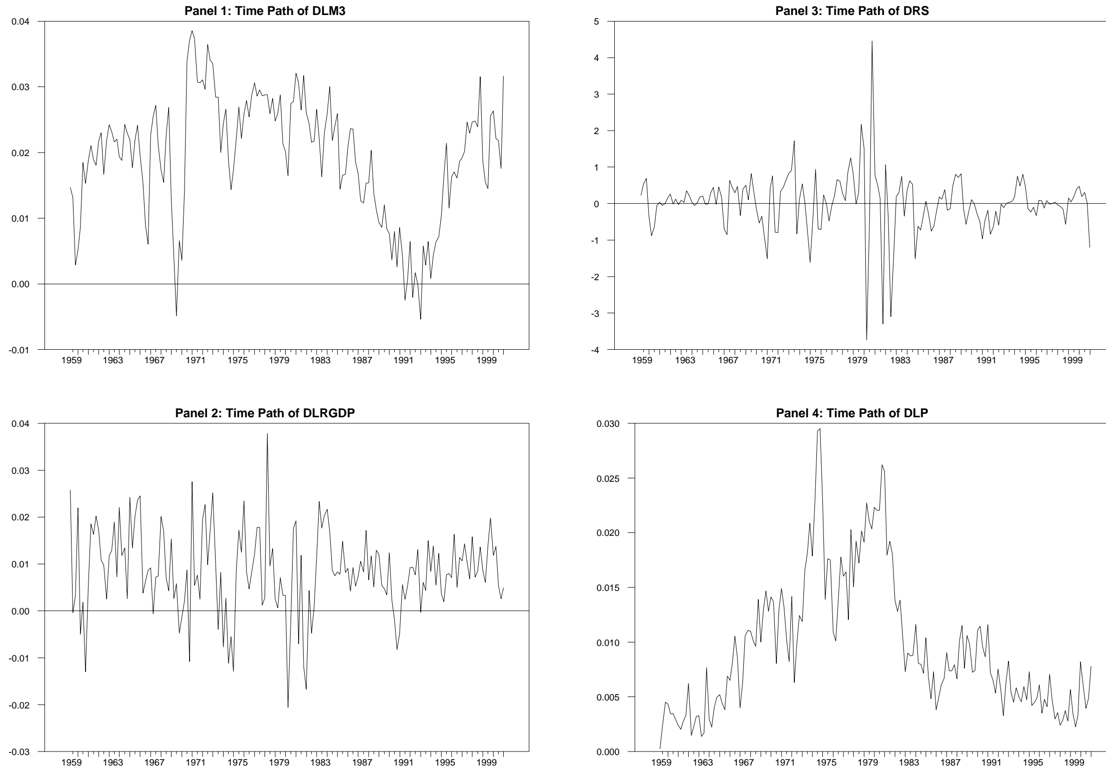
```
spgraph(hea='Graph of Four Principal Series',hfi=2,vfi=2)
```

We want to loop over the series dlm3 dlrgdp dr and dlp. Since the series numbers bear no particular relationship to each other, it is simplest to use a DOFOR loop. However, we need to use a counter j to indicate how many loops have been completed. Before entering the DOFOR loop, the value of j is initialized to zero and is increased by one every pass through the DOFOR loop. The first time through the loop, i equals the integer representing the series dm3 and j = 1. Consider:

```
com j = 0
dofor i = dlm3 dlrgdp drs dlp
  com j = j + 1
```

The next instruction below creates the header Panel 1: Time path of dlm3. To understand, note that the variable called header is necessarily a string variable. The string is equal to ‘Panel’ plus the value of j (=1) plus the string ‘Time Path of’ plus the label of dlm3 (i.e., the label of series i). The GRAPH instruction creates a graph of series i using the header created in the previous COMPUTE instruction. After the first pass, i becomes the integer value of dlrgdp and j = 2. Next, the header Panel 2: Time Path of dlrgdp is created and the GRAPH instruction creates a graph of series dlrgdp using this header. The process continues until i has taken on the values corresponding to dr and dlp. The final instruction SPGRAPH(done) displays the output.

```
   com header = 'Panel ' + j + ': Time Path of ' + %l(i)
   gra(hea= header) 1 ; # i
end do i
```
spgraph(done)



6.1 DOFOR and Loops for Series

In Program 3.5, we created the logarithmic change of series 2 through 7 and stored the results in series 8 through 13. It is quite possible that you do not want to create a variable in the form $\Delta \ln x_t$ for every variable in our data set (this is especially true if a series has one or more negative values). Suppose that you wanted to create only the logarithmic changes in real gdp (series 3), m2 (series 4) and tb3mo (series 6). Program 3.7 illustrates how you can perform this task using the DOFOR instruction:

```
scratch 3 / scr_no
dofor i = rgdp m2 tb3mo
   com j = j + 1
   set scr_no+j = log(i{0}) - log(i{1})
   labels scr_no+j ; # 'DL'+%l(i)
end dofor
```

table / scr_no+1 scr_no+2 scr_no+3

```
Series      Obs      Mean         Std Error       Minimum        Maximum
DLRGDP      168  0.0084752669  0.0089595106 -0.0205982814  0.0378040869
DLM2        168  0.0170454250  0.0085532148 -0.0026327924  0.0530276961
DLTB3MO     168  0.0032859057  0.1097860047 -0.3322470533  0.3969315930
```

Now, only three series are 'created from scratch' beginning with scr_no+1. The first time through the loop, i is the integer value of rdgp (i.e., i = 3) and j = 1. The SET instruction creates series 8 (note the scr_no+1 = 8) as the logarithmic change in rgdp (series 3.) The second time through the loop, j is incremented by 1 and series 9 is created as the logarithmic change in M2. Similarly, the last time through the loop, series 10 is created as the logarithmic change in tb3mo.

One final way to improve the program is to use the %S(L) function. As mentioned earlier, %S(L) returns the series number corresponding to the label L. You can verify that rgdp is the third series in RATS' memory by entering the instructions:

```
com a = %s('rgdp') ; dis a
```

```
3
```

A very useful feature of %S(L) is that it will *create* a series with the name L if it does not already exist. Suppose you have read in MONEY_DEM.XLS. An alternative way to create the logarithmic changes in rgdp, m2 and tb3mo is:

```
dofor i = rgdp m2 tb3mo
   set %s('dl'+%l(i)) = log(i{0}) - log(i{1})
end dofor
```

The key to understanding the program is recall that %l(i) is a string equal to the label of series *i*. Hence 'dl' + %l(i) refers to the label *dl* plus the label of series *i*. The first time through the loop, *i* = 3. Since the label *dlrgdp* does not exist, %s('dl'+%l(i)) creates a series with the name *dlrgdp* as the logarithmic change in series i. The second time through the loop, *i* is equal to the integer value of *m2* and %l(i) is the string 'm2'. Since there is no series named *dlm2*, %s('dl'+%l(i)) creates this series as the logarithmic change in *m2*. Similarly, the third time through the loop, the logarithmic change in *tb3mo* is created. Unlike the LABELS instruction, the names creates by %S(label) can be used for input and for output.

6.2 DOFOR and ENTRIES

In Section 3.1, it was indicated that any RATS instruction containing supplementary cards in a regression format allows you to use the ENTRIES option. However, any object you include on the supplementary card (except #, $ or a space) is counted as an entry. Thus, dlrgdp{1} is counted as four entries: dlrgdp, {, 1, and }. Similarly, dlrgdp{1 to p} is counted as 6 entries: dlrgdp, {, 1, to, and p}.

Suppose you want to determine whether the contemporaneous and lagged growth rate of M3 (*dlm3*) affected the growth rate of real GDP. You want to include contemporaneous money growth and the possibility that lags 1 to 4 of money growth are important. Since it is unclear whether the AR(1) or AR(2) model for $dlrgdp_t$ is most appropriate, you might want to estimate all your equations using both lag lengths. You could read in MONEY_DEM.XLS and construct $dlrgdp_t$ and $dlm3_t$ as follows:

```
set dlrgdp = log(rgdp) - log(rgdp{1})
set dlm3 = log(m3) - log(m3{1})
```

Then you could estimate six regressions using the following supplementary cards.

```
# constant dlrgdp{1}
# constant dlrgdp{1 to 2}
# constant dlrgdp{1} dlm3
# constant dlrgdp{1 to 2} dlm3
# constant dlrgdp{1} dlm3 dlm3{1 to 4}
# constant dlrgdp{1 to 2} dlm3 dlm3{1 to 4}
```

Alternatively, you could use the following set of instructions contained in Program 3.8:

```
dofor i = 7 8 14
   do p = 1,2
   lin(noprint,entries=i) dlrgdp 6 *
   # constant dlrgdp{1 to p} dlm3 dlm3{1 to 4}
   compute aic = %nobs*log(%rss) + 2*(%nreg)
   compute sbc = %nobs*log(%rss) + (%nreg)*log(%nobs)
   dis  'The aic = ' aic ' and sbc = ' sbc
   end do p
end dofor i
```

The first time through the DOFOR loop, i = 7 so that the ENTRIES option reads the `constant` and `dlrgdp{1 to p}` entries on the supplementary card. Hence, inside the DO loop as p changes from 1 to 2, the pure AR(1) and AR(2) are estimated. Next i = 8, so that ENTRIES option reads the `constant dlrgdp{1 to p}` and `dlm3` entries on the supplementary card. Again, two regressions are estimated using one and two lags of dlrgdp. Finally, i = 14 so that the

ENTRIES option reads the `constant`, `dlrgdp{1 to p}, dlm3` and `dlm3{1 to 4}` entries on the supplementary card. If you run the program, your output should be:

```
The aic =     -725.43529  and sbc =     -719.23556
The aic =     -727.49692  and sbc =     -718.19732
The aic =     -728.35931  and sbc =     -719.05971
The aic =     -729.80200  and sbc =     -717.40253
The aic =     -724.17933  and sbc =     -702.48027
The aic =     -725.39124  and sbc =     -700.59230
```

The SBC selects the model with only one lag of $dlrgdp_t$. However, if you estimate the model selected by the AIC, your output will look like:

lin dlrgdp 6 *
 # constant dlrgdp{1 to 2} dlm3

```
   Variable     Coeff        Std Error        T-Stat     Signif
**********************************************************************
1.  Constant   0.0023202979 0.0016286184      1.42470  0.15619062
2.  DLRGDP{1}  0.2347026242 0.0766291214      3.06284  0.00257298
3.  DLRGDP{2}  0.1412306952 0.0766582885      1.84234  0.06727619
4.  DLM3       0.1475857849 0.0715417775      2.06293  0.04073517
```

Jazzing Up the Program

The problem with ENTRIES is that we have to count the number of entries to use from the supplementary card. As illustrated above, this can be tricky when you use braces { }. However, RATS allows you to replace the individual entries on a supplemental card with a vector. You use the ENTER instruction to manipulate the items in the vector. By changing the contents of the vector, you modify the information on the supplementary card. The first few examples in the chapter on matrices (Chapter 5) cover this technique in great detail.

7. Loops with While and Until

The DO loop is appropriate if you know exactly how many loops you want to make. However, there are circumstances in which the number of repetitions is unclear. For example, a common way to select the a lag length in an AR(p) model is to estimate the autoregression equation using the largest value of p deemed reasonable. If the t-statistic on the coefficient for lag p is insignificant at some pre-specified level, estimate an AR(p-1) and repeat the process until the last lag is statistically significant. If you used a DO loop, it would be necessary to estimate every autoregression from p to 1. A more efficient procedure is to stop the process once a significant lag is found. The syntax for a WHILE block is:

while *condition* {
    block of statements executed as long as condition is “true”
  }
end while ;* (omit if the WHILE block is nested inside another compiled section)
The syntax for an UNTIL block is:

until *condition* {
   block of statements executed as long as condition is “true”
 }
end until ;* (omit if the WHILE block is nested inside another compiled section)

The remaining portion of Program 3.8 uses the WHILE instruction to perform the type of lag length test discussed above. Suppose that we want to fit an AR(p) model for *drl* with no more than 12 lags and that we want the t-statistic for the last coefficient to be significant at the 5% level. The COMPUTE instruction initializes the variable lags to be 13. This counter will be decreased by 1 each time through the WHILE loop. The variable sign is used to store the significance level of the t-statistic for the coefficient for *drl*(*lags*). COMPUTE initializes the variable sign to be 0.5 (Note: sign can be initialized to be any real number greater than 0.05). As long as sign exceeds 0.05, RATS will loop through the WHILE-END WHILE block below:

```
com lags = 13, sign = .5
while sign > 0.05  {
     com lags = lags - 1
     lin(noprint) drl
     # constant drl{1 to lags}
     exclude(noprint) ; # drl{lags}
     com sign = %signif
     dis 'Significance of lag' lags  '= ' sign
    }
end while
```

```
Significance of lag 12 =        0.92989
Significance of lag 11 =        0.60897
Significance of lag 10 =        0.62405
Significance of lag  9 =        0.32442
Significance of lag  8 =        0.43783
Significance of lag  7 =        0.01255
```

The first time through the loop, the variable sign is compared to 0.05. Since sign exceeds 0.05 (i.e., since the condition sign > 0.05 is true) all of the instructions within the block are executed. Hence, lags is decreased from 13 to 12 and *drl* is estimated as an AR(12). EXCLUDE calculates, but does not display, the value of the F-statistic for the exclusion restriction that the coefficient on $drl_{t-12} = 0$. Of course, with only one restriction, an F-test is equivalent to a t-test. Note that EXCLUDE creates the internal variable %SIGNIF containing the significance level of the restriction. The key instruction is COMPUTE sign = %signif. The variable sign is equated to the significance level of the restriction. If at the end of any loop sign < 0.05, the WHILE-END loop is terminated. Notice that sign exceeds 0.05 for lags 12 through 8 so that looping over the instructions within the WHILE-END block continues. However, in the AR(7) model, the significance level of the coefficient on dlr_7 is less than 0.05 and RATS exits the loop (i.e., the condition sign > 0.05 is not true). The next two lines produce the AR(7) model:

lin drl
 # constant drl{1 to lags}

```
   Variable         Coeff       Std Error      T-Stat     Signif
*******************************************************************
1.  Constant     0.007234002  0.053259079      0.13583  0.89213923
2.  DRL{1}       0.281822834  0.080465922      3.50239  0.00060674
3.  DRL{2}      -0.343129641  0.083606666     -4.10409  0.00006620
4.  DRL{3}       0.274614957  0.088185213      3.11407  0.00220824
5.  DRL{4}      -0.035831951  0.090593547     -0.39552  0.69301422
6.  DRL{5}       0.016548074  0.087971257      0.18811  0.85104475
7.  DRL{6}      -0.030787141  0.083329087     -0.36946  0.71229913
8.  DRL{7}      -0.202414362  0.080118649     -2.52643  0.01255294
```

Jazzing Up the Program

The program can be made much more useful by incorporating it into a DOFOR loop. You can allow DOFOR to loop over each series. Within each of these loops, the WHILE-END block determines the lag length. The remaining portion of Program 3.8 constructs the variable *dlp.* The counter *i* in the DOFOR instruction loops over each for these five variables. Within the DOFOR loop, the WHILE-END block determines the lag length for series *i*. At the end of each WHILE loop, the label of series *i* and the lag length is displayed. If you want the *AR*(p) model for each series, remove the comment label (*i.e*., remove *) from the final LINREG instruction.

```
set price = gdp/rgdp
set dlp = log(price) - log(price{1})
dofor i = dlrgdp dlm3 drs drl dlp
   com lags = 13, sign = .5
   while sign > 0.05  {
    com lags = lags - 1
```

```
    lin(noprint) i
    # constant i{1 to lags}
    exclude(noprint) ; # i{lags}
    com sign = %signif
     }    ;* Note that END WHILE is removed since we are in a compiled section
     dis %label([series]i) 'Lag length =  lags
     *    lin i ; # constant i{1 to lags}
end dofor
```

```
DLRGDP Lag length =  1
DLM3 Lag length =  1
DRS Lag length =  11
DRL Lag length =  7
DLP Lag length =  3
```

Frequently Asked Questions

1. What would happen if *condition* was never true?

Answer: Big problems! One possibility is that looping continues indefinitely until you click the HALT icon on RATS Menu Bar. The other possibility is that you produce an error that causes (1) RATS stop execution and display an error message or (2) an inadvertent *condition* that is true. For example, suppose you used the program above on a series that behaves as a white noise process (so that none of the lags is significant at the 5% level). Hence, looping will continue until *lags* equals zero. At this point, the program regresses the series on itself so that the resulting *F*-statistic is necessarily significant at the 5% level (R^2 will be 1). Here, dumb luck yields the correct lag length of zero. Since you can't count on being this lucky all of the time, it is important to be careful that you do not get caught in an infinite loop or a loop that produces an error.

2. What is the difference between WHILE and UNTIL?

Answer: The WHILE instruction checks to determine if *condition* is true at the beginning of a loop (i.e., when WHILE *condition* is encountered). The UNTIL instruction checks at the end of the loop (*i.e.*, when END UNTIL or the closing brace } is encountered). Thus, the UNTIL block of instructions is always executed at least once.

If PROGRAM 5 had used UNTIL instead of WHILE, the first three lines could have been:

```
dofor i = dlrgdp dlm3 drs drl dlp
  com lags = 13
until sign < 0.05  {            ;* Note the reversed inequality
```

The first time UNTIL is encountered, *sign* is not defined (so the condition *sign* < 0.05 is not true). The program continues until a significance level less than 0.05 is encountered.

3. What other conditions are allowed in the WHILE and UNTIL instructions?

    Answer: As detailed in the very beginning of the next chapter, any of the standard conditional relationships are allowed including < > (*i.e.*, not equal) and = = (*i.e.*, equality) as well as compound conditions created with .AND. and .OR.

Chapter 4:
IF Statements and Monte Carlo Experiments

There are many instances in which we want to perform a set of instructions only if a particular condition is met. For example, in the last chapter, we used the WHILE instruction to reduce the order of an *AR*(*p*) model if the last coefficient was not significant at the 5% level. The process continued until a significant lag was found. At that point, the program produced the output of the final autoregressive model. In fact, RATS enables you to control the flow of a program using many types of conditional statements.

Suppose that A and B are two numbers of the same type (real, integer, …). RATS is able to check the following conditions involving the standard relational operators:

equality:	A = = B	or	A.EQ.B
not equal	A <> B	or	A.NE.B
greater than	A > B	or	A.GE.B
less than	A < B	or	A.LT.B
greater or equal to	A >= B	or	A.GE.B
less than or equal to	A <= B	or	A.LE.B

Moreover, RATS can create compound conditional statements that use AND, NOT and OR:

A.AND.B
A.NOT.B
A.OR.B

The simplest way to use relational operators is with the IF instruction. The basic syntax of IF is:

IF *condition*
  instruction to be executed if the condition is true

where: *condition* is any of the relational conditions listed above and *instruction* is any valid RATS instruction.

If the specified condition is met, RATS will execute the next instruction. If the condition is not met, the next instruction will be skipped. For example, consider the following instructions:

```
if x = = 2
  dis ‘The value of x equals 2’
end if
```

This set of instructions will display ‘The value of x equals 2’ only when the value of x equals 2. You need to be careful about the use of the double equal sign since you are not equating x with the value of 2. If, by mistake, you type IF $x = 2$, RATS will not interpret this as any of the

relational statements above. Instead, the equals sign will cause RATS set the value of x equal to 2; since the condition is TRUE, the message always be displayed.

Note that you do not use the END IF statement if you are within a compiled section of a RATS program. For example, an IF statement within a DO loop or within a RATS procedure does not need the END IF statement. Otherwise, you should include the END IF instruction.

Examples

1.

```
do t = 2,2000:4
  if %valid( x(t) )==0 ; com x(t) = 0.5( x(t-1) + x(t+1) )
end do t
```

I do not recommend it, but suppose you want to 'fix' the missing values in your data set by averaging. The %valid() function returns zero if the argument is a missing value. If the data runs from periods 1 through 2001:1, this routine will replace a missing value of $x(t)$ with the average of the two adjacent values. (Example 3 in the Section %IF(x,y,z) below, discusses a faster way to perform the averaging).

2.

```
sta(noprint,fractiles) drs
if %minimum > 0.
  log drs / ldrs
if %minimum.le.0
  dis 'I CAN ONLY TAKE THE LOG OF POSITIVE NUMBERS'
```

The STATISTICS instruction with the FRACTILES option stores the smallest value of *drs* in the internal variable %minimum. If the smallest value of *drs* exceeds zero, RATS forms the new series *ldrs* as the log of *drs*. If the smallest value of *drs* is negative or zero, RATS will display a warning message.

3.

```
if %converged <> 1 ; dis 'Model did not converge'
```

A number of RATS instructions including NLLS and MAXIMIZE produce the internal variable %converged. Note that %converged = 1 if the nonlinear estimation converged within the allowable number of iterations and is equal to zero otherwise. Here, the warning message *Model did not converge* is displayed if the previous nonlinear estimation did not converge.

4.

```
if %converged = = 0 ; dis 'Model did not converge'
```

The program will display *Model did not converge* if convergence is not obtained. However, the following contains a serious mistake:

```
if %converged = 0 ; dis 'Model did not converge'
```

The program will never display *Model did not converge* since the equal sign (instead of a double equal sign) was used. In my own programs, I try to avoid this possible source of error and use .EQ. instead of = = .

1. If-Then-Else Blocks

If you want to execute a block of statements when a condition is met, simply include them in braces as illustrated below:

```
IF x = = 0 {
  program statements
}
END IF   (omit the END when the IF occurs inside a compiled sections)
```

All of the program statement in braces will be executed if the value of x is equal to zero. Otherwise, the entire set of statements will be ignored. If you have a single instruction (with or without supplementary cards) you can place them on the same line as the IF statement. Consider:

```
if i.ge.4 ; lin y ; # constant y{1 to i}
```

The IF block will estimate an *AR*(*i*) model so long as the value of i is greater than or equal to 4. If *i* is less than *4*, the regression will not be estimated.

Examples

1.
```
if lags.gt.0 {
  dis 'The lag length is' lags
  lin(noprint) y ; # constant y{1 to lags}
  com aic = %nobs*log(%rss) + 2*(%nreg)
  dis 'The aic is' aic
}
```

If the value of *lags* is greater than zero, the routine will display 'The lag length is' *lags*. Moreover, an AR(*p*) model (with *p* = *lags*) will be estimated and the value of the *AIC* will be calculated and displayed.

2. Consider the following routine to determine the optimal lag length for an *AR* process for *dlrgdp*. The first time through the DO loop, *p* = 1 and an *AR*(1) model is estimated. The resulting *AIC* is compared to *aic_min*. On this first loop, *p* = 1, so that the lag length *p* = 1 is stored in the variable *p_opt* and the value of the *AIC* replaces *aic_min*. The next time through the loop, an AR(2) model is estimated and the *AIC* for this model is compared to *aic_min*. After the DO loop is complete, *p_opt* contains the lag length of the best fitting model and *aic_min* contains the *AIC* for that model.

```
do p = 1,12
  lin(noprint) dlrgdp 13 *
  # constant dlrgdp{1 to p}
  compute aic = %nobs*log(%rss) + 2*(%nreg)
  if p.eq.1.or.aic < aic_min {
     com p_opt = p
     com aic_min = aic
   }
end do p
```

The ELSE Block

If the condition for the IF Block fails, you may want the RATS to execute an alternative set of statements. The syntax is:

If *condition* {
    program statements
    }
Else {
    Alternate program statements
    }
End if

The first set of statements will be executed if the condition is true, otherwise the alternate set of program statements will be executed. Again, you do not need the braces for a single statement. You should not use the END IF instruction if you are within a compiled section of a program.

ELSE IF Blocks

Oftentimes in your programming, you will encounter a situation where there is not an either/or choice. You might have a number of possible states and want to execute different sets of instructions for each possible state. In such circumstances, you can use ELSE IF blocks—each consists of a set of instructions that is executed if the appropriate condition is met. The typical structure of a complex IF instruction is:

IF *condition 1* {
    first set of statements
    }
Else if *condition 2* {
    second set of statements
    }
…
Else {
    last set of statements
    }
End if (if not within a compiled program segment)

Note: RATS performs the first IF or ELSE IF condition that is true. If all are false, RATS will perform ELSE (if present).

Nested IF Statements

It is possible to nest IF statements. Consider the following example. If *condition 1* and *condition 2* hold, then *statement 1* is executed. *Statement 2* is executed if *condition 1* does not hold.

```
if condition 1 {
   if condition 2 ; statement 1
   }
else ; statement 2
```

With the appropriate use of braces { } you can write programs with many conditional statements embedded within others. If you want to use nested IF statements, you should refer to the RATS Reference Manual. My experience indicates that it is best to avoid nested IF statements since they are very difficult to debug. It is always possible to program precisely the same conditional statements without nested IF statements using the relational .OR. and .AND. conditions.

1.1 Sample Program: Lag Lengths Again

Reconsider the problem of using the *AIC* to select the lag length for an autoregressive model *of* $dlrgdp_t$. After reading in the data set MONEY_DEM.XLS, Program 4.1 on the file labeled PROGRAM4.PRG constructs the variables *dlrgdp*, *drs* and *dlrm2*. Since these should be familiar to you, we can jump to the next part of the program:

```
lin(noprint) dlrgdp 14 * ; # constant
com aic_min = %nobs*log(%rss) + 2*%nreg, p_opt = 0
do p = 1,12
     lin(noprint) dlrgdp 14 *
     # constant dlrgdp{1 to p}
     compute aic = %nobs*log(%rss) + 2*(%nreg)
     if aic < aic_min {
       com p_opt = p
       com aic_min = aic
     }
end do p
```

As discussed in Example 2 above, the variable *aic_min* will be used to contain the smallest value of the *AIC* and the integer *p_opt* will be used to contain the value of *p* yielding the lowest *AIC*. However, to allow for the case where the optimal lag length is zero, the initial values of *aic_min* and *p_opt* are obtained from regressing $dlrgdp_t$ on a constant. In the second line of the program, the variables *aic_min* and *p* hold the *AIC* and lag length for this regression. The first time

through the loop, p = 1 and the AR(1) model will be estimated. The value of the resulting AIC will be compared to *aic_min*—if the calculated AIC is less than *aic_min*, the two instructions in brackets will be executed.

If the optimal lag is zero, the condition on the IF instruction below is true. As such, RATS displays the message: *An AR model is inappropriate. The autocorrelations are.* The COR instruction then produces the first twelve autocorrelations of $dlrgdp_t$ and the associated Q-statistics. If *p_opt* is greater than zero, the ELSE block is executed. Hence, the autoregression using *p_opt* lags is displayed along with the value of the AIC and *p_opt*. END IF instruction is used since the IF-ELSE block is not within a compiled section of the program.

```
if p_opt = = 0 {
     dis ‘An AR model is inappropriate. The autocorrelations are’
     cor(number=12,span=4,qstats) dlrgdp
     }
else {
     lin dlrgdp / resids
     # constant dlrgdp{1 to p_opt}
     compute aic = %nobs*log(%rss) + 2*(%nreg)
     dis ‘The aic with’ p_opt ‘lags is’ aic
     }
end if
```

If you run the program, your output will be:

```
  Variable             Coeff        Std Error       T-Stat      Signif
***********************************************************************
1.  Constant      0.0051566068 0.0010217954      5.04661   0.00000119
2.  DLRGDP{1}     0.2508977521 0.0769801061      3.25925   0.00135976
3.  DLRGDP{2}     0.1362250820 0.0762100846      1.78749   0.07571568

The aic with 2 lags is    -732.28752
```

Jazzing Up the Program

1. You might want to use a routine such as this in a number of your programs. However, you might not always want to use a maximum possible lag length of 12. To generalize the program, you can create a variable called *max_lag* and a variable called *diffs* right before the DO p = 1,12 instruction. *Max_lag* contains the maximum lag length you are willing to consider and the value of *diffs* equals 1 if the variable has been differenced. You will also need to modify the DO instruction such that the index p runs through *max_lag*. Hence, if you want to consider a maximum of 16 lags using a variable that has been differenced once, use:

```
com max_lag = 16, diffs = 1
do p = 1,max_lag
     lin(noprint) dlrgdp max_lag+diffs+1 *
```

These instructions cause RATS to use a maximum of 16 lags. Since the variable has been differenced once (*diffs* = 1), the estimation for all 16 regressions will begin with observation 18.

2. You can easily modify the program so that it can be used with any variable. Simply replace every occurrence of *dlrgdp* with the symbol *x*. For example, the first portion of the program will become:

```
com max_lag = 16, diffs = 1
lin(noprint) x max_lag+diffs+1 * ; # constant
com aic_min = %nobs*log(%rss) + 2*%nreg, p_opt = 0
do p = 1,max_lag
      lin(noprint) x max_lag+diffs+1 *
      # constant x{1 to p}
      compute aic = %nobs*log(%rss) + 2*(%nreg)
      if aic < aic_min {
            com p_opt = p
            com aic_min = aic
       }
end do p
```

If you eliminate the END IF instruction, you can use the routine to find the optimal lag length of $dlrgdp_t$, $dlrm2_t$ and drs_t using:

```
dofor x = dlrgdp dlrm2 drs
     The modified program
End DOFOR
```

The complete program is on the final portion of Program 4.1.

2. The %IF(x,y,z) Function

There is a second type of conditional statement that is especially useful when working with a series. Consider:

%IF(*condition*,y,z)

This function returns *y* if the condition is true and returns *z* if the condition is false. Note that *y*, and *z* can be REAL or INTEGER.

Examples:

1. com x = %if(i.ge.5,1.0,-1.0)

Here the %IF( ) function is used with a real variable x. The value of x will be 1.0 if *i* is greater than or equal to 5 and will be –1.0 if *i* is less than 5.

2. set dummy = %if(t.ge.1992:1,1,0)

In conjunction with SET, the %IF( ) function works on each entry of a series. Here, each value of DUMMY is equal to 1 for t ≥ 1992:1 and is equal to zero otherwise. Thus, in contrast to example 1, the SET command applies the %IF( ) function operates to the entire series. In fact, this single statement works just like the following DO loop:

```
set dummy = 0
do t = 1,2001:1
    if t.ge. 1992:1 ; com dummy(t) = 1
end do i
```

Although both yield the identical values for *dummy*, there is one major difference. The actual DO loop works many times slower than the SET with %IF( ). The following two programs both create a dummy variable equal to zero for observations 1 to 249 and equal to one for observations 250 through 500. In order to ascertain the differential speed of the DO loop versus the implied DO loop if the SET with %IF(), each program performs this task 10000 times. Program 1 took 1 minute and 18 seconds to run, while Program 2 took only 12 seconds.

Program 1

```
all 500
set dummy = 0
do i = 1,10000
    do t = 1,500
            if t.ge.250; com dummy(t) = 1
    end do t
end do i
* 1 minute 18 seconds
```

Program 2

```
        all 500
set dummy = 0
do i = 1,10000
   set dummy = %if(t.ge.250,1,0)
        end do i

        * 12 seconds
```

The point is to avoid DO loops if possible since they are slow. The implied DO loop of the %IF( ) instruction used in conjunction with SET is an efficient programming technique.

3. Reconsider the example where we 'fixed' a missing value by averaging:

```
do t = 2,2000:4
        if  %valid( x(t) ) = = 0 ; com x(t) = 0.5( x(t-1) + x(t+1) )
end do t
```

The following is much faster:

```
set x 2 2000:4 = %if( %valid(x) , x, (x{1} + x{-1})/2)
```

4. It is possible to SET entry *i* of one series based on the value of a corresponding value of a second series. Consider:

SET plus = %if(*resids*{1}< 0, 1, 0)

The series *plus* is equal to 1 if the lagged value of *resids* is negative, and is 0 when the lagged *resids* is greater than or equal to zero. To better understand the output of the %IF( ) function, suppose that *resids* contains the residuals from an *AR*(2) model. As such, the first two entries of *resids* are missing since two usable observations are lost in estimating the AR(2).

