

RATS Programming Manual

2nd Edition

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

and

Thomas Doan
Estima
Evanston, IL 60201
tomd@estima.com

Draft
February 26, 2014

Copyright © 2014 by Walter Enders and Thomas Doan

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 holders.

Contents

Preface	v
1 Introduction	1
1.1 What Are Your Options?	2
1.2 Which Should You Use?	3
1.3 Three Words of Advice	4
1.4 General Stylistic Tips.	5
1.5 About This E-Book	7
2 Regression and ARIMA Models	9
2.1 The Data Set.	9
2.2 Linear Regression and Hypothesis Testing	12
2.2.1 Examples using RESTRICT	18
2.3 The LINREG Options.	19
2.4 Using LINREG and Related Instructions	20
2.5 ARMA(p,q) Models	26
2.6 Estimation of an ARMA(p,q) process with RATS.	27
2.6.1 Identification	27
2.6.2 Estimation	28
2.6.3 Diagnostic Checking	29
2.7 An Example of the Price of Finished Goods	29
2.8 Automating the Process.	33
2.8.1 Introduction to DO Loops	34
2.9 An Example with Seasonality	37
2.10 Forecasts and Diagnostic Checks.	41
2.11 Examining the Forecast Errors	44
2.12 Coefficient Stability.	49
2.13 Tips and Tricks	52
2.13.1 Preparing a graph for publication	52

2.13.2	Preparing a table for publication	52
2.1	Introduction to basic instructions	53
2.2	Engle-Granger test with lag length selection	55
2.3	Estimation and diagnostics on ARMA models	56
2.4	Automated Box-Jenkins model selection	57
2.5	Seasonal Box-Jenkins Model	58
2.6	Out-of-sample forecasts with ARIMA model	59
2.7	Comparison of Forecasts	60
2.8	Stability Analysis	61
3	Non-linear Least Squares	63
3.1	Nonlinear Least Squares	64
3.2	Using NLLS	67
3.3	Restrictions: Testing and Imposing.	72
3.4	Convergence and Convergence Criteria.	75
3.5	ESTAR and LSTAR Models	77
3.6	Estimating a STAR Model with NLLS	79
3.7	Smooth Transition Regression	87
3.8	An LSTAR Model for Inflation	91
3.9	Functions with Recursive Definitions.	98
3.10	Tips and Tricks	101
3.10.1	Understanding Computer Arithmetic	101
3.10.2	The instruction NLPAR	102
3.10.3	The instruction SEED	105
3.1	Simple nonlinear regressions	106
3.2	Sample STAR Transition Functions	108
3.3	STAR Model with Generated Data	109
3.4	Smooth Transition Break	111
3.5	LSTAR Model for Inflation	113
3.6	Bilinear Model	115

4	Maximum Likelihood Estimation	116
4.1	The MAXIMIZE instruction	117
4.2	ARCH and GARCH Models	122
4.3	Using FRMLs from Linear Equations	127
4.4	Tips and Tricks	133
4.4.1	The Simplex Algorithm	133
4.4.2	BFGS and Hill-Climbing Methods	135
4.4.3	The CDF instruction and Standard Distribution Functions	137
4.1	Likelihood maximization	139
4.2	ARCH Model, Estimated with MAXIMIZE	140
4.3	GARCH Model with Flexible Mean Model	141
5	Standard Programming Structures	143
5.1	Interpreters and Compilers	143
5.2	DO Loops	146
5.3	IF and ELSE Blocks	151
5.4	WHILE and UNTIL Loops	154
5.5	Estimating a Threshold Autoregression	159
5.5.1	Estimating the Threshold	161
5.5.2	Improving the Program	164
5.6	Tips and Tricks	169
5.1	Illustration of DO loop	172
5.2	Illustration of IF/ELSE	172
5.3	Illustration of WHILE and UNTIL	173
5.4	Threshold Autoregression, Brute Force	174
5.5	Threshold Autoregression, More Flexible Coding	176
6	SERIES and Dates	178
6.1	SERIES and the workspace	178
6.2	SERIES and their integer handles	181
6.3	Series Names and Series Labels	183
6.4	Dates as Integers.	184

6.5	Tips and Tricks	187
6.1	Series and Workspace Length	188
6.2	Series handles and DOFOR	189
6.3	Date calculations and functions	191
A	Probability Distributions	192
A.1	Univariate Normal	192
A.2	Univariate Student (t)	193
A.3	Chi-Squared Distribution	194
A.4	Gamma Distribution	195
A.5	Multivariate Normal	196
B	Quasi-Maximum Likelihood Estimations (QMLE)	197
C	Delta method	200
D	Central Limit Theorems with Dependent Data	201
	Bibliography	204
	Index	205

Preface

This is an update of the *RATS Programming Manual* written in 2003 by Enders. That was, and this is, a free “e-book” designed to help you learn better how to use the more advanced features of RATS. Much has changed with RATS over the intervening ten years. It has new data types, more flexible graphics and report-building capabilities, many new and improved procedures, countless new example files.

And the practice of econometrics has changed as well. It’s much more common (and almost expected) to use more “computational intensive” methods, such as simulations and bootstrapping, sample stability analysis, etc. These techniques will often require use of programming beyond the “pre-packaged” instructions and procedures, and that’s what this e-book is here to explain.

The econometrics used in the illustrations is drawn from Enders (2010), but there is no direct connection between the content of this e-book and the textbook. If you have questions about the underlying statistical methods, that book would be your best reference.

Because the goal is to help you understand how to put together usable programs, we’ve included the full text of each example in this book. And the running examples are also available as separate files.

Introduction

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 the program and how to make the standard data transformations. If you are interested in learning about any particular instruction, you can use RATS *Help Menu* or refer to the *Reference Manual* and *User's Guide*.

The emphasis here is on what we 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. To do state-of-the-art research, it is often necessary to go “off the menu.” 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. With the power of modern computers, it’s increasingly expected that researchers justify their choices of things like lag lengths, test robustness through bootstrapping, check their model for sample breaks, etc. While some of these have been standardized and are incorporated into existing instructions and procedures, many have not and in some cases *can* not because they are too specific to an application. Sometimes, the “programming” required is as simple as “throwing a loop” around the key instruction. But often it will require more than that.

Of course, it will be impossible to illustrate even a small portion of the vast number of potential programs you can write. Our intent is to give you the tools to write your own programs. Towards that end, we will discuss a number of the key instructions and options in the programming language and illustrate their use in some straightforward programs. We 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 us, it is too difficult to learn econometrics and the programming tools at the same time. As such, we will try not to introduce any sophisticated econometric methods or techniques. Moreover, all of the examples will use a single data set. All examples are compatible with RATS version 8.0 or later.

1.1 What Are Your Options?

If you need some calculation which can't be done by just reading the data, doing some transformations, running basic statistic instructions (such as least squares regressions) and reporting the results, then you have three general “platforms” in which to do your work:

General Purpose Programming Language

Forty years ago, almost all statistical programming was done in Fortran, which is still heavily used in science and engineering. (If you ever see an astronomer or particle physicist scroll through a program, it will probably be Fortran). Some economists still use Fortran, or C++, and most of the high level statistical packages are written in them. (RATS uses C++). These are “compiled” languages, which means that they look at the entire program, optimize it and produce an executable program for a specific machine. This makes them harder to write, even harder to debug (you have to make a change and regenerate the executable in order to test something), but as a result of being compiled are very fast. The great disadvantage is that they, by and large, manipulate individual numbers and pairs of numbers, rather than matrices. While you can obtain and use packages of subroutines that do specific matrix calculations, you don't use matrices in “formula translation” form. (The phrase FORMula TRANslation is the source of the Fortran name). Since matrix calculations form the backbone of most work on econometrics, this isn't convenient.

Math Packages

The two most prominent of these in econometrics are Matlab[®] and Gauss[™], but there are others. These are primarily designed as matrix programming languages. Since matrix manipulations are so important in statistics in general and econometrics in particular, this makes them much simpler to use than general purpose languages.

These are designed to do the relatively small number of their built-in functions very well. Some “hot spot” calculations are specially optimized so they are faster than the same thing done using the compiled language.

This does come at a cost. Particularly in time series work, a “data as matrix” view doesn't work well, because the data matrix for one operation with one set of lags is different from the one needed with a different set. If you look at a program for, for instance, the estimation of a Vector Autoregression, very little of it will be the matrix calculations like $B = (X'X)^{-1} * (X'Y)$ —most will be moving information around to create the X and Y matrices.

High Level Statistical Packages

RATS is an example of this. These have a collection of “built-in” instructions for specific statistical calculations. All of them, for instance, will have something

similar to the RATS **LINREG** instruction which takes as input a dependent variable and collection of explanatory variables and performs a multiple linear regression, producing output, which typically includes summary statistics on residuals and fit, standard errors and significance levels on the coefficients. While the calculations for a simple **LINREG** are straightforward, what RATS and similar programs are doing for you simplifies considerably what would be required to do the same in a math package:

1. You can adjust the model by just changing the list of variables, instead of having to re-arrange the input matrices.
2. You don't have to figure out yourself how to display the output in a usable form. That can be a considerable amount of work if you want your output to make sense by itself, without having to refer back to the program.
3. While most of the summary statistics are simple matrix functions of the data and residuals, some common ones (such as the Durbin-Watson and Ljung-Box Q) aren't quite so simple, and thus are often omitted by people using math packages.

In addition to these, RATS also takes care of adjusting the sample range to allow for lags of the explanatory variables and for any other type of missing values. RATS also allows you to use lagged values without actually making a shifted copy of the data, which is a necessary step both in using a math package or a statistical package which isn't designed primarily for time series work.

Which high-level commands are built into a particular statistical package depends upon the intended market and, to a certain extent, the philosophy of the developers. The TS in RATS stands for Time Series, so RATS makes it easy to work with lags, includes instructions for handling vector autoregressions, GARCH models, computing forecasts and impulse responses, estimating ARIMA models, state-space models and spectral methods. While it also handles cross-sectional techniques such as probit and tobit models, and has quite a bit of support for working with panel data, you are probably using RATS because you're working with some type of dynamic model.

Most high-level statistical packages include some level of programmability, with looping, conditional execution, matrix manipulations, often some type of "procedure" language. RATS has these and also user-definable menus and dialogs for very sophisticated types of programs. CATS is the best example of this—it is entirely written in the RATS programming language. We will cover all of these topics as part of this book.

1.2 Which Should You Use?

Your goal in doing empirical work should be to get the work done correctly while requiring as little *human* time as possible. Thirty years ago, computing time was often quite expensive. Estimating a single probit model on 10000

data points might cost \$100, so it was much more important to choose the right platform and check and double-check what you were doing before you even submitted a job for computation. Computer time was more valuable than human time—a good research assistant would keep the computer center bills down. With modern computers, computing time for almost anything you might want to do is almost costless. No matter how fast computers get, there will always be someone figuring out some type of analysis which will take several days to run on the fastest computers available, but that's not typical. Even complicated simulations with a large number of replications can now be done in under an hour.

As a general rule, the best way to achieve this goal is to use a programmable high-level statistical package. By doing that, you start out with what will be in most cases large pieces of what you need to do already written, already tested, already debugged. All software (including RATS) has some bugs in it, but bugs in mass-marketed software generally get discovered fairly quickly because of the sheer number of users. The software is also “vetted” by doing comparisons against calculations done using other software. By contrast, a function written in a math package for use for one paper is never actually checked by anyone, other than the writer. (And, unfortunately, sometimes not even by the writer, as we have discovered in doing paper replications).

What's important, though, is that you make use, as much as possible, of the features available in RATS—the existing instructions, procedures and functions. If you don't you're throwing away the single greatest advantage of the statistical package.

1.3 Three Words of Advice

Get to Know Your Data!

The best way to waste time is to plunge ahead with a complicated piece of analysis, only to discover that your data aren't right only when the results make no sense. (Even worse, of course, is to do the work, and write an entire paper only to have a referee or thesis advisor tell you that the results make no sense). In the interest of brevity, most published papers omit graphs of the data, tables of basic statistics, simple regression models to help understand the behavior of the data, but even if they don't make it into your final product, they should be a vital part of your analysis.

Don't Reinvent the Wheel!

Use the built-in instructions wherever possible—they're the single greatest advantage of using a high-level statistical package. Use the RATS procedure library. Understand how to use the procedures which already exist. We'll discuss how to write your own procedures, and that same information can be used to modify existing ones where necessary, but before you do either, see if the

existing ones can be used as is. RATS comes with over 1000 textbook and paper replication examples. See if something similar to what you want has already been done and, if so, use it as the base for your work.

Realize That Not All Models Work!

You may need to be flexible. The RATS **LINREG** instruction will take just about any set of data that you throw at it—it will handle collinear data by (in effect) dropping the redundant variables—so as long as you have at least one usable observation, it will give you *some* result. However with non-linear estimation instructions like **GARCH**, **MAXIMIZE**, **DLM**, there's no guarantee that they can handle a model with a given set of data. Some models have multiple modes, some have boundary problems, some have parameter scaling issues. Many have structural changes and so don't fit properly over a whole sample. If you read a published paper, you're generally looking at the models which worked, not the ones which didn't. And often there are many in the latter category. So be prepared to have to drop a country or industry, or to change up the data range if you need to.

1.4 General Stylistic Tips

Commenting

Your first goal is always to get the calculations correct. However, if you have a program (or part of one) that you expect that you might need again, it's a good idea to add comments. Don't overdo it—the following would be a waste of the time to do the comment and also is distracting:

```
*
* Take first difference of log M2
*
set dlm2    = log(m2) - log(m2{1})
```

Note the difference between this and

```
*
* Use first difference of log M2 as in Smith and Jones(2010)
*
set dlm2    = log(m2) - log(m2{1})
```

where the comment will help you in writing up the results later. And if you have a part of your program which you're not sure is correct, commenting it can often help you spot errors. (If you can't explain why you're doing something, that might be a good sign that you're doing it wrong).

Prettifying

The word *prettify* is used in programming circles to describe making the program easier to read by changing spacing, line lengths, etc. It doesn't change

how it works, just how it reads. Well-chosen spaces and line breaks can make it easier to read a program, and will go a long way towards helping you get your calculations correct. Even minor changes can help you do that. Compare

```
set dlr GDP = log(rgdp)-log(rgdp{1})
set dlm2 = log(m2)-log(m2{1})
set drs = tb3mo-tb3mo{1}
set drl = tb1yr-tb1yr{1}
set dlp = log(deflator)-log(deflator{1})
set dlppi = log(ppi)-log(ppi{1})
```

with

```
set dlr GDP = log(rgdp) - log(rgdp{1})
set dlm2 = log(m2) - log(m2{1})
set drs = tb3mo - tb3mo{1}
set drl = tb1yr - tb1yr{1}
set dlp = log(deflator) - log(deflator{1})
set dlppi = log(ppi) - log(ppi{1})
```

The only difference is a handful of spaces in each line, but it's much clearer in the second case that these are parallel transformations, and it would be much easier to spot a typo in any of those lines.