ENTRY	RESIDS	PLUS
1	NA	0
2	NA	0
3	-0.41281771461	0
4	-3.11837667346	1
5	0.20035497963	1
6	0.67954385844	0
7	2.80353550741	0
8	-0.78068210129	0
9	1.18970179911	1
10	0.97092277920	0

In RATS, all series begin with entry 1. The first entry of *plus* [i.e., *plus*(1)] is zero since the previous entry of *resids* is undefined (*i.e*., it is not true that *resids* for period 0 is negative). Both *plus*(2) and *plus*(3) are equal to zero since the first two entries of *resids* are NA (hence, they are NOT positive). Since *resids*(3) and *resids*(4) are both negative, the fourth and fifth entries of plus are set equal to 1. The same logic prevails throughout; *plus*(10) equals zero since *resids*(9) is positive.

In essence, the single statement above replaces the more complex (and much slower):

```
set plus = 0
do t = 2,N
        if resids(t) < 0; com plus(t-1) = 1
end do t
```

3. Estimating a Threshold Autoregression

The threshold autoregressive (TAR) model has become popular in that it allows for different degrees of autoregressive decay. Consider a two-regime version of the threshold autoregressive (TAR) model developed by Tong (1983):

$$y_t = I_t\left[\alpha_0 + \sum_{i=1}^{p}\alpha_i y_{t-i}\right] + (1-I_t)\left[\beta_0 + \sum_{i=1}^{p}\beta_i y_{t-i}\right] + \varepsilon_t$$

where: y_t is the series of interest, the α_i, and β_i are coefficients to be estimated, τ is the value of the threshold, p is the order of the TAR model and I_t is the Heaviside indicator function:

$$I_t = \begin{cases} 1 \ \mathit{if} \ y_{t-1} \geq \tau \\ 0 \ \mathit{if} \ y_{t-1} < \tau \end{cases}$$

The nature of the system is that there are two states of the world. In the one state of the world, y_{t-1} exceeds the value of the threshold τ so that $I_t = 1$ and $(1 - I_t) = 0$. As such, y_t follows the autoregressive process: $\alpha_0 + \Sigma\alpha_i y_{t-i}$. Similarly, in the low state, y_{t-1} falls short of the threshold τ, so that $I_t = 0$, $(1 - I_t) = 1$ and y_t follows the autoregressive process: $\beta_0 + \Sigma\beta_i y_{t-i}$. In a sense, there are two attractors or potential "equilibrium" values. In the 'high' state, the system is drawn toward $\alpha_0/(1-\Sigma\alpha_i)$ and in the 'low' state, the system is drawn toward $\beta_0/(1-\Sigma\beta_i)$. Moreover, the degree of autoregressive decay will differ across the two states if for any value of *i*, $\alpha_i \neq \beta_i$. The key feature of the TAR model is that a sufficiently large ε_t shock can cause the system to switch between states.

PROGRAM 4.3 in the file CHAPTER4_1.PRG illustrates the estimation of a TAR model for the $\{dlrgdp_t\}$ series. As usual, the first five lines of the program read in the data set and construct the variable *dlpgdp* using:

set *dlrgdp* = log(*rgdp*) – log(*rgdp*{1})

First, suppose that we want the value of the threshold τ to equal zero. This might be the case if we were certain the positive real *GDP* growth behaved differently from negative growth. We have already determined that it is reasonable to use a lag length of 2. Hence, we need to construct a number of variables. First, we can construct the indicator function using:

set plus = %if(dlrgdp{1}>= 0,1,0)

Note first that we denote the indicator I_t by a variable called *plus*—we cannot use the symbol *i* (since *i* is a reserved integer variable) to represent a series. For each possible entry in the data set, the SET instruction compares $dlrgdp_{t-1}$ to the value 0. IF $dprgdp_{t-1}$ is greater than zero, the value

of $plus_t$ is equal to 1, otherwise $plus_t$ is zero. You can see how the program works by printing the first 10 values of *plus* and *dlrgdp*.

pri 1 10 dlrgdp plus

```
ENTRY        DLRGDP          PLUS
 1959:01        NA              0
 1959:02   0.025797235495       0
 1959:03  -0.000428834862       1
 1959:04   0.003297294498       0
 1960:01   0.021945448475       1
 1960:02  -0.004947391752       1
 1960:03   0.001847653167       0
 1960:04  -0.012963339986       1
 1961:01   0.005763460460       0
 1961:02   0.018546572263       1
```

Note that *plus*(1959:1) is zero since *dlrgdp*(1958:1) is not in the data set. Since this value was not positive, *plus*(1959:1) is set equal to zero. Similarly, the initial value of *dlrgdp* is *NA* (we lost an observation by differencing); since *dlrgdp*(1959:1) is not positive, *plus*(1959:1) is set equal to zero. However, *dlrgdp*(1959:2) is positive. As such, *plus*(1959:3) is set equal to 1. Continuing through the observations, you can verify that $plus_t$ is 1 when $dlrgdp_{t-1}$ is positive.

Next we create $(1\text{-}I_t)$ as the series *minus* using:

set minus = 1 - plus

Now we can create the variables: $I_t{*}dlrgdp_{t\text{-}1}$, $I_t{*}dlrgdp_{t\text{-}2}$, $(1\text{-}I_t){*}dlrgdp_{t\text{-}1}$ and $I_t{*}dlrgdp_{t\text{-}2}$. Consider:

set y1_plus = plus*dlrgdp{1}
set y2_plus = plus*dlrgdp{2}
set y1_minus = minus*dlrgdp{1}
set y2_minus = minus*dlrgdp{2}

Now we can estimate the regression using:

lin dlrgdp
 # plus y1_plus y2_plus minus y1_minus y2_minus

```
Variable        Coeff        Std Error       T-Stat      Signif
*****************************************************************
1.  PLUS        0.005600681  0.001464144      3.82523  0.00018693
2.  Y1_PLUS     0.192048274  0.108292518      1.77342  0.07806129
3.  Y2_PLUS     0.174427730  0.086316981      2.02078  0.04497008
4.  MINUS       0.003903606  0.003147715      1.24014  0.21673934
5.  Y1_MINUS    0.146629915  0.338184772      0.43358  0.66517798
6.  Y2_MINUS   -0.010018863  0.165657994     -0.06048  0.95184946
```

At this point, you might want to pare down the number of coefficients since a number have *t*-statistics that are quite low. Nevertheless, our goal here is to illustrate programming techniques, not to obtain the best fitting TAR model for real *GDP*.

3.1 Estimating the Threshold

One problem with the above model is that the threshold may not be zero. When τ is unknown, Chan (1993) shows how to obtain a super-consistent estimate of the threshold parameter. For a TAR model, the procedure is to order the observations from smallest to largest such that:

$$y^1 < y^2 < y^3 ... < y^T$$

For each value of y^j, let $\tau = y^j$, set the Heaviside indicator according to this potential threshold and estimate a TAR model. The regression equation with the smallest residual sum of squares contains the consistent estimate of the threshold. In practice, the highest and lowest 15% of the $\{y^j\}$ values are excluded from the grid search so as to ensure an adequate number of observations on each side of the threshold. We can conduct this procedure using the following program segment. Consider:

```
compute low = 1959:2 + fix(.15*%nobs) , high = 2001:1 - fix(.15*%nobs)
```

This first instruction creates two variables; *low* is equal to the integer corresponding to the 15% of the observations following 1959:2 and *high* is equal to the integer corresponding to the last observation less 15% of the total number of observations. The first instruction below creates a variable *rss_test* that will be used to hold the residual sum of squares for the best fitting model. This value is initially set to be higher than any possible value of the estimated residual sum of squares. The second instruction creates the series that will hold the calculated residual sum of squares from each regression estimated.

```
compute rss_test = 1000000.0
set rss = 0.
```

Next, we need to create a series for the ordered values of *dlrgdp*. The first instruction below creates the series *thresh_test* that will hold the potential thresholds. Initially this series is SET equal to *dlrgdp*. The ORDER instruction orders the series from lowest to highest. *Thresh_test* now contains the ordered values of *dlrgdp*.

```
set thresh_test = dlrgdp
order thresh_test
```

Now we can use each value of *thresh_test* as a potential threshold. We begin by equating *thresh* with the very first value of *thresh_test*.

```
compute thresh = thresh_test(low)
```

Next, we begin the loop. For each value of *i* running from *low* to *high*, we take the associated value of *thresh_test* and use it as a potential threshold. Inside the DO loop, the program creates the series *plus*, *minus*, *y1_plus, y2_plus*, *y1_minus*, *y2_minus* and estimates a TAR model. The residual sum of squares is compared to *rss_test*. If the resulting residual sum of squares exceeds this value, two instructions in brackets are not executed, the value of *i* is incremented by 1 and the loop is repeated. However, if %rss is lower than *rss_test* (*i.e.*, if the residual sum of squares from the current regression is lower than any from the prior regressions) the bracketed instructions are executed. The value of *rss_test* is replaced by the value of %rss and the value of *thresh* is equated to the *thresh_test* (*i.e.*, current value of the test threshold). Once the loop is completed, *thresh* will hold the value of the threshold that yields the lowest residual sum of squares.

```
do i = low,high
  set plus = %if(dlrgdp{1}<thresh_test(i),0,1)
  set minus = 1 - plus
  set y1_plus = plus*dlrgdp{1}
  set y2_plus = plus*dlrgdp{2}
  set y1_minus = minus* dlrgdp{1}
  set y2_minus = minus* dlrgdp{2}

  lin(noprint) dlrgdp
  # plus y1_plus y2_plus minus y1_minus y2_minus

  com rss(i) = %rss
  if %rss < rss_test {
   compute rss_test = %rss
   compute thresh = thresh_test(i)
  }
end do i
```

Once the program loop exits the loop, we can display the consistent estimate of the threshold if we use:

```
dis 'We have found the attractor'
dis ' Threshold = ' thresh
```

```
   We have found the attractor
   Threshold =          0.01724
```

Finally, we can estimate the TAR model with the consistent estimate of the threshold using:

```
  set plus = %if(dlrgdp{1}<thresh,0,1)
  set minus = 1 - plus
  set y1_plus = plus*dlrgdp{1}
  set y2_plus = plus*dlrgdp{2}
```

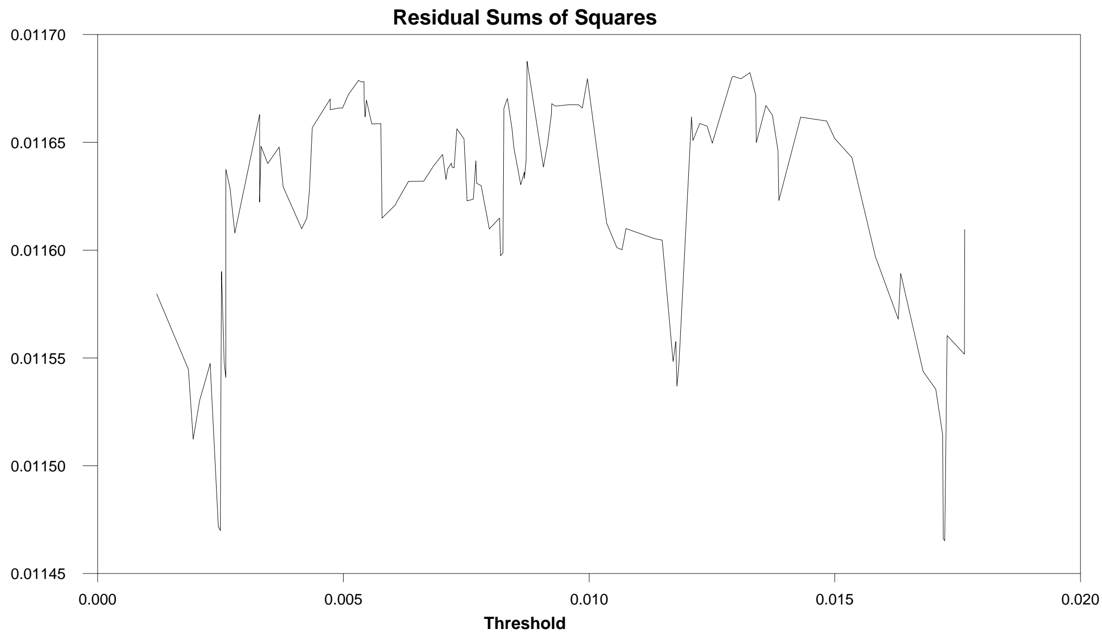
set y1_minus = minus* dlrgdp{1}
set y2_minus = minus* dlrgdp{2}

lin dlrgdp
 # plus y1_plus y2_plus minus y1_minus y2_minus

```
 Variable            Coeff       Std Error      T-Stat     Signif
*********************************************************************
1.  PLUS          0.022981605  0.009875828      2.32706  0.02121556
2.  Y1_PLUS      -0.443888007  0.406708488     -1.09142  0.27673082
3.  Y2_PLUS      -0.036693896  0.208404259     -0.17607  0.86046101
4.  MINUS         0.005019041  0.001047997      4.78918  0.00000379
5.  Y1_MINUS      0.237965600  0.104676378      2.27335  0.02433550
6.  Y2_MINUS      0.142325595  0.083235738      1.70991  0.08922107
```

Note that the series *rss* holds the residual sum of squares associated with each regression. The pattern of these values can be seen for every potential threshold value if we create a scatter diagram using:

scatter(Header='Residual Sums of Squares',style=lines,hlabel='Threshold') 1
 # thresh_test rss low high



It is the case that the threshold of `0.01724` does result in the lowest residual sum of squares. However, it is clear from the scatter diagram that there is a sharp local minimum around 0.0025. This suggests that there may be two thresholds implying the existence of a low, medium and high state.

4. Branching

IF-THEN-ELSE blocks allow you to have some control over the flow of your program. It is possible to have a set of instructions executed or completely skipped depending on a set of criteria that you specify. Nevertheless, program execution necessarily resumes with the first instruction following END IF. Note that in the example estimating the threshold model, RATS always displays "We have found the attractor."

Another flexible way to control the execution of your program is the BRANCH instruction. BRANCH enables you to jump to the particular point in your program that you specify. You can jump out of an IF-THEN-ELSE block, BRANCH back to some key instruction, or to jump to almost any other point in the program. The key rules are that you cannot jump back to the ALLOCATE instruction, into the middle of a nested block such as a DO loop or an IF-THEN-ELSE block.

Otherwise, there are only two rules about branching. The first is that BRANCH works only within a compiled section of a program. Second, to use BRANCH you must first label the point in the program where you want to branch to. You label this point by beginning that line with a colon and then immediately supplying the label. Consider the following three labels:

Examples:

```
:100
: jump_here
: exit
```

Any one of these three labels is a legitimate jump point—labels can be line numbers, multiple words or a word that conveys information about the execution of the program. The unconditional jump to *label* is performed by the instruction:

BRANCH *label*

4.1 Sample Program Using BRANCH

Suppose that you want to generate an *AR*(1) process for a series with 100 observations. In particular, suppose you want to generate a random-walk model in the form:

$$y_t = y_{t-1} + \varepsilon_t$$

where: ε_t is an i.d.d. random number with mean zero and variance equal to unity.

For each series you generate, you want to estimate an equation in the form:

$$y_t = \beta_0 + \beta_1 y_{t-1} + \varepsilon_t$$

Your goal is to save the estimate of β_1 and repeat the entire process 1000 times. Your aim is to obtain information about the distribution of the estimated values of β_1. A program that will perform this task is:

```
all 100
set y = 0.
set beta 1 1000 = 0.
do i = 1,1000
    set y 2 * = y{1} + %ran(1)
    lin(noprint) y ; # constant y{1}
    com beta(i) = %beta(2)
end do
table / beta
```

The first line of Program 4.4 sets the default length of all series to 100. The second line initializes all 100 values of $\{y_t\}$ to equal zero. The estimated values of β_1 will be stored in the series called *beta.* The third line initializes all 1000 values of *beta* to be zero. The initialization is necessary since RATS cannot create a series using COMPUTE. Instead, you create the series with SET so that later instructions can manipulate the individual entries of the series. There is a second interesting feature of the SET instruction. Although the default length for *beta* is 100, SET allows us to override this default by specifying the START and END values. The first instruction in the DO loop below generates a simulated random walk—the current value of *y* is equal to the previous value plus a random error term drawn from a normal distribution with mean zero and variance equal to 1. The second line in the loop estimates the model and the third equates the *i*-th value of the series *beta* with the estimated value of β_1. On exiting the loop, the TABLE instruction displays the sample statistics on the series beta.

If you run the program, your output will look something like:

```
Series     Obs        Mean        Std Error       Minimum        Maximum
BETA       1000 0.94693484752 0.04471912585 0.70122076370 1.04215477194
```

Note your output will be a bit different from mine in that your computer probably used different 'random numbers' than mine to generate the 1000 random-walk sequences. An issue that may arise concerns the use of random numbers in the program. We might want to ensure that the random numbers selected by RATS actually appear to be random. Towards this end, we can modify the routine as follows:

```
do i = 1,1000
:redraw
set epsilon = %ran(1)
    cor(number=4,qstats,noprint) epsilon
    if %signif < 0.005 ; branch redraw
set y 2 * = y{1} + epsilon
lin(noprint) y ; # constant y{1}
com beta(i) = %beta(2)
end do
```

Now the first instruction inside the loop is a label called :REDRAW. The program will jump back to this point and redraw a new set of ‘random-numbers’ if the current series does not appear to be random. The second instruction in the loop fills the series *epsilon* with 100 pseudo-random numbers. The next line obtains the first four autocorrelations of the *epsilon* series. The CORRELATE instruction creates the internal variable %signif that holds the significance level of the for the null hypothesis that these first four autocorrelations are all equal to zero. In the next line, we compare this significance level to 0.005. If the significance level for the Ljung-Box Q(4)-statistic is less than 0.001, the program jumps back to :REDRAW. Whenever such a jump occurs, RATS draws a new set of pseudo-random numbers.

5 Monte Carlo Experiments

A Monte Carlo experiment attempts to replicate an actual data generating process (DGP) using experimental means. You can use RATS to generate one or more series that characterize those produced by the actual data generating process in all key respects. A Monte Carlo experiment will generate a random sample of size T and the parameters and/or sample statistics of interest are calculated. This process in repeated N times (where N is a large number) so that the distribution of the desired parameters and/or sample statistics can be tabulated. These empirical distributions are used as estimates of the actual distributions. In fact, the previous Sample Program was a Monte Carlo experiment that can answer the following question: If the $\{y_t\}$ sequence is actually a random walk, what is the sampling distribution of the OLS estimate of β_1? Notice that the sample mean of the estimated values of β_1 (`0.94693484752`) is below the true value of unity and that the minimum value (`0.70122076370`) is further away from unity than the maximum value (`1.04215477194`).

As it turns out, Dickey and Fuller (1979) show that it is quite correct to infer that the estimate of β_1 is biased to be below unity. We will examine the Dickey-Fuller distribution in detail in two of the sample programs below. For now, suffice it to say that the use of the Monte Carlo method is justified by the Law of Large Numbers and various forms of the Central Limit Theorem. Consider the simplest case where v_t is an identically and independently distributed (*i.i.d.*) random number with mean μ and variance σ^2, *i.e.*,

$$v_t \sim (\mu, \sigma^2)$$

Consider the sample mean using T observations:

$$\bar{V} = (1/T)\sum_{t=1}^{T} v_i$$

As the sample size T grows sufficiently large:

a. $E(\bar{V}) \to \mu$
b. $E(\bar{V} - \mu)^2 \to \sigma^2 / T$
c. the distribution of $\bar{V}$ approaches a normal distribution with mean μ and variance σ^2/T.

Hence, the Monte Carlo mean ($\bar{V}$) is an unbiased estimate of the population mean with a variance of σ^2/T. Note that $\bar{V}$ is normally distributed around the true mean μ when T is large. The sample variance is an unbiased estimate of the population variance. Dividing the sample variance by T yields an unbiased estimate of the variance of the sample mean around the true mean. The point is that population means can be estimated using means of simple random samples—the accuracy is decreasing in σ^2/T so that the greater T, the greater the accuracy. (Note

that it is difficult to obtain a high degree of accuracy for the standard deviation since it is a function of $T^{-1/2}$ not T^{-1}).

When the distribution for v_t is more complicated, it is often not possible to obtain results in the form of *a*, *b*, and *c*. Moreover, it is often difficult to derive the properties of the distribution of the sample mean for the sample sizes typically used in econometric studies. This is when Monte Carlo analysis is most valuable.

There are two RATS functions that are especially useful for generating random numbers:

%RAN(X)	A random draw from a Normal (0, x) distribution
%UNIFORM(L,H)	A random draw from a uniform distribution ranging from lower bound L to upper bound H.

Examples

1. set x = %ran(1)

   This instruction equates each value of x with an i.i.d. random number drawn from a normal distribution with a mean of zero and a standard deviation equal to unity.

2. set x = %uniform(-1,1)

   This instruction equates each value of x with an i.i.d. random number drawn from a uniform distribution with a lower bound of –1 and an upper bound of 1.

3. Set y = 6 + %ran(1)
   set y 2 * = 3 + 0.5*y{1} + sqrt(0.75)*%ran(1)

   You might use these two statements to generate an *AR*(1) sequence with a mean of 6 [*i.e*., 6 = 3/(1 – 0.5)], an autoregressive parameter of 0.5 and a long-run variance of 1.0. The first statement initializes the series to equal the long-run mean plus a normally distributed random number with a variance equal to unity. The second equates the entries (beginning with entry 2) equal to the desired *AR*(1) process such that the unconditional variance of the y is unity.

5.1 A Simple Monte Carlo Experiment

A Modified Coin Tossing Problem**:** Suppose you toss a coin and a tetrahedron. For the coin, you get 1 point for a 'tail' and 2 points for a 'head.' The faces of the tetrahedron are labeled 1 through 4. For the tetrahedron, you get the number of points shown on the downward face. Your total score equals the number of points received for the coin and the tetrahedron. Of course, it is impossible to have a score of zero or 1. It is straightforward to calculate that the probabilities of scores 3, 4, and 5 equal 0.25 while the probabilities of scores 2 and 6 equal 0.125.

It is possible to simulate a roll of the coin and tetrahedron on the computer. Since the probability of a 'head' is 0.5, we can use the following program statement:

```
compute coin =  %if(%uniform(0,1) > .5,2,1)
```

The instruction equates the variable *coin* with 2 if a uniformly drawn random number over the interval [0, 1] exceeds 0.5. If the value of %uniform(0,1) is less than or equal to 0.5, the value of *coin* is 1. In this way, a 'head' is equal to 2 while a 'tail' is equal to 1. Similarly, we can simulate the toss of the tetrahedron.

```
com tet = fix(%uniform(1.0,5.0))
```

The first line equates x with a integer value of uniformly distributed random number over the interval [1, 5]. Note that FIX transforms a floating point number to an integer. As such, if the random number is between 1 and 2, the value of *tet* is the integer 1. Similarly, if the number is between 4 and 5, the value of *tet* is the integer 4. Thus, the probabilities that *tet* will equal 1, 2, 3 or 4 are all equal to 0.25.

We can obtain the total score using:

```
com score = coin + tet; dis score
```

To this point, we have done nothing but replicate the possible outcome of the game. However, it is simple to modify our program to replicate the game 1000 times. We can then calculate the proportion of instances in which we get scores of 2, 3, 4, 5, and 6. If the Monte Carlo method works, these sample proportions should come close to the true probabilities.

We will use the series *num* to hold the number of times we roll a 1, 2, 3, 4, 5 and 6 (Of course, the number of times a 1 is obtained will be zero). For example *num*(4) will equal the number of times a score of 4 is obtained. Consider the instructions from Program 4.5 on the file labeled CHAPTER4_1.PRG.

```
all 6
set num = 0
do j = 1,1000
   compute coin = fix(%if(%uniform(-1,1) > 0,2,1))
   com tet = fix(%uniform(1.0,5.0))
   compute num(coin+tet) = num(coin+tet) + 1   ;*  add 1 to sum(total # faces)
end do j
print / num
```

The ALLOCATE instruction sets the default series length equal to 6—we need only six entries to hold the series *num*. Line 2 initializes all entries of *num* to equal zero. The DO loop indicates that the lines 4 through 6 will be performed 1000 times. Line 4 is the simulated coin toss and line 5 is the simulated tetrahedron toss. To understand line 6, recall that *coin* + *tet* is equal to the score. Thus, line 6 increments the appropriate entry of score by 1. For example, if *coin* + *tet* = 5,

line 6 adds 1 to the value stored in num(5). Once we exit the loop, the values of *num* are printed to the screen. If you run the program yourself, you will obtain something like:

```
ENTRY          SUM
    1            0
    2          129
    3          262
    4          241
    5          252
    6          116
```

A score of 1 was never observed, a 2 was observed 12.9% of the time, a 3 was observed 26.2% of the time, and so on. In fact, the sample proportions are reasonably close to the true probabilities. If you increase the number of simulated rolls, you should obtain output that is closer to the true probabilities.

Of course, your answers will be somewhat different from mine. Your computer will not draw precisely the same random numbers as mine. However, this can be an undesirable feature of a program. Suppose we are debugging a program or want to perform a sensitivity analysis such that we want to use precisely the same random numbers from one computer run to the next. This can be done by seeding RATS' random number generator in the identical fashion from one run to the next. The instruction that does this is:

SEED *Integer*

If you include SEED 2001 immediately after the allocate statement, you should get the same output as that shown above.

5.2 Downward Bias in an AR Model

It is well known that the OLS estimates of a first-order autoregressive process are biased towards zero. The size of the bias increases as the magnitude of the autoregressive coefficient increases from zero to unity. Consider the following autoregressive model:

$$y_t = \alpha_0 + \alpha_1 y_{t-1} + \varepsilon_t$$

It is possible to write a simple program to measure the size of the bias and to see how it is affected by the magnitude of α_1. Towards this end, we could perform the following tasks:

Step 1: Generate a series in the form of the AR(1) model using a value of $\alpha_1 = 0.2$.

Step 2: Estimate the series using OLS and calculate the discrepancy between the estimated value of α_1 and 0.2.

Step 3: Repeat Steps 1 and 2 a total of 1000 times. Display the average value of the discrepancy.

Step 4: Repeat Steps 1 to 3 using alternative values of α_1.

Consider the first three lines of Program 4.6 in the file CHAPTER4_1:

```
all 100
set discrep 1 1000 = 0.
set y = 0.
```

Since the program uses only simulated data, we do not include a CALENDAR instruction. The ALLOCATE instruction sets the default length of any series we create equal to 100. For each value of α_1, we will store the 1000 discrepancies between the actual and estimated values in the series called *discrep*. The third instruction initializes all 100 observations in the simulated $\{y_t\}$ sequence to be zero.

Now we have to decide which values of α_1 to include. It might make sense to include 0.5 (since it is midway between 0. and 1.0) and two values near 1.0—say 0.9 and 0.99. Finally, we might want to include 0.0 for comparison purposes. Consider the next two instructions:

```
com alpha1 = 0.
dofor alpha1 = .2 .5 .9 .99 0.
```

The DOFOR instruction acts in a way that is similar to the DO instruction. Notice, however, that the index is not an integer and does not increase in any precise way from one loop to the next (In fact, for the last iteration, the index decreases from 0.99 to 0.) .The instruction, com alpha1 = 0. is necessary since we need to initialize the index to be a real number. Otherwise, RATS would have expected a list of integers for *alpha1*. Note that we could have used any real number (*e.g*., com *alpha1* = -.5) for the initialization.

The next instruction reseeds the random number generator each time α_1 takes on a new value. This way, the random numbers do not change as the values of the various values for α_i change. The first instruction inside the DO i loop, generates y_t as the current value of α_1 multiplied by y_{t-1} plus an *i.i.d.* normally distributed random number with mean zero and variance equal to 1. Since we are interested only in the discrepancy between the actual and estimated values of α_1, the output is suppressed. The last line in the DO *i* loop equates entry *i* of *discrep* with the difference between the actual and estimated values of α_1. On exiting the DO *i* loop, *discrep* will contain 1000 values of the discrepancy.

```
seed 2001
do i = 1,1000
   set y 2 * = alpha1*y{1} + %ran(1)
   lin(noprint) y ; # constant y{1}
   com discrep(i) =  alpha1 - %beta(2)
end i
```

The sample statistics of *discrep*, including the mean, are displayed using the TABLE instruction. If you want, you can display many interesting sample statistics using the command: statistics(fractiles) discrep. The final instruction closes the DOFOR loop. As such, α_1 changes from 0.2, to 0.5, to 0.9, to 0.99 and finally to 0.0.

table / discrep
end dofor

```
Series      Obs        Mean        Std Error      Minimum       Maximum
DISCREP    1000  0.0166418436  0.0972591579 -0.2935443023  0.3218321838
DISCREP    1000  0.0267019830  0.0905765140 -0.2170420309  0.3415792684
DISCREP    1000  0.0443060778  0.0615724835 -0.0715945104  0.3328153050
DISCREP    1000  0.0556321214  0.0456496779 -0.0321881468  0.2691217328
DISCREP    1000  0.0103641243  0.0974089395 -0.3157692119  0.2998128902
```

You can see that the mean value of *discrep* increases as α_1 increases. The smallest mean value of the discrepancy (*i.e.*, `0.0103641243`) occurs for $\alpha_1 = 0$. Why didn't this number turn out to be exactly zero—doesn't this show a bit of an 'upward bias' in the discrepancy? The answer involves the fact that we used only 1000 replications of the *AR*(1) process. In many Monte Carlo experiments, 100,000 replications are used in order to get a more precise estimate of the true sampling distribution.

5.3 Power of the Dickey-Fuller Test

The last example used the first-order *AR*(1) process:

$$y_t = \alpha_0 + \alpha_1 y_{t-1} + \varepsilon_t$$

where: $\{\varepsilon_t\}$ is generated from a white noise process.

In spite of the downward bias in the estimate of α_1, most applied econometricians would still use a standard *t*-test to determine whether α_1 is significantly different from zero. The bias is only large when α_1 is large. The situation is quite different if we want to test the hypothesis $\alpha_1 = 1$. Now, under the null hypothesis, the $\{y_t\}$ sequence is a non-stationary process. As such, it is inappropriate to use classical statistical methods to estimate and perform significance tests on the coefficient α. Dickey and Fuller (1979) used Monte Carlo methods to obtain the appropriate critical values to test for the presence of a unit root. The following program produces results that mimic their τ_μ distribution.

The logic of the method is identical to that for the *AR*(1) process used in the previous example. The difference is that we will generate random-walk sequences instead of mean-reverting autoregressive processes. The ALLOCATE instruction in Program 4.7 of the file labeled CHAPTER4_1.PRG, uses a sample size of 100. Line 2 initializes a series called *tstat*—this series will hold all of the *t*-statistics generated in the Monte Carlo experiment. Notice that we will perform the experiment 10000 times and will have 10000 *t*-statistics. As such we want *tstat* to

have a length of 10000. Line 3 initializes the *y* sequence to zero and sets the first entry equal to a normal distributed random number with a mean of zero and standard deviation of unity. Lines 5 – 9 are performed 10000 times. Line 5 generates a sequence that mimics the random-walk:

$$y_t = y_{t-1} + \varepsilon_t$$

Line 6 takes the first-difference of the sequence and calls the result *dy*. The usual form of the Dickey-Fuller test is to estimate the *AR*(1) equation in the form:

$$\Delta y_t = \alpha_0 + \rho y_{t-1} + \varepsilon_t$$

As such, lines 7 and 8 estimate *dy* on a constant and *y*{1}. Since $\rho = \alpha_1 - 1$, we want to know whether it is possible to reject the null hypothesis $\rho = 0$. The sampling distribution of the *t*-statistic for the null hypothesis $\rho = 0$ is stored as entry *i* of *tstat*. Once the 10000 replications of this Monte Carlo experiment are completed, *statistics*(*fractiles*) *tstat* produces the distribution of the *t*-statistics.

```
all 100
set tstat 1 10000 = 0.
set y = 0.0 ; compute y(1) = %ran(1)

do i = 1,10000
   set y 2 100 =  y{1} + %ran(1)
   diff y / dy
   linreg(noprint) dy * 100
   # constant y{1}
   compute tstat(i) = %tstats(2)
end do i
statistics(fractiles) tstat
```

```
Statistics on Series TSTAT
Observations   10000
Sample Mean    -1.5411735979          Variance                0.725122
Standard Error  0.8515412362          SE of Sample Mean       0.008515
t-Statistic     -180.98637            Signif Level (Mean=0) 0.00000000
Skewness           0.19591            Signif Level (Sk=0)   0.00000000
Kurtosis           0.35386            Signif Level (Ku=0)   0.00000000
Jarque-Bera      116.14118            Signif Level (JB=0)   0.00000000

Minimum     -4.8214558640             Maximum      2.3794080538
01-%ile     -3.4919869782             99-%ile      0.6158486194
05-%ile     -2.8861478731             95-%ile     -0.0669050993
10-%ile     -2.5748048837             90-%ile     -0.4212769335
25-%ile     -2.0988978255             75-%ile     -1.0183779154
Median      -1.5835459078
```

Your output will look a bit different from mine since we have not used the same SEED *integer*. Notice that 90% of the *t*-statistics exceeded -2.57, 95% exceeded -2.88 and 99% exceeded -3.49.

If you look at the Dickey-Fuller table, you will see that the at the 10%, 5% and 1% significance levels are -2.58, -2.89 and -3.51, respectively.

Thus, suppose you had a sample with 100 observations and estimated the series in question as an *AR*(1) process. If it turned out that the estimated value of $\rho = -.05$ (so that the estimated value of $\alpha_1 = 0.95$) with a standard error of 0.02, could you reject the null hypothesis $\rho = 0$? The answer is no! Although the estimate $\hat{\rho}$ is 2.5 standard deviations away from unity, it is not permissible to use a traditional *t*-statistic. Instead, the Monte Carlo results show that when the true value of $\rho = 0$ (*i.e.*, $\alpha_1 = 1$ so that the true data generating process is a random-walk), we would obtain a *t*-statistic that exceeds `-2.5748048837` (i.e., is less than `-2.5748048837` in absolute value) more than 90% of the time. As such, at the 1%, 5% and 10% significance levels, we cannot reject the null hypothesis $\rho = 0$.

Power of the Test

Now that we know critical values for the Dickey-Fuller test, it is instructive to show how to ascertain its power. Since the Dickey-Fuller confidence intervals exceed those for the usual *t*-test, it is to be expected that the power of the Dickey-Fuller test is low. Program 4.8 on the file labeled CHAPTER4_1 illustrates a way to determine how often the Dickey-Fuller test rejects a false null hypothesis. To be more specific, given that the true data generating process is such that $\rho < 0$ (*i.e.*, $\alpha_1 < 1$), the program calculates the probability of rejecting the false null hypothesis $\rho = 0$ and accepting the correct alternative hypothesis $\rho < 0$. The first two lines of the program set the default sample size of any series to 200 and seed the random number generator.

```
all 200
seed 237
```

Suppose we want to determine the power of the Dickey-Fuller test for various values of α_1 close to unity. The program uses the four values: $\alpha_1 = 0.8$, 0.9, 0.95 and 0.99. DOFOR will loop over real values if the index has been defined as a real number. This is accomplished using com aplha1 = 0.0. The first time through the DOFOR loop, *alpha1* = 0.8 and *rho* = -.2. The next line below initializes the variables *hits10, hits5* and *hits1* to equal zero; we will use these variables to hold the number of times the Dickey-Fuller test correctly rejects the null hypothesis of a unit-root at the 10%, 5% and 1% significance levels, respectively. The fifth line below initializes the $\{y_t\}$ sequence to equal its unconditional mean value of zero with a variance of 1.0.

```
com alpha1 = 0.0
dofor alpha1 = 0.80 0.90 0.95 0.99
com rho = alpha1 - 1.0

compute hits10 = hits5 = hits1 = 0
set y = %ran(1)
```

The DO *j* loop prepares RATS to do 10,000 Monte Carlo replications. Within each loop, the $\{y_t\}$ sequence is constructed as an *AR*(1) process with *i.i.d.* normally distributed errors. (Note that the value of *alpha1* is determined by the number of times the program has completed the DOFOR

loop). As constructed, the variance of the $\{y_t\}$ sequence is 1.0. If you take the variance of each side of the SET instruction you obtain var(y_t) = $(\alpha_1)^2$var(y_{t-1}) + [1 - $(\alpha_1)^2$]. Hence, var(y_t) = var(y_{t-1}) = 1.

```
do j = 1,10000
   set y 2 * = alpha1*y{1} + sqrt(1-alpha1**2)*%ran(1)
```

The next instruction creates the first-difference of $\{y_t\}$ and calls the resulting series $\{dy_t\}$. Next, dy_t is regressed on a constant and y_{t-1}. Notice that we use only the last 99 observations of the simulated $\{dy_t\}$ sequence for the regression. This technique is used to eliminate the problem of choosing the initial condition for the $\{y_t\}$ sequence. Since we do not want to impose any particular initial condition on the simulated sequence, we generate 200 values for $\{y_t\}$ and use only the last 100. Since we lose one additional observation by differencing, the regression uses 99 observations so as to replicate a data set containing 100 observations. The *t*-statistic for the estimated value of α_1 is stored in the variable *df*.

```
  dif y / dy
  linreg(noprint) dy 102 *
  # constant y{1}
  compute df = %tstats(2)
```

The three IF statements below compare the calculated *t*-statistic to the Dickey-Fuller critical values.[28] If the *t*-statistic is less than -2.58, the null hypothesis of $\rho = 0$ is rejected at the 10% significance level. Thus, if *df* < -2.58 the value of *hits10* is increased by one. Similarly, if *df* < -2.89, *hits5* is increased by one and if *df* < -3.51, *hits1* is increased by one. After the 10,000 DO *j* loops, we can use these three numbers to indicate the number of times that the test correctly rejected the null hypothesis of a unit-root.

```
  if df < -2.58 ; compute hits10 = hits10 + 1
  if df < -2.89 ; compute hits5 = hits5 + 1
  if df < -3.51 ; compute hits1 = hits1 + 1
end do j
```

The last section of the program displays the key output; for each value of ρ, *hits10, hits5* and *hits1* are shown.

```
display ' rho = ' rho
display '   10%     5%     1%    '
display ###### hits10 ######  hits5  ######  hits1
display '     '
end dofor
```

[28] The values used are those reported in Dickey and Fuller (1979), not those obtained in Program 4.7 above.

```
rho =        -0.20000
   10%      5%      1%
 9601   8730   5131

rho =        -0.10000
   10%      5%      1%
 5206   3256    865

rho =        -0.05000
   10%      5%      1%
 2350   1272    269

rho =        -0.01000
   10%      5%      1%
 1154    561    109
```

When ρ = -0.2, the test does reasonably well—at the 5% significance level, the false null hypothesis of a unit-root is rejected in 87.30% of the Monte Carlo replications. However, when ρ = -0.05, the probability of correctly rejecting the null hypothesis of a unit-root is estimated to be only 12.72%.

Jazzing Up the Program

A Monte Carlo analysis can take a very long time to execute. We could, for example, run the program for a number of sample sizes and for additional values of *alpha1*. Your computer screen can remain blank for a very long time before you see any output. You have probably noticed an INFOBOX that quickly appears and disappears when you read in a data set. Similarly, it is possible to 'keep track' of the number of iterations completed by using the INFOBOX instruction. It is necessary to use INFOBOX three times in the program; once to DEFINE the box, a second time to update or MODIFY the box and a third time to REMOVE the box. The syntax for INFOBOX is:

INFOBOX(ACTION=, *other options*) '*messagestring*'

where:
ACTION = DEFINE/[MODIFY]/REMOVE

With ACTION = DEFINE, the key options are:

PROGRESS/[NOPROG]	This option determines whether the box will include a progress bar. You must specify values for LOWER and UPPER when you use PROGRESS.
LOWER =	Integer value for the lower bound.
UPPER =	Integer value for the upper bound.

With ACTION = MODIFY, use the option:

CURRENT = The current integer value for the progress bar

Thus, to keep track of the number of iterations over j that have been completed, immediately before the instruction DO j = 1,10000, include:

infobox(action=define,progress,lower=1,upper=10000) 'Replications Completed'

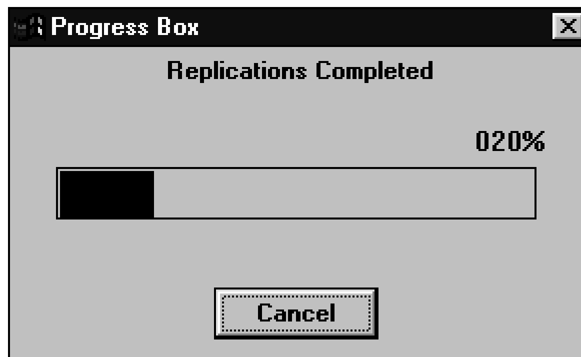
Immediately after the instruction DO j = 1,10000, include:

infobox(current=j)

Immediately after the instruction END DO j, include:

infobox(action=remove)

If you now run the program you should see something that looks like:



5.4 The Enders-Granger Statistic

In addition to shamelessly promoting my own work, the following program illustrates the use of several other BRANCHING and IF statements within a Monte Carlo study. In Enders and Granger (1998), we generalized the Dickey-Fuller methodology to consider the null hypothesis of a unit-root against the alternative hypothesis of a threshold autoregressive (TAR) model. The simple version TAR model is:

$$\Delta y_t = I_t\rho_1 y_{t-1} + (1 - I_t)\rho_2 y_{t-1} + \varepsilon_t$$

where: $I_t = \begin{cases} 1 \; if \;\; y_{t-1} \geq 0 \\ 0 \; if \;\; y_{t-1} < 0 \end{cases}$

The basic idea is that autoregressive decay might not be symmetric. If $y_{t-1} < 0$, the indicator function $I_t = 0$, so that: $\Delta y_t = \rho_2 y_{t-1} + \varepsilon_t$ and if $y_{t-1} \geq 0$, $I_t = 1$ so that $\Delta y_t = \rho_1 y_{t-1} + \varepsilon_t$. Notice that if $\rho_1 = \rho_2 = 0$, the process is a random walk. However, as in the Dickey-Fuller test, it is not possible

to use a classical F-statistic to test the null hypothesis $\rho_1 = \rho_2 = 0$. Instead, the following program can be used to accomplish the following tasks:

1. Generate a random-walk sequence.
2. Estimate the sequence as a threshold autoregressive model.
3. BRANCH if the simulated sequence does not cross the threshold
4. Calculate the sample F-statistic for the null hypothesis $\rho_1 = \rho_2 = 0$.
5. Obtain the distribution of the sample F-statistics.

There is one technical task that must be done. It is necessary to ensure that there are a sufficient number of observations on each side of the threshold to estimate the TAR model. One way to check is immediately after task 3. If either of the t-statistics for ρ_1 or ρ_2 is zero (or undefined), the simulated series needs to be eliminated from the study. Thus, after estimating the TAR model, we check to see is the absolute value of the product of the two t-statistics is less than 0.0000001. If so, we branch to step 1 to obtain a replacement series. Program 4.9 of the file CHAPTER4_2.PRG contains the program that we used to calculate the Monte Carlo values.

The first four lines perform some bookkeeping tasks. The default length of a series is 200. Also, the random number generator is seeded and the series f is initialized with 50000 values of zero. The sample values of the F-statistic for the null hypothesis $\rho_1 = \rho_2 = 0$ will be stored in f. Line 4 initializes the value of the $\{y_t\}$ sequence to zero.

```
all 200
seed 2001
set f 1 50000 = 0.
set y = 0.
```

The next line of the program begins the DO loop and the following line is labeled *reset*. The program jumps back to this line if the simulated series does not cross the threshold (*i.e.*, the program will BRANCH to *reset* if the two t-statistics are too small.

```
do j = 1,50000
   :reset
```

Task 1 is performed by the next three lines of the program. The next two lines in the program simulate a random-walk sequence of 200 observations. The third instruction takes the first-difference of the simulated series.

```
   * TASK 1: COMPUTE the y series
   set y 2 200 = y{1} + %ran(1)
   diff y 2 200 dy
```

Next, the %IF instruction is used to set the indicator—called *plus*—to equal zero if y_{t-1} is negative and to equal 1 otherwise. The second and third instructions shown below create *zplus* (the indicator multiplied by the lagged value y_{t-1}) and *zminus* (one minus the indicator multiplied by the lagged value y_{t-1}).

```
* TASK 2: Estimate the TAR Model
set plus = %if(y{1}<0,0,1)
set zplus = plus*y{1}
set zminus = (1-plus)*y{1}
```

In the two instructions below, RATS estimates the TAR model using only observations 102 through 200. Thus, as in the previous program, this mimics a data set with 100 total observations.

```
linreg(noprint) dy 102 200
# zplus zminus
```

The next two instructions are used to check the two *t*-statistics to see if they are both different from zero. If the product of the two is sufficiently close to zero, the program branches back to the line labeled :*reset*. Hence, if the two *t*-statistics are sufficiently low, the program branches back to the point where a new sequence is generated.

```
*  TASK 3
compute t1 = %tstats(1), t2 = %tstats(2)
if abs(t1*t2) < 0.0000001
     branch reset
```

If the program does not BRANCH to *reset*, the next three lines of the program obtain the sample *F*-value for the null hypothesis $\rho_1 = \rho_2 = 0$. The last line within the loop stores this sample *F*-value as entry *j* of the series *f*. At the end of 50,000 replications, the program exits the END DO *j* loop. The STATISTICS instruction with the FRACTILES option produces the output shown below. Be aware that the program takes a long time to complete the 50000 replications.

```
     * TASK 4: Calculate the Sample F-statistic
     exclude(noprint)
     # zplus zminus
     compute f(j) = %cdstat
end do j

* TASK 5
statistics(fractiles) f
```

```
Statistics on Series F
Observations   50000
Sample Mean     1.6524588263          Variance                 1.378597
Standard Error  1.1741364996          SE of Sample Mean        0.005251
t-Statistic      314.70023            Signif Level (Mean=0) 0.00000000
Skewness           1.73656            Signif Level (Sk=0)   0.00000000
Kurtosis           5.01475            Signif Level (Ku=0)   0.00000000
Jarque-Bera    77521.44422            Signif Level (JB=0)   0.00000000

Minimum      0.0013651902             Maximum     14.4743025649
01-%ile      0.1977081049             99-%ile      5.6967829958
05-%ile      0.3681509883             95-%ile      3.9233591336
10-%ile      0.5025774653             90-%ile      3.1857908038
25-%ile      0.8187794721             75-%ile      2.1691250789
Median       1.3579015615
```

Suppose that you estimated a series as a threshold process:

$$\Delta y_t = I_t\rho_1 y_{t-1} + (1 - I_t)\rho_2 y_{t-1} + \varepsilon_t$$

$$\text{where: } I_t = \begin{cases} 1 \textit{ if } \ y_{t-1} \geq 0 \\ 0 \textit{ if } \ y_{t-1} < 0 \end{cases}$$

If the sample F-statistic of the null hypothesis $\rho_1 = \rho_2 = 0$ was 3.5, you would be able to reject the null hypothesis at the 10% significance level but not the 5% level. You can modify the program by (*i*) including an INFOBOX, (*ii*) obtaining the critical values for additional sample sizes, and (*iii*) obtaining the critical values when Chan's (1993) method (see Section 3.1 in this chapter) is used to estimate the threshold.

5.5 Inference in a Cointegrated System

This is one of my favorite programs. It illustrates a number of RATS advanced features and the folly of using traditional distribution theory to perform hypothesis tests on a cointegrating vector. Suppose that x_t and y_t are two non-stationary time-series variables that are cointegrated of order 1. The error-correction representation of the system is:

$$y_t = y_{t-1} - \alpha_1 [\ \beta_0 + y_{t-1} - \beta_1 x_{t-1}\ ] + A_{11}(L)\Delta y_{t-1} + A_{12}(L)\Delta x_{t-1} + e_{1t}$$

$$x_t = x_{t-1} + \alpha_2 [\ \beta_0 + y_{t-1} - \beta_1 x_{t-1}\ ] + A_{21}(L)\Delta y_{t-1} + A_{22}(L)\Delta x_{t-1} + e_{2t}$$

It is assumed that e_{1t} and e_{2t} are serially uncorrelated but the covariance $Ee_{1t}\,e_{2t}$ need not be zero. As in Chapter 2, if the variances and covariance are time-invariant, we can write the variance/covariance matrix as:

$$\Sigma = \begin{bmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{21} & \sigma_{22} \end{bmatrix}$$

where: $\mathrm{Var}(e_{it}) = \sigma_{ii}$ and $\mathrm{Cov}(e_{1t},e_{2t}) = \sigma_{12} = \sigma_{21}$.

The nature of the system is such that $\{y_t\}$ and $\{x_t\}$ are unit-root processes that are linked by the cointegrating vector $\beta_0 + y_{t-1} - \beta_1 x_{t-1}$. Suppose you estimate the regression equation:

$$y_t = \hat{\beta}_0 + \hat{\beta}_1 x_t + e_t$$

Stock (1987) proves that the OLS estimates of the coefficient of a cointegrating vector converge faster than similar estimates for stationary variables. As such, it is inappropriate to use standard *t*-statistics and confidence intervals to perform inference on $\hat{\beta}_1$. The following Monte Carlo experiment illustrates the problem:

Step 1. Generate $\{x_t\}$ and $\{y_t\}$ as a cointegrated system. For simplicity, we will generate the series without an intercept in the cointegrating vector and with all values of $A_{ij}(L) = 0$. Hence, the simulated model is:

$$y_t = y_{t-1} - \alpha_1 [\, y_{t-1} - \beta_1 x_{t-1} \,] + e_{1t}$$

$$x_t = x_{t-1} + \alpha_2 [\, y_{t-1} - \beta_1 x_{t-1} \,] + e_{2t}$$

Step 2. Estimate the cointegrating vector:

$$y_t = \hat{\beta}_0 + \hat{\beta}_1 x_t + e_t$$

and use the *t*-distribution to form a 95% confidence interval around $\hat{\beta}_1$. The point of the exercise is to determine how well (or poorly) the *t*-distribution works for cointegrated variables. If this confidence interval includes the actual value of β_1 used in STEP 1, add 1 to the variable *success*. If the *t*-distribution is appropriate, *success* should be increased in 95% of the Monte Carlo replications.

Also, store the value of $\hat{\beta}_1$ so that it can be compared to the actual value of β_1.

Step 3. Repeat the experiment 2000 times and examine the variable *success*.

Step 4. Repeat Steps 1 – 3 for different values of α_1, α_2, β_1 and the elements of Σ.

The program relies on the EQUATION and the SIMULATE instructions. If you are familiar with these instructions, you can skip the remainder of this section. As used in the program, the syntax for the EQUATION instruction is:[29]

---

[29] If you have an equation with no *AR* or *MA* terms omit these fields and the MORE option.

EQUATION(COEFFS=*coeffs*,*other options) equation depvar ARlags MAlags*
list of explanatory variables

where:

equation	Equation name.
depvar	Dependent variable.
ARlags	Number of *AR* lags
MAlags	Number of *AR* lags
coeffs	Vector of coefficient values assigned to the *constant*, *AR* terms, *MA* terms, and *explanatory variables*, respectively.

Other Principal Options

CONSTANT /NOCONSTANT	Include a constant in the equation (CONSTANT is the default for ARMA models).
MORE/[NOMORE]	For *ARMA* equations, MORE indicates that other explanatory variables are on a supplementary card.
VARIANCE=	Value for the variance of the residual series.

SIMULATE creates a Monte Carlo replication of a model using normally distributed error terms. The syntax used in the program is:

simulate(model=*modelname,*results=*results*) * *number start V*

where:

number	The number of observations in the simulated model
start	The start date for the first entry.
V	The variance/covariance matrix of the residuals.
results	The simulated values of the first equation are stored in *results(1),* the simulated values of the second equation are stored in *results(2)*, and so on.

The Program

The first four instructions of Program 4.10 on the file CHAPTER4_2.PRG set the default size of a series to 150 entries, SEED the random number generator, and initialize the *y* and *x* series to equal zero. The COMPUTE instruction sets $\beta_1 = 1.0$ and $\alpha_1 = \alpha_2 = 0.1$.

```
allocate 150
seed 2002
set y = 0.0
set x = 0.
com beta1 = 1.0 , alpha1 = 0.1 , alpha2 = 0.1

equation(noconstant,more,coeffs=|| 1.-alpha1, alpha1*beta1 ||) eq1 y ; # y{1} x{1}
equation(noconstant,more,coeffs=|| 1.-alpha2*beta1, alpha2 ||) eq2 x ; # x{1} y{1}
```

The first EQUATION instruction creates an equation in the form:

$y_t = (1 - \alpha_1)y_{t-1} + \alpha_1\beta_1 x_{t-1} + e_{1t}$

Notice that NOCONSTANT is used. Hence, the value of $(1 - \alpha_1)$ is assigned to the *AR*(1) coefficient and the value of $\alpha_1\beta_1$ is assigned to the coefficient for x_{t-1}. [*Note*: MORE indicates that an explanatory variable appears on a supplementary card]. The VARIANCE= option is not used. Similarly, the second EQUATION instruction creates an equation of the form:

$x_t = (1 - \alpha_2\beta_1)x_{t-1} + \alpha_2 y_{t-1} + e_{2t}$

GROUP creates a model called *unit* using the equations *eq1* and *eq2*. The first COMPUTE instruction creates the symmetric matrix *v*; the elements of *v* represent the elements of Σ. Here, $\sigma_{11} = \sigma_{22} = 1$ and $\sigma_{12} = \sigma_{21} = 0$. The second COMPUTE instruction initializes the variable *success* to be zero and the SET instruction initializes the 2000 entries of the series *betahat* to be zero; *betahat* is used to store the estimated values of β_1.

```
group unit eq1 eq2
com [symmetric]v = || 1.0 , 0.0 | 0.0 , 1.0 ||
com success = 0.
set betahat 1 2000 = 0.
```

The instructions within the DO *i* loop are performed 2000 times. SIMULATE creates a Monte Carlo replication of the model *unit* using the error structure specified by *v*. The simulated series contain 149 observations and begin with entry 2. Note that *sims(1)* contains the simulated valued of $\{y_t\}$ and *sims(2)* contains the simulated values of $\{x_t\}$. The LINREG instruction estimates a regression of *sims(1)* on a constant and *sims(2)*. Only the last 100 observations are used to avoid any problems concerning the initial conditions used for y_1 and x_1. COMPUTE stores the estimated value of β_1 in *betahat*(*i*).

```
do i = 1,2000
   sim(model=unit,results=sims) * 149 2 v
   lin(noprint) sims(1) 51 * ; # constant sims(2)
   com betahat(i) = %beta(2)
   if beta1.gt.%beta(2)-1.96*%stderrs(2).and. beta1.le.%beta(2)+1.96*%stderrs(2)
        com success = success+1
   dis beta1-1.96*%stderrs(2)    beta1+1.96*%stderrs(2)  success
end do i
```

The IF instruction needs a bit of explanation. Recall that %STDERRS(2) holds the standard error of %BETA(2). Hence, %beta(2) – 1.96*%STDERRS(2) is the lower bound of the 95% confidence interval constructed using the *t*-distribution. Similarly, %beta(2) + 1.96*%STDERRS(2) is the upper bound of the confidence interval. If the actual value β_1 exceeds the lower bound *and* is less than the upper bound, β_1 is within the confidence interval. Thus, if the condition is TRUE, *success* is increased by 1. The DISPLAY instruction displays the lower and the upper bound for each replication along with the current value of *success*.

After the loop is completed, the summary STATISTICS for *betahat* are displayed along with the percentage of instances in which β_1 fell within the confidence interval.

sta(fractiles) betahat
dis 'The percentage of successes is:' success/20.

```
Statistics on Series BETAHAT
Observations   2000
Sample Mean     0.6271085179       Variance               0.070829
Standard Error  0.2661369878       SE of Sample Mean      0.005951
t-Statistic      105.37861         Signif Level (Mean=0) 0.00000000
Skewness          -0.46258         Signif Level (Sk=0)   0.00000000
Kurtosis           0.07801         Signif Level (Ku=0)   0.47716964
Jarque-Bera       71.83512         Signif Level (JB=0)   0.00000000

Minimum     -0.3874254319              Maximum      1.3107230084
01-%ile     -0.0732848428              99-%ile      1.1432856243
05-%ile      0.1554922902              95-%ile      1.0196162470
10-%ile      0.2617774667              90-%ile      0.9400523351
25-%ile      0.4552469550              75-%ile      0.8244537550
Median       0.6507239420

The percentage of successes is:      16.85000
```

Notice that 16.8% of the confidence intervals contained the true value of β_1. Moreover, the average value was `0.6271085179` whereas the actual value was 1.0. Clearly, it is inappropriate to use the usual confidence intervals for $\hat{\beta}_1$. Now, if you rerun the program using $\beta_1 = 0.5$, you will obtain:

```
The percentage of successes is:      33.30000
```

Hence, the level of β_1 affects the accuracy of the 95% confidence interval. Moreover, the value of the elements of Σ are also important. If you replace the elements of *v* with com [symmetric]v = || 1. , -.5 | -5. , 1. ||, you will obtain:

```
The percentage of successes is:      56.95000
```

6. Antithetic Random Variables

A complete Monte Carlo experiment involves many replications and a substantial amount of computer time. It is typical to obtain a sampling distribution over a number of different parameter values and sample sizes. In the Enders-Granger (1998) example discussed above, we examined various types of TAR models using sample sizes ranging from 50 to 1000 with 100,000 Monte Carlo replications for each; I would turn on my computer, go to lunch and hope it would be done when I returned. The problem is the critical values will change from one replication to the next; we wanted to use a number of replications such that the critical values stabilized in the second decimal place. One technique for reducing the sampling variance inherent in a Monte Carlo study (and reduce the number of replications) is to use 'antithetic' random numbers.

The basic idea is to pool two different unbiased estimates of the parameters of interest so as to obtain an estimate with a small variance. Suppose that $\alpha(1)$ and $\alpha(2)$ are the two different estimates of the parameter α from replications 1 and 2. Consider the pooled estimator, $\overline{\alpha}$, that is the simple average of the two:

$$\overline{\alpha} = 0.5[\ \alpha(1) + \alpha(2)\ ]$$

The variance of $\overline{\alpha}$ is:

$$var(\overline{\alpha}) = 0.25\{\ var[\ \alpha(1)\ ] + var[\ \alpha(2)\ ] + 2\ cov[\alpha(1), \alpha(2)\ ]\ \}$$

If the two estimates have a negative covariance, the pooled estimator will have a far smaller variance than those from the two independent draws. The problem is to find a way to ensure that the estimates have a negative covariance (*i.e.*, are antithetic). In certain circumstances, the solution is remarkably simple—use the same set of numbers *with the sign reversed* for the second replication. Intuitively, if the first replication gives an estimate of α that is too high, the second should give an estimate that is too low.

6.1 Bias in NLLS Estimates

Antithetic random variables can be quite effective when working with nonlinear models. The sample program illustrated below replicates the results of a Monte Carlo experiment reported in Davidson and MacKinnon (1993). The issue is to find the bias, if any, of the exponent α in the nonlinear regression model:

$$y_t = \beta x_t^{\alpha} + \varepsilon_t$$

Davidson and MacKinnon (1993) use a sample size of 50 with a single set of the $\{x_t\}$ sequence drawn from a uniform distribution on the interval [5, 15] using parameter values $\beta = 1$, $\alpha = 0.5$ and ε_t drawn from an *i.i.d.* standardized normal distribution. The program will use the following four instructions to set up the nonlinear estimation:

```
    NONLIN alpha beta
    FRML monte y = beta*x**alpha
    COM alpha = 0.48, beta = 0.98
    NLLS(frml=monte,noprint) y
```

Program 4.11 of the file labeled CHAPTER4_2 performs the Monte Carlo experiment without using antithetic variates. The first three lines define the default size of a series to equal 50, seed the random number generator, and create the 50 entries of the sequence $\{x_t\}$ as random draws from a uniform distribution on the interval [5, 15]. The next two lines initialize 1000 values of *alpha_hat* and *beta_hat* to zero. These two series will be used to store the estimates of α and β, respectively.

```
all 50
seed 2001
set x = %uniform(5,15)
set alpha_hat 1 1000 = 0.
set beta_hat 1 1000 = 0.
```

The next three instructions prepare RATS to perform a nonlinear estimation of the model and provide initial guesses of α and β. The statements within the DO loop will be performed 1000 times. The 50 values of y_t are created as $x_t^{0.5}$ plus an *i.i.d.* disturbance drawn from a standardized Normal distribution. The NLLS instruction actually perform the estimation.

```
NONLIN alpha beta
FRML monte y = beta*x**alpha
COM alpha = 0.48, beta = 0.98

do i = 1,1000
    set y = x**0.5 + %ran(1)
    NLLS(frml=monte,noprint) y
```

The next instruction stores the estimated value of α in the *i*-th entry of *alpha_hat* and the estimated value of β as the *i*-th entry of *beta_hat.* On exiting the DO loop, the TABLE instruction is used to provide the summary statistics for *alpha_hat* and *beta_hat*.

```
    com alpha_hat(i) = alpha, beta_hat(i) = beta
end do i
table / alpha_hat beta_hat
```

```
Series       Obs       Mean        Std Error      Minimum       Maximum
ALPHA_HAT    1000 0.50061759618 0.14924811080 0.00760025999 1.00199740576
BETA_HAT     1000 1.06013032231 0.37423173774 0.30586651854 2.97669650556
```

The first half of the sample is quite similar to the second half of the sample. Consider:

table * 500 alpha_hat beta_hat

```
Series      Obs       Mean        Std Error      Minimum       Maximum
ALPHA_HAT   500 0.50526653814 0.14860337610 0.00760025999 0.93395005105
BETA_HAT    500 1.04927811557 0.38210832158 0.33895348276 2.97669650556
```

table 501 * alpha_hat beta_hat

```
Series      Obs       Mean        Std Error      Minimum       Maximum
ALPHA_HAT   500 0.49596865421 0.14989449989 0.07604389005 1.00199740576
BETA_HAT    500 1.07098252906 0.36624672687 0.30586651854 2.74200765785
```

Notice the mean values of the estimates are quite close to their true values. However, the standard errors of *alpha_hat* and *beta_hat* are very large.

Results with Antithetic Variables**:** The remaining portion of the program takes advantage of antithetic random numbers. The next two instructions create the series *alpha_bar* and *beta_bar*; these two series will hold the averaged estimates of α and β.

```
set alpha_bar 1 500 = 0.
set beta_bar 1 500 = 0.
```

Again, the next three instructions prepare RATS to perform a nonlinear estimation of the model and provide initial guesses of α and β.

```
nonlin alpha beta
frml monte y = beta*x**alpha
com alpha = 0.48, beta = 0.98
```

Next, two DO loops are created. The instructions within the DO *i* loop are executed 500 times. The series *eps* will contain 50 *i.i.d.* draws from a standardized normal distribution. The next portion of the program is the critical part:

```
do i = 1,500
  set eps = %ran(1)
  do j = 0,1
    set y = x**0.5 + (1-j)*eps - j*eps
    nlls(frml=monte,noprint) y
```

Each time through the DO j loop, y_t is set equal to $x^{0.5}$ = (1-*j*)**eps* - *j***eps*. The first time through the DO *j* loop, *j* = 0 so that:

$y_t = x_t^{0.5} + \varepsilon_t$

The second time through the DO *j* loop, *j* = 1 so that:

$y_t = x_t^{0.5} - \varepsilon_t$

The NLLS instruction estimates the nonlinear equation using the formula previously defined as *monte*. Next, the values of *alpha_bar*(*i*) and *beta_bar*(*i*) are updated—50% of the estimated value of *alpha* is added to *alpha_bar*(*i*) and 50% of the estimated value of *beta* is added to *beta_bar*(*i*). At the end of the DO loop, entry *i* of each of these series contains the desired pooled estimate. After the process is performed 500 times, the DO *i* loop is completed. The TABLE instruction is used to obtain the distribution of *alpha_bar* and *beta_bar*.

```
    com alpha_bar(i) = alpha_bar(i) + 0.5*alpha
    com beta_bar(i) = beta_bar(i) + 0.5*beta
end do j
end do i
tab /  alpha_bar beta_bar
```

```
    Series       Obs        Mean        Std Error       Minimum        Maximum
    ALPHA_BAR    500 0.50192972485 0.00880740910 0.45061006127 0.56465375430
    BETA_BAR     500 1.05368592894 0.07719055534 0.99439278963 1.65755600806
```

The mean values are quite similar to those of *alpha_hat* and *beta_hat*. However, the standard errors of *alpha_bar* and *beta_bar* are about 17 times and 5 times smaller than those of *alpha_hat* and *beta_hat,* respectively.