At a minimum, you should get into the habit of indenting loops and the like. This makes it much easier to follow the flow of the program, and also makes it easier to skip more easily from one part of the calculation to the next.

Two operations on the *Edit* menu can be helpful with this. *Indent Lines* adds (one level) of indentation at the left; *Unindent Lines* removes one level. The number of spaces per level is set in the preferences in the *Editor* tab. All the programs in this book are done with 3 space indenting, which seems to work well for the way that RATS is structured. In the following, if you select the five lines in the body of the loop and do *Edit-Indent*

```
do i = 1,8
  linreg(noprint) dresids 1962:2 *
  # resids{1} dresids{1 to i}
  com aic = %nobs*log(%rss) + 2*(%nreg)
  com sbc = %nobs*log(%rss) + (%nreg)*log(%nobs)
  dis "Lags: " i "T-stat" %tstats(1) aic sbc
end do i
```

you'll get

```

do i = 1,8
  linreg(noprint) dresids 1962:2 *
  # resids{1} dresids{1 to i}
  com aic = %nobs*log(%rss) + 2*(%nreg)
  com sbc = %nobs*log(%rss) + (%nreg)*log(%nobs)
  dis "Lags: " i "T-stat" %tstats(1) aic sbc
end do i

```

1.5 About This E-Book

Examples

The full running examples are included both in the text and are distributed as separate files with the e-book. The names for these files are `RPMn.m.RPF`, where `RPM` is “RATS Programming Manual”, `n` is the chapter number and `m` the example number. We would suggest that you use the separate example files rather than trying to copy and paste whole programs out of the PDF—if you do the latter, you can often end up extra information from the page layout.

Typefaces

To help you understand how RATS works and in particular, what is happening in the sample programs, we will use several conventions for typefaces.

Elements of RATS programs (instructions and options) are in `Courier` font. Within text, they will be in upper case to stand out, with instruction names in bold face and options or variable names in regular face:

We want to suppress the usual **LINREG** output from the regressions with different lags, so we'll use `NOPRINT`. Then we'll use **DISPLAY** to show the test statistic and the two criteria.

However, stand-alone examples of code will be in lower case for readability:

```

do q=0,3
  do p=0,3
    boxjenk(constant,ar=p,ma=q) y
  end do p
end do q

```

Standard procedures which are distributed with RATS will be shown (in bold upper case `Courier`) with the standard `@` prefix, the way that you would use them in practice: **@DFUNIT**, **@REGCRITS**. Since almost all procedures are on a file named “procedure name”.src (that is, `dfunit.src` for **@DFUNIT**), we won't talk about where you might find the code for a procedure unless it's on a file other than the expected one.

Output taken straight out of a RATS output window will be in smaller fixed font (to keep the information aligned) with a box around it:

Null Hypothesis : The Following Coefficients Are Zero	
DRS	Lag(s) 5 to 7
F(3,196)=	9.44427 with Significance Level 0.00000739

Wizards

We won't talk much about the use of RATS "wizards". While some of these remain useful even to very experienced RATS programmers (the *Data (Other Formats)* and *Standard Functions* wizards in particular), they're generally designed to help with one-off calculations for less-experienced users, and not for calculations with variables for ranges, and looping calculations that we'll be doing here.

Tips and Tricks

If there is a subject which you might find interesting, but which would interrupt the flow of a chapter, we will shift that into a *Tips and Tricks* section at the end of the chapter.

Exercises

The point of this book is to help you learn better how to accomplish more advanced tasks using RATS. To that end, we will occasionally insert "exercises", which are ask you to think about how to do a slightly different example or how to recode what we've already presented.

Regression and ARIMA Models

This chapter begins with a quick overview of some of the basic RATS instructions used in estimating linear regression and ARMA models. This book is definitely not an econometrics text; instead, the aim is to refresh your memory and to introduce you to some basic RATS tools. Towards that end, a number of key RATS instructions are illustrated in some straightforward programs.

2.1 The Data Set

The file labeled `QUARTERLY(2012).XLS` contains quarterly values for the 3-month and 1-year treasury bill rates, real GDP, potential real GDP, the GDP deflator, the seasonally adjusted money supply (M2), the producer price index of finished goods (PPI), and currency in circulation for the 1960:1 – 2012:4 period. The data were 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:

DATE	Tb3mo	Tb1yr	RGDP	Potent	Deflator	M2	PPI	Curr
1960Q1	3.87	4.57	2845.3	2824.2	18.521	298.7	33.2	31.8
1960Q2	2.99	3.87	2832.0	2851.2	18.579	301.1	33.4	31.9
1960Q3	2.36	3.07	2836.6	2878.7	18.648	306.5	33.4	32.2
1960Q4	2.31	2.99	2800.2	2906.7	18.700	310.9	33.7	32.6
1961Q1	2.35	2.87	2816.9	2934.8	18.743	316.3	33.6	32.1
1961Q2	2.30	2.94	2869.6	2962.9	18.785	322.1	33.3	32.1
1961Q3	2.30	3.01	2915.9	2991.3	18.843	327.6	33.3	32.7
1961Q4	2.46	3.10	2975.3	3019.9	18.908	333.3	33.4	33.4

If you open up Example **2.1** (file `RPM2_1.RPF`), you'll see the following lines, which read in the entire data set:

```
cal(q) 1960:1
all 2012:4
open data quarterly(2012).xls
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 and limit the output to only two decimal places, your output should be:

```
table(picture="*.##")
```

Series	Obs	Mean	Std Error	Minimum	Maximum
TB3MO	212	5.03	2.99	0.01	15.05
TB1YR	212	5.58	3.18	0.11	16.32
RGDP	212	7664.75	3390.65	2800.20	13665.40
POTENT	212	7764.87	3511.54	2824.20	14505.40
DEFLATOR	212	61.53	31.59	18.52	116.09
M2	212	3136.84	2648.84	298.70	10317.70
PPI	212	99.97	49.13	33.20	196.20
CURR	212	327.91	309.02	31.83	1147.62

Many of the examples presented will use the growth rates of M2 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 or the PPI). You can create these six variables using:

```
set dlrgdp = log(rgdp) - log(rgdp{1})
set dlm2   = log(m2) - log(m2{1})
set drs    = tb3mo - tb3mo{1}
set drl    = tb1yr - tb1yr{1}
set dlp    = log(deflator) - log(deflator{1})
set dlppi  = log(ppi) - log(ppi{1})
```

Notice that we've chosen to notate the change in a variable as a prefix of *d*, the growth rate of a variable by *dl*, and the suffixes *s* and *l* refer to the short-term and long-term interest rates, and the logarithmic change in price (called *dlp*) is the quarterly inflation rate.

We can create graphs of the series (Figure 2.1) using:¹

```
spgraph(footer="Graphs of the Series",hfields=2,vfields=2)
  graph(header="Panel 1: The Interest Rates",key=below,nokbox) 2
  # tb3mo
  # tb1yr
  graph(header="Panel 2: Real and Potential GDP",key=upleft) 2
  # rgdp
  # potent
  graph(header="Panel 3: Time path of money growth",noaxis) 1
  # dlm2
  graph(header="Panel 4: Time path of Inflation",noaxis) 1
  # dlp
spgraph(done)
```

Recall that the typical syntax of the **GRAPH** instruction is:

```
GRAPH( options ) number
# series start end
```

¹The growth rate of the PPI and CURR are not shown here—both are considered in more detail later in the chapter.

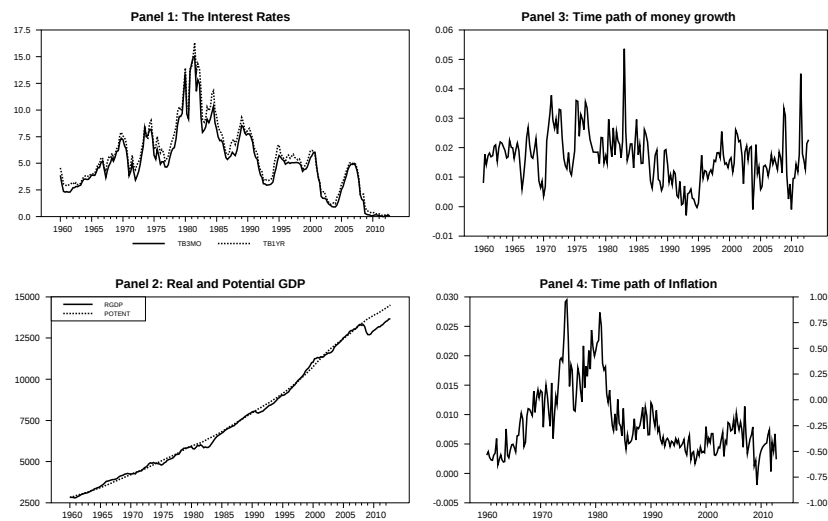


Figure 2.1: Graphs of the series

number The number of series to graph. The names of the series are listed on the supplementary cards (one card for each series).

series The name of the series to graph

start end Range to plot. If omitted, RATS uses the current sample range.

The graphs shown in Figure 2.1 illustrate only a few of the options available in RATS. The commonly used options are (brackets[] indicate default choice):

HEADER=*header string (centered above graph)*

FOOTER=*footer string (left-justified below graph)*

KEY=*the location of the key*

Some of the choices you can use are [NONE], UPLEFT, LORIGHT, ABOVE, BELOW, RIGHT. Some (such as UPLEFT and LORIGHT) are inside the graph box, others (such as ABOVE and RIGHT) are outside.

STYLE=*graph style*

Some of the choices include: [LINE], POLYGON, BAR, STACKEDBAR.

DATES/NODATES

RATS will label the horizontal axis with dates (rather than entry numbers) unless the NODATES option is specified.

The program also 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 there are two fields horizontally (HFIELD=2) and two vertically (VFIELD=2). The option **FOOTER** produces “Graphs of the Series” as the footer for the full page. The headers on the four **GRAPH** instructions produce the headers on the individual panels. Nothing is actually shown until the **SPGRAPH (DONE)**.

2.2 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:

$$drs_t = \alpha_0 + \sum_{i=1}^7 \alpha_i drs_{t-i} + \varepsilon_t \quad (2.1)$$

The next two lines of the program (this is still `RPM2_1.RPF`) 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 1962:01 To 2012:04				
Usable Observations	204			
Degrees of Freedom	196			
Centered R ²	0.2841953			
R-Bar ²	0.2586309			
Uncentered R ²	0.2843637			
Mean of Dependent Variable	-0.011617647			
Std Error of Dependent Variable	0.759163288			
Standard Error of Estimate	0.653660810			
Sum of Squared Residuals	83.745401006			
Regression F(7,196)	11.1168			
Significance Level of F	0.0000000			
Log Likelihood	-198.6489			
Durbin-Watson Statistic	1.9709			
Variable	Coeff	Std Error	T-Stat	Signif

1. Constant	-0.011903358	0.045799634	-0.25990	0.79521316
2. DRS{1}	0.390010248	0.069644459	5.60002	0.00000007
3. DRS{2}	-0.380186642	0.074718282	-5.08827	0.00000084
4. DRS{3}	0.406843358	0.078304236	5.19567	0.00000051
5. DRS{4}	-0.159123423	0.082740231	-1.92317	0.05590809
6. DRS{5}	0.193334248	0.078290297	2.46945	0.01438724
7. DRS{6}	-0.089946745	0.074692035	-1.20423	0.22995107
8. DRS{7}	-0.220768119	0.069358921	-3.18298	0.00169542

Almost every piece of information in this output can be retrieved for future calculations—in the descriptions below the variable name (if it exists) is in bold. These are all saved to full precision, not just to the number of decimal places shown in the output. You need to be careful since these are replaced every time you estimate a regression, and some may be recomputed by other instructions as well. `%NOBS`, for instance, is replaced by almost any statistical instruction.

Usable Observations (**`%NOBS`**)

This doesn't count observations lost to differencing and lags. If there were missing values within the sample it wouldn't count those either.

Degrees of Freedom (**%NDF**)

Number of observations less number of (effective) regressors. If you happened to run a regression with collinear regressors (too many dummies, for instance), the number of effective regressors might be less than the number you listed.

Centered R^2 (**%RSQUARED**)

The standard regression R^2 .

 R^2 -Bar (**%RBARSQ**)

R^2 corrected for degrees of freedom.

Uncentered R^2

R^2 comparing sum of squared residuals to sum of squares of the dependent variable, without subtracting the mean out of the latter.

Mean of Dependent Variable (**%MEAN**)

The mean of the dependent variable computed only for the data points used in the regression. Thus it will be different from what you would get if you just did a **TABLE** or **STATISTICS** instruction on the dependent variable.

Std Error of Dependent Variable (**%VARIANCE**)

This is the standard error of the dependent variable computed only for the data points used in the regression. **%VARIANCE** is its square.

Standard Error of Estimate (**%SEESQ**)

The standard degrees-of-freedom corrected estimator for the regression standard error. Its square (that is the variance estimated) is in the **%SEESQ** variable.

Sum of Squared Residuals (**%RSS**)Regression F (**%FSTAT**)

This is the F test for the hypothesis that all coefficients in the regression (other than the constant) are zero. Here, the sample value of F for the joint test $\alpha_1 = \alpha_2 = \alpha_3 = \dots = \alpha_7 = 0$ is 11.1168. Its output also shows the numerator and denominator degrees of freedom of the test.

Significance Level of F (**%FSIGNIF**)

This is the significance level of the regression F , which here is highly significant.

Log Likelihood (**%LOGL**)

This is the log likelihood assuming Normal residuals. Note that RATS includes all constants of integration in any log likelihood calculations.

Durbin-Watson Statistic (**%DURBIN**)

The Durbin-Watson test for first-order serial correlation in the residuals. This is computed and displayed even though the standard small sample limits don't apply to a regression like this with lagged dependent variables—it's

mainly used as an informal indicator of serial correlation if it differs substantially from the theoretical value of 2.0 for serially uncorrelated residuals.

In the regressor table at the bottom, we have the coefficient estimate (Coeff), the standard error of estimated 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). The fetchable information here are saved in `VECTORS`, each of which would have 8 elements in this case. The coefficients are in `%BETA`, the standard errors in `%STDERRS` and the t -statistics in `%TSTATS`. (The significance levels aren't saved). Thus `%BETA(2)` is the coefficient on the first lag of `DRS` (roughly .3900), `%STDERRS(5)` is the standard error on the estimate of `DRS{4}` (.0827), and `(%TSTATS(8))` is the t -statistic on `DRS{8}` (-3.183).

There are a several other variables defined by `LINREG` which don't show directly on the output:

<code>%NREG</code>	Number of regressors
<code>%NMISS</code>	Number of skipped data points (between the start and end)
<code>%TRSQUARED</code>	Number of observations times the R^2 . Some LM tests use this as the test statistic.
<code>%TRSQ</code>	Number of observations times the <i>uncentered</i> R^2 . Also sometimes used in LM tests.
<code>%SIGMASQ</code>	Maximum likelihood estimate (that is, not corrected for degrees of freedom) of the residual variance.
<code>%XX</code>	The $(X'X)^{-1}$ matrix, or (in some cases) the estimated covariance matrix of the coefficients. This is a $k \times k$ SYMMETRIC matrix where k is the number of regressors. <code>%XX(i, j)</code> is its i, j element.

For a time series regression, it is always important to determine whether there is any serial correlation in the regression residuals. The `CORRELATE` instruction calculates the autocorrelations (and the partial autocorrelations) of a specified series. The syntax is:

```
CORRELATE( options ) series start end corrs
```

where

<code>series</code>	The series for which you want to compute correlations.
<code>start end</code>	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=*number of autocorrelations to compute*

The default is the integer value of $T/4$

RESULTS=*series used to save the correlations*

PARTIAL=*series for the partial autocorrelations*

If you omit this option, the PACF will not be calculated.

QSTATS

Produces the Ljung-Box Q-statistics

SPAN=*interval width for Q-statistics*

Use with **QSTATS** to set the width of the intervals. For example, **SPAN=4** produces $Q(4)$, $Q(8)$, $Q(12)$, and so forth.

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

```
corr(number=24,partial=partial,qstats,span=4,pic="###.###") resids
```

The options also include a degrees of freedom correction. Here, you could include the option **DFC=7** since the residuals are generated from a model with seven autoregressive coefficients.

Correlations of Series RESIDS									
Quarterly Data From 1962:01 To 2012:04									
Autocorrelations									
1	2	3	4	5	6	7	8	9	10
0.015	0.002	-0.019	-0.021	-0.043	0.044	-0.069	0.105	-0.096	0.061
11	12	13	14	15	16	17	18	19	20
-0.137	0.022	-0.067	0.006	-0.131	-0.047	-0.092	0.076	-0.025	0.001
21	22	23	24						
0.041	0.028	-0.045	0.049						
Partial Autocorrelations									
1	2	3	4	5	6	7	8	9	10
0.015	0.002	-0.019	-0.020	-0.043	0.045	-0.071	0.107	-0.103	0.067
11	12	13	14	15	16	17	18	19	20
-0.145	0.033	-0.069	-0.004	-0.123	-0.077	-0.063	0.032	0.004	-0.058
21	22	23	24						
0.083	-0.035	0.005	-0.002						
Ljung-Box Q-Statistics									
Lags	Statistic	Signif	Lvl						
4	0.209	0.994898							
8	4.390	0.820339							
12	11.371	0.497450							
16	16.661	0.407869							
20	20.035	0.455719							
24	21.637	0.600931							

All of the autocorrelation and partial autocorrelations are small and the Ljung-Box Q -statistics do not indicate the values are statistically significant. Other diagnostic checks include plotting the residuals using (for instance)

```
graph 1
# resids
```

A concern is that the model is over-parameterized since it contains a total of eight coefficients. While the t -statistics allow you to determine the significance levels of individual coefficients, 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. The following does a joint test on the final three lags:

```
exclude
# drs{5 to 7}
```

Null Hypothesis : The Following Coefficients Are Zero	
DRS	Lag(s) 5 to 7
F(3,196)=	9.44427 with Significance Level 0.00000739

This can be rejected at conventional significance levels.

With **EXCLUDE** (and similar instructions) you can suppress the output with the **NOPRINT** option, or you can “improve” the output using the **TITLE** option to give a clearer description. Whether or not use print the output or not, they define the variables

%CDSTAT The test statistic

%SIGNIF The significance level

%NDFTEST The (numerator) degrees of freedom. (The denominator degrees of freedom on a F will be the **%NDF** from the previous regression.)

SUMMARIZE has the same syntax as **EXCLUDE** but is used to test the null hypothesis that the *sum* of the list coefficients is equal to zero. In the following example, the value of t for the null hypothesis $\alpha_5 + \alpha_6 + \alpha_7 = 0$ is -1.11460 . As such, we do not reject the null hypothesis that the sum is zero.

```
summarize
# drs{5 to 7}
```

Summary of Linear Combination of Coefficients			
DRS	Lag(s) 5 to 7		
Value	-0.1173806	t-Statistic	-1.11460
Standard Error	0.1053116	Signif Level	0.2663855

In addition to %CDSTAT and %SIGNIF, **SUMMARIZE** defines %SUMLC and %VARLC as the sum of the coefficients and the estimated variance of it.

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 $\alpha_i, \alpha_j, \dots$ and α_k equal the values $r_i, r_j, \dots$ and r_k , respectively. Formally, suppose you want to test the restrictions

$$\alpha_i = r_i, \alpha_j = r_j, \dots, \text{ and } \alpha_k = r_k$$

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(3,196)=	15.77411 with Significance Level 0.00000000
-----------	---

RATS displays the F -value and the significance level of the joint test. 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 (i.e., $\alpha_0 = 0$) and that $\alpha_1 = 0.4, \alpha_2 = -0.1, \alpha_3 = 0.4$, use:

```
test
# 1 2 3 4
# 0. 0.4 -0.1 0.4
```

F(4,196)=	4.90219 with Significance Level 0.00086693
-----------	--

RESTRICT is the most powerful of the hypothesis testing instructions. It 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:

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

where the α_i are the coefficients of the estimated model (i.e., each coefficient is referred to by its assigned number), the 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 requires 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 .

2.2.1 Examples using RESTRICT

1. To test the restriction that the constant equals zero (which *could* be done with **EXCLUDE** or **TEST**) 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 $1.0 \times \alpha_0 = 0$.

2. To test the restriction that $\alpha_1 = \alpha_2$, we rearrange that to $\alpha_1 - \alpha_2 = 0$ and 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 $1.0 \times \alpha_1 - 1.0 \times \alpha_2 = 0$.

3. If you reexamine the regression output, it seems as if $\alpha_1 + \alpha_2 = 0$. We'll also include several other restrictions which aren't quite as clear: $\alpha_3 + \alpha_4 = 0$ and $\alpha_4 + \alpha_5 = 0$. To test the combination of these three restrictions use:

```
restrict(create) 3 resids
# 2 3
# 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. Whenever **CREATE** is used, you can save the new regression residuals simply by providing RATS with the name of the series in which to store the residuals—here **RESIDS**. (%**RESIDS** is also redefined). The test is shown *above* the new regression output.

F(3,196)= 3.74590 with Significance Level 0.01197151				
Linear Model - Estimation by Restricted Regression				
Dependent Variable DRS				
Quarterly Data From 1962:01 To 2012:04				
Usable Observations	204			
Degrees of Freedom	199			
Mean of Dependent Variable	-0.011617647			
Std Error of Dependent Variable	0.759163288			
Standard Error of Estimate	0.667052922			
Sum of Squared Residuals	88.546960661			
Durbin-Watson Statistic	1.9307			
Variable	Coeff	Std Error	T-Stat	Signif

1. Constant	-0.013441955	0.046725758	-0.28768	0.77389289
2. DRS{1}	0.378316399	0.060581719	6.24473	0.00000000
3. DRS{2}	-0.378316399	0.060581719	-6.24473	0.00000000
4. DRS{3}	0.266709604	0.065399412	4.07817	0.00006562
5. DRS{4}	-0.266709604	0.065399412	-4.07817	0.00006562
6. DRS{5}	0.266709604	0.065399412	4.07817	0.00006562
7. DRS{6}	-0.112088351	0.075915937	-1.47648	0.14139579
8. DRS{7}	-0.175829465	0.069193109	-2.54114	0.01181155

Here the F -statistic (with three degrees of freedom in the numerator and 196 in the denominator) is 3.74 with a significance level of 0.01197. Hence, we would reject the null hypothesis at the 5% significance level and conclude that restriction is binding. At the 1% significance level, we can (just barely) accept the null hypothesis.

Note that when you do **RESTRICT (CREATE)**, the t -statistics in the new output (and any other further tests that you do) take the set of restrictions used as given. Thus the t -statistic on $DRS\{1\}$ tests whether the coefficient on the first lag is zero, *given that the first two lags sum to zero*, which means that it actually is restricting both coefficients to zero (hence the matching (up to sign) t statistics).

2.3 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
```

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 residuals in a separate series. RATS always saves

the residuals in a series in a series called %RESIDS. You can use this series just as if you named the series. However, be aware that %RESIDS is overwritten each time a new **LINREG** instruction (or similar instruction) is performed.

list The list of explanatory variables.

The most useful options for our purposes are:

DEFINE=*name of EQUATION* to define
[PRINT]/NOPRINT

LINREG also has options for correcting standard errors and *t*-statistics for hypothesis testing. **ROBUSTERRORS/ [NOROBUSTERRORS]** computes a consistent estimate of the covariance matrix that corrects for heteroscedasticity as in White (1980). **ROBUSTERRORS** and **LAGS=** produces various types of Newey-West estimates of the coefficient matrix. You can use **SPREAD** is for weighted least squares and **INSTRUMENTS** for instrumental variables. The appropriate use of these options is described in Chapter 2 of the RATS User's Guide.

2.4 Using LINREG and Related Instructions

To illustrate working with the **LINREG** and related instructions, it is useful to consider the two interest rate series shown in Panel 1 of Figure 2.1.² Economic theory suggests that long-term and short-term rates have a long-term equilibrium relationship. Although the two series appear to be nonstationary, they also seem to bear a strong relationship to each other. We can estimate this relationship using:

```
linreg tb1yr / resids
# constant tb3mo
```

²The analysis from this section is in Example 2.2, file RPM2_2.RPF.

Linear Regression - Estimation by Least Squares				
Dependent Variable TB1YR				
Quarterly Data From 1960:01 To 2012:04				
Usable Observations		212		
Degrees of Freedom		210		
Centered R ²		0.9868383		
R-Bar ²		0.9867756		
Uncentered R ²		0.9967863		
Mean of Dependent Variable	5.5787735849			
Std Error of Dependent Variable	3.1783132737			
Standard Error of Estimate	0.3654972852			
Sum of Squared Residuals	28.053535759			
Regression F(1,210)	15745.3945			
Significance Level of F	0.0000000			
Log Likelihood	-86.4330			
Durbin-Watson Statistic	0.5766			
Variable	Coeff	Std Error	T-Stat	Signif

1. Constant	0.2706392926	0.0491897079	5.50195	0.00000011
2. TB3MO	1.0547609616	0.0084057656	125.48065	0.00000000

An important issue concerns the nature of the residuals. We can obtain the first 12 residual autocorrelations using:

```
corr(num=8, results=cors, partial=partial, picture="##.###", qstats) resids
```

Correlations of Series RESIDS								
Quarterly Data From 1960:01 To 2012:04								
Autocorrelations								
1	2	3	4	5	6	7	8	
0.711	0.410	0.260	0.133	-0.042	-0.141	-0.069	-0.013	
Partial Autocorrelations								
1	2	3	4	5	6	7	8	
0.711	-0.193	0.103	-0.109	-0.176	0.003	0.179	-0.039	
Ljung-Box Q-Statistics								
Lags	Statistic	Signif	Lvl					
8	169.465	0.000000						

As expected, the residual autocorrelations seem to decay reasonably rapidly. Notice that we used the `QSTATS` option—this option produces the Ljung-Box Q -statistic for the null hypothesis that all 8 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:

```
corr(num=12, results=cors, partial=partial, span=4, qstats) resids
```

Since we are quite sure that the autocorrelations differ from zero, we won't use that here. We can graph the ACF and PACF (Figure 2.2) using:

```
graph(nodates, number=0, style=bar, key=below, footer="ACF and PACF") 2
# cors
# partial
```

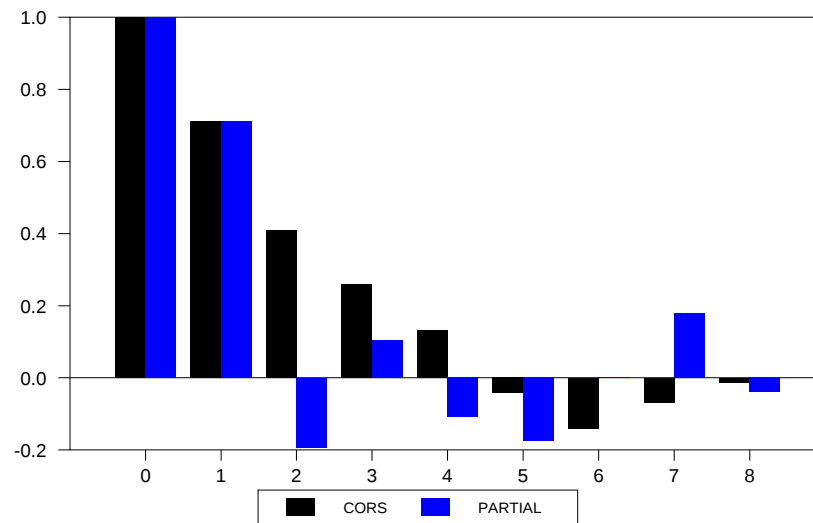


Figure 2.2: Correlations from interest rate regression

Notice that we used the `NODATES` and `NUMBER=` options. We want the x -axis to be labeled with integers ranging from 0 to 24 instead of calendar dates since these aren't data, but a sequence of statistics. 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`:

```
diff resids / dresids
```

Now estimate the dynamic adjustment process as:

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

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 (that is, under the null of no cointegration between the two rates), we cannot use the usual t -distributions—this is the Engle-Granger test from Engle and Granger (1987). And to apply this, we need to choose p to “eliminate” the serial correlation in the residuals. p is clearly not zero, so we must come up with some method to choose it.³

The ACF suggests that we can look at a relatively short lag lengths although the partial autocorrelation coefficient at lag 6 appears to be significant. We could do the test allowing for two full years' worth of lags (that is, 8) with:

³This is a heavily-used test, and RATS provides procedures for doing this, as will be discussed below. But for now, we'll look at how to do it ourselves.