7. Bootstrapping

Programming for bootstrapping is similar to that for a Monte Carlo experiment. However, there is an essential difference. In a Monte Carlo study, you generate the random variables from a given distribution such as the Normal. The bootstrap takes a different approach—the random variables are drawn from their observed distribution. This is quite useful if you are working with data that is not Normally distributed. In essence, the bootstrap uses the *plug-in principle*—the observed distribution of the random variables is the best estimate of their actual distribution.

The idea of the bootstrap was developed in Efron (1979). Suppose you have a data set of size T and want to estimate the mean μ and the standard deviation of the mean σ_μ. The key point made by Efron is that the observed data set is a random sample of size T drawn from the actual probability distribution generating the data. As such, the empirical distribution function is defined to be the discrete distribution that places a probability of $1/T$ on each of the observed values. It is the empirical distribution function—and not some pre-specified distribution such as the Normal—that is used to generate the random variables. The bootstrap sample is a random sample of size T drawn **with** replacement from the observed data putting a probability of $1/T$ on each of the observed values. Once the bootstrap sample has been drawn, it is possible to estimate the mean of the bootstrap sample. Call this estimate using the first bootstrap sample μ_1^*. If N bootstrap samples are drawn, there are N bootstrap estimates of μ that we denote by μ_1^* through μ_N^*. The bootstrap estimate of σ_μ is the standard deviation of the $\{\mu_i^*\}$ sequence.

To be more specific, suppose that we have the following 10 values of x_t:

t	1	2	3	4	5	6	7	8	9	10
x_t	0.8	3.5	0.5	1.7	7.0	0.6	1.3	2.0	1.8	-.05

The sample mean is 1.87 and the standard deviation is 2.098. Next, we will draw 100 bootstrap samples in order to estimate the standard deviation of the mean. Each bootstrap sample consists of 10 randomly selected values of x_t drawn with replacement—each of the 10 values listed above is drawn with a probability of 0.1. It might seem that this resampling repeatedly selects the same sample. However, by sampling with replacement, some elements of x_t will appear more than once in the bootstrap sample. The first three bootstrap samples might be:

t	1	2	3	4	5	6	7	8	9	10	μ_i^*
x_1^*	3.5	1.7	-0.5	0.5	1.8	2.0	1.7	0.6	0.6	7.0	1.89
x_2^*	-0.5	0.6	0.6	0.8	1.7	7.0	1.8	3.5	1.8	0.8	1.81
x_3^*	0.5	0.6	7.0	1.3	1.3	7.0	1.3	1.8	3.5	0.6	2.49

where: x_i^* denotes bootstrap sample i.

Notice that 0.6 and 1.7 appear twice in the first bootstrap sample, 0.6, 0.8 and 1.8 appear twice in the second and that 1.3 appears three times in the third bootstrap sample. As such, the sample means are not identical. For 100 such bootstrap samples, the standard deviation of the 100 values of μ_i^* is the estimate of the standard deviation of the mean σ_μ. The algorithm is as follows:

1. Select N = 100 bootstrap samples each consisting of 10 data points (N should be between 25 and 200 for estimating standard errors).

2. Evaluate the parameter of interest (μ_i^*) for each bootstrap sample.

3. Estimate the standard deviation of the mean by the sample standard deviation of the N replications.

Efron (1979) shows that such bootstrap estimates converge to the population standard deviation as N goes to infinity. Program 4.12 will estimate the standard deviation of the mean using the bootstrap. The first two instructions set the default size of a series to 10 and seed the random number generator. The third line generates x_t. Note that x_t is not quite normally distributed since *fix*(10*%*ran*(2)) generates the integer value of 10 times an *i.i.d.* normally distributed random number with a standard deviation of 2. Dividing this result by 10 and adding 2 yields the x_t series listed above: 0.8, 3.5. 0.5, 1.7, 7.0. 0.6, 1.3, 2.0, 1.8 and -0.5. The series *mean* is created; the 100 entries of mean will hold the bootstrap mean values μ_1^* through μ_{100}^*.

```
all 10
seed 2002
set x = 2+fix(10*%ran(2))/10.
set mean 1 100 = 0.
```

The instructions in the DO loop will be performed 100 times. A series y of length 10 is created. Note that *fix*(%uniform(1,11)) draws the integer value of a uniformly distributed random variable on the interval (1, 11). Thus, *fix*(%uniform(1,11)) will be one of the integers from 1 to 10 each selected with a probability of 0.1. Suppose these integers happen to be 2, 1, 5, 6, 10, 10, 4, 5, 7, and 2. The values of y will be such that y(1) = x(2), y(2) = x(1), y(3) = x(5), … , y(10) = x(2). Hence, y is a bootstrap sample--it consists of randomly selected values of x drawn with replacement (the probability of selecting any particular value is 0.1). The STATISTICS instruction generates the mean of y and this value is stored in entry i of *mean.* At the end of the DO loop, mean contains the 100 bootstrap means. The standard deviation of these means is obtained using the TABLE instruction.

```
do i = 1,100
   set y = x(fix(%uniform(1,11)))
   sta(noprint) y ; com mean(i) = %mean
end do i
table / mean
```

```
   Series      Obs       Mean         Std Error       Minimum        Maximum
   MEAN        100 1.90330000000 0.61347998082 0.78000000000 4.31000000000
```

Thus, the bootstrap estimate of the standard deviation of the mean is about `0.61348`. Notice this is quite similar to the standard deviation of x (= 2.098) divided by the square root of the number of observations (2.098/10**.5 = 0.663) .

7.1 Bootstrapping Regression Coefficients

Suppose you have a data set with T observations and want to estimate the effects of variable x on variable y. Towards this end, you might estimate the linear regression:

$$y_t = \beta_0 + \beta_1 x_t + \varepsilon_t$$

Although the properties of the estimators are well-known, you might not be confident using standard t-tests if the estimated residuals do not appear to be normally distributed. One way to estimate the sample properties of $\hat{\beta}_0$ and $\hat{\beta}_1$ is to use the method of *Bootstrapped Residuals*.[30] After estimating the model, you perform the following steps:

Step 1: Calculate the residuals as: $e_t = y_t - \hat{\beta}_0 - \hat{\beta}_1 x_t$.

Step 2: Generate a bootstrap sample of the error terms e^* containing the elements e_1^*, e_2^*, … , e_T^*. Use the bootstrap sample to calculate a bootstrapped y series (called y^*). For each value of i running from 1 to T, calculate y_i^* as:

$$y_i^* = \hat{\beta}_0 + \hat{\beta}_1 x_i + e_i^*$$

Note that the estimated values of the coefficients are treated as fixed. Moreover, the values of x_i are treated as fixed quantities so that they remain the same in the bootstrap sample.

Step 3. Use the bootstrap sample to estimate new values of β_0 and β_1 calling the resulting values β_0^* and β_1^*.

Step 4. Repeat Steps 2 and 3 many times and calculate the sample statistics for the estimated values $\hat{\beta}_0$ and $\hat{\beta}_1$ using the sample properties of β_0^* and β_1^*.

Of course, it is possible to apply the identical procedure to a nonlinear regression model as well. Program 4.13 of the file CHAPTER4_2.PRG finds bootstrap confidence intervals for the nonlinear regression model discussed in conjunction with antithetic variates:

$$y_t = \beta x_t^{\alpha} + \varepsilon_t$$

---

[30] The alternative method is to bootstrap the paired (y_i, x_i) combinations.

Instead of using two actual series from MONEY_DEM.XLS, the program uses artificially generated data for $\{x_t\}$ and $\{y_t\}$. As such, we will know how the two series data are actually generated and the true values of α and β. As in Program 6, the first two lines set the default length of a series to 50, seed the random number generator and generate the $\{x_t\}$ and $\{y_t\}$ sequences. Hence, the true parameter values are $\beta = 1$ and $\alpha = 0.5$.

```
all 50
seed 2001
set x = %uniform(5,15)
set y = x**0.5 + %ran(1)
```

The trick to understanding the program is to pretend that we do not know the actual data generating process. The next four lines of the program use the NONLIN-NLLS block to estimate α and β. Note that the NLLS instruction saves the residuals in series *e*. The estimated values of α and β (i.e., $\hat{\alpha}$ and $\hat{\beta}$) are saved as *alpha_hat* and *beta_hat*.

```
nonlin alpha beta
frml monte y = beta*x**alpha
com alpha = 0.48, beta = 0.98
nlls(frml=monte) y / e
```

```
   Variable     Coeff        Std Error      T-Stat     Signif
*************************************************************
1.  ALPHA      0.4194759184 0.1538968309    2.72570  0.00892917
2.  BETA       1.1695671363 0.4194023378    2.78865  0.00756525
```

The parameter estimates are reasonable. Note that stopping at this point would require us to use a normal approximation to obtain a confidence interval for the coefficients. For a 90% confidence interval, 1.64 standard deviations on either side of *alpha* gives us an interval of 0.16709 to 0.67187, and two standard deviations on either side of beta gives us an interval of 0.48175 to 1.85739.

Thus, Step 1 has been completed. In preparation for Step 2, the series *alpha_star* and *beta_star* are initialized to contain 1000 entries.

```
com alpha_hat = %beta(1) , beta_hat = %beta(2)
set alpha_star 1 1000 = 0.
set beta_star 1 1000 = 0.
```

The first instruction inside the DO loop constructs the bootstrap sample of the residuals *e_star* (note that the BOOT instruction can also be useful for these situations). The series *e_star* will consist of 50 values drawn from the regression residuals *e*. Each value of *e_star* is a random draw with replacement from *e*. Note that each element of *e* has a 1/50 chance of being selected since *fix*(*%uniform*(1,51)) generates an integer on the interval (1, 50). The next instruction uses the bootstrap residuals to construct the bootstrap *y* series (y^*). Notice that the coefficient values

used in the construction are those originally estimated from the data. Moreover, the values of x_t have not been modified or resampled. Thus, Step 2 is completed.

```
do i = 1,1000
   set e_star = e(fix(%uniform(1,51)))
   set y_star = beta_hat*x**alpha_hat + e_star
```

Step 3 requires us to use the bootstrapped *y_star* sequence to obtain new coefficient estimates. This is accomplished in the four instructions in the NONLIN-NLLS block below. The coefficient estimates for α^* and β^* are stored in entry *i* of *alpha_star* and *beta_star*, respectively.
On exiting the DO loop, these two series will each contain 1000 values of bootstrapped coefficients. The sample properties of these coefficients are obtained using the STATISTICS instruction with the FRACTILES option.

```
   nonlin alpha beta
   frml monte y_star = beta*x**alpha
   com alpha = alpha_hat, beta = beta_hat
   nlls(frml=monte,noprint) y_star
   com alpha_star(i) = %beta(1) , beta_star(i) = %beta(2)
end do i
```

sta(fractiles) alpha_star

```
Statistics on Series ALPHA_STAR
Observations   1000
Sample Mean     0.4179067424          Variance               0.023137
Standard Error  0.1521082297          SE of Sample Mean      0.004810

Minimum     -0.0878398657             Maximum      1.0489024933
01-%ile      0.0706360305             99-%ile      0.8022149869
05-%ile      0.1715603199             95-%ile      0.6706139537
10-%ile      0.2215775497             90-%ile      0.6020511077
25-%ile      0.3161065322             75-%ile      0.5150522575
Median       0.4178528143
```

sta(fractiles) beta_star

```
Statistics on Series BETA_STAR
Observations   1000
Sample Mean    1.24610274529          Variance               0.198084
Standard Error 0.44506578336          SE of Sample Mean      0.014074

Minimum     0.26104747424             Maximum     3.85257134145
01-%ile     0.48336068574             99-%ile     2.59363073397
05-%ile     0.64414171297             95-%ile     2.08050680611
10-%ile     0.75625461184             90-%ile     1.84697927718
25-%ile     0.94211587333             75-%ile     1.48805437120
Median      1.17862968423
```

We can use the distribution for *alpha_star* to form a symmetric 90% confidence interval for $\hat{\alpha}$. Since 5% of the values lie below `0.1715603199` and 5% lie above `0.6706139537`, the 90% confidence is the range `0.1715603199` to `0.6706139537`. Similarly, 90% confidence interval for $\hat{\beta}$ using the bootstrap distribution is `0.64414171297` to `2.08050680611`. Notice that this confidence interval is much wider than that obtained using the normal approximation.

7.2 The *AR* Coefficients of Real *GDP* Growth

Step 2 needs to be modified for a time series model due to the presence of lagged dependent variables. As such, the bootstrap y^* is constructed in a slightly different manner. Consider the simple *AR*(1) model:

$$y_t = \beta_0 + \beta_1 y_{t-1} + \varepsilon_t$$

As in Step 2, we can construct a bootstrap sample of the error terms e^* containing the elements e_1^*, e_2^*, … , e_T^*. Now, the bootstrap y^* sequence using this sample of error terms. In particular, given the estimates of β_0 and β_1 and an initial condition for y_1^*, the remaining values of the y^* sequence can be constructed using:

$$y_i^* = \hat{\beta}_0 + \hat{\beta}_1 y_{i-1} + e_i^*$$

For higher order *AR*(*p*) models, initial conditions for y_1^* through y_p^* are needed. Typically, these values are selected by random draws from the $\{y_t\}$ sequence.[31]

Program 4.14 uses this method to bootstrap the coefficients of an *AR*(2) model of the *dlrgdp* in the form:

$$dlrgdp_t = \beta_0 + \beta_1 dlrgdp_{t-1} + \beta_2 dlrgdp_{t-2} + \varepsilon_t$$

The issue is that the coefficient of $dlrgdp_{t-2}$ has a significance level of 0.0756. However, since the residuals are not normally distributed, it might be wise to use the bootstrap to obtain confidence intervals for $\hat{\beta}_2$. The first six lines of Program 4.14 read in the data set MONEY_DEM.XLS, seed the random number generator with 2001 and estimate the model:

lin dlrgdp / resids
 # dlrgdp{1 to 2} constant

---

[31] A second, although less common, bootstrapping technique used in time series models is called "Moving Blocks." For an *AR*(*p*) process, select a length *L* that is longer than *p*; *L* is the length of the block. To construct the bootstrap y^* series, randomly select a group of *L* adjacent data points to represent the first *L* observations of the bootstrap sample. In total, you need to select *T/L* of these samples to form a bootstrap sample with *T* observations. The idea of selecting a block is to preserve the time-dependence of the data. However, observations more than *L* apart will be nearly independent. Use this bootstrap sample to estimate the bootstrap coefficients β_0^* and β_1^*.

```
   Variable           Coeff       Std Error      T-Stat     Signif
*******************************************************************
1.  DLRGDP{1}       0.2508977521 0.0769801061    3.25925  0.00135976
2.  DLRGDP{2}       0.1362250820 0.0762100846    1.78749  0.07571568
3.  Constant        0.0051566068 0.0010217954    5.04661  0.00000119
```

The next two instructions do some bookkeeping. The first line below stores the estimates $\hat{\beta}_1$, $\hat{\beta}_2$ and $\hat{\beta}_0$ in the variables *beta1_hat*, *beta2_hat and beta0_hat*, respectively. The second line stores the standard error and *t*-statistic of $\hat{\beta}_2$ in the variables *se_2* and t2, respectively.

```
com beta1_hat = %beta(1), beta2_hat = %beta(2) , beta0_hat = %beta(3)
com se_2 = %STDERRS(2), t2 = %tstats(2)
```

If we use the normal approximation, we can obtain the lower (*l*) and upper (*u*) limits of the 90%, 95% and 99% confidence intervals for $\hat{\beta}_2$ using:

```
dis 'Normal Approximation'
com l = beta2_hat - 1.644*se_2 , u = beta2_hat + 1.644*se_2 ; dis '  90% '  l h
com l = beta2_hat - 1.96*se_2 ,  u = beta2_hat + 1.96*se_2 ; dis '  95% '  l h
com l = beta2_hat - 2.57*se_2 ,  u = beta2_hat + 2.57*se_2 ; dis '  99% '  l h
```

```
Normal Approximation
 90%        0.01094       0.26151
 95%       -0.01315       0.28560
 99%       -0.05963       0.33208
```

In order to see if the residuals appear normal, we can use the STATISTICS instruction:

sta resids

```
Statistics on Series RESIDS
Quarterly Data From 1959:04 To 2001:01
Observations   166
Sample Mean    0.00000000000          Variance            7.095529e-05
Standard Error 0.00842349604          SE of Sample Mean       0.000654
t-Statistic        0.00000            Signif Level (Mean=0) 1.00000000
Skewness           0.05245            Signif Level (Sk=0)   0.78455071
Kurtosis           1.40625            Signif Level (Ku=0)   0.00029372
Jarque-Bera       13.75405            Signif Level (JB=0)   0.00103121
```

Although the residuals do not appear to be skewed, there is excess kurtosis and the Jarque-Bera test clearly rejects normality. In preparation for the construction of bootstrap confidence intervals, a bit more bookkeeping is necessary. The integer *boot_num*, containing the number of replications, is set equal to 1000. If you want to alter the number of replications simply change this single program statement. The next instruction creates the series *beta2_star*; the 1000 bootstrap estimates of β_2 (*i.e.*, the values of β_2^*) will be stored in this series. The third instruction creates the values of the y^* sequence.

```
com boot_num = 1000
set beta2_star 1 boot_num = 0
set y_star = 0.
```

The instructions in the DO loop will be executed 1000 times. The first instruction in the loop randomly selects an integer in the range 1959:2 to 2001:1. The value of ii serves as a randomly selected entry value. The COM instruction uses this value to equate the first two values of y_star (i.e., y_1^* and y_2^*) with two consecutive values of *dlrgdp*. Next, the bootstrapped e^* series is created from residuals of the *AR*(2) model. Each value e_1^*, e_2^*, … , e_T^* is a random draw with replacement from *resids.* Note that each element of *resids* has a 1/166 chance of being selected since *fix*(*%uniform*(1959:4,2001:1+1)) generates integers on the interval (4, 169).

```
do k = 1,boot_num
    com ii = fix(%uniform(1959:2,2001:1))
    com y_star(1) = dlrgdp(ii), y_star(2) = dlrgdp(ii+1)
    set e_star = resids(fix(%uniform(1959:4,2001:1+1)))
```

The first instruction below creates the series *y_star* for entries 1959:4 through 2001:1 using the bootstrapped residuals *e_star*. As such, the first two entries of *y_star* remain the values SET by the first instruction inside the DO loop. Notice that the estimated values of $\hat{\beta}_1$, $\hat{\beta}_2$ and $\hat{\beta}_0$ are used in the construction of *y_star*. The second instruction below uses the bootstrap sample to estimate β_1^*, β_2^* and β_0^*. The value of β_2^* is stored as the *i*-th entry of *beta2_star*:

```
    set y_star 3 * = beta1_hat*y_star{1} + beta2_hat*y_star{2} + beta0_hat + e_star
    lin(noprint) y_star ; # y_star{1 to 2} constant
    compute beta2_star(k) = %beta(2)
end do k
```

At this point, *beta2_star* contains the 1000 values of β_2^*. We can use the STATISTICS instruction to obtain the FRACTILES of *beta2_star*:

sta(fractiles) beta2_star

```
Statistics on Series BETA2_STAR
Minimum     -0.1559817399              Maximum      0.3371048578
01-%ile     -0.0558588474              99-%ile      0.2886134655
05-%ile     -0.0113366577              95-%ile      0.2431975939
10-%ile      0.0172751868              90-%ile      0.2141146300
25-%ile      0.0661351327              75-%ile      0.1684400633
Median       0.1204672933
```

From the output, it is clear that a symmetric 90% confidence interval includes zero--the lower and upper boundaries are -0.0113366577 and 0.2431975939, respectively. This suggests that we can exclude the second lag of *dlrgdp*. The FRACTILES option may not provide all of the information you would like concerning the distribution of *beta2_star*. One way to obtain

whatever set of FRACTILES you want is to order *beta2_star* from the smallest to the largest value:

order beta2_star

Now the 50-*th* value (5% of 1000 replications) and the 950-*th* (95% of 1000) can be used to obtain the lower and upper bounds of a 90% confidence interval. Similarly, the same logic can be used to construct 95% and 99% confidence intervals. The following four instructions produce these three confidence intervals:[32]

```
dis 'Confidence intervals for beta2'
dis ' 10% ' beta2_star(fix(.05*boot_num))  beta2_star(fix(.95*boot_num))
dis '  5% ' beta2_star(fix(.025*boot_num)) beta2_star(fix(.975*boot_num))
dis '  1% ' beta2_star(fix(.005*boot_num)) beta2_star(fix(.995*boot_num))
```

```
Confidence intervals for beta2
  10%      -0.01137      0.24317
  5%       -0.03765      0.25895
  1%       -0.09205      0.3096
```

[32] An alternative way to construct confidence intervals it to use the bootstrapped *t*-statistics for the null hypothesis $\beta_2^* = \hat{\beta}_2$. Program 4.14a (not discussed here) shows how to construct confidence intervals using the "Bootstrapped T-Statistics."

Chapter 5:
Vector and Matrix Manipulations

Although RATS is not intended to be a matrix programming language, it has evolved to the point where you can use it to perform very complicated econometric tasks entirely in matrix notation. In fact, you can create various vectors and matrices from your data set. For example, you can create a Y vector, an X matrix and obtain the ordinary least squares (OLS) coefficient estimates from $(X'X)^{-1}X'Y$. I will show you how to do that, and more, in the section entitled *Making Matrices from Your Data*. However, most RATS users will not want their programs to consist entirely of matrix manipulations. One of the strengths of RATS is that you can use vectors and matrices to complement the existing instruction set. Programming in a pure matrix language can be cumbersome; it is a lot easier to let RATS perform some of the matrix manipulations for you. For example, the LINREG instruction creates the coefficient vector (%beta) and the $(X'X)^{-1}$ matrix. You can call and manipulate both of these matrices without having to construct them yourself. Similarly, many RATS instructions accept a vector or a matrix as inputs. In essence, you pass information to the instructions using matrices. In fact, there are so many RATS instructions that utilize or create matrices that advanced RATS programmers often incorporate some matrix manipulations into their overall program.

1. Creating Matrices and Vectors

In traditional matrix algebra, the elements of a matrix are numbers. RATS allows you to be much more flexible. Not only can the elements of a matrix be real numbers, complex numbers or integers, they can also be strings, labels or series. Be aware that many RATS instructions have options allow you to create matrices. For example, the ESTIMATE instruction discussed in Chapter 2 can have the form:

ESTIMATE(OUTSIGMA=*V*,residuals=*resids*,coefficients=*coef*)

where:

OUTSIGMA=	Computes and saves the covariance matrix of the residuals.
COEFFICIENTS=*coef*	Creates a matrix of the coefficients. Column *i* contains the coefficients of the *i*-th equation.
RESIDUALS=*resids*	Creates a vector of series. The residuals from the first equation are stored in the series called resids(1), the residuals from the second equation are stored in the series called resids(2), and so forth.

Also, you can create vectors and matrices in RATS using the DECLARE, COMPUTE and MAKE instructions. The main use of each is:

DECLARE	The most general instruction for creating matrices.
COMPUTE	Useful for creating small matrices or vectors.
MAKE	Creates a matrix (or vector) from data series.

Whenever you create a matrix using the DECLARE instruction, you need to inform RATS about three features of the matrix. First, you can create different types of matrices. The types will usually be:

rectangular:	an r x c matrix where r = # rows and c = # columns
vector:	a one dimensional array
symmetric:	a symmetric r x r matrix

Second, you need to DIMENSION the matrix by indicating the number of rows and columns it contains. You can DIMENSION the matrix directly on the DECLARE instruction or on a DIM instruction.[33] It is useful to know that matrices can be redimensioned within any program. However, you cannot DECLARE a matrix as a different data type. For example, if *A* is a vector of integers, you cannot DECLARE *A* to be a vector of real numbers or a RECTANGULAR matrix of integers. Third, you need to instruct RATS concerning the type of information contained in the matrix. A matrix might contain one or more series or a set of integers, real numbers or string variables.

1.1 Declare

DECLARE is the most general instruction for creating a matrix. DECLARE allows you to completely specify the type of the matrix (*i.e.*, rectangular, vector or symmetric) along with its dimension and the contents of the elements. The syntax for DECLARE is:

DECLARE *matrix type list of names*

where:

matrix type	Will usually be *rectangular, vector,* or *symmetric*. By default, the elements are real numbers. You indicate other element types by the use of brackets [ ]. The most typical element types are INTEGER, SERIES, LABELS or STRING.
list of names	The names of the matrices you wish to create. Note that you can include the DIMENSION in parentheses.

It is important to note that within a compiled procedure, you cannot dimension a matrix on the DECLARE instruction. Instead, you must use a separate DIMENSION statement.

---

[33] Within a procedure, you cannot DECLARE and DIMENSION a matrix on the same instruction. See Chapter 6 for details on writing your own procedures.

Examples

1. To create a vector *x* that can hold 100 real numbers, you can use:

```
declare vector x(100)
```

or:

```
declare vector x
dim x(100)
```

NOTE: Within a procedure, you must use the second set of instructions.

2. To create a vector *x* that can hold 100 integers and a vector *y* that can hold 50 integers, you can use:

```
declare vector[integer] x(100) y(50)
```

or:

```
declare vector[integer] x y
dim x(100) y(50)
```

3. To create a vector *x* that can hold 10 series, you can use:

```
declare vector[series] x(10)
```

4. To create a vector *x* that can hold 100 real numbers and a 10 x 20 matrix *b* containing real numbers, you can use:

```
dec vector x(100)
dec rectangular b(10,20)
```

or:

```
dec vector x
dec rectangular B
dim x(100) B(10,20)
```

Note: You can include different types of matrices on a DIMENSION instruction but not on a DECLARE instruction.

1.2 COMPUTE

In many ways, you create and manipulate matrices just as scalars. You PRINT a series but you DISPLAY (or WRITE) a scalar and a matrix. Similarly, you manipulate a series using SET but you can manipulate scalars and matrices using COMPUTE. In fact, the simplest way to create and manipulate matrices is with the COMPUTE instruction. COMPUTE allows you to implicitly DECLARE and DIMENSION a matrix so that there is no need to have an explicit DECLARE instruction. Moreover, COMPUTE also allows you to completely specify the data type. There are two rules you need to know when using compute:

1. COMPUTE creates a RECTANGULAR matrix if you do not specify *vector* or *symmetric*. You can specify (i.e., explicitly DECLARE) the type of the matrix using braces [ ].

2. COMPUTE determines the type of the elements (*e.g*., *integer, real, string*) from the expression you enter.[34]

Examples

1. com a = ||'dlrgdp', 'dlm3' , 'drs' ||
dis a

```
dlrgdp dlm3 drs
```

The COMPUTE instruction creates a 1 x 3 RECTANGULAR matrix containing the string variables *dlrgdp, dlm3* and *drs*. You can refer to each element by its position in the matrix. For example:

dis a(1,2)

```
dlm3
```

Suppose that the residuals from a VAR using *dlrgdp, dlm3* and *drs* are stored in series 1, 2, and 3. You could graph each of the residuals series using:

```
do i = 1,3
        graph(header='Residuals from '+a(1,i)) 1 ; # i
end do
```

The headers of the graphs would be: *Residuals from dlrgdp*, *Residuals from dlm3* and *Residuals from drs.*

---

[34] Note that if you use brackets to explicitly create a vector or symmetric matrix, you should also specify the data type (i.e., integer, string, series, …). For example: [vector[string]]

2. The syntax of the supplementary card in LINREG is:
   # *list of explanatory variables*

   The list of explanatory variables can include labels and/or series numbers. Suppose that you want to regress the log of *M2* (*lm2*) on a constant, the log of *RGDP* (*lrgdp*), the log of the *GDP* deflator (*lp*) and a short-term interest rate. Program 5.1 in the file CHAPTER5.PRG reads in the seven series from the data set MONEY_DEM.XLS and creates these variables. If you enter the TABLE instruction, you will see that *tb3mo* is series 6, *tb1yr* is series 7, *lrgdp* is series 8, *lm2* is series 9, and *lp* is series 10. All of the following produce the identical regression output:

```
lin lm2
# constant lrgdp lp tb3mo

lin lm2
# 0 8 10 6      ;* NOTE: 0 is equivalent to constant. If you type: pri / 0 you get constant

com a = || 0, 8, 10, 6 ||
lin lm2
# a
```

   You could embed this routine in a DOFOR loop to obtain a regression of *lm2* on each of the short-term interest rates:

```
dofor i = tb3mo tb1yr
    com a = || 0, 8, 10, i ||
    lin lm2
    # a
end do for
```

   The first time though the loop, $i = 6$ and the regressor list uses *tb3mo*. The second time through the loop, $i = 6$ and the regressor list uses *tb1yr*.

3. Program 4.1 also illustrates the difference between different data types, Consider:

```
com names = || 'rgdp' , 'tb3mo' ||
dis names(1, 2)
          tb3mo
```

   The 1 x 2 rectangular matrix *names* consists of the string variables *rgdp* and *tb3mo*. DISPLAY element *a*(1, 2) produces the string `tb3mo`. PRINT operates on one or more series. As such, if you try to PRINT *names* or an element of *names*, you will get an error message:

```
pri / names
## SX22. Expected Type SERIES, Got RECTANGULAR(STRING) Instead
>>>> pri / names<<<<
```

Now create a 1 x 2 rectangular matrix *b* that contains the series numbers of series *rgdp* and *tb3mo*.

com b = || rgdp, tb3mo ||
pri / b

```
ENTRY          RGDP            TB3MO
 1959:01        2273.0      2.773333333333
 1959:02        2332.4      3.000000000000
 .......
 2000:04        9393.7      6.016666666667
 2001:01        9439.9      4.816666666667
```

pri / b(1, 2)

```
ENTRY          TB3MO
 1959:01   2.773333333333
 1959:02   3.000000000000
 1959:03   3.540000000000
 .......
 2000:03   6.016666666667
 2000:04   6.016666666667
 2001:01   4.816666666667
```

Because *b* consists of a integer for which there are corresponding series, you can PRINT *b* or an element of *b*. If you wanted to DISPLAY the second ENTRY of *tb3mo* [*i.e.* entry *tb3mo*(1959:02)] you could enter:

dis b(1, 2)(2)

```
3.00000
```

Finally, create the VECTOR *c* consisting of the strings *rgdp* and *tb3mo*. Note that we needed to use [vector[string]] *c*. The default for COMPUTE is RECTANGULAR and the default for VECTOR is REAL. The double bracket [vector[string]] overrides both of these defaults. We can display either the entire vector or a single element of the vector. Note that a VECTOR uses only a single subscript.

com [vector[string]] c = || 'rgdp', 'tb3mo' ||
dis c

```
   rgdp tb3mo
```

dis c(2)

```
         tb3mo
```

4. com d = || 1.0, 2 | 3 , 4|| , e = || 1, 2 ||
dis d e

```
    1.00000        2.00000
    3.00000        4.00000
        1  2
```

A single COMPUTE instruction can be used to create several matrices and/or vectors. Here, *a* is a 2 x 2 matrix of real numbers and *b* is a 1 x 2 rectangular matrix of integers. Note that COM [vect] b = || 1 , 2 || creates a vector of real numbers

com a = || 1.0, 2 | 3 , 4|| , [vect[integer]] b = || 1, 2 ||

If you are going to explicitly declare the type of matrix and the type of data on the COMPUTE instruction, it might be simpler to use the DECLARE instruction. Nevertheless, the next several examples might prove instructive.

5. com [vect] x = || 1.0, 4.0, -3.9 , 2.0 ||

   Since [*vect*] is specified, the COMPUTE instruction creates the vector *x* consisting of four real numbers. Since the default for a matrix argument is a RECTANGULAR matrix,

   com y = || 1.0, 4.0, -3.9 , 2.0 ||

   creates a 1 x 4 rectangular matrix of real numbers.

   The difference is that a matrix has double subscripts that reference the row and column while a vector has a single subscript. For example, you reference the value 4.0 in each using:

   dis x(2) y(1,2)
   ```
           4.00000        4.00000
   ```

6. The following instruction creates the 1 x 4 RECTANGULAR matrix *inum* consisting of four integers:

   com inum = || 1, 4, -3 , 2 ||

   In contrast:

   com [vect] inum = || 1, 4, -3 , 2 ||

   creates a vector of real numbers since the default data type for vector is real numbers.

   com [vect[integer]] inum = || 1, 4, -3 , 2 ||

   creates a vector of integers since both types are specified. Note the appropriate use of double brackets: vect[integer] must be enclosed in brackets.

7. You can use COMPUTE in association with previously defined variables. Consider:

   com i11 = 1, i12 = 2 , i13 = 3 , i14 = 4
   com [rect[integer]] inum = ||i11, i12 | i13, i14||
   dis inum

```
1   2
3   4
```

Note that once the data type is made explicit, you must take care not to use a different type. If you let *j13* equal 3.9, and enter:

```
com j11 = 1, j12 = 2 , j13 = 3.9 , j14 = 4
com [rect[integer]] inum = ||j11, j12 | j13, j14||
```

```
## SX22. Expected Type INTEGER, Got REAL Instead
>>>>, j12 | j13, j14||<<<<
```

2. Matrix Operations

Suppose that A and B are conformable matrices. The following are just some of the matrix operations you are allowed to perform:

COM C = A + B (or A – B)	Addition and subtraction
COM C = A*B	Multiplication A and B
TR(A)	Transpose of A
INV(A)	Inverse of A
%DECOMP(A)	Choleski decomposition (of SYMMETRIC only)
%KRONEKER(A,B)	Kroneker product of A and B.
%CORR(A,B)	Correlation of A and B
%DET(A)	Determinant
%DOT(A,B)	Dot product
%DIAG(A)	n x n diagonal matrix from an (n x 1) or (1 x n)
%SOLVE(A,B)	Solves the problem Ax = B, where A is an NxN array of known values, B is an Nx1 array of known values, and x is an Nx1 array of unknowns. The function returns x as an Nx1 RECTANGULAR array.

In addition to the usual conformability restrictions, you need to be sure that the matrices contain the same type of variables. For example, if *A* is a 2 x 2 matrix of real numbers and *I1* is a 2 x 2 identity matrix of integers, *A*I1* is not permissible.

2.1 Operations on Subcomponents of a Matrix

One way to manipulate an element is through the COMPUTE instruction. Consider the following examples:

1. com a(1,5) = 3.0

   Here the COMPUTE instruction equates the element in the first row of the fifth column of the matrix A with the real number 3.0.

2. com x = a(1,5)

   Here the COMPUTE instruction equates the variable x with the element in the first row of the fifth of the matrix A.

Sometimes there is a particular relationship among the elements of a matrix. You can exploit this relationship using the EWISE instruction. To use EWISE, you must dimension the matrix. For a vector, EWISE contains an implied DO loop. Consider:

EWISE *a*(i) = *formula in i*

Suppose you wanted to construct a vector of integers running from 1 to 10. One way to do this would be to:

```
declare vector[integer] a(10)
do i = 1,10
   com a(i) = i
end do i
       1  2  3  4  5  6  7  8  9  10
```

The first time through the loop i = 1 and RATS performs the operation COM $a(1) = 1$. The second time through the loop i = 2 and RATS performs the operation COM $a(2) = 2$. This procedure continues through i = 10. A more efficient way to perform the same task is to use:

```
declare vector[integer] a(10)
ewise a(i) = i
```

Here EWISE sets element $a(1) = 1$, element $a(2) = 2$, … . The index i runs from 1 though the dimension of the array.

For RECTANGULAR and SYMMETRIC arrays, EWISE is equivalent to the use of multiple DO loops. Consider:

EWISE *a*(*i,j*) = *formula in i and j*

Here, EWISE sets element (*i,j*) according to the specified formula. For example:

```
declare rect[integer] ix(3,3)
ewise ix(i, j) = i – j
dis ix
   0  -1   -2
   1   0   -1
   2   1    0
```

Note that you do not need to be too careful about the fact that i and j are integers while the elements of the matrix *ix* might be real. Consider:

```
declare rect[real] ix(3,3)
ewise ix(i, j) = i – j
dis ix
      0.00000      -1.00000      -2.00000
      1.00000       0.00000      -1.0000
      2.00000       1.00000       0.00000
```

2.2 Selecting ARMA Coefficients

The BOXJENK instruction also allows you to use vectors to input information. The syntax and principal options of the BOXJENK instruction are:

boxjenk(options) *depvar start end residuals*

where:

depvar	The dependent variable
start end	The range to use in the estimation.
residuals	The name of the series to call the residuals.

The important options for our purposes are:

CONSTANT /[NOCONSTANT]	You must specify *constant* if you want to include an intercept.
AR=	List of autoregressive coefficients. [Default = 0]
MA=	List of moving-average coefficients. [Default = 0]

Thus, the program line below will estimate the model $y_t = a_0 + a_1y_{t-1} + a_2y_{t-2} + \varepsilon_t + \beta_1\varepsilon_{t-1} + \beta_2\varepsilon_{t-2}$ over the sample period 3 through 100 (since two observations are lost due to the 2 AR coefficients) and saves the residuals in a series called *resids*.

box(constant,ar=2,ma=2) y 3 100 resids

It is important to know that the options *ar* = *p* and *ma* = *q* include *AR* coefficients for lags 1 through *p* and *MA* coefficients 1 through *q*. In contrast, you can use *ar* = || *list* || and *ma* = || *list* || to include only those coefficients enumerated in *list*. For example, *ar* = 4 calls for the inclusion of autoregressive coefficients a_1, a_2, a_3, and a_4. Instead, *ar* = ||1, 4|| calls for the inclusion of autoregressive coefficients a_1 and a_4 but not a_2 or a_3. Thus, to estimate the model $y_t = a_0 + a_1y_{t-1} + a_2y_{t-2} + \varepsilon_t + \beta_2\varepsilon_{t-2} + \beta_4\varepsilon_{t-4}$ use:

box(constant,ar=2,ma=||2, 4||) y 3 100 resids

What you are doing is entering a vector of integers (i.e., the vector [2 , 4]) into the instruction. This method works well if the vector list is short. Since you are creating a vector, you must type each integer value that you want to include. Hence, you cannot enter: *ma* = || 1 to 4, 5||.

Moreover, if you embed the instruction in a procedure, you want to give the user the flexibility to input any possible parameter set. The way to avoid this problem is to use the method shown below to create your own integer vector. Although it is not especially efficient to use three lines of code instead of one, you could estimate the model above using:

```
declare vector[integer] mas
com mas = || 2, 4||
box(constant,ar=2,ma=mas) y 3 100 resids
```

In other circumstances, the method can be particularly useful. A friend of mine working with energy-price tick data wanted to estimate an MA model with coefficients at lags 1 - 60, 120, 180 and 240. A simple way to create the vector is:

```
declare vector[integer] mas(63)
ewise mas(i) = i
com mas(61) = 120, mas(62) = 180, mas(63) = 240
box(constant,ar=2,ma=mas) y 3 100 resids
```

Here, you declare the vector *mas* and fill the vector with the coefficient values you want in the estimation. The EWISE instruction fills all 63 entries with integer values running from 1 to 63. The COMPUTE instruction corrects entries 61, 62 and 63 such that they equal 120, 180 and 240, respectively. The final instruction estimates a model with 2 autoregressive coefficients and the 63 moving average coefficients—the *MA* terms at lags 1-60, 120, 180 and 240.

2.3 Manipulating the Output of a VAR

In Chapter 2, we were able to create a 3-variable VAR using:

```
system(model=chap2)
var dlrgdp dlrm2 drs
lags 1 to 12
det constant
end(system)
```

The necessary instructions to set up the VAR are repeated in Program 5.2. The next statement instructs RATS to estimate the VAR, and create the variance/covariance matrix *v*. The regression residuals are stored in the series *resids*(1), *resids*(2) and *resids*(3) and the coefficients are stored in the matrix *ca.*

```
estimate(sigma,outsigma=v,residuals=resids,coeffs=ca)
```

Note that *ca* is a 37 x 3 RECTANGULAR matrix. You can view the coefficients using:

dis ca

```
  0.04961       0.02504      27.24470
 -0.02046       0.06033       6.04229
 -0.16255       0.05159     -12.42336
  0.12126      -0.07211       9.71846
 -0.15415       0.00161      -2.79501
     ……..
```

Element *ca*(1,1) is the regression coefficient of *dlpgdp* on its own first lag and element *ca*(37, 2) is the intercept in the *dlrm2* equation. Similarly, PRINT the residuals using:

pri * 16 resids

```
 ENTRY       RESIDS(1)       RESIDS(2)       RESIDS(3)
  1962:02  -0.004738155830  0.004968068719 -0.651980315507
  1962:03  -0.001610217811  0.000793836267 -0.117657508925
  1962:04  -0.010007743363  0.005422212237  0.101551774645
```

Notice that the residual series begin with 1962:02 since one usable observation is lost by differencing and twelve are lost by using lags in the regression equations. The PRINT instruction caused RATS to print three series: *resids*(1), *resids*(2) and *resids*(3). In essence, *resids* is a vector containing three series. You can print any one of the three series using *resids*(*i*). For example, you can print the residuals from the second regression equation using:

pri * 16 resids(2)

```
 ENTRY       RESIDS(2)
  1962:02   0.004968068719
  1962:03   0.000793836267
  1962:04   0.005422212237
```

Thus, `0.004968068719` is the first defined entry of the series called *resids*(2) and `0.000793836267` is the second defined entry of the same series. You can reference the individual elements as follows:

dis resids(2)(14)

```
        0.00497
```

Next, we want to decompose the variance/covariance matrix using a Choleski decomposition. You can display the variance/covariance matrix *v* and the Choleski decomposition of *v* using:

dis v

```
    4.25841e-05
    8.43189e-06   2.83336e-05
    7.56122e-04  -6.56400e-04       0.30310
```

dis %decomp(v)

```
    0.00653       0.00000       0.00000
    0.00129       0.00516       0.00000
    0.11587      -0.15611       0.51507
```

Hence:

v = B*B'

where: *B* = %decomp(v)

To obtain impulse responses, it is desirable to use the transpose of this matrix. You can compute the matrix *G* as the transpose of %decomp(v) with:

com g = tr(%decomp(s))
dis 'Decomposed Matrix' g

```
Decomposed Matrix
0.00653       0.00129       0.11587
0.00000       0.00516      -0.15611
0.00000       0.00000       0.51507
```

You can obtain the autocorrelations of the residuals using the DO loop below. The first time through the loop, *i* = 1 so that the autocorrelations of *resid*s(1) are displayed.

```
do i = 1,3
  cor(qstats,span=4) resids(i)
end do
```

Sections 4 through 6 of Chapter 2 contain a number of examples involving structural VARs. In a structural VAR, you model the individual elements of the matrix *g*.

3. Example: ENTER and Supplementary Cards

RATS allows you to replace the individual entries on a supplemental card with a vector. You use the ENTER instruction to manipulate the items in the vector. By changing the contents of the vector, it is easy to add, subtract or (in almost any reasonable way) modify the information on the supplementary card.

You first have to create the vector that will hold the information. If we are going to modify the various series listed on the card, it is necessary to create a vector of integers (Recall that series can be referenced by their labels or integers). The syntax of ENTER needed to perform this task is:[35]

ENTER(varying) *vector*
variables for vector

Use the VARYING option since the length of the *vector* will vary as you add or delete variables. We will return to the issue of automating model selection in a VAR in Sections 3.1 and 3.2 below. Before considering the full VAR system, suppose you want to estimate the three regression equations:

$$dlrgdp_t = \alpha_0 + A_{11}(L)dlrgdp_{t-1} + \varepsilon_t$$
$$dlrgdp_t = \alpha_0 + A_{11}(L)dlrgdp_{t-1} + A_{12}(L)dlrm2_{t-1} + \varepsilon_t$$
$$dlrgdp_t = \alpha_0 + A_{11}(L)dlrgdp_{t-1} + A_{12}(L)dlrm2_{t-1} + A_{13}(L)drs_{t-1} + \varepsilon_t$$

The next instruction in Program 5.2 is used to DECLARE the integer vector *reglist*. This vector will hold the list of regressors we want to place on the supplementary card. The second line above places '*constant*' in the vector *reglist*.

```
dec vector[integer] reglist
compute reglist = || constant ||

dofor i = dlrgdp dlrm2 drs
   enter(varying) reglist
   # reglist i{1 to 12}
   lin dlrgdp
   # reglist
end dofor i
```

The first time through the DOFOR loop, i = the integer assigned to the series *dlrgdp*. Hence, *dlrgdp*{1 to 12} is added to *reglist*. Notice that the supplementary card on the ENTER instruction also contains *reglist*. This is because we want the new list of regressors to include

[35] ENTER can also be used when passing information to a procedure. Here, we consider only the use of ENTER to create a variable list.

everything in the previous list (*i.e*., a constant) and the variable we are adding to the list. Next, *i* = the integer assigned to *dlrm2* and a new list is created. The new vector *reglist* contains everything in the previous list (*i.e*., a constant and *dlrgdp*{1 to 12}) and dlrm2{1 to 12}. In this way the program estimates the regression equations.

If you do not want a constant in any of the regressions, you can replace the first line of the routine with:

```
dec vector[integer] reglist(0)
```

Now reglist has been dimensioned such that it contains no entries.

3.1 Automating Model Selection in a VAR

You can use ENTER to modify the list of deterministic variables in a VAR. Suppose that we want to determine whether to include seasonal dummy variables in the 3-variable 12-lag VAR. We want a routine that will make three loops. In the first loop, the system is estimated without the seasonal dummies and the seasonal dummies are used in the second loop. The VAR output is not displayed at this stage. In the third loop, the program estimates the 'best fitting' of the two models and displays the output. The first line of the program creates a seasonal dummy variable that is 1 in the 4-th quarter of each year and zero in all other quarters. The next instruction of PROGRAM2 on the file CHAPTER5.PRG initializes a switch called *print* that is OFF (*i.e*., = 0) in the first two loops and is ON (*i.e*., = 1) in the third loop. It also initializes the variable *aic_min*—this variable will be used to hold the *aic* for the 'best-fitting' model.

```
sea seasons
com print = 0  , aic_min = 100000000.
```

Next, we create two vectors; one will hold the deterministic regressors we want to keep in the VAR (*perm_det*) and the other holds the current regressor list (*temp_det*). These two integer vectors are initialized to hold only a constant.

```
dec vector[integer] temp_det perm_det
com temp_det = ||constant||
com perm_det = ||constant||
```

Next we begin the three DO loops. On the first loop, *i* = 1 and the bracketed conditional statement is ignored. Hence, the only deterministic regressor is the constant. On the second loop, *i* = 2 and three seasonal dummy variables are added to the list of temporary variables. On the third loop, *i* = 3 and the first set of conditional statements is ignored but the PRINT switch is turned ON.

```
do i = 1,3
   if i == 2 {
   enter(varying) temp_det
```

```
# perm_det seasons{1 to 3}
  }
if i == 3 ; com print = 1
```

Next the VAR system is estimated. On the first loop, only the constant is included in the deterministic regressors. On the second loop, the list will include the seasonal dummy variables.

```
system 1 to 3
vars dlrgdp dlrm2 drs
lags 1 to 12
det temp_det
end(system)
estimate(print=print)
```

After the system is estimated, the multivariate *AIC* is computed. The function %eqnsize(*equation*) returns the number of regressors in the specified equation. To obtain the number of coefficients estimated in the system, find the number of regressors in equation 1 (i.e., the first equation in the system) and multiply by 3. If the resulting value of the *AIC* is less than *aic_min*, the list of permanent regressors is equated to the current regressor list and *aic_min* is replaced by the current *AIC*. Otherwise these two instructions are skipped.

```
com aic = %nobs*%logdet + 2*(%eqnsize(1))*3; dis aic
if aic < aic_min {
  enter(varying) perm_det
  # temp_det
  com aic_min = aic
  }
```

Next, the temporary list is replaced by the permanent list and the loop is completed.

```
    enter(varying) temp_det
    # perm_det
end do
```

If you run the program, you will find that the model without the seasonal dummy variables is selected.

3.2 Creating a Near-VAR Using ENTER

Suppose that we want to forecast a series $\{y_t\}$ using lagged values of a number of macroeconomic variables. One way to determine which variables to include in the forecasting equation is to use Granger-causality tests. Suppose that we have determined that a four-lag model is most appropriate and that a tentative forecasting equation for y_t is:

$$y_t = \alpha_0 + a_{11}y_{t-1} + a_{12}y_{t-2} + a_{13}y_{t-3} + a_{14}y_{t-4} + a_{21}x_{t-1} + a_{22}x_{t-2} + a_{23}x_{t-3} + a_{24}x_{t-4} + \ldots + \varepsilon_t$$

where: $\{x_t\}$ is one of the series that we might want to include in our final forecasting equation.

Since there is likely to be a fair amount of correlation among the regressors, it is standard to rely on F-tests to determine whether or not to include a series in the forecasting equation. It is said that $\{x_t\}$ Granger-causes $\{y_t\}$ if it is possible to reject the null hypothesis:

$$a_{21} = a_{22} = a_{23} = a_{24} = 0$$

Hence, if we cannot reject this null hypothesis, we exclude all values of $\{x_t\}$ from the forecasting equation. Of course, we can also determine if $\{y_t\}$ Granger-causes itself by testing the null hypothesis $a_{11} = a_{12} = a_{13} = a_{14} = 0$. One way to proceed is to estimate a very general forecasting equation using all variables in the data set and then to eliminate variables based on Granger-causality tests. The other way is to begin with only an intercept and sequentially add variables keeping only those that pass the causality test. The purpose of this section is to illustrate the use of the ENTER instruction, not to determine which of the two methods is best. As such, we will use the second method to obtain a forecasting equation for the logarithmic change in *M3*.

As in the previous section, it is necessary to keep track of two regressor lists. The first—called *reglist*—holds only the variables that we want to keep for the final forecasting equation. These are the variables that have already passed the Granger-causality test. The second—called *templist*—holds the variables in *reglist* plus the variable that is currently under consideration for inclusion. Only if this variable passes the causality test will it be added to *reglist*.

The final section of Program 5.2 creates the variables *dlm3* and *dlp* and the two integer vectors *templist* and *reglist*.

```
dec vector[integer] templist reglist
compute templist=||constant||
compute reglist=||constant||
```

Next, the program loops over the series *dlrgdp, dlm3, drs and dlp*. The DOFOR instruction below uses the fact that RATS allows you to refer to a series by its name or by its number. The first time through the loop, *templist* and *reglist* contain only the constant term. The ENTER instruction creates *templist* as the constant (*i.e.*, the contents of *reglist*) plus the first four lags of *dlrgdp*. Next, the LINREG instruction estimates a regression of $dlm3_t$ on the contents of *templist*.

```
dofor i = dlrgdp dlm3 drs dlp
    enter(varying) templist
    # reglist i{1 to 4}
    lin(noprint) dlm3
    # templist
```

The EXCLUDE instruction is used to perform the *F*-test that the coefficients on the four lags of $dlrgdp_t$ are statistically significant from zero. If the null hypothesis that the coefficients are jointly equal to zero is significant at the 5% level, the bracketed instructions are executed. The bracketed ENTER instruction adds lags 1 to 4 of $dlpgdp_t$ to *reglist*. If the null hypothesis cannot be rejected, the bracketed ENTER instruction is skipped so that the lagged values of $dlpgdp_t$ are not added to *reglist*.

```
    exclude(noprint)
    # i{1 to 4}
    if %signif < .05 {
        enter(varying) reglist
        # reglist i{1 to 4}
    }
```

Next, *templist* is updated such that the regressors in *templist* are identical to that in *reglist*. As it turns out, the null hypothesis cannot be rejected so that we conclude that the $\{dlrgdp_t\}$ sequence does not Granger-cause $dlm3_t$. At the end of the DOFOR loop, *reglist* contains only the constant.

```
    enter(varying) templist
    # reglist
end dofor i
```

On completing the first pass through the DOFOR loop *i* = *dlm3* [To be more precise, the entire process is repeated for *i* = the integer corresponding to series number of *dlm3*]. As such, four lags of $dlm3_t$ are added to *templist* and a regression of $dlm3_t$ on its own four lags and a constant is estimated. The entire process discussed above is repeated for $dm3_t$ and subsequently for dr_t and dlp_t. It turns out that only the lags of $dlm3_t$ help to forecast the current value of $dlm3_t$. On completing the four loops, the final two instructions below estimate a regression containing only those variables passing the Granger-causality test.

```
lin dlm3
# reglist
```

If you execute the program, your output will be:

```
   Variable      Coeff       Std Error      T-Stat     Signif
********************************************************************
1.  Constant   0.002906841  0.000993077     2.92711  0.00392329
2.  DLM3{1}    0.824726885  0.080688781    10.22108  0.00000000
3.  DLM3{2}    0.004028644  0.104551220     0.03853  0.96931127
4.  DLM3{3}   -0.079967942  0.104224480    -0.76727  0.44406097
5.  DLM3{4}    0.109141754  0.079475991     1.37327  0.17160277
```

Jazzing Up the Program

It is straightforward to modify the program to use each of the four variables as the dependent variable in the regression equation. Thus, the final output will be a four-variable near-VAR in the sense that only the variables that are “causal” remain in the system. Towards this end, we will let the index *j* loop over *dlrgdp, dlm3, drs and dlp*. Consider:

```
dec vector[integer] templist reglist

dofor j = dlrgdp, dlm3, drs and dlp
compute templist=||constant||
compute reglist=||constant||
```

Now, each time through the DOFOR j loop, *templist* and *reglist* will be initialized to contain only a constant. There are only three other required modifications to the program. As indicated below, the two LINREG instructions are changed to indicate that the dependent variable in the regression is series *j*. The final line in the program closed the DOFOR *j* loop.

```
dofor i = dlrgdp dlm3 drs dlp
   enter(varying) templist
   # reglist i{1 to 4}
   lin(noprint) j                ;* NOTE the change in this line
   # templist

   exclude(noprint)
   # i{1 to 4}
   if %signif < .05 {
   enter(varying) reglist
   # reglist i{1 to 4}
   }
```

```
enter(varying) templist
    # reglist
end dofor i
lin j                           ;* NOTE the change in this line
# reglist
end dofor j                     ;* NOTE the addition of this line
```

If you execute the program, you will obtain four regression equations. Instead of reproducing a rather large amount of output, simply note that:

Equation	Causal Variables
dlrgdp	dlrgdp, drs
dlm3	dlm3
drs	dlrgdp, drs, dlp
dlp	dlrgdp, dlm3, drs, dlp

4. Example: Moving Average Representations

Suppose that we have an *ARMA*(p,q) model and want to calculate the coefficients of its infinite-order moving average representation. Specifically, suppose that we are working with the model:

$$y_t = \alpha_0 + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \ldots + \alpha_p y_{t-p} + \varepsilon_t + \beta_1 \varepsilon_{t-1} + \ldots + \beta_q \varepsilon_{t-q}$$

As long as the equation is invertible, it is possible to express the $\{y_t\}$ sequence in terms of the $\{\varepsilon_t\}$ sequence as:

$$y_t = \phi_0 + \sum_{i=0}^{\infty} \phi_i \varepsilon_{t-i}$$

One way to obtain the values of the $\{\phi_i\}$ sequence is to use the Method of Undetermined Coefficients. It should be clear that $\phi_0 = \alpha_0/(1 - \alpha_1 - \alpha_2 - \ldots - \alpha_p)$ and that $\phi_1 = 1$. However, finding the remaining values of the $\{\phi_i\}$ sequence can be complicated. In particular, the formulas for the coefficients are given by:

$$\phi_0 = 1$$

$$\phi_1 = \beta_1 + \phi_0\alpha_1$$

$$\phi_2 = \beta_2 + \phi_1\alpha_1 + \phi_0\alpha_2$$

$$\phi_3 = \beta_3 + \phi_2\alpha_1 + \phi_1\alpha_2 + \phi_0\alpha_3$$

…

Thus, with knowledge of the $\{\alpha_i\}$ and $\{\beta_i\}$ the values of the various $\{\phi_i\}$ can be found by iteration using the formula:

$$\phi_j = \beta_j + \sum_{k=1}^{j} \alpha_k \phi_{j-k} \qquad \text{for } j = 1, 2, 3, \ldots, 24$$

We want the program to contain the following steps:

1. Allow the user to enter any number of autoregressive coefficients and/or moving average coefficients.

2. Calculate the ϕ_j iteratively using the formula above.

3. Create a bar graph of the first 24 values of the $\{\phi_i\}$ sequence.

There are many ways to perform these three tasks; the program developed below is designed to illustrate matrix manipulations. Consider the first five lines of Program 5.3 on the file CHAPTER5.PRG:

```
compute number = 24
all number

com alpha = ||1.1, -.4, .2  ||
com beta = ||-.7, .3||
```

The first two lines set the default length of any series to 24. You can change the value of "number" to obtain a smaller or larger number of impulse responses.

The third and fourth lines create the vectors *alpha* and *beta*. The two vectors hold the coefficients of the *ARMA* model. Here you can enter as many or as few values for α_i and β_i as desired. In the example, $\alpha_1 = 1.1$, $\alpha_2 = -0.4$, $\alpha_3 = 0.2$, $\beta_1 = -.7$ and $\beta_2 = 0.3$. Hence, these first four lines have accomplished task 1. However, we did not DECLARE either *alpha* or *beta* to be vectors—refer to the individual elements of each as *alpha*(1,1), *alpha*(1,2), *alpha*(1,3), *beta*(1,1) and *beta*(1,2).

Now we need to do some bookkeeping. Our first bookkeeping task is to compute values of p and q, *i.e.*, the dimensions of the *alpha* and *beta* vectors. The function %cols(A) returns the number of columns in matrix A. Hence, we can obtain p and q using:

```
compute p = %cols(alpha)
compute q = %cols(beta)
```

Unless otherwise specified, creation of a literal vector using COMPUTE causes RATS to create a **row** vector. Hence, *alpha* has 1 row and 3 columns while *beta* has 1 row and 2 columns.

Next, we need to set up a vector—that we call *phi*—to hold the twenty-four values of the $\{\phi_i\}$. However, there is a small problem in that a vector cannot have an element zero. Thus, we cannot use the notation ϕ_0, ϕ_1 … because we cannot have an element of a vector designated as *phi*(0). Instead, we need to store the first value of ϕ in *phi*(1), the second value in *phi*(2), … . Thus, *phi*(1) will equal 1, *phi*(2) will equal $\beta_1 + \alpha_1$*phi*(1), … . In essence, we need to create the *phi* vector such that *phi*(1) = ϕ_0, *phi*(2) = ϕ_1, … Hence, we will replace the usual formula for calculating impulse responses with:

$$\phi_1 = 1$$

$$\phi_{j+1} = \beta_j + \sum_{k=1}^{j} \alpha_k \phi_{j+1-k} \qquad \text{for } j = 1, 2, 3, \dots, 23$$

We can DECLARE the phi vector to contain 24 elements and initialize the first value *phi*(1) = 1 using:

```
dec vect phi(number)
com phi(1) = 1.
```

We cannot directly use the formula above since α_k is undefined for $k > p$ and β_j is undefined for $j > q$. The final bookkeeping task concerns treatment of these values. One straightforward method is to define two new vectors of dimension 24. We can let the first p elements of vector A hold the elements of *alpha* and set the remaining values to zero. Similarly, we can let the first q elements of vector B hold the elements of *beta* and set the remaining values to zero.

```
dec vect a(number) b(number)
com a = %const(0.) , b = %const(0.)
ewise a(i) = %if(i<=p, alpha(1,i), 0.0)
ewise b(i) = %if(i<=q ,beta(1,i), 0.0)
```

Hence, the DECLARE instruction in the program segment above creates the vectors A and B and sets the dimension of each to 24. COMPUTE uses the %CONST(x) instruction to set all values of A and B to equal the constant zero.[36] Next, looping over p, equates the first p values of A with the corresponding elements of *alpha*. Similarly, looping over q sets the first q elements of B equal to the corresponding elements of *beta*. Finally, we can write a routine that calculates the remaining values of *phi*.

```
do j = 1,number-1
   com phi(j+1) = b(j)
   do k = 1,j  ; com phi(j+1) = phi(j+1) + phi(j+1-k)*a(k) ; end do k
end do j
```

The first loop initializes *phi*(j+1) equal to β_j. Each time through the inner loop (i.e., for each value of k), $\alpha_k\phi_{j+1-k}$ is added to *phi*(j+1). Exiting this inner loop yields the desired sum:

$$\phi_{j+1} = \beta_j + \sum_{k=1}^{j} \alpha_k \phi_{j+1-k}$$

This process is repeated for each value of j up to and including $j = 23$.

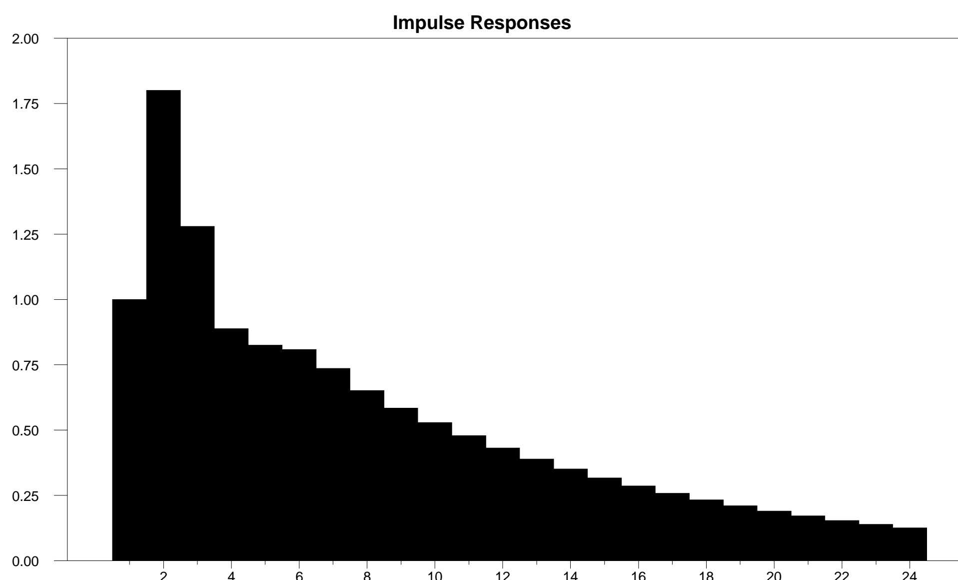
The last task is to create a bar graph of the phi sequence. Since *phi* is a vector, it cannot be graphed directly. However, we can convert *phi* into a series called *response* and graph *response* using:

```
set response = phi(t)
gra(style=bar,header='Impulse Responses') 1 ; # response
```

If you run the program as shown, your output will be:

---

[36] You cannot use com a = 0; this instruction would cause A to equal the number 0. %CONSTANT sets each element of A to zero.