```
diff resids / dresids
linreg dresids
# resids{1} dresids{1 to 8}
```

Linear Regression - Estimation by Least Squares				
Dependent Variable DRESIDS				
Quarterly Data From 1962:02 To 2012:04				
Usable Observations		203		
Degrees of Freedom		194		
Centered R ²		0.2552704		
R-Bar ²		0.2245599		
Uncentered R ²		0.2552866		
Mean of Dependent Variable		-0.001310240		
Std Error of Dependent Variable		0.281667715		
Standard Error of Estimate		0.248033987		
Sum of Squared Residuals		11.935046594		
Log Likelihood		-0.4213		
Durbin-Watson Statistic		2.0048		
Variable	Coeff	Std Error	T-Stat	Signif

1. RESIDS{1}	-0.379234416	0.084147774	-4.50677	0.00001136
2. DRESIDS{1}	0.241016426	0.092645733	2.60148	0.00999787
3. DRESIDS{2}	0.005897591	0.091245574	0.06463	0.94853175
4. DRESIDS{3}	0.125011741	0.083397849	1.49898	0.13550420
5. DRESIDS{4}	0.129833220	0.079674704	1.62954	0.10482111
6. DRESIDS{5}	0.008635668	0.078080089	0.11060	0.91204778
7. DRESIDS{6}	-0.142481467	0.075908766	-1.87701	0.06201793
8. DRESIDS{7}	0.050886186	0.071685366	0.70985	0.47864666
9. DRESIDS{8}	0.085033097	0.071088272	1.19616	0.23309320

where we can read off the E-G test statistic as the *t*-stat on the lagged residual (-4.50677). That wouldn't be an unreasonable procedure, but then at least those last two lags and perhaps all but the first lag on `DRESIDS` look like they may be unnecessary. Since each added lag costs a usable data point, and unneeded coefficients tend to make the small-sample behavior of tests worse, it would be useful to see if we can justify using fewer.

There are several possible ways to “automate” lag selection in a situation like this. Here, we'll demonstrate use of Information Criteria. Using the variables defined by a `LINREG`, values for the Akaike Information Criterion (AIC) and the Schwartz Bayesian Criterion (SBC) (often called the Bayesian Information Criterion or BIC) can be computed using:

```
com aic = -2.0*%logl + %nreg*2
com sbc = -2.0*%logl + %nreg*log(%nobs)
```

We want to suppress the usual `LINREG` output from the regressions with different lags, so we'll use `NOPRINT`. Then we'll use `DISPLAY` to show the test statistic and the two criteria.

When you use Information Criteria to choose lag length, it's important to make sure that you use the same sample range for each regression—if you don't, the sample log likelihoods won't be comparable. We can pick up the range from the 8 lag regression using

```
compute egstart=%regstart()
```

and use that as the start period on the other regressions:

```
do i = 0,8
  linreg(noprint) dresids egstart *
  # resids{1} dresids{1 to i}
  com aic = -2.0*%logl + %nreg*2
  com sbc = -2.0*%logl + %nreg*log(%nobs)
  dis "Lags: " i "T-stat" %tstats(1) "The aic = " aic " and sbc = " sbc
end do i
```

Lags: 0	T-stat	-5.87183	The aic =	30.68736	and sbc =	34.00057
Lags: 1	T-stat	-6.60326	The aic =	24.73007	and sbc =	31.35648
Lags: 2	T-stat	-5.38733	The aic =	24.58260	and sbc =	34.52222
Lags: 3	T-stat	-5.63637	The aic =	23.88187	and sbc =	37.13469
Lags: 4	T-stat	-6.24253	The aic =	19.43505	and sbc =	36.00108
Lags: 5	T-stat	-5.67614	The aic =	21.43312	and sbc =	41.31236
Lags: 6	T-stat	-4.37831	The aic =	16.65477	and sbc =	39.84721
Lags: 7	T-stat	-4.34237	The aic =	18.33419	and sbc =	44.83984
Lags: 8	T-stat	-4.50677	The aic =	18.84250	and sbc =	48.66135

Whether we use the 6-lag model selected by the minimum 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

It is important to note that there are many equivalent ways to report the AIC and SBC for linear regressions, which is fine as long as you use the same formula in comparing models. The following eliminate the additive terms from $-2 \log L$ term that depend only upon the number of observations:

$$AIC = T \log(RSS) + 2k$$

$$SBC = T \log(RSS) + k \log T$$

and can be computed with

```
com aic = %nobs*log(%rss)+%nreg*2
com bic = %nobs*log(%rss)+%nreg*log(%nobs)
```

You can also divide through the formulas by the number of observations. Since the number of observations should match in models that you are comparing with the information criteria, neither of these changes will affect the rank orderings of models.

We can now re-run the regression with the 6 lags picked using AIC:

⁴With 203 usable observations, the 5% critical value is -3.368 .

```
linreg dresids
# resids{1} dresids{1 to 6}
```

Note that the t -statistic on the lagged residual is slightly different here from what it was for six lags when we did the loop (-4.44300 vs -4.37831). This is because the earlier regression used the sample range that allowed for eight lags, while this one has re-estimated using the longer range allowed by only six lags. It's a fairly common practice in this type of analysis to pick the number of lags based upon a common range (which is necessary for using the information criteria), then re-estimate with the chosen lag length using as much data as possible.

Two other standard procedures can be helpful in avoiding some of the programming shown above. `@REGCRITS` produces four model selection criterion (including the AIC and SBC). Note that it uses a “per observation” likelihood-based version of the criteria:

$$AIC = -2 \log(L)/T + 2k/T$$

$$SBC = -2 \log(L)/T + k \log(T)/T$$

Again, this is fine as long as you use the same formula for each model that you are comparing.

`@REGCORRS` produces an analysis of the residuals—with the options shown below it creates both a graph of the correlations, and a table of “running” Q statistics.

```
@regcrits
@regcorrs (number=24, qstats, report)
```

As mentioned earlier, the Engle-Granger test is important enough that there is a separate procedure written to do the calculation above. That's `@EGTEST`. To choose from up to 8 lags using AIC, you would do the following:

```
@egtest (lags=8, method=aic)
# tbl1yr tlb3mo
```

Note that this matches what we did, and gives the test statistic from the re-estimated model:

```
Engle-Granger Cointegration Test
Null is no cointegration (residual has unit root)
Regression Run From 1961:04 to 2012:04
Observations          206
With 6 lags chosen from 8 by AIC
Constant in cointegrating vector
Critical Values from MacKinnon for 2 Variables

Test Statistic -4.44300**
1%(**)         -3.95194
5%(*)          -3.36688
10%            -3.06609
```

2.5 ARMA(p,q) Models

Instead of the pure autoregressive process represented by equation (2.1), most time-series models are based on the stochastic difference equation with p autoregressive terms and q moving average terms. Consider

$$y_t = a_0 + a_1 y_{t-1} + a_2 y_{t-2} + \dots + a_p y_{t-p} + \varepsilon_t + \beta_1 \varepsilon_{t-1} + \dots + \beta_q \varepsilon_{t-q}$$

where y_t is the value of the variable of interest in time period t , the values of a_0 through a_p and β_1 through β_q are coefficients, and ε_t is a white-noise stochastic disturbance with variance σ^2 .

As a practical matter, the order of the ARMA process is unknown to the researcher and needs to be estimated. The typical tools used to identify p and q are the autocorrelation function (ACF) and the partial autocorrelation function (PACF). The autocorrelation function is the set of correlations between y_t and y_{t-i} for each value of i . Thus, the ACF is formed using

$$\rho_i = \gamma_i / \gamma_0$$

where γ_i is the covariance between y_t and y_{t-i} . As discussed in Enders (2010), some of the key properties of the ACF are:

1. White-noise (i.e., $a_i = 0$ and $\beta_i = 0$): All autocorrelations are zero.
2. AR(1): For $a_1 > 0$, the values of ρ_i decay geometrically with $\rho_i = a_1^i$. For negative values of a_1 , the decay is oscillatory.
3. MA(q): The autocorrelations cut to zero after lag q .
4. AR(2): The decay pattern can contain trigonometric components.
5. ARMA(1, q): If $a_1 > 0$ and $q = 1$, geometric decay after lag 1; if $a_1 < 0$ there is oscillating geometric decay after lag 1.
6. ARMA(p , q): The ACF will begin to decay at lag q . The decay pattern can contain trigonometric components.

In contrast to the autocorrelation ρ_i , the partial autocorrelation between y_t and y_{t-i} holds constant the effects of the intervening values of y_{t-1} through y_{t-i+1} . A simple way to understand the partial autocorrelation function is to suppose that the y_t series is an ARMA(p , q) process has been demeaned. Now consider the series of regression equations

$$y_t = \theta_{11} y_{t-1} + e_t$$

$$y_t = \theta_{21} y_{t-1} + \theta_{22} y_{t-2} + e_t$$

$$y_t = \theta_{31} y_{t-1} + \theta_{32} y_{t-2} + \theta_{33} y_{t-3} + e_t$$

where e_t is an error term (that may not be white-noise).

The partial autocorrelation function (PACF) is given by the values θ_{11} , θ_{22} , θ_{33} , etc., that is, the coefficient on the final lag. For a pure $AR(p)$ process, $\theta_{p+1,p+1}$ is necessarily zero. Hence, the PACF of an $AR(p)$ process will cut to zero after lag p . In contrast, the PACF of a pure MA process will decay.

2.6 Estimation of an ARMA(p,q) process with RATS

The Box-Jenkins methodology is a three-step procedure: Identification, Estimation, and Diagnostic Checking. Each is described below.

2.6.1 Identification

The first step in the Box-Jenkins methodology is to identify several plausible models. A time-series plot of the series and a careful examination of the ACF and PACF of the series can be especially helpful. Be sure to check for outliers and missing values. If there is no clear choice for p and q , entertain a number of reasonable models. If the series has a pronounced trend or meanders without showing a strong tendency to revert to a constant long-run value, you should consider differencing the variables. As discussed in later chapters, the current practices in such circumstances involve testing for unit roots and/or structural breaks.

Although you can create the plot the correlations and partial correlations using **CORRELATE** and **GRAPH**, it's much quicker to use the **@BJIDENT** procedure:

```
@BJIDENT( options ) series start end
```

series Series used to compute the correlations.

start end Range of entries to use. The default is the entire series.

The principal options are:

DIFF=number of regular differences [0]

SDIFFS=number of seasonal differences [0]

TRANS= [NONE] /LOG/ROOT

Chooses the preliminary transformation (if any). **ROOT** means the square root.

NUMBER=number of correlations to compute

The default is the integer value of $T/4$

2.6.2 Estimation

Although it is straightforward to estimate an $AR(p)$ process using **LINREG**, the situation is more complicated when MA terms are involved. Since, the values of $\varepsilon_t, \varepsilon_{t-1}, \dots$ are not observed, it isn't possible to let the lagged values of these error terms be regressors in an OLS estimation. Instead, models with MA terms are generally estimated using maximum likelihood techniques. The form of the RATS instruction used to estimate an ARMA model is:

```
BOXJENK( options ) series start end residuals
```

For our purposes, the important options are:

```
AR=number of autoregressive parameters [0]
MA=number of moving average parameters [0]
DIFFS=number of regular differences [0]
CONSTANT/[NOCONSTANT]
```

Note: by default, a constant is not included in the model.

```
SAR=number of seasonal autoregressive parameters [0]
SMA=number of seasonal moving average parameters [0]
DEFINE=name of the EQUATION to define from this
```

As with the **LINREG** instruction, **BOXJENK** creates a number of internal variables that you can use in subsequent computations. A partial list of these variables includes the coefficient vector %BETA, the vector of the t -statistics %TSTATS, the degrees of freedom %NDF, the number of observations %NOBS, the number of regressors %NREG, and the residual sum of squares %RSS.

BOXJENK also creates the variable %CONVERGED. %CONVERGED = 1 if the estimation converged and %CONVERGED = 0 if it didn't.

It is important to remember that the default is to *not* include an intercept term from the model. Moreover, the reported value of **CONSTANT** is actually the estimate of the mean (not the estimate of a_0).⁵ The relationship between the mean, μ , and the intercept, a_0 , is

$$\mu = a_0 / (1 - a_1 - a_2 - \dots - a_p)$$

After the candidate set of models has been estimated, they should be compared using a number of criteria including:

Parsimony

Select models with low values of p and q . Large values of p and q will necessarily increase the fit, but will reduce the number of degrees of freedom. Poorly

⁵This parameterization makes it simpler to do more general regressions with ARIMA errors.

estimated coefficients have large standard errors and will generally result in poor forecasts. Moreover, high order ARMA(p, q) models can usually be well-approximated by low-order models.

Goodness of Fit

The most popular goodness-of-fit measures are the Akaike Information Criterion (AIC) and the Schwartz Bayesian Criterion (SBC). You can construct these measures using the same code as for the **LINREG** instruction.

2.6.3 Diagnostic Checking

It is important to check the residuals for any remaining serial correlation. The **GRAPH**, **STATISTICS**, and **CORRELATE** instructions applied to the residuals can help you determine whether or not the residuals are well-described as a white-noise process. Any pattern in the residuals means that your equation is misspecified. As described below, you can use recursive estimations to check for coefficient stability.

2.7 An Example of the Price of Finished Goods

The ideas in the previous section can be illustrated by considering an extended example of the producer price index. This is **Example 2.3**, file `RPM2_3.RPF`. We'll again read in the data with

```
cal(q) 1960:1
all 2012:4
*
open data quarterly(2012).xls
data(org=obs, format=xls)
```

If you look at a time series graph of PPI,⁶ it should be clear that it is not stationary and needs to be differenced. It turns out that it is best to work with the logarithmic difference. This variable can be created using:

```
log ppi / ly
dif ly / dly
```

We could do this in one step, but we'll also have use for the `LY` series itself. Now graph the (log differenced) series and obtain the ACF and PACF using