Jazzing Up the Program

It is likely that you might want to find the impulse responses to an *ARMA*(*p, q*) model that you estimated using the BOXJENK instruction. BOXJENK estimates a model and stores the resulting coefficients in a vector called %BETA. The first element of %BETA is always the constant (if any), then the *AR*(*p*) coefficients and finally the *MA*(*q*) coefficients. We need to make the following modifications to the program:

1. Program 5.4 illustrates the process by estimating *drs* from the data set MONEY_DEM.XLS. Consider:

```
cal 1959 1 4
all 2001:1
open data a:\money_dem.xls
data(org=obs,format=xls)
dif tb3mo / drs
com number = 24
```

The first five lines instruct RATS to read the data set and create the variable *drs*. The sixth line indicates that we want 24 impulse responses. If you use the Box-Jenkins methodology, you can convince yourself that a plausible model for the $\{drs_t\}$ sequence is:

$$drs_t = \alpha_2 drs_{t-2} + \alpha_7 drs_{t-7} + \varepsilon_t + \beta_1 \varepsilon_{t-1} + \beta_3 \varepsilon_{t-3}$$

Nevertheless, in order to illustrate some vector manipulations, we will estimate this model including an intercept. We can estimate the model with:

```
com ar = ||2,7||
com ma = ||1,3||
```
box(constant,ar=ar,ma=ma) drs

```
   Variable   Coeff        Std Error        T-Stat    Signif
*************************************************************
1.  CONSTANT  0.012981373  0.053771900      0.24142  0.80955008
2.  AR{2}    -0.268737166  0.076271837     -3.52341  0.00055917
3.  AR{7}    -0.335461768  0.073223473     -4.58134  0.00000940
4.  MA{1}     0.385063544  0.078041223      4.93410  0.00000205
5.  MA{3}     0.183914603  0.074495456      2.46880  0.01463518
```

Formulating the model in this form makes our programming problem a bit more complicated than estimating an *ARMA*(7,3) model. The first autoregressive coefficient estimated is α_2 and the second is α_7. Similarly, the first *MA* coefficient estimated in β_1 and the second is β_3. In fact, the five estimated coefficients are contained in the vector %BETA. Note that %BETA has only 5 elements: %BETA(1) = 0.012981373, %BETA(2) = -0.268737166,%BETA(3) = -0.335461768, %BETA(4) = 0.385063544 and %BETA(5) = 0.183914603. We need to transfer these five values to the appropriate elements of the *A* and *B* matrices in the formula used to calculate *phi*.

The next two instructions use the variable *p* to store the number of *AR* coefficients and *q* to store the number of *MA* coefficients. The third instruction sets up an indicator called *flag* equal to the number of coefficients estimated minus ($p + q$). Note that %BETA is a column vector so that %ROWS(%BETA) indicates the total number of coefficients estimated. Thus, *flag* equals 1 if there is an intercept (since the number of coefficients estimated will exceed $p + q$ by 1) and is zero if no intercept is present. If this was the only time you were going to use the program, this step would be unnecessary. However, you might want to use the same routine for other estimations that might not include an intercept. As such, it becomes convenient to allow the program to determine whether of not an intercept was included in the estimation.

```
compute p = %cols(ar)
compute q = %cols(ma)
com flag = %rows(%beta) - p -q
```

As in the original program above, the next two instructions fill the vectors *A* and *B* with zeroes.

```
dec vect a(number) b(number)
com a = %const(0.) , b = %const(0.)
```

Now we need to transfer the values of %BETA(2) to *A*(2), %BETA(3) to *A*(7), %BETA(4) to *B*(1) and %BETA(5) to *B*(3). One way to do this would be to use:

com a(2) = %beta(2), a(7) = %beta(3), b(1) = %beta(4) and b(3) = %beta(5).

However, a more general way to do this is to recall that the first element of *AR* is the integer 2, the second element of *AR* is the integer 7, the first element of *MA* is the integer 1 and the second element of *MA* is the integer 3. Hence, we can write:

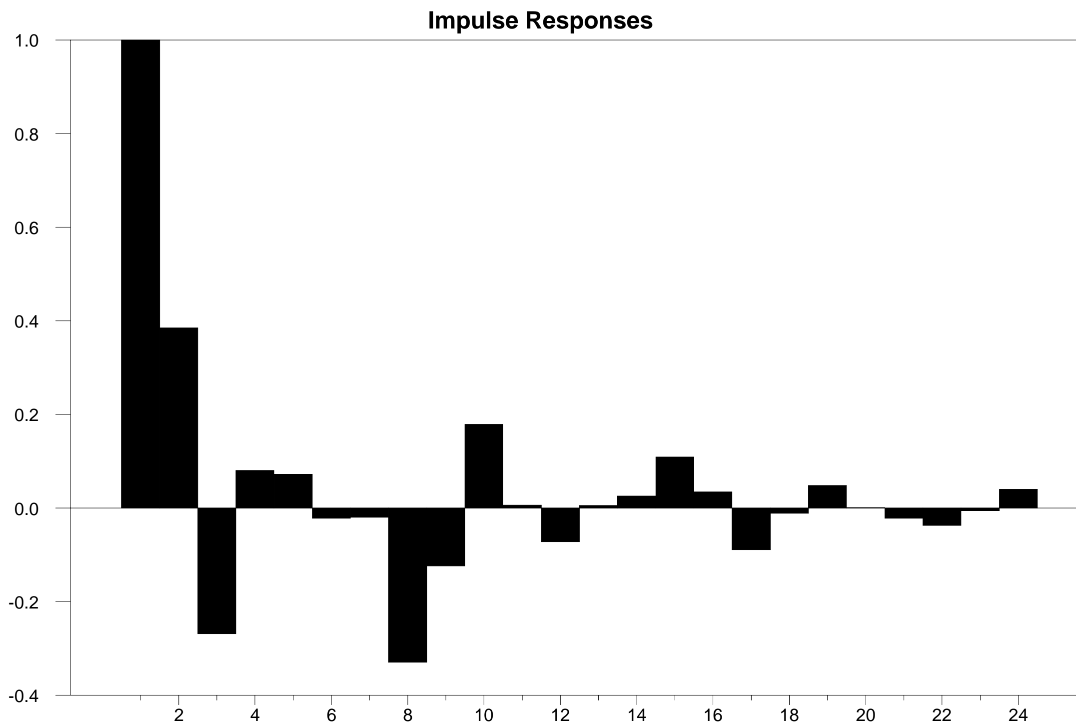
```
do i = 1,p ; com a(ar(1,i)) = %beta(i+flag) ; end do i
do i = 1,q ; com b(ma(1,i)) = %beta(i+p+flag) ; end do i
```

Since *flag* = 1, the first loop equates $A(2)$ with %BETA(2) and $A(7)$ with %BETA(3) and the second loop equates $B(1)$ with %BETA(4) and $B(3)$ with %BETA(5). The key point to note is that this set of instructions will work for any pattern of ARMA coefficients. The next six lines of code are identical to that of the program above:

```
dec vect phi(number)
com phi(1) = 1.

do j = 1,number-1
   com phi(j+1) = b(j)
   do k = 1,j  ; com phi(j+1) = phi(j+1) + phi(j+1-k)*a(k) ; end do k
end do j
```

The remainder of the program is unchanged:

```
set response = phi(t)
gra(nodates,style=bar,Header='Impulse Responses') 1 ; # response
```



4.1 Impulse Responses in a First-Order VAR

Suppose you estimated a $dlrgdp_t$, $dlrm2_t$ and drs_t as a first-order VAR using:[37]

estimate(sigma,outsigma=v,residuals=resids,coeffs=ca)

You could obtain the impulse responses using:

errors(impulses,model=*modelname*) 3 24 v

or

impulse(model=*modelname*) 3 24 * V

An alternative way to obtain the identical answers is to use RATS matrix instructions.
The matrix *ca* contains the slope coefficients and the intercept terms. Since the impulse responses represent deviations from equilibrium, you will want to create a new matrix, called *A*, containing only the slope coefficients. Consider the next three instructions:

```
dec rect a(3,3)
ewise a(i,j) = ca(i,j)
dis 'A = ' a
```

The DECLARE instruction creates *A* as a 3 x 3 rectangular matrix. The EWISE instruction equates each element of *A* with the corresponding element of *ca*. The third instruction allows you to view the newly created matrix *A*. Next, create a matrix to contain the impulse responses. We will create an 8 x 3 matrix called *imp* to hold the eight impulse responses of a shock to the first variable on the time path of the three variables in the system:

```
dec rect imp(8,3)
```

Thus, *imp*(4,2) will contain the impulse response of a shock to $dlrgdp_t$ on the value of $dlm2_{t+4}$. Next, we create the impulse responses themselves. Since we consider only a first-order VAR, after the first period, the system evolves as:

$$imp_t = imp_{t-1}\, a$$

where: the 1 x 3 vector imp_t are the impulse responses of ($dlrgdp_t$ dlm_t dr_t) to a real gdp shock and *A* is the coefficient matrix created above.

---

[37] Although the 12-order VAR is more appropriate, the example here is intended to be illustrative.

```
ewise imp(i,j) = g(1,j)
do i = 2,8
   com c = tr(%xrow(imp,i-1))*a
   do j = 1,3 ; com imp(i,j) = c(1,j) ; end do j
end do i
dis ##.########   ##.####### ##.######## imp
```

The logic of the program is to initialize the impulses with a 1-unit shock to *dlrgdp*. Within the first loop, *c* is computed as the transposed value of the *i*-1 row of *imp* (*i.e*., the impulses for period *i*-1) multiplied by the coefficient matrix *a*. Then the current value of imp for real gdp, money and interest rates are computed as the associated value of the 1 x 3 vector *c*. Displaying *imp* yields identical answers to the ERRORS(impulses) instruction.

5. Creating Matrices from Your Data

If you are going to use matrix manipulations on your data set, it is important to realize that RATS does not treat a series in the same way as a vector. For example, a vector is a one-dimensional array such that the elements have subscripts that run from 1 to *N*. In order to manipulate the vector, each element needs to be defined. Unlike a vector, you can manipulate a series even if it has ranges that are undefined or *NA*. The key point is that RATS treats the two differently: to perform any matrix manipulations on your data set, you need to create vectors from your series. You can also create a RECTANGULAR matrix of series such that you can refer to each element by its row and column. In a RECTANGULAR matrix, columns represent variables and rows represent observations. Thus, element *i,j* is the element in the *i*-th row of the *j*-th column. Similarly, element *i,j* is the *i*-th observation of variable *j*.

You can create matrices from series using the MAKE instruction. The syntax is:

MAKE *array start end numobs numvars*
list of variables

numobs:	INTEGER used by RATS to return the number of observations (*i.e.*, the number of rows)
numvars:	INTEGER used by RATS to return the number of variables (*i.e.*, the number of columns)

Options:

EQUATION=	equation supplying variables
LASTREG:	use regressors from last regression.

NOTE: Omit the supplementary card with either of the above.

TRANS:	set up the transpose of the observation array

Examples

For all of the examples below, read in the data set MONEY_DEM.XLS and create *dlrgdp* using PROGRAM 5 of CHAPTER5.PRG. Next, estimate *dlrdgp* as an *AR*(2) process using:

lin dlrgdp / resids
 # constant dlrgdp{1 to 2}

```
   Variable        Coeff       Std Error     T-Stat     Signif
****************************************************************
1.  Constant    0.0051566068 0.0010217954    5.04661  0.00000119
2.  DLRGDP{1}   0.2508977521 0.0769801061    3.25925  0.00135976
3.  DLRGDP{2}   0.1362250820 0.0762100846    1.78749  0.07571568
```

Now we can perform the identical estimation using matrices. First, we can make the matrix *x* containing the regressors from the *AR*(2) using:
make(lastreg) x 4 2001:1

Unlike a series, you can show the contents of a matrix using the DISPLAY instruction. If you DISPLAY *x*, you will see that the first 5 rows of matrix *x* are:

dis x
```
1.00000  -4.28835e-04   0.02580
1.00000   0.00330      -4.28835e-04
1.00000   0.02195       0.00330
1.00000  -0.00495       0.02195
1.00000   0.00185      -0.00495
```

You can see that the first column consists of all 1's, the second column contains *dlrgdp*{1} and the second column contains *dlrgdp*{2}. You can display any particular element of x by referring to its row and column. For example:

dis x(2,3)
```
-4.28835e-04
```

Next, create the matrix *y* containing the dependent variable (*i.e*., the contemporaneous values of *dlrgdp*). Care needs to be taken about the conformability of the *x* and *y* matrices. It is necessary to begin with observation 4 since one usable observation is lost by using first differences and two more are lost as a result of estimating a model with two lags. Hence:

```
make y 4 2001:1
# dlrgdp
```

If you enter DISPLAY *y*, you will see that the first five rows are:

dis y
```
   0.00330
   0.02195
  -0.00495
   0.00185
  -0.01296
```

You can display the individual elements of *y* by referring to their row and column. For example:

dis y(3,1)
```
  -0.00495
```

5.1 Estimating the Regression Coefficients

We want to compute and display the matrix of coefficients as: $(x'x)^{-1}x'y$. Consider the following program statements:

```
com xx = tr(x)*x
com xx_inv = inv(xx)
```

```
com xy = tr(x)*y
com beta = xx_inv*xy
```
dis beta
```
0.00516
0.25090
0.13623
```

We could have written all of the above in one step. However, it is instructive to consider each of the program statements. The first line creates xx as $x'x$. The second line takes the inverse $(x'x)^{-1}$ and the third creates $x'y$. The fourth line creates β as $(x'x)^{-1}x'y$ and the fifth displays β. Here is how you could have written all of the above in the single step:

```
dis inv(tr(x)*x)*tr(x)*y
```

Next, call $\hat{y}_t$ the predicted value of y_t. The matrix $x(x'x)^{-1}x'$ is often called the projection matrix P since:

$$\hat{y}_t = X_t\hat{\beta}$$

and $\hat{\beta} = (x'x)^{-1}x'y$

Moreover, since the error term $e_t = y_t - \hat{y}_t$, it follows that:

$$e_t = y_t - \hat{y}_t = (I - P)y_t = My_t.$$

We can calculate the projection matrix P as:

```
com p = x*xx_inv*tr(x)
```

Students of econometrics will recall that P is idempotent (By definition, the square matrix A is idempotent if $A*A = A$). You can verify that P is idempotent using:

```
com test = p*p - p ; dis test
```

Since P has the dimensions 166 x 166, you will see a tremendous amount of output displayed to the screen. Nevertheless, all of the values of test are approximately equal to zero. Although, each value should be exactly equal to zero, rounding errors are present. As another exercise, you might recall that the rank of an idempotent matrix equals the trace of the matrix. You can display the trace of P using:

dis %trace(p)
```
3.00000
```

You can use the projection matrix to obtain the predicted values of y_t and the error terms using:

```
com y_hat = p*y  ; dis y_hat
com e = y - y_hat ; dis e
```

If you display *e* and print *resids*, you should obtain exactly the same results. Additionally, you can obtain the orthogonal complement (M) of P by forming:

$$M = I - P.$$

Notice that M is another useful way to calculate the residuals. Since:

$$e_t = y_t - \hat{y}_t$$

it follows that:

$$e_t = y_t - Py_t = (I - P)y_t = My_t$$

We can calculate M using:

```
com m = %identity(166) – P
```

Exercises:

1. Verify that %identity(166) is an identity matrix with 166 rows and columns.

   ```
   dis %identity(166)
   ```

2. Verify that M and P are orthogonal by entering:

   ```
   dis m*p
   ```

3. Verify that M is idempotent by entering:

   ```
   com test = m*m - m ; dis test
   ```

5.2 Hypothesis Testing in the Regression Model

We can calculate the variance of the residuals (*i.e.*, the squared standard error of the estimate) as:

$$\hat{\sigma}^2 = e'e/(T-3)$$

```
com v = %scalar(tr(e)*e)/163 ; dis v
* An alternative is to use com v = %dot(e, e)/163
dis %sqrt(v)
```

Note that we need to covert the 1 x 1 matrix tr(e)*e into a scalar before dividing by the scalar value 163. Alternatively, we could use %dot(e, e) to produce the inner product. If you compare the answer to that from the regression model, you should find the same answer. Next, we can find the standard errors of the coefficients using that fact that:

$$\text{var}(\hat{\beta}) = \text{var}(e)^* (X'X)^{-1}$$

```
com v_beta = %scalar(v)*xx_inv
dis v_beta
```
dis %sqrt(v_beta(1,1)) %sqrt(v_beta(2,2)) %sqrt(v_beta(3,3))
```
  0.00102
  0.07698
  0.07621
```

Notice that these are the same as the standard errors of the coefficients that we obtained using the LINREG instruction. Typically, we estimate regressions in order to perform a Wald test on the regression coefficients. The simplest way to perform such a test is to use an EXCLUDE or RESTRICT instruction. However, to further illustrate matrix manipulations, consider a set of linear restrictions of the form:

$$R\hat{\beta} = c$$

where: R = q x k
$\hat{\beta}$ = estimated coefficients
c = constants

and q = number of restrictions, and k = number of estimated parameters. The F-statistic is:

$$F(q,T\text{-}k) = (c - R\hat{\beta})'[\, R(X'X)^{-1}R'\,]^{-1}(c - R\hat{\beta})/(q\hat{\sigma}^2)$$

Now consider the null hypothesis that the intercept term is zero. We can write this restriction as:

$$\begin{pmatrix}1 & 0 & 0\end{pmatrix}\begin{bmatrix}\beta_0\\ \beta_1\\ \beta_2\end{bmatrix} = 0$$

```
com q = 1
com R = || 1. , 0. , 0. ||
com c = || 0.||
com f =  tr(c - R*beta)*inv( r*xx_inv*tr(r) )*(c - r*beta)
com f1 = %scalar(f)/(q*v) ; dis f1
```

We can obtain the t-statistic for the null hypothesis using:

dis %sqrt(f1)

```
5.04661
```

You can easily verify this value as the same as that obtained using the LINREG instruction. We can test the more complicated hypothesis: $\beta_1 = \beta_2 = 0$ using:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

com n = 2
com R = || 0. , 1. , 0. | 0., 0., 1. ||
com c = || 0. | 0.||
com f = tr(c - R*beta)*inv(r*xx_inv*tr(r))*(c - r*beta)
com f1 = %scalar(f)/(2*v) ; dis f1

```
9.17622
```

We obtain precisely the same value using LINREG ad EXCLUDE. Consider:

exclude
 # dlrgdp{1 2}

```
Null Hypothesis : The Following Coefficients Are Zero
DLRGDP           Lag(s) 1 to 2
F(2,163)=      9.17622 with Significance Level 0.00016735
```

5.3 Creating Series from a Matrix

There are some instructions that operate only on series. As such, you may want to create a series from a vector or several series from matrix. For example, suppose that you want to crate a graph of the regression residuals contained in the 166 x 1 vector *e*. Since GRAPH requires a series, we need to DECLARE a series that can be used to hold the residuals. The first instruction creates the vector *errors* containing a single series. The second instruction sets each element of *errors*(1) equal to the corresponding element of *e*. The PRINT instruction allows you to compare the difference between the *errors*(1) and the *resids* series:

dec vector[series] errors(1)
set errors(1) = e(t,1)
pri 1 5 errors(1) resids

```
ENTRY       ERRORS(1)        RESIDS
1959:01  -0.005265949134       NA
1959:02   0.016019975946       NA
1959:03  -0.016059236473       NA
1959:04  -0.005057184696 -0.005265949134
1960:01  -0.017909559980  0.016019975946
```

Notice that the first three entries of *resids* are NA; one observation was lost because we used the first difference of *dlrgdp* and two more were lost because we used two AR coefficients in the LINREG instruction. Notice that the first entry of *errors*(1) corresponds to fourth entry of *resids*. To explain, note that *x* was constructed with no missing observations. Hence, first element of *e* [i.e., e(1,1)] contains the difference between y_3 and $\hat{y}_3$. Except for the two-period shift, the *resids* and *errors*(1) series are identical.

Chapter 6: Writing Your Own Procedures

In RATS, a procedure is a set of instructions that resides 'outside' of the program itself. In writing a procedure, you are effectively writing your own RATS instruction along with options, supplementary cards and choices concerning the series and variables to use. It is clear why you might want to write a procedure. Suppose that there is a particular task involving a set of instructions that you frequently invoke. It would be desirable if you could simply type a few keystrokes that instructed RATS to perform this more complicated set of instructions. In this way you could automate your task within any particular program. More importantly, by writing a procedure you can 'call up' this set of instructions in a number of programs. Since the procedure resides in a file, you can send the file to your co-author, students or post it on your website. You might think of a procedure as a 'macro' in WORD. Older programmers, like myself, might want to think of a procedure as an external function in FORTRAN. Now, the advantage is that you can customize RATS by writing (or downloading) a procedure.

RATS comes with a number of useful procedures and you can download many more from the Estima website (www.estima.com). Even if you do not want to write your own procedures from scratch, the material in this chapter should be useful to the advanced RATS user. It is often quite simple to edit existing procedures to tailor them to your needs. In order to use a procedure, it must be complied. The syntax for compiling a procedure stored on an external file is:

source *name* (The sole option is *noecho.*)

If you use the *noecho* option, you will not see anything displayed on the screen after compiling the procedure. If you omit *noecho*, you will see each line of the 'source code' preceded by the location in memory where the beginning of the code resides. This is useful for two reasons. If you need to debug a procedure, knowing the location in computer memory can be useful. Also, the fact that you get to see each line of the procedure allows you to see the instructions and comments contained in the procedure. Good programmers will include a set of comments in the procedure (usually at the beginning) that describe the proper use of the procedure.

If you are an experienced RATS user, you certainly have used procedures many times. You know that a procedure needs to be compiled only once within any program. Once it has been compiled, the procedure can be used any number of times. In Chapter 2, we discussed the procedure BJIDENT.SRC. Recall that you can use the procedure to construct the autocorrelations and partial autocorrelations of a series using:

@bjident(options) *series start end*

where:

start end	The range of the series to use for constructing the autocorrelations and partial autocorrelations

Some of the options for the procedure are:

DIFF =	Maximum regular differencings [0].
TRANS = [NONE]/LOG/ROOT	Transformation to apply to data
[GRAPH]/NOGRAPH	Do High-resolution graphs?

The key point to note is that the procedure allows you to make a number of important choices. You choose the series to use along with the *start* and *end* dates. Moreover, the procedure contains three different types of options. The DIFF = option uses an integer value, the TRANS = option is an OPTION CHOICE allowing you to select a particular type of data transformation and GRAPH/NOGRAPH is a SWITCH option. A good portion of this chapter will develop the syntax allowing you to pass information concerning series names, entries and options to a procedure. All of the procedures developed here are available on the file labeled CHAPTER6.PRG

1. A Procedure to Display the AIC and SBC

It is likely that your RATS sessions require you to calculate and display the *aic* and *sbc* a number of times. After estimating a regression equation, you need to enter the following three lines:

```
compute aic = %nobs*log(%rss) + 2*(%nreg)
compute sbc = %nobs*log(%rss) + (%nreg)*log(%nobs)
display  'aic = ' aic ' bic = ' sbc
```

Once you have typed the three lines, you never need to type them again. Instead of retyping, you probably scroll upward in your program and re-execute the three lines. However, this can be a bit of a hassle, and, if you are like me, you might even lose your place in a complicated program. A simple way to avoid this is to write a procedure containing the lines that you want to execute.

Every procedure should begin and end with its own name. For example, you can write a procedure called BIC as the following five lines:

```
procedure bic
  compute aic = %nobs*log(%rss) + 2*(%nreg)
  compute bic = %nobs*log(%rss) + (%nreg)*log(%nobs)
  display  'aic = ' aic 'bic = ' sbc
end bic
```

If you write the procedure in RATS, you can just save the procedure in a separate file somewhere on your hard drive. I save all of my procedures in the same directory containing RATS32S.EXE. If you write the procedure using WORD or some other word processing program, be sure to save the program in ASCII (*i.e.*, **.txt*) format). Good programming style dictates that similar files all have the same extension. As such, most RATS programmers use the extension *.SRC to indicate a file containing source code. Say that the five lines are saved as c:\winrats\bic.src.

To compile the procedure use:

```
source c:\winrats\bic.src
```

or, if you do not want the source code and location numbers displayed to the screen, use

```
source(noecho) c:\winrats\bic.src
```

Once you have estimated a regression, you can obtain the *aic* and *sbc* by simply typing:

```
@bic
```

2. Using SWITCH Options

The procedure BIC.SRC is quite simple since we did not need to pass any information to the procedure. Unlike BJIDENT.SRC, the procedure BIC.SRC contains all of the necessary information to compute the *aic* and the *sbc*. The RATS instruction LINREG (or BOXJENK) creates the regression variables %nobs, %rss and %nreg. These are the only three pieces of information needed to create the *aic* and *sbc*. Since all are stored internally in RATS, we do not need to 'send' or pass them to the procedure. However, the most useful procedures perform far more complicated tasks on variables, matrices, and/or entire series (or set of series). Since a procedure is external to RATS, we need a mechanism to pass information from RATS to the procedure itself.

As discussed in Chapter 1, a number of RATS instructions, such as LINREG, have SWITCH and CHOICE options. It is straightforward to include such options within your own procedure. Recall that a SWITCH option allows only two choices: ON or OFF. **For all RATS switching options, you can turn on the switch by equating its value to 1 and turn off the switch by equating its value to 0.** The appropriate syntax to include a SWITCH option in a procedure is:

OPTION SWITCH *option name default value* (Note: The default value must be 0 or 1)

If you turn back to Section 3.1 of Chapter 1, you will see that LINREG has a SWITCH option named PRINT. Since PRINT = 1 is the default, all of the following will cause the regression output to be displayed:

```
lin drs ; # constant drs{1 to 7}

lin(print) drs; # constant drs{1 to 7}

lin(print=1) drs; # constant drs{1 to 7}

com ii = 1
lin(print=ii) drs; # constant drs{1 to 7}
```

Similarly, all of the following will suppress printing the regression output:

```
lin(noprint) drs; # constant drs{1 to 7}

lin(print=0) drs; # constant drs{1 to 7}

com ii = 0
lin(print=ii) drs; # constant drs{1 to 7}
```

We can illustrate the use of a SWITCH option in a procedure by returning to BIC.SRC. Note that a number of authors calculate the values of the *aic* and the *sbc* using the following two formulas:[38]

$$aic' = T\ln\left[\sum_{t=1}^{T} e_t^2\right] + 2k - T\ln T \text{ and } sbc' = T\ln\left[\sum_{t=1}^{T} e_t^2\right] + k\ln(T) - T\ln(T)$$

It is simple to modify BIC.SRC to allow us to select the desired form. The modified procedure shown below uses OPTION SWITCH with the name *altform;* notice that the default value of *altform* is 0. The IF-ELSE block uses *altform* to determine which form to calculate and display. If *altform* = 0, the procedure will calculate and display the *aic* and the *sbc* as in our original procedure. Otherwise, the procedure will calculate and display *aic'* and *sbc'*.

```
procedure bic
option switch altform 0
   if altform = = 0 {
   compute aic = %nobs*log(%rss) + 2*(%nreg)
   compute sbc = %nobs*log(%rss) + (%nreg)*log(%nobs)
   display  ‘ aic = ' aic ' bic = ' sbc
   }
else {
   com aic = %nobs*log(%rss)  + 2*(%nreg) - %nobs*log(%nobs)
   com sbc =  %nobs*log(%rss) + (%nreg)*log(%nobs) - %nobs*log(%nobs)
   display “aic’ = ”  aic “bic’  = ”  sbc
   }
end bic
```

Suppose you have just estimated *drs* using: LIN *drs* ; # constant *drs*{1 to 7}. Since the default value of *altform* is 0, all of the following will cause the procedure to display the *aic* and *sbc* in logarithmic form:

```
@bic
@bic(noaltform)
@bic(altform=0)
com ii = 0; @bic(altform=ii)
```

Similarly, all of the following will display the alternate form of the *aic* and *sbc:*

@bic(altform=1)	[Since *altform* =1, the ELSE Block is executed]
@bic(altform)	[Turns ON the *altform* OPTION]
@bic(altform=2)	[Since *altform* ≠ 0, the ELSE portion of the procedure is executed].

[38] Note that *aic* and *aic'* will necessarily select the same model since *aic* is a monotonic transformation of *aic'*. Similarly *sbc* and *sbc'* will select the same model.

2.1 Integer and Choice Options

Oftentimes you will want something more complicated than an ON/OFF switch. At times, it will be convenient to pass a particular number to the procedure. For example, BJIDENT.SRC allows you to obtain the *ACF* and the *PACF* using first-differences of the data using DIFF = *1*. Similarly, BJIDENT.SRC allows you to select a logarithmic transformation of the data transformation using TRANS=*log*.

The syntax for an integer choice (such as the number of lags to use in the *ACF* and *PACF*) is:

OPTION INTEGER option name default value

To allow for a non-numerical set of choices (such as TRANS = log) use:

OPTION CHOICE option name default number list of choices

Examples

1. Suppose you want your procedure to estimate the series *y* as an AR(*p*) model where *p* is selected by the user. Since the number of lags is an integer, use the INTEGER option. To set the default number of lags equal to 1 use:

   OPTION INTEGER lags 1

   You should protect the user from inadvertently entering lags=0. Somewhere in your procedure, you could use the following set of instructions:

   ```
   if lags.ge.0 {
   lin y ; # constant y{1 to lags}
   }
   ```

2. Suppose you want the user to determine whether to graph series y in levels, first differences, or in logarithmic first differences. If you want the default to be such that the series is displayed in levels, you can use:

OPTION CHOICE trans 1 levels diff growth

   Your procedure should contain instructions that are similar to:

```
if trans.eq.1 {
  gra 1 ; y
}
if trans.eq.2 {
  dif y / dy ; gra 1 ; # dy
}
if trans.eq.3 {
  log y / ly ; dif ly / dly
  gra 1 ; dly
}
```

If the option trans is left unspecified or set equal to 1, the routine will produce a graph of *y*. If the user sets trans=2, the time path of the first difference of *y* will be shown and if the user sets trans=3, the time path of the growth rate of *y* will be shown.

3. Passing Series to a Procedure

Usually you will want a procedure to perform one or more operations on a series. As an experienced RATS user, you will have passed a series along with its *start* and *end* dates to a procedure. For example, we can use BJIDENT.SRC to construct the *ACF* and the *PACF* for the series *dlgdp* over the sample period 1959:1 through 1985:4 using:

```
@bjident dlrgdp * 1985:4
```

In essence, you are passing the sample values of the series *lgdp* along with the integer values of the *start* and *end* dates to the procedure. There must be a way for the procedure to 'recognize' the type of information it is being sent. This is done by listing all of the parameters (series, integer values and matrices) that can be passed on the first line of the procedure. If you actually open BJIDENT.SRC, you will see that the first thirteen lines are:

```
proc bjident series start end
type series series
type integer start end

option integer diff   0
option integer sdiff  0
option choice  trans  1  none  log  root
option switch  graph  1
option integer span

local integer nbeg nend spanl i j
local series  corrs partials
local series  transfrm  diffed     upper lower
local integer number

inquire(series=series) nbeg>>start nend>>end
```

As in BIC.SRC, the first line names the procedure. However, the first line also contains the three parameters SERIES, START and END. The second line declares that SERIES is a series. The third line declares START and END to be integers. This illustrates the general organization of a procedure. The first set of instructions classifies the parameters being passed by the procedure. Next, the various options used in the procedure are enumerated. Note that the number of differences (DIFF=), seasonal differences (SDIFF=), and the seasonal span (SPAN=) are all INTEGER OPTIONS. The form of the data transformations (TRANS=) is a CHOICE OPTION and whether or not to display a graph is a SWITCH OPTION. The third set of instructions indicates whether or not the variables will be local (defined only within the procedure) or global (accessible throughout the main program and procedures). Fourth, you might want to inquire about the nature of the series being passed to the procedure—it is especially useful to obtain the

starting and ending entries. Finally come the set of instructions you want the procedure to perform. The general structure of any procedure is:

PROCEDURE *procedure name list of parameters*
TYPE instructions
OPTION instructions
LOCAL instructions
INQUIRE instructions
  set of instructions to be performed
END *procedure name*

The procedure is executed by @*procedure name list of parameters*. Note that the order of the parameter list must match that used within the procedure.

4. Writing a Procedure to Test for Unit Roots

We will illustrate the process by writing a procedure called UNIT.SRC that will perform an augmented Dickey-Fuller test. The final procedure will be similar to that in DFUNIT.SRC that comes with RATS. However, by writing such a procedure from scratch, it will be possible to illustrate the structure and key instructions of any procedure. We begin simply. At first, the procedure will only estimate a model of the form:

$$\Delta y_t = \beta_0 + \rho y_{t-1} + \beta_1 \Delta y_{t-1} + \varepsilon_t$$

The entire regression output will not be displayed. Instead, the procedure will only display the estimate of ρ and the *t*-statistic for the null hypothesis $\rho = 0$.