```
spgraph(footer="Price of Finished Goods", hfield=2, vfield=1)
  graph(header="Panel a: Quarterly Growth Rate") 1
  # dly
  @bjident(separate, number=12) dly
spgraph(done)
```

⁶The quickest way to do that is by doing the menu operation *View-Series Window*, click on the `PPI` series and the *Time Series Graph*  toolbar icon.

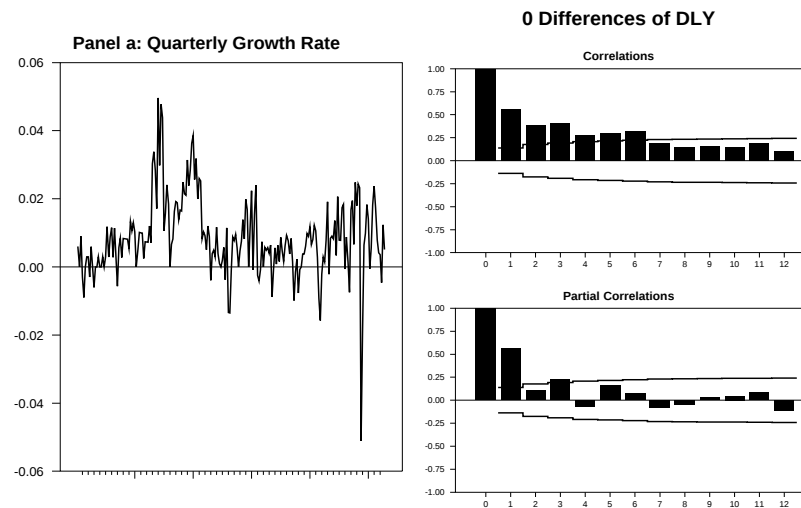


Figure 2.3: Price of Finished Goods

This produces Figure 2.3. Notice that we wrapped an **SPGRAPH** around our own **GRAPH**, and the graphs produced by **@BJIDENT**. **@BJIDENT** *also* uses **SPGRAPH** to split a graph space vertically between the ACF on top and the PACF on the bottom. These end up splitting the right pane in the **SPGRAPH** that *we* define. This is how nested **SPGRAPHS** work.

The plot of the series, shown in Panel a of Figure 2.3 indicates that there was a run-up of prices in the early 1970s and a sharp downward spike in 2008:4. However, for our purposes, the series seems reasonably well-behaved. Although the ACF (shown in the upper right-hand portion of the figure) decays, the decay does not seem to be geometric. Notice that the PACF has significant spikes at lags 1 and 3. As such, at this point in the analysis, there are several possible candidates:

1. AR(3): The ACF does not exhibit simple geometric decay so an AR(1) is likely to be inappropriate. Moreover, the PACF does not display a simple decay pattern; instead, the values of θ_1 and θ_3 are significant.
2. Low-order ARMA(p, q): Neither the ACF nor the PACF display simple decay patterns. As such, the process may be mixed in that it contains AR and MA terms.

Estimating the series as an AR(3) can be done using **LINREG** or **BOXJENK**. To illustrate the use of **BOXJENK**, we have

```
boxjenk(constant, ar=3) dly
```

The coefficient block of the output is:

Variable	Coeff	Std Error	T-Stat	Signif

1. CONSTANT	0.008431166	0.002141902	3.93630	0.00011349
2. AR{1}	0.478711560	0.068156413	7.02372	0.00000000
3. AR{2}	-0.008559549	0.076086058	-0.11250	0.91053898
4. AR{3}	0.228904929	0.068504836	3.34144	0.00099121

If we reestimate the model without the insignificant AR(2) term, we obtain:

```
boxjenk(constant,ar=||1,3||) dly
```

Variable	Coeff	Std Error	T-Stat	Signif

1. CONSTANT	0.0084324694	0.0021480622	3.92562	0.00011810
2. AR{1}	0.4752570939	0.0607005314	7.82954	0.00000000
3. AR{3}	0.2253908909	0.0608218529	3.70576	0.00027113

Checking the residuals for serial correlation, it should be clear that the model performs well. Note the use of the option `DFC=%NARMA`. **BOXJENK** sets the internal variable `%NARMA` with the number of AR + MA coefficients, which is required to correct the degrees of freedom for the *Q*-statistics when they are computed for the residuals from an ARMA estimation. Here, `%NARMA` is equal to 2.

```
corr(number=8,qstats,span=4,dfc=%narma,picture=".#.###") %resids
```

Correlations of Series %RESIDS							
Quarterly Data From 1961:01 To 2012:04							
Autocorrelations							
1	2	3	4	5	6	7	8
0.022	-0.037	-0.003	-0.123	0.067	0.150	-0.035	-0.067
Ljung-Box Q-Statistics							
Lags	Statistic	Signif	Lvl				
4	3.610	0.164446					
8	10.678	0.098844					

The fit of the model can be obtained using:

```
com aic = -2.0*%logl + %nreg*2
com sbc = -2.0*%logl + %nreg*log(%nobs)
display "aic = " aic "bic = " sbc
```

aic =	-1353.86042	bic =	-1343.84781
-------	-------------	-------	-------------

Next, estimate the series as an ARMA(1, 1) process. Since the two models are to be compared head-to-head, they need to be estimated over the same sample period. The estimation for the ARMA(1,1) is constrained to begin on 1961:1 (the first usable observation for the AR model with three lags).

```
boxjenk(constant,ar=1,ma=1) dly 1961:1 *
com aic = -2.0*%logl + %nreg*2
com sbc = -2.0*%logl + %nreg*log(%nobs)
display "aic = " aic "bic = " sbc
```

Variable	Coeff	Std Error	T-Stat	Signif

1. CONSTANT	0.008516961	0.002099853	4.05598	0.00007093
2. AR{1}	0.810358229	0.066987586	12.09714	0.00000000
3. MA{1}	-0.393822229	0.105012809	-3.75023	0.00022985

aic = -1346.07120 bic = -1336.05858				

The AR(1,3) is the clear favorite of the information criteria⁷

In addition, the residual correlations for the ARMA model are unsatisfactory:

```
corr(number=8,qstats,span=4,dfc=%narma,picture=".#####") %resids
```

Correlations of Series %RESIDS							
Quarterly Data From 1961:01 To 2012:04							
Autocorrelations							
1	2	3	4	5	6	7	8
0.045	-0.140	0.092	-0.105	0.040	0.176	-0.040	-0.070
Ljung-Box Q-Statistics							
Lags	Statistic	Signif	Lvl				
4	8.751	0.012583					
8	17.174	0.008665					

Note that you can also use @REGCORRS to do the analysis of the correlations, displaying the autocorrelations (Figure 2.4), the Q and the AIC and SBC:⁸

```
@regcorrs(number=8,qstats,dfc=%narma,footer="ARMA(1,1) Model")
```

Exercise 2.1 *Experiment with the following:*

1. `@bjident(report,qstats,number=8) resids`
2. `box(ar=5,constant) dly / resids`
`versus`
`box(ar=||1,3||,constant) dly / resids`
3. `box(ar=2,ma=1,constant) dly / resids`
`dis \%converged`

⁷Since the number of estimated parameters in the two models is the same, the two criteria must agree on the ordering.

⁸The AIC and SBC on the output from @REGCORRS have been divided by the number of observations which doesn't change the ordering, and makes them easier to display.

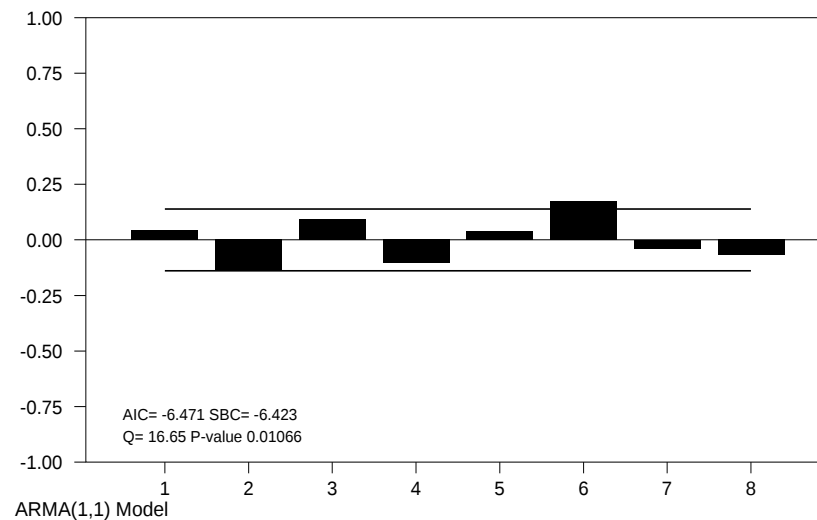


Figure 2.4: Residual analysis for ARMA(1,1)

2.8 Automating the Process

The Box-Jenkins methodology provides a useful framework to arrive at a carefully constructed model that can be used for forecasting. However, to some, the method is not scientific in that two equally skilled researchers might come up with slightly different models. Moreover, it may not be practical to carefully examine each series if there are many series to process. An alternative is to use a completely data-driven way to select p and q . The method is to estimate every possible ARMA(p, q) model and to select the one provides the best fit. Although it might seem time-intensive to estimate a model for every possible p and q , RATS can do it almost instantly. This is done in Example 2.4, file RPM2_4.RPF.

The procedure `@BJAUTOFIT` allows you to specify the maximum values for p and q and estimates all of the varied ARMA models.⁹ You can select whether to use the AIC or the SBC to use as the criterion for model selection.

```
@BJAUTOFIT( options ) series start end
```

The important options are

```
PMAX=maximum value of p
QMAX=maximum value of q
CRIT=[AIC]/SBC
DIFFS=number of regular differencings [0]
SDIFFS=number of regular differencings [0]
```

⁹It does not allow for “zeroing out” intermediate lags. An AR(3) is considered but not an AR({1,3}). There are simply too many possibilities if lags are allowed to be skipped.

CONSTANT/ [NOCONSTANT]

It is instructive to understand the programming methods used within the procedure.

2.8.1 Introduction to DO Loops

The **DO** loop is the simplest and most heavily used of the programming features of RATS (and any other statistical programming language). It's a very simple way to automate many of your repetitive programming tasks. The usual structure of a **DO** loop is:

```
do index=n1,n2,increment
  program statements
end do index
```

where **N1**, **N2** and **INCREMENT** are integer number or variables. If **INCREMENT** is omitted, the default value of 1 is used. The “index” is a name that you assign to be the counter through the loop. Typically, this is a (very) short name, with **I**, **J** and **K** being the most common ones. **I** and **J** are, in fact, defined variables in RATS precisely because they are such common loop indexes (which is why you can't use the **I** name for an “investment” series). There is, of course, nothing preventing you for using something more descriptive like **LAG** or **DRAW**.

The **DO** loop is really a counter. The first time that the **DO** instruction is executed, the index variable takes on the value **N1** and the block of program statements is executed. On reaching the end of the loop, the index is increased by **INCREMENT** (that is, index is now **N1+INCREMENT**). If the index is less than or equal to **N2**, the block of program statements is executed again. On reaching the end of the loop, the value of the index is again increased by **INCREMENT** and again compared to **N2**. This process is repeated until the value of the index exceeds **N2**. At that point, RATS exits the loop and subsequent instructions can be performed.

There are two differences among **DO** loop implementations in different languages that you need to keep in mind:

1. In RATS, if **N1**>**N2**, the loop is not executed at all (that is, the test is at the top of the loop). In a few languages, the loop is always executed once (the test is at the bottom).
2. In RATS, the loop index has the value from the last executed pass through the statements, *not* the value that would force the termination of the loop. This can be quite different from language to language (in some, it's not even necessarily defined outside the loop).

DO loops are particularly useful because they can be nested. Consider the following section of code:

```
do q=0,3
  do p=0,3
    boxjenk(constant,ar=p,ma=q) y
  end do p
end do q
```

The key point to note is that the AR and MA options of the **BOXJENK** instruction do not need to be set equal to specific *numbers*. Instead, they are set equal to the counters P and Q. The first time through the two loops, P and Q are both zero, so the **BOXJENK** estimates an ARMA(0,0) model (that is, just the mean). Since the P loop is the *inner* loop, the value of P is incremented by 1 but Q remains at 0. Hence, RATS next estimates an AR(1) model. Once the AR(3) model is estimated, control falls out of the **DO P** loop, and Q is incremented. The **DO P** loop is started again with P=0, but now with Q=1. In the end, all 16 combinations of ARMA(p,q) models with $0 \leq p \leq 3$ and $0 \leq q \leq 3$ will be estimated.

The output produced by this small set of instructions can be overwhelming. In practice, it is desirable to suppress most of the output except for the essential details. A simple way to do this is to use the **NOPRINT** option of **BOXJENK**. The following estimates the 16 models over a common period (1961:1 is the earliest that can handle 3 lags on the difference), displays the AIC, SBC, and also shows the value of %CONVERGED. The results from an estimation that has not converged are clearly garbage, though it's very rare that a model where the estimation would fail to converge would be selected anyway, since, for an ARMA model, a failure to converge is usually due to "overparameterization", which is precisely what the AIC and SBC are trying to penalize.

```
do q=0,3
  do p=0,3
    boxjenk(noprint,constant,ar=p,ma=q) dly 1961:1 *
    com aic=-2*logl+%nreg*2
    com sbc=-2*logl+%nreg*log(%nobs)
    disp "Order("+p+", "+q+")" "AIC=" aic "SBC=" sbc "OK" %converged
  end do p
end do q
```

Order(0,0)	AIC=	-1265.14664	SBC=	-1261.80910	OK	1
Order(1,0)	AIC=	-1342.37368	SBC=	-1335.69861	OK	1
Order(2,0)	AIC=	-1342.78978	SBC=	-1332.77717	OK	1
Order(3,0)	AIC=	-1351.87333	SBC=	-1338.52317	OK	1
Order(0,1)	AIC=	-1326.95443	SBC=	-1320.27935	OK	1
Order(1,1)	AIC=	-1346.07120	SBC=	-1336.05858	OK	1
Order(2,1)	AIC=	-1346.06399	SBC=	-1332.71383	OK	1
Order(3,1)	AIC=	-1352.68025	SBC=	-1335.99256	OK	1
Order(0,2)	AIC=	-1327.74892	SBC=	-1317.73630	OK	1
Order(1,2)	AIC=	-1349.46788	SBC=	-1336.11773	OK	1
Order(2,2)	AIC=	-1353.41486	SBC=	-1336.72717	OK	1
Order(3,2)	AIC=	-1351.42556	SBC=	-1331.40034	OK	1
Order(0,3)	AIC=	-1345.26392	SBC=	-1331.91377	OK	1
Order(1,3)	AIC=	-1350.81865	SBC=	-1334.13096	OK	1
Order(2,3)	AIC=	-1351.41686	SBC=	-1331.39164	OK	1
Order(3,3)	AIC=	-1353.87773	SBC=	-1330.51496	OK	1

The minimum AIC is (3,3) (just barely better than the (2,2)), while SBC shows a clear choice at (3,0). If we want to use AIC as the criterion, we should probably try a higher limit since as it is, it's being minimized at the upper bound in both parameters.

A similar analysis can be done with the `@BJAUTOFIT`:

```
@bjautofit(constant,pmax=3,qmax=3,crit=aic) dly
```

AIC analysis of models for series DLY				
MA				
AR	0	1	2	3
0	-1287.8512	-1349.7689	-1350.5439	-1368.6927
1	-1365.5246	-1369.9612	-1373.2313	-1374.6395
2	-1365.9782	-1371.0569	-1377.5783*	-1375.7936
3	-1375.0658	-1375.5717	-1375.7714	-1377.5110

You'll note that the values (and the decision) are somewhat different. That's because `@BJAUTOFIT` uses maximum likelihood (rather than conditional least squares), and, because it uses the (more complicated) maximum likelihood estimator, it can use the full data range for `DLY` from 1960:2 on. You'll also note that it's presented in a more convenient table rather than a line at a time listing that we got using `DISPLAY`. `@BJAUTOFIT` (and most other standard RATS procedures) uses the `REPORT` instructions to format their output. We'll learn more about that later in the book.

Exercise 2.2 *You can see how well the method works using the following code:*

```
set eps = %ran(1)
set(first=%ran(1)) y1 = 0.5*y1{1} + eps
@bjautofit(pmax=3,qmax=3,constant) y1
```

Repeat using

```

set y2 = 0.
set y2 3 * = 0.5*y2{1} + 0.25*y2{2} + eps }
@bjautofit (pmax=3,qmax=3,constant) y2}
set y3 = 0.5*eps{1} + eps }
@bjautofit (pmax=3,qmax=3,constant) y3}
set (first=%ran(1)) y4 = 0.5*y4{1} + eps + 0.5*eps{1}
@bjautofit (pmax=3,qmax=3,constant) y4

```

Be aware that the model with the best in-sample fit may not be the one that provides the best forecasts. It is always necessary to perform the appropriate diagnostic checks on the selected model. Autofitting techniques are designed to be a tool to help you select the most appropriate model.

2.9 An Example with Seasonality

Many series have a distinct seasonal pattern such that their magnitudes are predictably higher during some parts of the year than in others. In order to properly model such series, it is necessary to capture both the seasonal and nonseasonal dependence. Since the Federal Reserve injects currency into the financial system during the winter quarter, we would expect the stock of currency in the current winter quarter to aid in predicting the stock for the next winter. Fortunately, the autocorrelations and partial autocorrelations often reflect pattern of the seasonality. In Example 2.5 (file `RPM2_5.RPF`), we again pull in the data set `QUARTERLY(2102).XLS`:

```

cal(q) 1960:1
all 2012:4
open data quarterly(2012).xls
data(org=obs,format=xls)

```

We then compute the log difference and log combined regular and seasonal differences of currency with:

```

set ly = log(curr)
dif ly / dly
dif(sdiffs=1,dif=1) ly / m

```

In terms of lag operators, $m = (1 - L^4)(1 - L)ly$. The following graphs (Figure 2.5) the ACF and PACF of the two types of differenced data:

```

spgraph(footer="ACF and PACF of dly and m",hfields=2,vfields=1)
  @bjident dly
  @bjident m
spgraph(done)

```

The ACF and PACF of the dly series (shown in the left-hand side panel) indicate that the ρ_1 is significant at the 5% level. However, the pronounced features of

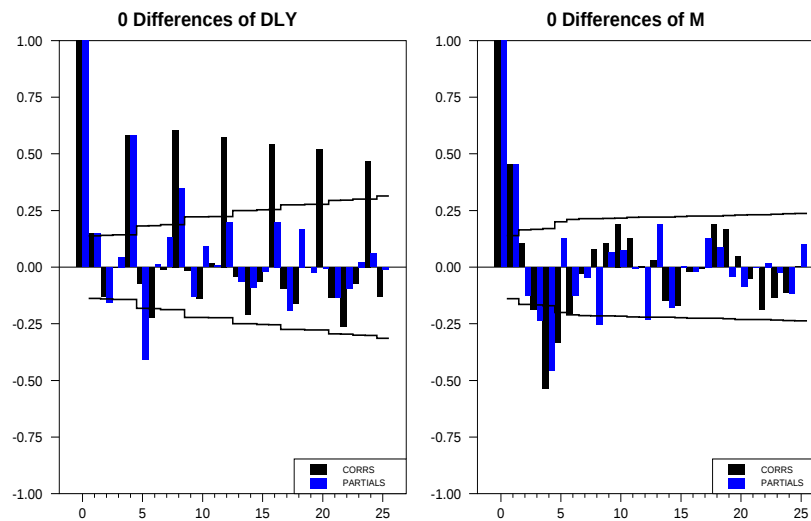


Figure 2.5: ACF/PACF of Differences of Log Currency

the figure are the spikes at lags in the autocorrelations at lags 4, 8, 12, 16, 20 and 24. Obviously, these correlations correspond to the seasonal frequency of the data. If we focus only on the autocorrelations at the seasonal frequency, it should be clear that there is little tendency for these autocorrelations to decay. In such circumstances, it is likely that the data contains a seasonal unit-root. Hence, we can transform the data by forming the seasonal difference of dly_t as: $m_t = dly_t - dly_{t-4}$.

If you are unsure as to the number of differences to use, the procedure `@BJDIFF` can be helpful. The following reports the Schwartz criteria allowing for a maximum of one regular difference and one seasonal difference with and without a constant:

BJDiff Table, Series CURR					
Reg	Diff	Seas	Diff	Intercept	Crit
0		0	No		-0.104162
0		0	Yes		-2.558801
0		1	No		-8.222783
0		1	Yes		-8.994149
1		0	No		-8.524110
1		0	Yes		-9.098987
1		1	No		-9.477942*
1		1	Yes		-9.453089

As indicated by the asterisk (*), the log transformation using one seasonal difference, one regular difference, and no intercept seems to be the most appropriate.¹⁰

Because there are now four choices that need to be made (regular AR and MA, seasonal AR and MA) rather than just two, selecting a model for a seasonal

¹⁰You can formally test for a unit root and a seasonal unit root using the Hylleberg, Engle, Granger, and Yoo (1990) test. The procedure `@HEGY` will perform the test using quarterly data.

ARMA can be quite tricky. In general, it's a good idea to start with simple models, estimate and see if there's residual autocorrelation remaining. Here, we'll start with four: $(1, 1, 0) \times (0, 1, 1)$, $(0, 1, 1) \times (0, 1, 1)$ (sometimes known as the "airline model"), $(1, 1, 0) \times (1, 1, 0)$ and $(0, 1, 1) \times (1, 1, 0)$. This covers all four cases with exactly one parameter in the regular polynomial (either one AR or one MA) and one in the seasonal (one SAR or one SMA). We'll use @REGCORRS to compute the Q , and the information criteria for each. For comparison purposes, we need to estimate the models over the same sample period. Note that the model with one AR term and one seasonal AR term can begin no earlier than 1962:3, with losses due to both the differencing and the lags for the AR and SAR parameters. We'll use NOPRINT while we're just doing a crude check of various models.

```
boxjenk(noprint, constant, ar=1, sma=1) m 1962:3 *
@regcorrs(title="(1,1,0)x(0,1,1)", dfc=%narma)
display "aic = " %aic "bic = " %sbc "Q(signif)" *.### %qsignif
*
boxjenk(noprint, constant, ma=1, sma=1) m 1962:3 *
@regcorrs(title="(0,1,1)x(0,1,1)", dfc=%narma)
display "aic = " %aic "bic = " %sbc "Q(signif)" *.### %qsignif
*
boxjenk(noprint, constant, ar=1, sar=1) m 1962:3 *
@regcorrs(title="(1,1,0)x(1,1,0)", dfc=%narma)
display "aic = " %aic "bic = " %sbc "Q(signif)" *.### %qsignif
*
boxjenk(noprint, constant, ma=1, sar=1) m 1962:3 *
@regcorrs(title="(0,1,1)x(1,1,0)", dfc=%narma)
display "aic = " %aic "bic = " %sbc "Q(signif)" *.### %qsignif
```

aic =	-7.11001	bic =	-7.06088	Q(signif)	0.566
aic =	-7.10238	bic =	-7.05325	Q(signif)	0.442
aic =	-6.93147	bic =	-6.88234	Q(signif)	0.002
aic =	-6.92417	bic =	-6.87504	Q(signif)	0.001

The first of the four models seems to be slightly preferred over the second, with the other two being inadequate. We can now take the NO off the NOPRINT to look at it more carefully:

```
boxjenk(print, constant, ar=1, sma=1) m 1962:3 *
@regcorrs(title="(1,1,0)x(0,1,1)", dfc=%narma)
display "aic = " %aic "bic = " %sbc "Q(signif)" *.### %qsignif
```

Box-Jenkins - Estimation by LS Gauss-Newton				
Convergence in 10 Iterations. Final criterion was 0.0000032 <= 0.0000100				
Dependent Variable M				
Quarterly Data From 1962:03 To 2012:04				
Usable Observations	202			
Degrees of Freedom	199			
Centered R ²	0.5102749			
R-Bar ²	0.5053530			
Uncentered R ²	0.5105079			
Mean of Dependent Variable	0.0002124435			
Std Error of Dependent Variable	0.0097610085			
Standard Error of Estimate	0.0068650289			
Sum of Squared Residuals	0.0093785957			
Log Likelihood	721.1113			
Durbin-Watson Statistic	1.9192			
Q(36-2)	38.9946			
Significance Level of Q	0.2551573			
Variable	Coeff	Std Error	T-Stat	Signif

1. CONSTANT	0.000071690	0.000209506	0.34219	0.73257113
2. AR{1}	0.432725378	0.063065071	6.86157	0.00000000
3. SMA{4}	-0.767040488	0.047004350	-16.31850	0.00000000

The Q statistic is fine¹¹ and there are no obvious signs of problems in the individual autocorrelations on the graph (not shown). The only change to the model that seems likely to make a difference would be to take the constant out. @BJDIFF chose a model with no constant, and this would confirm that that seemed to be the correct recommendation.¹²

The procedure for automating selection of a seasonal ARMA model (given the choice of differencing) is @GMAUTOFIT. This puts an upper bound, but not lower bound, on the number of parameters in each polynomial, so it includes models with no parameters. Even with a limit of 1 on each, there are quite a few candidate models. And, fortunately, it agrees with our choice:

```
@gmautofit (regular=1, seasonal=1, noconst, full, report) m
```

AR	MA	AR(s)	MA(s)	LogL	BIC
0	0	0	0	666.1212169	-1332.24243
0	0	0	1	713.3733549	-1421.41399
0	0	1	0	700.2558277	-1395.17894
0	0	1	1	714.0155643	-1417.36569
0	1	0	0	689.8974268	-1374.46213
0	1	0	1	736.2116913	-1461.75795
0	1	1	0	720.3065378	-1429.94764
0	1	1	1	736.3675337	-1456.73691
1	0	0	0	689.7674212	-1374.20212
1	0	0	1	737.3251643	-1463.98489*
1	0	1	0	721.2150088	-1431.76458
1	0	1	1	737.4792222	-1458.96029
1	1	0	0	690.6894892	-1370.71354
1	1	0	1	737.9898838	-1459.98161
1	1	1	0	721.7517229	-1427.50529
1	1	1	1	738.1317848	-1454.93269

¹¹The Q in the output above and the Q in the @REGCORRS output use different numbers of autocorrelations, but they lead to the same conclusion.

¹²A model with two differences (in any combination) plus a constant would have a quadratic drift, which isn't likely.

2.10 Forecasts and Diagnostic Checks

Perhaps the primary use of the Box-Jenkins methodology is to develop a forecasting model. The selection of the proper AR and MA coefficients is important because any model specification errors will be projected into the future. If your model is too small, it will not capture all of the dynamics present in the series. As such, you want to forecast with a model that does not contain any remaining correlation in the residuals. Clearly, one way to eliminate serial correlation in the residuals is to add one or more AR or MA coefficients. However, you need to be cautious about simply increasing the number of estimated parameters. All parameters estimated are subject to sampling error. If your model includes a parameter with a large standard error, this estimation error will be projected into the future. To take a specific example, suppose that y_t is properly estimated as an AR(1) process. However, suppose that someone estimates the series as $y_t = a_0 + a_1y_{t-1} + a_2y_{t-2} + \varepsilon_t$. Now, if a_2 turned out to be exactly zero, there would not be a problem forecasting with this model. In point of fact, the probability that the estimated value of a_2 takes on the value zero is almost surely equal to zero. On average, forecasts from the model will contain the error a_2y_{t-2} .

Most diagnostic checks begin with a careful examination of the residuals. Plot the residuals and their ACF to ensure that they behave as white-noise. This is most easily done using the **REGCORRS** procedure discussed earlier. For now, we consider the issue of how to use a properly estimated ARIMA model to forecast.

Out of Sample Forecasts

Probably the most important use of time-series models is to provide reliable forecasts of the series of interest. After all, once the essential properties of the dynamic process governing the evolution of y_t have been estimated, it is possible to project these properties into the future to obtain forecasts. To take a simple example, suppose that the evolution of y_t has been estimated to be the ARMA(1,1) process:

$$y_t = a_0 + a_1y_{t-1} + \beta_1\varepsilon_{t-1} + \varepsilon_t$$

so that

$$y_{t+1} = a_0 + a_1y_t + \beta_1\varepsilon_t + \varepsilon_{t+1}$$

Given the estimates of a_0 , a_1 , and β_1 along with the estimated residual series, the conditional expectation of y_{t+1} is

$$\begin{aligned} E_ty_{t+1} &= E_t[a_0 + a_1y_t + \beta_1\varepsilon_t + \varepsilon_{t+1}] \\ &= a_0 + a_1y_t + \beta_1\varepsilon_t \end{aligned}$$

and the conditional expectation of y_{t+2} is

$$E_ty_{t+2} = a_0 + a_0a_1 + a_1\beta_1\varepsilon_t + a_1^2y_t$$

The arithmetic gets quite messy with larger models and for larger forecasting horizons. However, the important point is that RATS can readily perform all of the required calculations using the **UFORECAST** and **FORECAST** instructions. Although the **FORECAST** instruction is the more flexible of the two, **UFORECAST** is very easy to use for single-equation models. The syntax of **UFORECAST** is:

```
UFORECAST ( options ) series start end
```

series This is where the forecasts will be saved.

start end The range to forecast. You can use this, or the **FROM**, **TO** and **STEPS** options to set this—choose whichever is simplest in a given situation.

The most important options are

EQUATION=*name of the equation to use for forecasting*
FROM=*starting period of the forecasts*
TO=*ending period of the forecasts*
STEPS=*number of forecast steps to compute*
ERRORS=*series containing the forecast errors*
STDERRS=*series of standard errors of the forecasts*
PRINT/[**NOPRINT**]

Note that the default is to *not* print the forecasts.

Notice that **UFORECAST** requires the name of a previously estimated equation to use for forecasting. Whenever you estimate a linear regression using **LINREG** or an equation using **BOXJENK**, use the option to **DEFINE** the model. **UFORECAST** uses this previously defined model for forecasting purposes. If you omit the **EQUATION**= option, **UFORECAST** will use the most recently estimated equation. In Example 2.6 (file **RPM2_6.RPF**), we'll take the model chosen in Section 2.7 and use it to forecast the logarithmic change of the PPI two years beyond the end of the sample (2012:4).

We need to define an **EQUATION** usable for forecasting when we estimate the model, so we add the **DEFINE** option to the **BOXJENK**.