After compiling, the procedure can be invoked using @unit *series*. For example, you can use to procedure to test for a unit root in the variable *lrgdp* using:

@unit *lrgdp*

We begin by writing UNIT.SRC as:

```
procedure unit y
type series y
set dy = y - y{1}
lin(noprint) dy
# y{1} dy{1} constant
dis 'The estimate of rho = ' %beta(1)
dis 'the t-statistic for the null hypothesis rho = 0 is ' %tstats(1)
end unit
```

The interpretation of lines 1 and 8 is straightforward; these are simply the starting and ending lines of the procedure. In line 1, we inform RATS that we will pass a parameter to UNIT.SRC.

Line 2 needs a bit of explanation. In general, you will want to pass series, integers, real variables, matrices, etc., to the procedure. RATS needs a mechanism by which to determine the type of parameter that is being passed. Line 2 indicates that the parameter we pass (i.e., *y*) is a series. This is done using the TYPE instruction. The syntax is:

TYPE *data type* *parameter list*

where:

data type	can be any RATS date type such as SERIES, INTEGER, REAL, SYMMETRIC, …
parameter list	is the list of parameters you which to declare as this particular date type.

Notes:

1. Use one TYPE instruction for each data type.

2. The TYPE instructions should begin in line 2 of a procedure (*i.e.*, immediately after the PROC *name parameter list* instruction).

3. We cannot include *dy* on a TYPE instruction since we do not pass this series to the procedure. Instead, *dy* is created within the procedure.

If you are not familiar with programming, it might seem troublesome that we want to pass *lgdp* to the procedure, but the procedure seems to use only the series denoted by *y*. Actually, *y* is only a placeholder—it will accept any series that is passed to it.

Line 3 creates the first-difference of *y* and the regression is estimated in lines 4 and 5. Line 7 displays the point estimate of ρ and line 8 displays the *t*-statistic for the null hypothesis $\rho = 0$. Thus, if we use @unit *lrgdp*, we will obtain the estimate of ρ, the *t*-statistic for the null $\rho = 0$ for the equation:

$$\Delta lrgdp_t = \beta_0 + \rho lrgdp_{t-1} + \beta_1 \Delta lrgdp_{t-1} + \varepsilon_t$$

The procedure seems to work as intended. However, there is a potential problem that needs to be addressed. Just because a procedure works within your particular program does not mean it will work in all possible programs. Suppose that you posted a copy of UNIT.SRC on your class webpage and one of your students had a data set that included a variable called *dy*. Every time she invoked UNIT.SRC, the procedure would write over her variable. In order to separate variables used in the main body of a program from those used in a procedure, RATS allows you to designate variables used in a procedure as local variables.

4.1 Creating Local Variables

Unless otherwise stated, all variables in RATS can be used in the main program or in a procedure. Thus, by default, all variables are global. If you modify a variable within a procedure, it will be modified when control returns to the main body of the program. In many circumstances, this can be desirable. In fact, you might want a procedure to perform a particular transformation on a variable and then return control to the main program. Other times, you might create a variable within a procedure that is already named within the main body of the program. Thus, the procedure would overwrite the values of the variable. A local variable is one that is used only in the procedure that defines it. Thus, changing the value of a local variable changes its value only within the procedure.

There is another reason to use local variables. Suppose a procedure includes a series that is not defined as a local series. If you attempt to compile the procedure prior to the ALLOCATE instruction, RATS will display the error message:

```
## SR1. ALLOCATE Instruction Needed Before Series or Equations Can Be Used
```

The point is that RATS will have no way of knowing the length of the series appearing in the procedure. By declaring the variable as a local variable, the procedure can be compiled before the ALLOCATE instruction is encountered.

To define a variable as local, use:

LOCAL SERIES *series name*

or

LOCAL INTEGER *name*

Thus, we want to include the following line in UNIT.SRC

LOCAL SERIES dy

The precedence of statements is that DECLARE instructions should precede TYPE, LOCAL and OPTION instructions. All of these must precede the executable instructions. Hence, we want the first three lines of UNIT.SRC to be:

procedure unit y
type series y
local series dy

Notice that we do not need to define *y* as a local variable since *y* is a parameter (i.e., the symbol *y* acts only as a placeholder).

4.2 Adding Options

To turn UNIT.SRC into a procedure that you might want to use in your own research, we need to make three changes to the basic program. The user should be able to select the deterministic regressors to include in the model, select the lag length to use in the augmented form of the Dickey-Fuller test, and to choose whether to display a graph of the residuals.

The Choice of Lag Length
Since the number of lags is an integer, we need to make only two modifications to the procedure:

1. Include the instruction: OPTION INTEGER LAGS 1.
2. Replace the supplementary card in the LINREG instructions with:

    # y{1} dy{1 to lags} constant

Since the default value of LAGS is 1, if you use @unit *lrgdp* or @unit(lags=1) *lrgdp*, you estimate a regression only one lag of Δ*lrgdp*. If you invoke the procedure using @unit(lags=4) *lrgdp,* you estimate a regression with four lags of Δ*lrgdp.*

The Choice of Deterministic Regressors

The selection of the deterministic regressors is a clearly a CHOICE option; the deterministic regressors can be NONE, CONSTANT or TREND (*i.e*., constant plus trend). As such, we need to make four modifications to the program:

1. Include the instruction: OPTION CHOICE DET 2 NONE INTERCEPT TREND

   Notice that default value of DET is 2. If you use @unit(det=none) *lrgdp*, the value of DET will be 1, @unit *lrgdp* or @unit(det=intercept) *lrgdp* the value of DET will be 2, and @unit(det=trend) *lrgdp* the value of DET will equal 3.

2. In order to create the time trend, we need to include the instruction:

   set time = t

3. Since the procedure creates the series *time*, we should include this in the list of local series using: LOCAL SERIES dy time.

4. We need to include a set of IF statements to select the regressors:

   If det.eq.1 ; lin(noprint) dy ; # y{1} dy{1 to lags}
   If det.eq.2 ; lin(noprint) dy ; # y{1} dy{1 to lags} constant
   If det.eq.3 ; lin(noprint) dy ; # y{1} dy{1 to lags} constant time

The Choice to Display a Graph of the Residuals

We can use a SWITCH OPTION to allow the user to turn ON or OFF a display of the graph. We need to make the following modifications to the procedure:

1. Include the instruction OPTION SWITCH graph 0

   As formulated, the default the value of GRAPH is zero. In order to turn the option on (*i.e*., in order to set GRAPH = 1), we can use:
   @unit(graph) *lrgdp*
   @unit(graph=1) *lrgdp*

2. We need to save the residuals from the selected regression equation. Hence, we modify our IF instructions such that the residuals are saved in the series called *resids*:

   If det.eq.1; lin(noprint) dy / resids ; # y{1} dy{1 to lags}
   If det.eq.2 ; lin(noprint) dy / resids ; # y{1} dy{1 to lags} constant
   If det.eq.3; lin(noprint) dy / resids ; # y{1} dy{1 to lags} constant time

3. We obtain the *ACF* of *resids* using:

```
cor(noprint,number=24) resids / corrs
```

4. If GRAPH = 1, we create a time series plot of the residuals and the ACF of the residuals using:[39]

```
cor(noprint,number=24) resids / corrs
if graph==1 {
  spgraph(hfields=1,vfields=2,header='RESIDUAL ANALYSIS')
  gra(header='Time Path of the Residuals') 1 ; # resids
  gra(max=1.0,min=-1.0,style=bar,number=0,nodates,header='Residual ACF') 1
 # corrs
 spgraph(done)
      }
```

5. Notice that the procedure creates the series *corrs* and *resids*. We want to include these newly created series on the list of local series using:

```
local  series  dy resids corrs time
```

The complete procedure is listed below:

```
procedure unit y
type series y

option switch graph 0
option integer lags 1
option choice det 2 none intercept trend
local series dy resids corrs time
set dy = y - y{1}
set time = t
if det.eq.1; lin(noprint) dy / resids ; # y{1} dy{1 to lags}
if det.eq.2 ; lin(noprint) dy / resids ; # y{1} dy{1 to lags} constant
if det.eq.3; lin(noprint) dy / resids ; # y{1} dy{1 to lags} constant time

dis 'The estimate of Rho = ' %beta(1)
dis 'The t-statistic for the null hypothesis Rho = 0 is ' %tstats(1)
cor(noprint,number=24) resids / corrs
if graph==1 {
spgraph(hfields=1,vfields=2,header='Residual Analysis')
```

---

[39] At this point in the construction of the procedure, you might want to place the COR instruction inside of the IF block. As such, the correlations would be computed only if the GRAPH option is selected. However, when we extend the program in later sections we will want to place COR outside of the IF block.

```
gra(header='Time Path') 1 ; # resids
gra(max=1.0,min=-1.0,style=bar,number=0,nodates,header='Residual ACF') 1 ; # corrs
spgraph(done)
}
end unit
```

5. Retrieving START and END entry values

If you pass a series to a procedure, you might not want the procedure to operate on the entire range of entries. The INQUIRE instruction is specifically designed to allow the user to select the starting and ending values of any series passed to a procedure. It is important to note that procedures are often designed to work with a variety of data sets. As such, it is most useful to write procedures that use integers, as opposed to dates, to represent the entries of a series. You can use the INQUIRE instruction to obtain the defined range of a series as follows:

INQUIRE (SERIES = *series name*) *value1 value2*

After execution, *value1* will contain the starting entry value of *series name* and *value2* will contain the ending entry value. To illustrate, suppose you read in the data set MONEY_DEM.XLS and enter the following TABLE instruction:

table / rgdp tb1yr

```
Series   Obs        Mean        Std Error      Minimum       Maximum
RGDP     169   5142.36449704 1950.84049366 2273.00000000 9439.90000000
TB1YR    167      6.15387226    2.39362220    2.71333333   14.38000000
```

Both series are quarterly running from 1959:1 to 2001:1; as such, 1959:1 is entry 1 and 2001:1 is entry 169. Recall that the first two values of *tb1yr* are NA. Next, enter:

inquire(series=rgdp) v1 v2
inq(series=tb1yr) v3 v4
dis v1 v2 v3 v4

```
1 169 3 169
```

Hence, *v1* and v2 contain the first and last entries of *rgdp*, respectively. However, *tb1yr*(1) and *tb1yr*(2) are NA; as such, *v3* = 3 and *v4* = 169.

You can also use INQUIRE with the SEASONAL option to obtain the seasonal frequency of the data. The following instruction returns the seasonal frequency in the variable *value*:

INQUIRE(SEASONAL) *value*

Since we used CALENDAR to instruct RATS that MONEY_DEM.XLS contains quarterly data, we can store the seasonal frequency in the variable *v* using:

inq(seasonal) v; dis v

```
4
```

It is unlikely that you will ever need to use INQUIRE outside of a procedure. In fact, the more useful form of INQUIRE is:

INQUIRE(SERIES = *series name*)   *v1 >>start v2 >>end*

where: *start* and *end* are the *start* and *end* integer values supplied by the user when the procedure is executed. The expressions *>>start* and *>>end* instruct RATS to equate *v1* with *start* and to equate *v2* with *end* if explicit values are given by the user.

Examples:

BJIDENT.SRC contains the instruction:

```
inquire(series=series) nbeg>>start nend>>end
```

1. @bjident rgdp 1959:1 2001:1

   Here *start* = 1 and *end* = 2001:1. Similarly, *nbeg* = 1 and *nend* = 161. If the user does not provide the values *start* and *end*, *nbeg* and *nend* will automatically default to the first and last dates for which the series is defined.

2. @bjident rgdp * 2001:1

   Here *start* is 'undefined' so *nbeg* becomes the first defined value and *nend* is 169. Since, the first entry for *rgdp* is 1, *nbeg* = 1.

3. @bjident tb1yr 1959:1 1999:1

   Now *start* = 1 and *end* = 161 so that *nbeg* = 1 and *nend* = 161. Since the first entry of *tb1yr* is 3, RATS uses entries 3 through 161 to form the autocorrelations.

Suppose you want the user of your procedure to supply a *series* along with the *start* and *end* dates. The typical format will be:

@procname series start end

Your procedure should have the following structure:

procedure *procname seriesname start end*
type series *seriesname*
type integer *start end*
option instructions
local instructions
inquire (series = *series*) *v1>>start v2>>end*

As in the previous examples, the PROCEDURE instruction contains the name of the procedure (*procname*) along with a list of all of the parameters what will be passed to the procedure. Here, we pass a series (*seriesname*) and the *start* and *end* entries. The second instruction declares *seriesname* to be a SERIES and the third declares *start* and *end* to be integers. The fourth instruction equates *v1* and *v2* with the *start* and *end* values selected by the user. If the user does not provide the values *start* and *end*, *v1* and *v2* will automatically default to the first and last dates for which the series is defined.

To alter UNIT.SRC such that we can pass starting and ending dates, we need to make the following five modifications to the procedure:

1. We need to list all parameters passed to the procedure. Since we pass *start* and *end* values, the first line of the UNIT.SRC should be:

   procedure *unit y start end*

2. We need to instruct RATS that *start* and *end* are integers. This is done using:

   type integer *start end*

3. We need to use the INQUIRE instruction to store the *start* and *end* values in the variables *v1* and *v2*. Note that INQUIRE is an executable instruction; it should be placed after all TYPE, OPTION, and LOCAL instructions.

   inquire(series=y) *v1>>start v2>>end*

4. UNIT.SRC creates the two integer variables *v1* and *v2*. We can declare then as LOCAL variables using:

   local integer v1 v2

5. Modify the LINREG, GRAPH and CORRS instructions such that *v1* and *v2* are used as the starting and ending entries:

   if det.eq.1; lin(noprint) dy v1 v2 resids ; # y{1} dy{1 to lags}
   if det.eq.2 ; lin(noprint) dy v1 v2 resids ; # y{1} dy{1 to lags} constant
   if det.eq.3; lin(noprint) dy v1 v2 resids ; # y{1} dy{1 to lags} constant time

The graph and autocorrelations can be obtained from:

```
gra(header='Time Path') 1 ; # resids * v2
cor(noprint,number=24) resids * v2 corrs
```

The source code for the first nine lines of the procedure is:

```
procedure unit y start end
type series y
type integer start end
option switch graph 0
option integer lags 1
option choice det 2 none intercept trend
local series dy resids corrs  time
local integer v1 v2
inquire(series=y) v1>>start v2>>end
```

Now, the procedure can be quite useful. For example, open the file CHAPTER6.PRG and read in the data set using Program 6.1. As the program illustrates, you can embed the UNIT.SRC in a DO loop so that you that you perform augmented Dickey-Fuller tests on *tb1yr* using lags 1 through 12:

do i = 1,12; @unit(lags=i) tb1yr ; end do i

```
The estimate of rho with  1  lags =       -0.06280
The t-statistic for the null hypothesis rho = 0 is       -2.49752

The estimate of rho with  2  lags =       -0.04931
The t-statistic for the null hypothesis rho = 0 is       -1.98906

. . .

The estimate of rho with  11  lags =       -0.05338
The t-statistic for the null hypothesis rho = 0 is       -2.00540

The estimate of rho with  12  lags =       -0.05416
The t-statistic for the null hypothesis rho = 0 is       -2.00671
```

5.1 Passing Information by Address

To this point, we have thought in terms of passing information to a procedure. However, in many circumstances you will want to pass a variable from a procedure to the main program. This is not particularly difficult if you are working with a global variable; a global variable is accessible from any point in RATS. Of course, if you sent the procedure to someone else, that person would need to know how you named the variable in question. It would be most convenient if you

allowed the user to fetch the variable in question using any name she selected. The way to do this is to pass the information by address (not by name). This is done by placing an asterisk * immediately preceding the variable name on the TYPE instruction. Suppose, for example, that you wanted to pass the residual autocorrelations created by CORR back to the main program. To illustrate, suppose we modify UNIT.SRC by removing *corrs* from the list of LOCAL series. Once you invoke the procedure using @unit tb1yr, you can print the *ACF* of the regression residuals by entering the instruction: print / corrs. The two potential problems with this method are: (1) the user needs to know the name of the series containing the correlations and (2) you might not want the name *corrs* to refer to a global variable. A way to rectify this problem is to modify the PROCEDURE, TYPE SERIES and LOCAL SERIES instructions such that:

procedure unit y start end corrs	Add corrs to the parameter list
type series y *corrs	Define *corrs* to be a series that can be passed back to the main program.
local series dy resids time	Remove *corrs* as from the list of LOCAL series.

No other instructions need to be modified or added. Now, you can invoke the procedure and print the ACF of the residuals using:

@unit *series start end name for the correlations*

Note that the user can select any valid series *name for the correlations*; *corrs* is simply a placeholder for the name specified by the user. For example, the following three instructions all yield the identical output:

@unit tb1yr 3 169 corrs ; pri / corrs

@unit tb1yr 1 2000:1 zz ; pri / zz

@unit tb1yr / x ; pri / x

5.2 Optional Fields

Notice that the following three instructions produce quite different results:[40]

@unit tb1yr

```
The estimate of rho with  1  lags =       -0.05355
The t-statistic for the null hypothesis rho = 0 is       -2.31170
```

@unit(graph) tb1yr / corrs

```
The estimate of rho with  1  lags =       -0.05355
The t-statistic for the null hypothesis rho = 0 is       -2.31170
```

[40] The version of UNIT.SRC distributed with this book contains the ‘fix’ described below. Hence, it will not produce the error message.

@unit(graph) tb1yr

```
The estimate of rho with  1  lags =       -0.05355
The t-statistic for the null hypothesis rho = 0 is       -2.31170
## SX22. Expected Type SERIES, Got Function Instead
The Error Occurred At Location 1060 of UNIT
```

The reason is that the field for the correlations *(i.e., name for the correlations*) must be specified if a graph of the *ACF* is to be displayed. Recall that the GRAPH instruction uses:

gra(max=1.0,min=-1.0,style=bar,number=0,nodates,header='Residual ACF') 1; # corrs

Hence, the graph cannot be created unless a name for the field is specified by the user. The way to fix the problem is to use one series to pass the correlations and another series on the COR instruction. Consider the following modifications to the procedure:

1. LOCAL SERIES *dy resids xx time*

   A variable *xx* is defined to be a local series. This series will hold the correlations.

2. cor(noprint,number=24) resids v1 v2 xx
   if %defined(corrs) ; set corrs 1 25 = xx

   The autocorrelations of *resids* are now stored in the series *xx*. Note that the RATS function %DEFINED(*name*) returns the status of a procedure parameter or option called *name*. Here, if the field for *corrs* is specified by the user, %defined = 1 and the SET instruction is performed. If the field is not specified, %defined = 0 and the SET instruction is not performed. No error message is created since the series *corrs* is used only when the field is specified by the user.

3. gra(max=1.0,min=-1.0,style=bar,number=0,nodates,header='Residual ACF') 1; # xx

   If the GRAPH option is selected, a graph of *xx* is produced.

Examples:

1. Suppose you want to perform three unit root tests on *tb1yr*. The first does not have any deterministic regressors, the second has an intercept, and the third has an intercept and a trend. Use:

   do det = 1,3 ; @unit(det=det,lags=7) tb1yr ; end do det

   The procedure produces the desired result since the first time through the loop, *det* = 1, the second time *det* = 2, and the third time *det* = 3.

2. To perform the test with an intercept in the equations for *tb1yr* and *tb3mo* use:

   dofor j = tb1yr tb3mo ; @unit(lags=7) j ; end dofor

3. @unit(lags=7,graph) tb1yr 1980:1 * corrs

   A unit root test containing an intercept and seven lagged values of *tb1yr* is performed. A graph of the residuals is displayed and the autocorrelations of the residuals are returned in the series *corrs*. Note that the test is performed beginning with 1980:1. The results for the *t*-test are identical to those obtained from:

```
dif tb1yr / drl
lin drl 1980:1 * ; # constant tb1yr{1} drl{1 to 7}
```

6. A Procedure for Computing Lag Lengths

Anyone working with time-series data routinely selects the optimal lag length to use in an autoregressive model. It is straightforward to write a procedure to automate this process. The procedure below, called LAGLENGTH.SRC, estimates a number of *AR*(*p*) models and reports the one selected as 'best' by the SBC. The procedure is executed using:

@laglength(option) *y start end laglength*

where:

y	Series to estimate as an AR(*p*) model
start and *end*	The range to use in the estimation. Note that one observation will be lost for each lag.
laglength	(Optional) If *laglength* is specified, the procedure returns the integer value of the lag selected by the *SBC*.

The sole option for the procedure is:

MAXLAG =	Maximum number of lags to use in the lag length test.

In the first line below, the PROCEDURE instruction contains the name of the procedure and a parameter list containing *y*, *start end*, *laglength*. The next two lines define the TYPE of these parameters. Line 2 indicates that *y* is a series and line 3 indicates that *start*, *end* and *laglength* are all integers. Notice the asterisk preceding *laglength*. If this field is specified by the user, *laglength* will return the integer value of the optimal lag to the main body of the program.

```
procedure laglength y start end laglength
type series y
type integer start end *laglength
```

The next section of a procedure should contain the OPTIONS. Here, there is a single option called *MAXLAG* with a default value equal to 1. The user can use the option *maxlag* = *p* to select the maximum value of *p* to use in the estimated models autoregressive models.

```
option integer maxlag 1
```

The third section of procedure declares the LOCAL variables. Here, *v1, v2, bestlag* and *i* are all declared as local integers and *sbc, sbcmin* are local real numbers.[41]

```
local integer v1 v2 bestlag i
local real sbc sbcmin
```

---

[41] Note that LAGLENGTH.SRC does not create any series; as such, we do not define any local series. The series *y* is not created by the procedure; *y* is just a placeholder for the series passed to LAGLENGTH.SRC by the user.

In any procedure, the executable instructions follow the declaration of the local variables. The first executable instruction is INQUIRE. The integers *v1* and *v2* will contain the *start* and *end* dates selected by the user. The next three lines initialize several key variables. The LINREG regresses *y* on a constant, calculates the *SBC* (called *sbcmin*) for this regression and computes *bestlag* = 0. In essence, the procedure estimates and calculates the *SBC* for an *AR*(0) model.

```
inq(series=y) v1>>start v2>>end

lin(noprint) y v1+maxlag v2
# constant
compute sbcmin = %nobs*log(%rss) + (%nreg)*log(%nobs) , bestlag = 0
```

The next section of the procedure estimates an *AR*(*p*) model for lag length running from 1 to *maxlag*. Notice that all estimations are performed over the same sample period; the start date for each is *v1+maglag* and the end date is *v2*. The first time through the DO loop, *i* = 1 and an *AR*(1) model is estimated. The value of the *SBC* is compared to that of the *AR*(0), if the *AR*(1) has the smaller *SBC*, *bestlag* is equated to *i* and *sbcmin* is replaced by *sbc*. Thus, if the *AR*(1) provides a better fit than the *AR*(0), *bestlag* = 1 and *sbcmin* contains the *SBC* of the *AR*(1). The next time through the loop, *i* = 2. The *SBC* from the *AR*(2) is compared to that of the previously selected best fitting model. If the *AR*(2) fits better than the model indicated by *bestlag*, the value of *bestlag* is set equal to 2. On exiting the loop, *bestlag* contains the lag length of the model providing the best fit.

```
do i = 1,maxlag
   lin(noprint) y v1+maxlag v2
   # constant y{1 to i}
   compute sbc = %nobs*log(%rss) + (%nreg)*log(%nobs)
   if sbc < sbcmin ; compute bestlag = i , sbcmin = sbc
end do i
```

If *bestlag* = 0, the regression is not estimated and the procedure displays: DO NOT USE LAGS. If *bestlag* is greater than zero [*i.e.*, if any of the *AR*(*p*) models fit better than the *AR*(0)], the *AR*(*bestlag*) model is estimated and the output is displayed. Notice that the sample period runs from *v1+bestlag* to *v2*.

```
if bestlag.EQ.0 ; dis 'DO NOT USE LAGS'
if bestlag.GT.0
lin y v1+bestlag v2 ; # constant y{1 to bestlag}
```

If the user specifies *laglength*, the procedure returns the value of *bestlag*. Consider:

```
if %defined(laglength) ; com laglength = bestlag
end laglength
```

7. Interacting With Procedures

RATS contains a number of instructions that allow you to interact with a procedure. The most straightforward of these instructions is MESSAGEBOX. You can use MESSAGEBOX to halt the execution of the procedure and display an ‘Alert’ that contains a message. Execution of the procedure will resume when the user enters the appropriate button. The most commonly used syntax for MESSAGEBOX is:

messagebox(status=*value*,style=*style*) '*Your Message*'

where:

STATUS = *value*	The integer variable *value* will set equal to a particular number corresponding to the button selected by the user.
STYLE=[ALERT]	Displays a MESSAGEBOX that will look like the following:

STYLE = YESNO	Halts execution of the program and sets *value* = 1 if YES is selected and *value* = 0 if NO is selected.
STYLE=OKCANCEL	Halts execution of the program and sets *value* = 1 if OK is selected and *value* = 0 if CANCEL is selected.
STYLE=YNCANCEL	Halts execution of the program and sets *value* = 1 if YES is selected, *value* = 0 if NO is selected and *value* = -1 if CANCEL is selected.

In most instances, you will use a SWITCH or CHOICE OPTION to allow the user to make a selection. The advantage of MESSAGEBOX is that it allows the user to make a choice in the midst of the execution of the procedure.

The QUERY instruction works in a similar fashion. QUERY halts the execution of the procedure and prompts the user to to input the values for a list of variables. The most commonly used syntax is:

query(prompt=‘*Your message*’) *variable list*

where:

variable list — The *list of variables* whose values are input by the user. The individual variables can be any combination of integer, real, string, or label variables.

Note that QUERY uses a dialog box similar to that shown above. The user's response is entered such that each value is separated by commas or spaces. Notice that QUERY does not display a separate line for each value to be entered. As such, I recommend using a separate QUERY instruction for each value to be input. Also, the TYPE of variable to be input (*e.g.*, *real, string,* or *integer*) must be specified prior to the QUERY instruction. COMPUTE or DECLARE, for example, can be used to determine the variable TYPE.

Examples

1. Suppose you reach the end of LAGLENGTH.SRC and see the message: DO NOT USE LAGS. You might be concerned that the *SBC* selected a model that was too parsimonious. You could insert a MESSAGEBOX to prompt the user for further instructions. Consider the following modifications to the procedure:

   a) if bestlag.EQ.0 ; mes(style=alert) 'DO NOT USE LAGS'

   This modification will inform the user of the problem by halting execution of the procedure and display an ALERT.

   b)
   ```
   if bestlag.EQ.0 {
     mes(style=YESNO,status=status) 'Do you want to enter the lag length?'
     if status.eq.1  ; query(prompt='Enter the lag length') bestlag
   }
   ```

   The procedure will display a MESSAGEBOX with YES and NO buttons along with the message: *Do you want to enter the lag length?* If the YES button is selected, the user will see the prompt: *Enter the lag length.* The variable integer variable *bestlag* is set equal to the value input by the user.

2. You can add the following two lines to LAGLENGTH.SRC:

   ```
   messagebox(style=yesno,status=status) 'Do you want to run a unit root test?'
   if status.eq.1 ; @unit(lags=bestlag-1) y v1 v2
   ```

   If you position the two lines after the IF %DEFINED(laglength) instruction, the variable *bestlag* contains the lag length selected by the *SBC*. The MESSAGEBOX instruction prompts the user with the question: *Do you want to run a unit root test?* If the *YES* button in the MESSAGEBOX is selected, the variable *status* = 1. Since the condition on the IF instruction is TRUE, LAGLENGTH.SRC calls UNIT.SRC. Hence, UNIT.SRC performs a unit root test on the variable *y* over the sample period *v1* to *v2* using the lag length contained in *bestlag*. If the *NO* button is selected, *status* = 0 and the unit root test is not performed.

Read in the data set MONEY_DEM.XLS and compile the procedures UNIT.SRC and LAGLENGTH.SRC. It is important to compile the procedures in this order: If procedure A makes reference to procedure B, B must be compiled before A. Now enter the following instruction and select the YES button:

@laglength(maxlag=8) tb1yr

```
   Variable       Coeff       Std Error       T-Stat     Signif
******************************************************************
1.  Constant     0.327483556  0.148546781      2.20458  0.02892885
2.  TB1YR{1}     1.259723680  0.077483059     16.25805  0.00000000
3.  TB1YR{2}    -0.608960910  0.119220538     -5.10785  0.00000093
4.  TB1YR{3}     0.552438699  0.119096075      4.63860  0.00000731
5.  TB1YR{4}    -0.255732154  0.077249623     -3.31046  0.00115404

With 3 lags, the estimate of rho =       -0.05253
The t-statistic for the null hypothesis rho = 0 is       -2.33526
```

LAGLENGTH.SRC selects a lag-length of 4; if you selected the *YES* button, you see the output from UNIT.SRC such that the augmented Dickey-Fuller test contains (*bestlags* – 1) lags. If you want to include these two procedures in your research, you could make several small modifications.

a) Notice that LAGLENGTH.SRC displays regression output with 4 lags and UNIT.SRC displays "`With 3 lags, the estimate of rho =`. You can modify LAGLENGTH.SRC such that it displays the message: `In the augmented Dickey-Fuller test.` The output will look like:

```
In the augmented Dickey-Fuller test
With 3 lags, the estimate of rho =       -0.05253
The t-statistic for the null hypothesis rho = 0 is       -2.33526
```

b) It is important that the user compile UNIT.SRC prior to compiling LAGLENGTH.SRC. One way to ensure this is done correctly is to include both procedures on a single file *in the desired order*. Save the file using a descriptive name and the extension *.*SRC*. When the user compiles the file, all of the procedures will be compiled in the appropriate order. For example, you might you have a single file structured as follows:

```
procedure unit y start end corrs_save
type series y *corrs_save
type integer start end
```

other program statements

```
end unit
**********
procedure laglength y start end laglength
type series y
type integer start end *laglength
```

other program statements

```
end laglength
```

8. Creating a Menu

RATS allows you to present the user with menu of choices in one of two ways. The first presents the user with a box listing the choices. After the user selects a choice from the list, a specified block of instructions can be performed. The typical syntax for the MENU block is:

MENU ‘*Message to Display*’

CHOICE ‘*Name of First Choice*’
  Instruction block for first choice

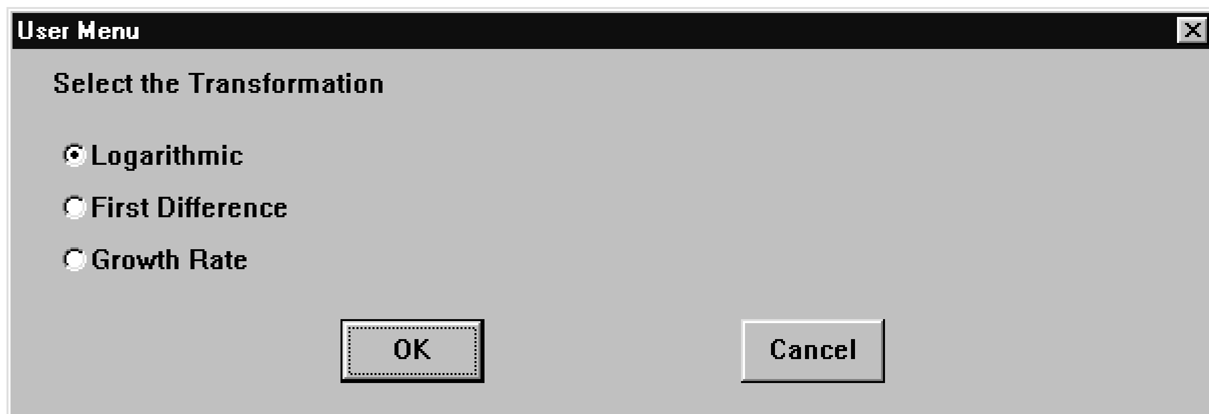
CHOICE ‘*Name of Second Choice*’
  Instruction block for second choice
  …
END MENU

Example

The following procedure, called *TRANSFORM*, uses the MENU instruction to prompt the user for a particular type of data transformation. After compiling *TRANSFORM.SRC*, the procedure is executed using:

@transform *series*

The user will see a dialog box that looks something like:



Once the desired transformation is selected, the procedure will create a graph of the transformed series. The label of the transformed series will be the *series label* appended with an *L* for the logarithmic transformation, *D* for the first-difference, or *G* for the growth rate. Notice that the TYPE instruction listed below declares the parameter *y* to be a series. The instructions in the MENU—END MENU block create the dialog box shown above.

```
procedure transform y
type series y

menu 'Select the Transformation'

choice 'Logarithmic'
{
sta(noprint,fractiles) y
  if %minimum > 0. {
    com a$ = 'L' + %label(y)
    set %s(a$) = log(y)
  }
 if %minimum.le.0
   mes(style=alert) "I CAN TAKE THE LOG OF POSITIVE NUMBERS ONLY"
}

choice 'First Difference'
{
com a$ = 'D' + %label(y)
set %s(a$) = y - y{1}
}

choice 'Growth Rate'
{
com a$ = 'G' + %label(y)
set %s(a$) = y/y{1} - 1
}

end menu
graph(header='Time Path of ' + a$) 1 ; # %s(a$)

end transform
```

If *Logarithmic* is selected, a simple check is performed before the routine attempts to create the log of series. The STATISTICS instruction is used to obtain the minimum value of the *series*. If this value is negative, an ALERT is displayed informing the user that it is not possible to obtain the log of a negative number. Only if the smallest value of *series* is positive will the procedure create the new series. The label attached to the newly created series is *L* + *series label*. Similarly, if *First Difference* is selected, the instructions in the second CHOICE block are executed. The first difference of *series* is created with the label *D* + *series label*.

The final instruction creates the graph of the transformed series. Notice that the choice *Growth Rate* does not check for the possibility of values that equal zero. If any value of *series* does equal zero, the associated entry value for the transformed series will be NA. The next two lines of Program 6.1 compile the procedure and EXECUTE the procedure using *rgdp*:

source(noecho) c:\winrats\transform.src
@transform rgdp

Select *Growth Rate* and use the instruction:

table /

Series	Obs	Mean	Std Error	Minimum	Maximum
DATE	169	1979.876331	12.232185	1959.100000	2001.100000
GDP	169	3572.739053	2873.158128	496.100000	10243.600000
RGDP	169	5142.364497	1950.840494	2273.000000	9439.900000
M2	169	1904.835266	1399.706717	287.800000	5043.710000
M3	169	2414.462229	1916.764710	290.053333	7260.136667
TB3MO	169	5.915148	2.590483	2.303333	15.053333
TB1YR	167	6.153872	2.393622	2.713333	14.380000
GRGDP	168	0.008551	0.009027	-0.020388	0.038528

Notice that the series GRGDP has been created. Since this variable has *not* been declared as a LOCAL SERIES, it is accessible from the main body of the program.

8.1 Creating a USERMENU

A second way to create an interactive procedure is to use the USERMENU instruction. USERMENU allows you to add a user-defined menu to the RATS menu bar. The user clicks on the pull-down menu and the list of choices appears. Once a USERMENU is activated, it controls the flow of the program until a selection is made. Typically, you will use the USERMENU instruction at least three times within a procedure.[42] The first use is to DEFINE the structure of the menu, the second causes the menu to appear on the menubar and the third is to REMOVE the menu.

The typical syntax to DEFINE the structure a USERMENU is:

usermenu(action=define,title=‘*Title*’) 1>>‘*string 1*’ 2>>‘*string 2*’ … *n*>>‘*string n*’

The ACTION=DEFINE option is used to create a pull-down menu to the right of the RATS Help menu with the title determined by the string *Title*. The choices on the USERMENU appear as *string 1*, *string 2*, … through *string n*. The menu does not actually appear on the menubar until the simple instruction USERMENU is encountered. The user can now select one of the choices. Once a choice is selected, the variable %MENUCHOICE is set equal to integer value of the selection. At this point, a series of IF instructions can be used to control program execution.

Finally, the menu is removed using:

[42] The next section discusses how to turn on and off choices using the ENABLE option.

```
USERMENU(action=remove)
```

It is possible to rewrite the procedure *TRANSFORM* so that it uses the USERMENU instruction instead of the MENU instruction. Consider the modified procedure called *TRANSFORM_USER*. The procedure is executed using:

@transform_user *series*

As in the original procedure, the user passes a series to the procedure; the TYPE instruction defines this parameter as a series. The first USERMENU instruction defines a menu with the title *Transform*. Clicking on this pull-down menu will reveal the three choices *Logarithmic, First Difference,* and *Growth Rate*. Notice that the instruction is completely contained on a single program statement; the use of the $ sign as a continuation is simply to make the program easier to read.

```
procedure transform_user y
type series y

usermenu(action=define,title='Transform') $
1>>'Logarithmic' $
2>>'First Difference' $
3>>'Growth Rate'
```

The menu does not appear until USERMENU with the (default) option ACTION=RUN is encountered. If the user selects *Logarithmic*, %menuchoice=1 and the instructions in the first IF block are executed. Again, there is a check that prevents the user from trying to take the log of a number that is not positive.

```
usermenu

if %menuchoice.eq.1 {
sta(noprint,fractiles) y
  if %minimum > 0. {
    com a$ = 'L' + %label(y)
    set %s(a$) = log(y)
  }
  if %minimum.le.0
    mes(style=alert) "I CAN ONLY TAKE THE LOG OF POSITIVE NUMBERS"
}
```

The integer variable %menuchoice equals 2 if the user selects *First Difference* and equals 3 if the user selects *Growth Rate*. After the appropriate IF block is completed, a graph of the transformed series is created. The instruction USERMENU(action=REMOVE), removes the menu from the menubar. In order to make another transform or to transform another variable, it is necessary to execute the procedure a second time.

```
if %menuchoice.eq.2 {
  com a$ = 'D' + %label(y)
  set %s(a$) = y - y{1}
}

if %menuchoice.eq.3 {
  com a$ = 'G' + %label(y)
  set %s(a$) = y/y{1} - 1
}

graph(header='Time Path of ' + a$) 1 ; # %s(a$)
usermenu(action=remove)
end transform_user
```

Jazzing up the Procedure

To allow several transformations on the same variable, it is possible to use a LOOP. Place the LOOP instruction immediately preceding USERMENU and an END LOOP instruction before USERMENU(action=remove). Of course, it is necessary to have a way to break out of the loop. This is easily accomplished by adding a fourth menu choice and a fourth IF block. The key changes to the procedure are:

```
usermenu(action=define,title='Transform') $
1>>'Logarithmic' $
2>>'First Difference' $
3>>'Growth Rate'   $                ;* New continuation sign
4>>'Done'                           ;* New choice

loop                                ;* Begin loop here
usermenu

   THE ORIGINAL INSTRUCTION SET BELONGS HERE

if %menuchoice.eq.4 ; break         ;* If Done is selected, terminate the loop

graph(header='Time Path of ' + a$) 1 ; # %s(a$)
end loop
usermenu(action=remove)
end transform_user
```

Program 6.1 compiles and EXECUTES the procedure using:

```
source(noecho) c:\winrats\transform_user.src
@transform_user rgdp
```

Notice the new selection on the Menu bar entitled **Transform.** You can select the desired transformation from the menu. Note that you cannot execute any other instructions until you select ‘Done.’

9. An Interactive Procedure with Menu and USERMENU

This section will illustrate a simple program that uses both the MENU and the USERMENU instructions. The procedure allows the user to estimate a regression equation with seasonal dummy variables and a nonlinear time trend of the form:

$$y_t = a_0 + a_1 time + a_2 time^2 + a_3 time^3 + b_1 D_1 + b_2 D_2 + \ldots + b_{s-1} D_{s-1} + \varepsilon_t$$

where: D_1 through D_{s-1} are seasonal dummy variables and s is the seasonal span of the series.

The USERMENU instruction allows the user to select the dependent variable. The MENU instruction is used to create a dialog box that allows the user to enter the degree of the polynomial. After the procedure is executed, RATS creates a menu called *Trends* on the menubar. If the user clicks the mouse on this *Trends* menu, a list of three potential selections appears: *Select a Variable to Estimate, Estimate the Trend,* and *Done*. The flow of the program is as follows:

1. If the user chooses *Select a Variable to Estimate,* a list of all of the series in RATS memory is displayed. Once the selection is made, a graph of the time path of the series is displayed. Initially, the user is not allowed to select *Estimate the Variable*. The program requires the user to choose *Select a Variable to Estimate* before the selection *Estimate the Variable* is enabled.

2. Once a variable has been selected, the user is allowed to select any of the three choices from the *Trends* menu. Hence, it is possible to select an alternative variable by returning to *Select a Variable to Estimate*, estimate the selected variable with *Estimate the Variable* or to exit the procedure with *Done*.

3. If the user selects *Estimate the Variable*, another menu appears. However, this second menu is actually a dialog box with 'buttons' that allow the user to select one of four choices. The user can select whether to estimate the series without a trend, with a linear trend, with a quadratic trend or with a cubic trend. Whichever choice is made, the program displays the regression output and shows a graph of the actual and the fitted values of the series.

The program consists of two separate procedures with the names ESTIMATE and SEASONS. SEASONS creates the USERMENU called *Trends*. No parameters are passed to SEASONS and the procedure contains no OPTIONS, CHOICES or SWITCHES. As such, the first instruction defines a LOCAL INTEGER variable called *depvar*. *Depvar* is the integer value corresponding to the series selected by the user (*i.e.*, the *dependent variable* in the regression equation). The first USERMENU instruction defines the menu *Trends* and creates the three possible selections. The second USERMENU instruction illustrates the use of the ACTION = MODIFY option. Notice that choice 2 is not enabled; the user will not be able to select *Estimate the Trend* until RATS encounters a USERMENU instruction enabling choice 2.

```
procedure seasons
local integer depvar
usermenu(action=define,title='Trends')   $
1>>'Select a Variable to Estimate' $
2>>'Estimate the Trend' $
3>>'Done'

usermenu(action=modify,enable=no) 2
```

If the user chooses *Select a Variable to Estimate*, menuchoice = 1 so that the instructions in the first IF block are executed. The SELECT(series) depvar instruction presents the user a list of all series in memory. The integer value corresponding to the selected series is assigned to *depvar* and the label of *depvar* is stored in the string variable a$. The remaining instructions in the first IF-block create a graph of *depvar* and enable the choice *Estimate the Trend*.

```
loop
  usermenu
  if %menuchoice.eq.1 {
     select(series) depvar
     compute a$ = %l(depvar)
     gra(header= 'Time Path of ' + a$) 1
     # depvar
     usermenu(action=modify,enable=yes) 2
     }
```

If the user selects *Estimate the Trend*, SEASONS invokes the procedure ESTIMATE. This second procedure interacts with the user to request information concerning the type of time trend to estimate. Once ESTIMATE has completed its functions, program control returns to LOOP-ENDLOOP block. The instructions within this block are continually executed until the BREAK instruction is encountered. As long as the user does not select DONE (i.e., as long as %menuchoice does not equal 3), the user can continually select variables and estimate regression equations. Once BREAK is encountered, the program exits the loop and the USERMENU is removed from the menubar.

```
  if %menuchoice.eq.2
     @estimate depvar
  if %menuchoice.eq.3
     break
end loop
usermenu(action=remove)

end seasons
```

The procedure ESTIMATE is invoked from SEASONS. Hence, it is necessary to compile ESTIMATE before SEASONS. Since SEASONS passes the parameter *depvar* to ESTIMATE*,* it is necessary to declare *depvar* as a series. The program needs to create variables for the linear,

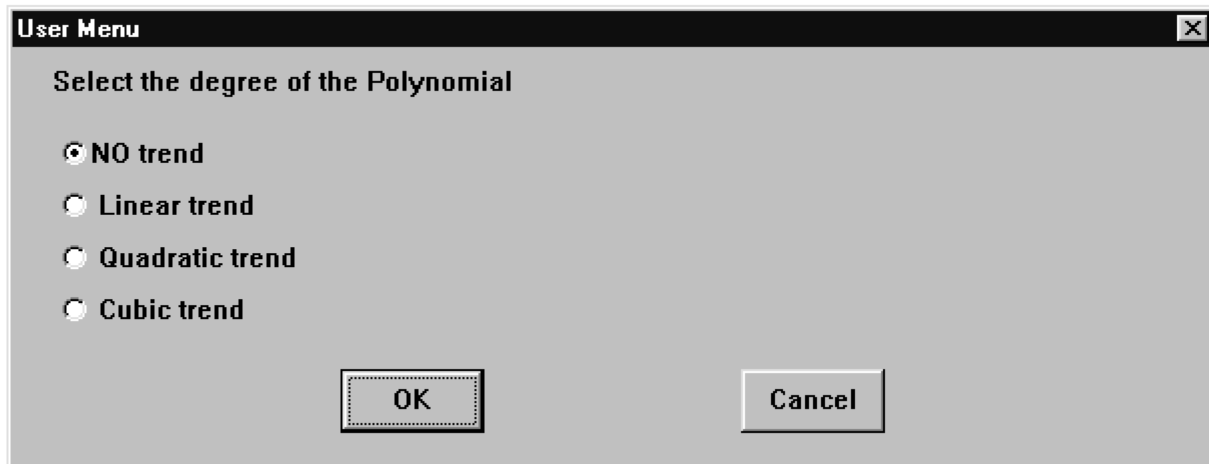
quadratic and cubic time trends and for the fitted values from the regression. The next instruction declares *time, t2, t3* and *fitted* to be local series.

```
proc estimate depvar
type series depvar
local series time t2 t3 fitted
```

The next instruction creates an integer vector (called *reglist*) that will hold the variables to be used in the regression equation. The second instruction below uses COMPUTE to include a constant in the list of regressors. The linear, quadratic and cubic time trends are created by the subsequent SET instructions.

```
dec vector[int] reglist
compute reglist=||constant||
set time = t ; set t2  = t*t ; set t3  = t*t*t
```

The MENU instruction below creates a dialog box with the title *Select the Degree of the Polynomial*. The user will be presented with four choices in a dialog box that looks like:



The selection will affect the string variable *b$* and the vector *reglist*. For example, if *NO Time Trend* is selected, b$ = 'No Time Trend' and the vector *reglist* is unaltered (hence, *reglist* contains only the constant). Instead, if *Cubic Trend* is selected, *b$* = 'Cubic Time Trend' and *reglist* contains a constant, *time, t2,* and *t3*.

```
menu 'Select the degree of the Polynomial'
 choice 'NO trend '
    com b$ = 'No Time Trend'
 choice ' Linear trend'
 {
   enter(varying) reglist ; # reglist time
   com b$ = 'Linear Time Trend'
 }
 choice ' Quadratic trend'
 {
   enter(varying) reglist ; # reglist time t2
   com b$ = 'Quadratic Time Trend'
 }
 choice ' Cubic trend '
 {
  enter(varying) reglist ; # reglist time t2 t3
  com b$ = 'Cubic Time Trend'
 }
end menu
```

Next, the linear regression is estimated using the *depvar* as the dependent variable and the variables in *reglist* as the independent variables. The PRJ instruction is used to create a series of the *fitted* values. The final three instructions obtain the label assigned to *depvar*, and create a graph of the series and the fitted values from the regression. On completion of the graph, control passes back to the LOOP—ENDLOOP block in the procedure SEASONS.

```
lin depvar ;# reglist
prj fitted

compute a$ = %l(depvar)
gra(header='Trend Estimate of ' + a$,subheader=b$,key = below,patterns, $
klabel=||'Fitted','Actual'||) 2  ; # fitted ; # depvar

end estimate
```

Jazzing up the Procedure 1

The procedure is called *SEASONS* because we are going to modify it to allow for seasonal dummy variables. If the seasonal span of the data is *s*, the procedure will estimate a model of the form:

$$y_t = a_0 + a_1 time + a_2 time^2 + a_3 time^3 + b_1 D_1 + b_2 D_2 + \dots + b_{s-1} D_{s-1} + \varepsilon_t.$$

The first step is to add the choice *Estimate the Trend and Seasonals* to the USERMENU. The procedure will set the variable *s_* = 0 if the user selects *Estimate the Trend* and will set *s_* = 1 if the user selects *Estimate the Trend and Seasonals.* As such, the modified procedure adds *s_* to

the list of local integer variables. Moreover, note that the third choice on the USERMENU is *Estimate the Trend and Seasonals* and that *Done* has been moved to the fourth position.

```
procedure seasons
local integer depvar s_
usermenu(action=define,title='Trends')   $
1>>'Select a Variable to Estimate' $
2>>'Estimate the Trend' $
3>>'Estimate the Trend and Seasonals' $
4>>'Done'
```

We do not want to ENABLE either of the *Estimate* choices unless the user has chosen *Select a Variable to Estimate*. As such, the following two lines are used to prevent the user from selecting choice 2 or choice 3.

```
usermenu(action=modify,enable=no) 2
usermenu(action=modify,enable=no) 3
```

Next, it is necessary to modify the statements that are executed with each %MENUCHOICE. Consider the program segment below. If *Select a Variable to Estimate* is selected, %MENUCHOICE=1. The two USERMENU instructions enable the *Estimate the Trend* and the *Estimate the Trend and Seasonals* choices.

If the user selects *Estimate the Trend* (%MENUCHOICE =2), *s_* = 0, and if the user selects *Estimate the Trend and Seasonals* (%MENUCHOICE =3), *s_* = 1. Both of these choices pass *depvar* and the value of *s_* to the procedure *ESTIMATE*. If *Done* is selected, %MENUCHOICE = 4 and the procedure exits the LOOP.

```
loop
usermenu
if %menuchoice.eq.1 {

    Original Instructions

    usermenu(action=modify,enable=yes) 2
    usermenu(action=modify,enable=yes) 3
    }
if %menuchoice.eq.2 {
    com s_ = 0
    @estimate depvar s_
}
if %menuchoice.eq.3 {
    com s_ = 1
    @estimate depvar s_
}
```

```
if %menuchoice.eq.4
    break
end loop
```

It is also necessary to modify ESTIMATE. Note that *s_* is included on the parameter list and is defined as an integer.[43] Moreover, *span* is a local integer that will be set equal to the seasonal span of the data (*i.e.*, *span* = 4 for quarterly data, 12 for monthly data, … . The seasonal span is determined on the CALENDAR instruction).

```
proc estimate depvar s_          ;* s_ is included on the parameter list
type integer s_                  ;* s_ is an integer
type series depvar
local integer span               ;* Span is an integer representing the seasonal span of the data
local series time t2 t3 fitted

dec vector[int] reglist
compute reglist=||constant||
```

The following IF-block is added to the procedure. If the user selected *Estimate the Trend, s_* = 0 so that this new section of the procedure is bypassed. If the user selected *Estimate the Trend and Seasonals, s_* = 1. As such, the procedure uses the INQUIRE instruction to obtain the seasonal span. Next, a seasonal dummy variable called *seasons* is created. The current value of *seasons* plus *span*-2 leads are added to the regressor list. For example, if *span* = 4, there will be three seasonal dummy variables in addition to the intercept. The remaining portions of the ESTIMATE procedure are unaltered.

```
if s_.eq.1 {
inquire(seasonal) span
seasons seasons
enter(varying) reglist ; # reglist seasons{0 to -span+2}
}
```

Jazzing up the Procedure 2

It is straightforward to modify the procedure so that it incorporates the features of TRANSFORM. The source code below indicates the necessary modifications of SEASONS. Notice that a new choice called *Transform a Variable* has been added to the USERMENU instruction. As such, *Estimate the Trend* moves to choice 3 and *Estimate the Trend and Seasonals* moves to choice 4. As such, it is necessary to change the instructions using the ENABLE = option to reflect the new position.

```
usermenu(action=define,title='Trends')   $
1>>'Transform a Variable'  $                          ;* New Choice
2>>'Select a Variable to Estimate' $
```

---

[43] Another possible way to write the procedure is to use *s_* as an OPTION in ESTIMATE.

```
3>>'Estimate the Trend' $
4>>'Estimate the Trend and Seasonals' $
5>>'Done'

usermenu(action=modify,enable=no) 3        ;* Estimate the Trend is choice 3
usermenu(action=modify,enable=no) 4        ;* Estimate the Trend and Seasonals is choice 4
```

If *Transform a Variable* is selected, %MENUCHOICE = 1 and the user is presented with a list of variables. Selection of the variable invokes the procedure TRANSFORM. Once the graph of the transformed variable is displayed, program control returns to the LOOP block of instructions. The user can make another transformation, *Select a Variable to Estimate*, or exit the procedure by selecting *Done*. Given the new positions for the menu choices, only the remaining IF %MENUCHOICE instructions need be modified. Consider:

```
loop
usermenu
if %menuchoice.eq.1 {
   select(series) depvar
   @transform depvar
}

if %menuchoice.eq. 2 {                        ;* Select a Variable to Estimate is now choice 2
        select(series) depvar
   compute a$ = %l(depvar)
   gra(header= 'Time Path of ' + a$) 1
   # depvar
   usermenu(action=modify,enable=yes) 3   ;* Enable Estimate the Trend
   usermenu(action=modify,enable=yes) 4   ;* Enable Estimate the Trend and Seasonals
   }
if %menuchoice.eq.3  {                             ;* Estimate the Trend is choice 3
   com s_ = 0
   @estimate depvar s_
}
if %menuchoice.eq.4  {                             ;* Estimate the Trend and Seasonals
   com s_ = 1
   @estimate depvar s_
}
```

```
if %menuchoice.eq.5                           ;* Done is now choice 5
    break

end loop
usermenubreak(action=remove)
end seasons
```

Cleaning up

I have used SEASONS.SRC in one portion of my undergraduate forecasting class. The experience taught me the importance of anticipating every possible choice a user can make. My recommendation for debugging the procedure is to experiment with various combinations of choices that might seem implausible *to you*. For example, SEASONS.SRC will catch a mistaken attempt to take the log of a series containing negative numbers. However, this might not deter someone from trying to take the growth rate of the same series. If you 'fool around' with the procedure, you will discover the glitch. One way to remedy the problem is to modify TRANSFORM.SRC such that portion that calculates the growth rate requires all entries to be positive.[44] To make the change, you do not have to do much more than copy the instructions from the logarithm choice and paste them into the 'Growth Rate' section:

```
choice 'Growth Rate'
{
sta(noprint,fractiles) y
  if %minimum > 0. {
    com a$ = 'G' + %label(y)
    set %s(a$) = y/y{1} - 1
}
  if %minimum.le.0
    mes(style=alert) "I CANNOT PERFORM THE DESIRED TRANSFORMATION"
}
```

A user might also select "CANCEL" when selecting a variable to transform or estimate. In no other series has been selected previously, SEASONS.SRC will attempt transform (or estimate) a constant. To clean up the program, you can use the STATUS option on the two SELECT instructions. The option STATUS=INTEGER returns the integer value 0 if the user clicks the "Cancel" button and a 1 if the user clicks the "OK" button. Thus, in the segment below, TRANSFORM.SRC is called only if the user clicks the "OK" button:

```
if %menuchoice.eq.1 {
     select(series,status=cancel) depvar
     if cancel.eq.1
       @transform depvar
}
```

---

[44] Of course, you could modify the procedure to allow all entries to be negative.

The file SEASONS.SRC includes a similar check in case the user cancels the choice *Select a Variable to Estimate*. Program 6.2 in the file CHAPTER6.PRG EXECUTES the procedure using:

```
source(noecho) c:\winrats\seasons.src
@seasons
```

Index

Akaike information criterion............................ 19
ARIMA models iv
automating model selection 182
Blanchard-Quah decomposition........... 71, 82, 86
BOXJENK .... iv, 35, 103, 104, 178, 192, 207
Box-Jenkins methodology 104, 192
BRANCH................................ iv, 133, 147, 148
CDF .... iv, 40, 41
CHOICE options... 207
Choleski decomposition.... 53, 59, 63, 69, 70, 71, 72, 75, 83, 84, 176, 180
Cointegration 249
COMPUTE .... iv, 21, 22, 30, 32, 34, 37, 38, 44, 71, 84, 98, 110, 115, 116, 134, 147, 151, 152, 168, 171, 173, 174, 176, 179, 190, 191, 230, 241
CORRELATE................................ iv, 7, 30, 135
CVMODEL.................................... iv, 75, 76, 78
DECLARE.................. iv, 66, 75, 168, 169, 170, 171, 174, 182, 190, 191, 195, 203, 215, 230
Dickey-Fuller tests........................ 136, 141, 142, 143, 144, 146, 213, 215, 222, 231
differencing.....6, 36, 38, 103, 105, 129, 144, 180
DIMENSION................................ 169, 170, 171
DISPLAY 70, 91, 152, 171, 172, 173, 198
DO 19, 99, 100, 103, 105, 106, 107, 108, 109, 110, 111, 113, 115, 116, 120, 121, 124, 125, 126, 127, 131, 133, 134, 138, 140, 141, 143, 144, 146, 147, 148, 152, 155, 156, 157, 159, 161, 162, 166, 172, 176, 177, 181, 182, 183, 185, 186, 187, 222, 227, 230
DOFOR................................ 110, 111, 113, 116, 125, 140, 141, 143, 172, 182, 185, 186, 187
Downward Bias 139
ENABLE...................................... 235, 243, 244
Engle-Granger procedure................................ 18
ENTER .... iv, 114, 182, 183, 185, 186
ENTRIES........................... iv, 12, 100, 113, 114
EQUATION................ iv, 22, 150, 151, 152, 197
EQV Instruction........................... iv, 96, 97, 109
Error-correction models............... 64, 65, 73, 149
ERRORS................................ iv, 53, 58, 62, 66, 71, 75, 83, 84, 196, 203
ESTIMATE....................... iv, 48, 52, 53, 58, 65, 67, 71, 84, 85, 105, 168, 239, 240, 243, 244
EWISE............................. iv, 176, 177, 179, 195
EXCLUDE................8, 9, 10, 116, 186, 201, 202
EXECUTE... 100, 234
FORECAST................................. iv, 60, 62, 63
FRML 21, 22, 24, 30, 32, 33, 34, 35, 36, 37, 38, 41, 42, 44, 46, 94, 155
Generalized ARCH (GARCH) 40, 41, 44, 46
Granger causality 48
GRAPH............................. iv, 5, 14, 15, 80, 101, 110, 203, 205, 216, 217, 221, 224
GROUP...................................... iv, 62, 63, 152
Heteroskedasticity... 249
IF .... iv, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 133, 144, 146, 147, 152, 208, 216, 217, 230, 235, 236, 237, 240, 244, 245
IGARCH 44, 45
impulse response function............... iv, 53, 68, 69
IMPULSES 53, 62, 66, 75, 83, 84
Inference 149, 249
INFOBOX.. iv, 145, 149
INQUIRE.......... iv, 212, 219, 220, 221, 227, 244
INTEGER option 209
interactive... 235
LABELS .... iv, 97, 109, 112, 169
lag length tests.................................. 1, 102, 105
likelihood ratio tests.............................. 51, 105
LINREG.................... iv, 6, 9, 12, 13, 14, 19, 21, 23, 30, 34, 35, 40, 44, 61, 62, 64, 65, 73, 94, 102, 108, 116, 152, 168, 172, 185, 187, 201, 202, 203, 207, 215, 221, 227
LOCAL .... iv, 212, 215, 216, 221, 223, 224, 226, 235, 239
local variables .... iv, 214, 227
LOOP...................... 20, 237, 240, 242, 243, 245
LSTAR model.................... 27, 28, 29, 30, 31, 34
MAKE.. iv, 168, 197
MAXIMIZE................ iv, 23, 32, 33, 34, 35, 36, 37, 38, 41, 43, 44, 45, 46, 120
MENU.............................. iv, 233, 236, 239, 241
MESSAGEBOX .... iv, 229, 230
Monte Carlo methods................... 119, 136, 137, 138, 141, 142, 143, 145, 146, 147, 150, 151, 152, 154, 155, 158
NLLS .... iv, 1, 21, 22, 23, 25, 30, 31, 34, 35, 120, 154, 155, 157, 161, 162
NLPAR .... iv, 23, 25, 33
NONLIN .... iv, 21, 24, 30, 32, 33, 34, 36, 37, 38, 41, 42, 44, 45, 75, 76, 78, 155, 161, 162
OPTION.................................. 47, 48, 205, 207, 208, 209, 211, 212, 215, 216, 221, 229, 244
ORDER... 130
Power 141, 143
Procedures... 204, 229
Q-statistics 7, 17, 41, 124
QUERY... iv, 229, 230
RATIO .... iv, 51, 56
Regression................................ 6, 160, 198, 200
RESTRICT......................... iv, 8, 10, 11, 33, 201
ROBUSTERRORS ..... iv, 12, 13, 14, 23, 33, 100
Schwartz Bayesian Criterion............................ 19

SCRATCH .... iv, 97
SEED 139, 142, 151
SIMULATE .... iv, 150, 151, 152
SPGRAPH 5, 110
Structural VARs 71
SUBFORMULA 35, 36, 38, 44
SUMMARIZE 8, 10
SUR .... iv, 47, 61, 62
SYSTEM 47, 48, 49, 53, 60, 64, 105
TEST 8, 9, 10, 33
Unit root tests
  Dickey-Fuller 136, 141, 142, 143, 144, 146, 213, 215, 222, 231
UNTIL .... iv, 115, 117, 118
USERMENU .... iv, 235, 236, 237, 239, 240, 242, 243, 244
Vector autoregression (VAR) 1, 35, 37, 38, 39, 47, 48, 49, 50, 51, 52, 53, 55, 57, 58, 60, 61, 62, 63, 64, 69, 70, 75, 77, 82, 83, 84, 85, 86, 87, 105, 171, 179, 181, 182, 183, 184, 185, 187, 195
Vectors 168, 249
WHILE .... iv, 115, 116, 117, 118, 119

References

Bernanke, B. (1986). “Alternative Explanations of Money-Income Correlation.” *Carnegie-Rochester Conference Series on Public Policy* 25, pp. 49 – 100.

Blanchard, O. and D. Quah (1989) "The Dynamic Effects of Aggregate Demand and Supply Disturbances." *American Economic Review* 79, pp. 655-673.

Chan, K.S. (1993). “Consistent and Limiting Distribution of the Least Squares Estimator of a Threshold Autoregressive Model.” *Annals of Statistics*, pp. 520 - 533.

Davidson, R. and J. G. MacKinnon (1993). *Estimation and Inference in Econometrics*. (Oxford University Press: Oxford).

Dickey, D. A. and W. A. Fuller (1979). "Distribution of the Estimates for Autoregressive time Series With a Unit Root." *Journal of the American Statistical Association* 74, pp. 427-431.

Efron, B. (1979). “Bootstrap Methods: Another Look at the Jackknife.” *Annals of Statistics* 7, pp. 1 – 26.

Enders, W. and C.W.J. Granger (1998). “Unit-Root Tests and Asymmetric Adjustment With an Example Using the Term Structure of Interest Rates.” *Journal of Business and Economic Statistics* 16, pp. 304-11.

Engle, R. (1982). "Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation." *Econometrica* 50, pp. 987-1007.

Engle R., D. Lilien, and R. Robbins (1987). "Estimating Time Varying Risk Premium in the Term Structure: The ARCH-M Model. *Econometrica* 55, pp. 391 - 407.

Engle, R. and C.W.J. Granger (1987). “Cointegration and Error Correction: Representation, Estimation and Testing”, *Econometrica* 55, pp. 251-76.

Sims, C. (1986). “Are Forecasting Models Usable for Policy Analysis.” *Federal Reserve Bank of Minneapolis Quarterly Review*, pp. 3 - 16.

Stock, J. (1987). “Asymptotic Properties of Least-Squares Estimators of Cointegrating Vectors.” *Econometrica* 55, pp. 1035 – 56.

Teräsvirta, T. and H. M. Anderson (1992). “Characterizing Nonlinearities in Business Cycles using Smooth Transition Autoregressive Models.” *Journal of Applied Econometrics* 7, pp. S119 - S139.

Tong, Howell (1983). *Threshold Models in Non-Linear Time Series Analysis*. (Springer-Verlag: New York).

White, H. (1980). “A Heteroskedasticity-Consistent Covariance Matrix Estimator and Direct Test for Heteroskedasticity.” *Econometrica* 48, pp. 817-838.