```
boxjenk(constant,ar=||1,3||,define=ar1_3) dly
```

In order to forecast, the equation is defined as **AR1_3**. Now **UFORECAST** can be used to instruct RATS to produce the 1- through 8-step ahead forecasts using

```
uforecast(equation=ar1_3,print) forecasts 2013:1 2014:4
```

Entry	DLY
2013:01	0.003903092
2013:02	0.007167626
2013:03	0.007082460
2013:04	0.006769988
2014:01	0.007357279
2014:02	0.007617198
2014:03	0.007670298
2014:04	0.007827904

You can get the same results with

```
ufore(equation=ar1_3,print,steps=8) forecasts
```

as **UFORECAST** by default starts the forecasts one period beyond the (most recent) estimation range, so this requests 8 forecast steps. You pick the combination of **STEPS**, **FROM** and **TO** options, or the *start* and *end* parameters that you find most convenient.

Although the **FORECAST** instruction is typically used for multiequation forecasts, it can also forecast using a single equation. Subsequent chapters will consider **FORECAST** in more detail. For now, it suffices to indicate that the identical output can be obtained using

```
forecast(print,steps=8,from=2013:1) 1
# ar1_3 forecasts
```

If you are going to forecast, there is an important difference between the following two instructions:

```
boxjenk(constant,ar=||1,3||,define=ar1_3) dly
```

and

```
boxjenk(define=ar_alt,constant,ar=||1,3||,dif=1) ly
```

The coefficients are identical, the fits are identical, but in the second case, the original dependent variable is **LY**, not its difference **DLY**, and the equation it produces is one that has **LY** as the dependent variable, so the forecasts are of **LY**, not of **DLY**. You can use **DISPLAY** to look at the two equations:

```
disp ar1_3 ar_alt
```

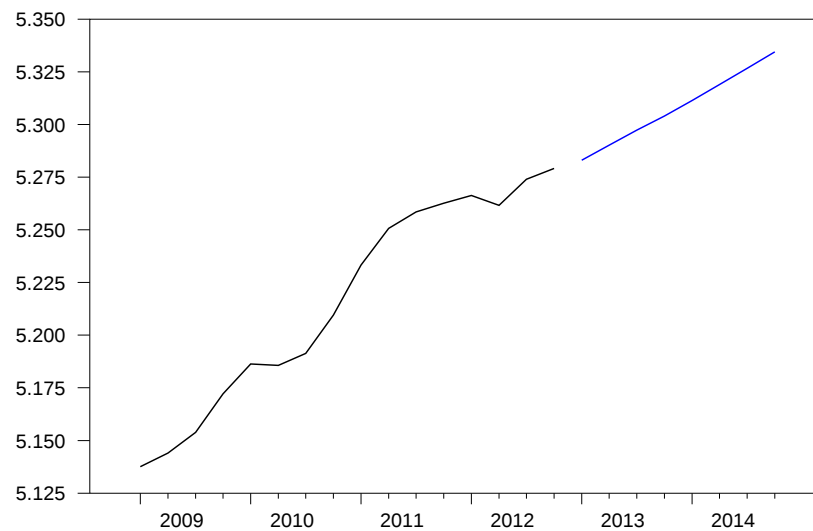


Figure 2.6: Actual and Forecasted Values of log PPI

Dependent Variable DLY		
Variable		Coeff

1. Constant		0.0025242767
2. DLY{1}		0.4752570939
3. DLY{3}		0.2253908909
Dependent Variable LY		
Variable		Coeff

1. Constant		0.002524277
2. LY{1}		1.475257094
3. LY{2}		-0.475257094
4. LY{3}		0.225390891
5. LY{4}		-0.225390891

You can verify that these are identical if you substitute out $DLY = LY - LY1$. We can forecast LY itself and graph the forecasts (with the final four years of actual data, Figure 2.6) using:

```
ufore(equation=ar_alt,print) forecasts 2013:1 2014:4
graph(footer="Actual and Forecasted Values of the log PPI") 2
# ly 2009:1 *
# forecasts
```

2.11 Examining the Forecast Errors

Given that the forecasts will contain some error, it would be desirable to have a model that produces unbiased forecasts with the smallest possible mean square error. Although it is not really possible to know the forecast errors, it is possible to put alternative models to a head-to-head test. With the PPI series, for example, you could compare the $AR(\{1,3\})$ and $ARMA(1,1)$ specifications by holding

back 50 observations (about 1/4 of the data set) and performing the following steps:

1. Construct two series to contain the forecast errors. For simplicity, we will let `error1` contain the forecast errors from the $AR(\{1,3\})$ and `error2` contain the forecast errors from the $ARMA(1,1)$.
2. For each time period between 2000:2 and 2012:3, estimate the two models from the start of the data set through that time. Construct one-step forecasts and save the errors.

Analyzing these two series can give you insight into which of the two models generates forecast errors with the most desirable properties. For example, if the mean of the forecast errors from the $AR(\{1,3\})$ model is closer to zero than those of the $ARMA(1,1)$, you might prefer to use the AR model to forecast beyond the actual end of the data set (i.e., 2012:4).

It's a good idea to avoid using "hard-coded" dates on the individual instructions if at all possible. If we wrote all this with 2000:2 and 2012:3, for instance, then, if we (or a referee) decided that we really should start in 1998:2 instead, we would have to change several lines and hope that we caught all the places that mattered. So in Example 2.7 (file `RPM2_7.RPF`), after reading the data and creating `DLY` as we've done before, we'll start with:

```
compute dataend=2012:4
compute baseend=dataend-50
```

which define `DATAEND` as the end of the observed data and `BASEEND` as 50 entries earlier. You can even avoid the hard-coded 2012:4 by using the `%ALLOCEND()` function, which gives the entry number of the end of the standard range of the workspace.

The error series are initialized with

```
set error1 baseend+1 * = 0.
set error2 baseend+1 * = 0.
```

This is necessary when you are filling entries of a series one-at-a-time using **COMPUTE** instructions, which is what we will be doing here with the forecast errors—RATS needs to set aside space for the information.

The working instructions are:

```
do end=baseend, dataend-1
  boxjenk(constant, ar=||1,3||, define=ar1_3) dly * end
  boxjenk(constant, ar=1, ma=1, define=arma)    dly * end
  ufore(equation=ar1_3, steps=1) f1
  ufore(equation=arma, steps=1) f2
  compute error1(end+1)=dly(end+1)-f1(end+1)
  compute error2(end+1)=dly(end+1)-f2(end+1)
end do t
```

This runs a loop over the end of the estimation sample, with the variable `END` used to represent that. Why is it through `DATAEND-1` rather than `DATAEND`? If we estimate through `DATAEND`, there is no data to which to compare a one-step-ahead forecast. There wouldn't be any real harm in doing it, since the forecast errors would just be missing values, but it's better to write the program describing what you actually need, rather than relying on the RATS missing value handlers to fix the mistake.

By default, the **UFORECAST** instructions forecast from one period beyond the end of the previous regression range, which is what we want, so we only need to indicate the number of steps. The forecast errors are computed using the actual data `DLY` and the forecasts (`F1` and `F2`) for the period `END+1`.

Exercise 2.3 *The loop above could just as easily have been written with the loop index running over the start of the forecast period rather than the end of the estimation period. What changes would be necessary to do that?*

```
table / error1 error2
```

Series	Obs	Mean	Std Error	Minimum	Maximum
ERROR1	50	-0.0004350166	0.0136450332	-0.0698340471	0.0214472245
ERROR2	50	-0.00041141885	0.0137877256	-0.0709608893	0.0186385371

Notice that the forecast errors from the $AR(\{1,3\})$ model have a larger mean (in absolute value) than those of the $ARMA(1,1)$ model but have a smaller standard error. A simple way to test whether the mean of the forecast errors is significantly different from zero is to regress the actual values on the predicted values. Consider:

```
linreg dly
# constant f1
test(title="Test of Unbiasedness of AR(3) forecasts")
# 1 2
# 0 1
```

If the forecasts are unbiased, the intercept should be equal to zero and the slope coefficient should be equal to unit, which is what the **TEST** instruction is doing. The results are:

Test of Unbiasedness of AR(3) forecasts F(2,48)= 3.32730 with Significance Level 0.04433528
--

Repeating for the ARMA forecasts:

```
linreg dly
# constant f2
test(title="Test of Unbiasedness of ARMA forecasts")
# 1 2
# 0 1
```

gives

Test of Unbiasedness of ARMA forecasts
 $F(2,48) = 3.55490$ with Significance Level 0.03633255

Although both models show some bias, there is a slight preference for the $AR(\{1,3\})$.

Mean Square Forecast Errors

Not only are we concerned about the bias of the forecast errors, most researchers would also want the dispersion of the errors to be as small as possible. Although the standard error of the forecasts from the $AR(\{1,3\})$ model is smaller than that from the $ARMA(1,1)$ model, we might want to know if it is possible to conclude that there is a statistical difference between the two (that is reject the null hypothesis that the variance of the `error1` series is equal to that of the `error2` series).

The Granger-Newbold test (Granger and Newbold (1973)) can be used to test for a difference between the sums of squared forecast errors:

$$d = (SSR_1 - SSR_2)/N$$

where SSR_1 and SSR_2 are the sums of squared forecast errors from models 1 and 2, respectively and N is the number of forecast errors. If the two models forecast equally well, this difference should be zero. Under the assumptions that

1. the forecast errors have the same means and are normally distributed and
2. the errors are serially uncorrelated

Granger and Newbold show that the following has a t -distribution with $N - 1$ degrees of freedom

$$\sqrt{N-1} \frac{r}{\sqrt{1-r^2}}$$

r is the correlation coefficient between x_t and z_t defined as $x_t = e_{1t} + e_{2t}$ and $z_t = e_{1t} - e_{2t}$ and e_{1t} and e_{2t} are the forecast errors from the alternative models. If you reject the null hypothesis $r = 0$, conclude that the model with the smallest residual sum of squares has the smallest mean square forecast errors. Instead of programming the test yourself, it is simpler to use the procedure `@GNEWBOLD`. In this case, the instruction is

```
@gnewbold dly f1 f2
```

giving us

```

Granger-Newbold Forecast Comparison Test
Forecasts of DLY over 2000:03 to 2012:04

Forecast Test Stat P(GN>x)
F1          -0.3295 0.62842
F2           0.3295 0.37158

```

Hence, we do not reject the null hypothesis that the two mean square forecast errors are equal and conclude that the forecast errors from the AR({1,3}) model has the same dispersion as the ARMA(1,1). The first set (the AR model) is slightly better since it's showing the negative test value, but that isn't statistically significant.

Diebold and Mariano (1995) have shown how to modify the Granger-Newbold test for the case in which the forecast errors are serially correlated. Although one-step-ahead forecasts like these ideally *shouldn't* be serially correlated, when we check them with:

```

corr(qstats,span=4) error1
corr(qstats,span=4) error2

```

we find that that doesn't seem to be the case here (note particularly the 4th lag):

```

Correlations of Series ERROR1
Quarterly Data From 2000:03 To 2012:04

Autocorrelations
 1      2      3      4      5      6      7      8      9     10
0.0884  -0.1558  -0.2303  -0.3086  0.0422  0.1078  -0.1068  -0.0497  -0.0260  -0.0212
11      12
0.2272   0.0192

Ljung-Box Q-Statistics
Lags  Statistic Signif Lvl
 4      10.046   0.039667
 8      11.678   0.166162
12      15.215   0.229873

```

The Diebold-Mariano test is done with the `@DMARIANO` procedure, which has similar syntax to `@GNEWBOLD` but requires a `LAGS` option. The procedure does a linear regression with a HAC covariance matrix—we recommend that you also include the option `LWINDOW=NEWKEY`, as the default “truncated” lag window (which was recommended in the original paper) doesn't guarantee a positive-definite covariance matrix estimator.

```
@dmariano(lags=4,lwindow=newey) dly f1 f2
```

```

Diebold-Mariano Forecast Comparison Test
Forecasts of DLY over 2000:03 to 2012:04
Test Statistics Corrected for Serial Correlation of 4 lags
Forecast   MSE      Test Stat P(DM>x)
F1         0.00018265  -0.5723 0.71645
F2         0.00018647   0.5723 0.28355

```

Again, the mean square forecast errors are so similar that null hypothesis of no significant difference is not rejected.

Note that the asymptotics of the Diebold-Mariano test break down if the two models are “nested” (one is a special case of the other). That’s not the case here. A full AR(3) vs the AR({1,3}) *would* be a pair of nested models and couldn’t be compared using a straight Diebold-Mariano test.

2.12 Coefficient Stability

If the model adequately reflects the data generating process, the coefficients should be stable over time. In other words, the coefficients should not change dramatically when we estimate the model over different sample periods. A simple way to check for coefficient stability is to use recursive estimates. That’s what we’ll do in Example 2.8 (file `RPM2_8.RPF`). Consider the following segment of code for the transformed *PPI* series. We first set up the “target” series: two for the recursive estimates of the coefficients (`INTERCEPT` for the constant term and `AR1` for the first autoregressive parameter), and two for their estimated standard errors.

```
set intercept = 0.
set ar1 = 0.
set sd0 = 0.
set sd1 = 0.
```

Next, loop over each entry from 1980:1 to 2012:4, estimating the model through that period and save the four items required. Note again that we add a `NOPRINT` option to the `BOXJENK` so we don’t produce pages of unnecessary output. If, however, you find that something seems to be wrong, don’t hesitate to re-activate the `PRINT` to see what’s happening. Because `BOXJENK` uses an iterative estimation process, this checks whether there is any problem with convergence—that’s unlikely for such a simple model, but it doesn’t hurt to be safe.

```
do end = 1980:1,2012:4
  boxjenk(constant,ar=||1,3||,noprint) dly * end
  com intercept(end) = %beta(1), ar1(end) = %beta(2)
  com sd0(end) = %stderrs(1) , sd1(end) = %stderrs(2)
  if %converged<>1
    dis "###DID NOT CONVERGE for " %datelabel(end)
  end do end
```

When we run this, we don’t get any messages about non-convergence. We can construct confidence intervals around the estimated coefficients by adding and subtracting 1.64 standard deviations¹³ to the estimated coefficient values. Consider:

¹³If you want greater accuracy, you can use `%INVNORMAL(.95)` in place of 1.64.

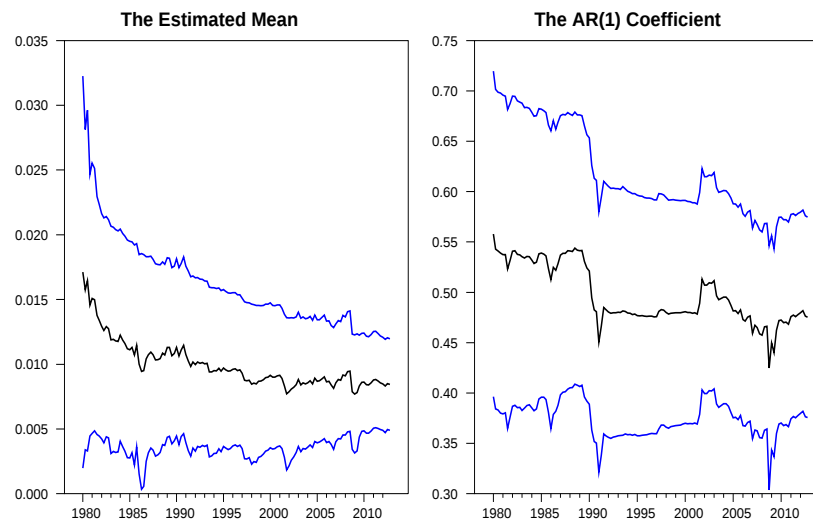


Figure 2.7: Coefficient Estimates with 90% Confidence Intervals

```
set plus0 = intercept + 1.64*sd0
set minus0 = intercept - 1.64*sd0
set plus1 = ar1 + 1.64*sd1
set minus1 = ar1 - 1.64*sd1
```

Now, we can graph the coefficients and the confidence intervals using:

```
spgraph(hfield=2,vfields=1,$
  footer="Coefficient Estimates with 90% Confidence Intervals")
graph(header="The Estimated Mean") 3
# intercept 1980:1 *
# plus0      1980:1 * 2
# minus0     1980:1 * 2
graph(header="The AR(1) Coefficient") 3
# ar1        1980:1 *
# plus1      1980:1 * 2
# minus1     1980:1 * 2
spgraph(done)
```

producing Figure 2.7. The recursive estimates for both coefficients are quite stable over time. Since the earliest estimates use a small number of observations, it is not surprising that the confidence intervals are widest for these early periods. Sometimes it is preferable to estimate recursive regressions using a rolling window instead of an expanding window. With an expanding window the number of observations increases as you approach the end of the sample. With a fixed, or rolling window, you use the same number of observations in each regression. The way to modify the code to have an expanding window is to allow the start date to increase by 1 and the end date to increase by 1 every time through the loop. To modify the code to have a rolling window with—say 75—observations use:

```

compute width=75
do end = 1980:1,2012:4
    boxjenk(constant,ar=||1,3||,noprint) dly end-width+1 end
    com intercept(end) = %beta(1), ar1(end) = %beta(2)
    com sd0(end) = %stderrs(1) , sd1(end) = %stderrs(2)
    if %converged<>1
        dis "DID NOT CONVERGE for t = " %datelabel(end)
    end do end

```

The first time through the loop, the estimation uses the 75 observations from 1961:3 (74 periods before 1980:1), the second time through observations from 1961:4 to 1980:2, and so on.

For models which are, in fact, linear in the parameters (like this one), another way to obtain the recursive estimates with an expanding window is to use the RATS instruction **RLS** (for Recursive Least Squares). The syntax is:

```

RLS( options ) series start end resids
# list of explanatory variables in regression format

```

The principal options are:

```

EQUATION=equation to estimate
COHISTORY=VECTOR[SERIES] of coefficient estimates
SEHISTORY=VECTOR[SERIES] of coefficient standard errors

```

Hence, similar output to that reported above can be obtained using:

```

rls(cohist=coeffs,sehist=serrs,equation=ar1_3) dly
set plus0 = coeffs(1) + 1.64*serrs(1)
set minus0 = coeffs(1) - 1.64*serrs(1)
set plus1 = coeffs(2) + 1.64*serrs(2)
set minus1 = coeffs(2) - 1.64*serrs(2)

```

There are several differences between **RLS** and what you get by the **DO** loop method:

1. The residuals in the first method (either `%RESIDS` or the series saved by the *resids* parameter) are recomputed for the full sample each time through. **RLS** produces *recursive residuals* where the time t residual is the (standardized) predictive error for period t given the previous estimates, and $t - 1$ and earlier are left alone at time t . Recursive residuals have many nice properties for analyzing model stability.
2. Because **BOXJENK** uses a different parameterization for the intercept/mean that a linear regression, those won't be directly comparable. The autoregressive coefficients will be the same however.
3. **RLS** cannot be used with MA terms, so we cannot estimate the ARMA(1,1) model using **RLS**.

2.13 Tips and Tricks

2.13.1 Preparing a graph for publication

You may have noticed that all of our graphs used a `FOOTER` option on the **GRAPH** (if it was stand-alone) or on the outer **SPGRAPH**. This is used in preference to a **HEADER**, which is generally only used in inner graphs for **SPGRAPH** setups. You may also have noticed that in almost all cases, that footer was gone in the version that we included in the book. Most publications will put their *own* caption on a graphic, so you probably don't want something similar to show up in the graph itself. Since the graphic labeling is usually below, the footer, rather than header, comes closest to the final appearance. The footer also uses a smaller font, more similar to what will be used.

It wouldn't be a good idea to strip the footer (or header) out of the graph while you're still doing the empirical work. The footer/header is used in the title bar of the graph window, and in the *Window* menu to identify graphs for a closer look. What we do is to use the **GSAVE (NOFOOTER)** instruction which was added with RATS 8.2. This strips the outer footer out of a graph, *but only when it is exported in some way* (either by being exported to a file or to the clipboard with *Edit-Copy*).

2.13.2 Preparing a table for publication

If you check the *Window-Report Windows* menu after running one of the example programs, you'll see anywhere from 1 to 20 reports queued up. These are generated by instructions like **LINREG**, **BOXJENK** or **FORECAST** or procedures like **@BJDIFF** or **@REGCRITS**. The ones at the top of the list will be the last ones created. Note that these are only created if you `PRINT` the output; if you do `NOPRINT`, RATS saves time by not formatting up the reports. If you select one of the reports, it will load it into a window. However, unlike the standard output that goes into the text-based *output window*, this is organized into a spreadsheet-like table of rows and columns. And, even though you can't see it, this has the full precision at which the calculations were done.

We will show later in the course how to use the RATS **REPORT** instruction to generate a table with the specific information that you want, but in many cases, you may be able to get by using just the re-loaded standard formatted report. Any "text-based" copy and paste operation (to TeX or format like comma-delimited) will copy the numbers with the rounding shown in the window. (Excel and similar formats will get full-precision). You can reformat any contiguous block of cells to show whatever numerical format you want. Select the cells you want to reformat, and choose *Edit-Change Layout*, or *Reformat* on the right-click menu. Then select the cells you want to export and copy-and-paste or export to a file. Note that *Copy- $\dot{\iota}$ TeX* copies a TeX table into the clipboard, so you can paste into a TeX document.

Example 2.1 Introduction to basic instructions

```

cal(q) 1960:1
all 2012:4
*
open data quarterly(2012).xls
data(org=obs,format=xls)
table(picture="*.##")

set dlrgdp = log(rgdp) - log(rgdp{1})
set dlm2    = log(m2) - log(m2{1})
set drs     = tb3mo - tb3mo{1}
set dr1     = tb1yr - tb1yr{1}
set dlp     = log(deflator) - log(deflator{1})
set dlppi   = log(ppi) - log(ppi{1})

spgraph(footer="Graphs of the Series",hfields=2,vfields=2)
  graph(header="Panel 1: The Interest Rates",key=below,nokbox) 2
  # tb3mo
  # tb1yr
  graph(header="Panel 2: Real and Potential GDP",key=upleft) 2
  # rgdp
  # potent
  graph(header="Panel 3: Time path of money growth",noaxis) 1
  # dlm2
  graph(header="Panel 4: Time path of Inflation",noaxis) 1
  # dlp
spgraph(done)
*
linreg drs / resids
# constant drs{1 to 7}
*
corr(number=24,partial=partial,qstats,span=4,pic="##.###") resids
graph 1
# resids

exclude
# drs{5 to 7}

summarize
# drs{5 to 7}

test
# 6 7 8
# 0.1 0.1 0.1

test
# 1 2 3 4
# 0. 0.4 -0.1 0.4

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

```

```
# 1. 1. 0.  
# 5 6  
# 1. 1. 0.
```

Example 2.2 Engle-Granger test with lag length selection

```

cal(q) 1960:1
all 2012:4
*
open data quarterly(2012).xls
data(org=obs, format=xls)
*
set dlrgdp = log(rgdp) - log(rgdp{1})
set dlm2    = log(m2) - log(m2{1})
set drs     = tb3mo - tb3mo{1}
set dr1     = tb1yr - tb1yr{1}
set dlp     = log(deflator) - log(deflator{1})
set dlppi   = log(ppi) - log(ppi{1})
*
* Estimate "spurious regression"
*
linreg tb1yr / resids
# constant tb3mo

corr(num=8, results=cors, partial=partial, picture="##.###", qstats) resids

graph(nodates, number=0, style=bar, key=below, footer="ACF and PACF") 2
# cors
# partial
*
* Do E-G test with fixed lags
*
diff resids / dresids
linreg dresids
# resids{1} dresids{1 to 8}
*
* Do E-G test with different lag lengths
*
compute egstart=%regstart()
do i = 0, 8
    linreg(noprint) dresids egstart *
    # resids{1} dresids{1 to i}
    com aic = -2.0*%logl + %nreg*2
    com sbc = -2.0*%logl + %nreg*log(%nobs)
    dis "Lags: " i "T-stat" %tstats(1) "The aic = " aic " and sbc = " sbc
end do i

linreg dresids
# resids{1} dresids{1 to 6}
@regcrits
@regcorrs(number=24, qstats, report)
*
@egtest(lags=8, method=aic)
# tb1yr tb3mo

```

Example 2.3 Estimation and diagnostics on ARMA models

```

cal(q) 1960:1
all 2012:4
*
open data quarterly(2012).xls
data(org=obs,format=xls)
*
log ppi / ly
dif ly / dly
*
spgraph(footer="Price of Finished Goods",hfield=2,vfield=1)
      graph(header="Panel a: Quarterly Growth Rate") 1
      # dly
      @bjident(separate,number=12) dly
spgraph(done)
*
boxjenk(constant,ar=3) dly
boxjenk(constant,ar=||1,3||) dly
corr(number=8,qstats,span=4,dfc=%narma,picture=".#.#.###") %resids
@regcorrs
com aic = -2.0*%logl + %nreg*2
com sbc = -2.0*%logl + %nreg*log(%nobs)
display "aic = " aic "bic = " sbc

boxjenk(constant,ar=1,ma=1) dly 1961:1 *
com aic = -2.0*%logl + %nreg*2
com sbc = -2.0*%logl + %nreg*log(%nobs)
display "aic = " aic "bic = " sbc

corr(number=8,qstats,span=4,dfc=%narma,picture=".#.#.###") %resids
@regcorrs(number=8,qstats,dfc=%narma,footer="ARMA(1,1) Model")

```

Example 2.4 Automated Box-Jenkins model selection

```
cal(q) 1960:1
all 2012:4
*
open data quarterly(2012).xls
data(org=obs,format=xls)

log ppi / ly
dif ly / dly

do q=0,3
  do p=0,3
    boxjenk(noprint,constant,ar=p,ma=q) dly 1961:1 *
    com aic=-2*%logl+%nreg*2
    com sbc=-2*%logl+%nreg*log(%nobs)
    disp "Order("+p+", "+q+") " "AIC=" aic "SBC=" sbc "OK" %converged
  end do p
end do q
*
@bjautofit(constant,pmax=3,qmax=3,crit=aic) dly
```

Example 2.5 Seasonal Box-Jenkins Model

```

cal(q) 1960:1
all 2012:4
open data quarterly(2012).xls
data(org=obs, format=xls)

set ly = log(curr)
dif ly / dly
dif(sdiffs=1,dif=1) ly / m

spgraph(footer="ACF and PACF of dly and m",hfields=2,vfields=1)
  @bjident dly
  @bjident m
spgraph(done)

@bjdiff(diff=1,sdiffs=1,trans=log) curr

boxjenk(noprint,constant,ar=1,sma=1) m 1962:3 *
@regcorrs(title="(1,1,0)x(0,1,1)",dfc=%narma)
display "aic = " %aic "bic = " %sbc "Q(signif)" *.### %qsignif
*
boxjenk(noprint,constant,ma=1,sma=1) m 1962:3 *
@regcorrs(title="(0,1,1)x(0,1,1)",dfc=%narma)
display "aic = " %aic "bic = " %sbc "Q(signif)" *.### %qsignif
*
boxjenk(noprint,constant,ar=1,sar=1) m 1962:3 *
@regcorrs(title="(1,1,0)x(1,1,0)",dfc=%narma)
display "aic = " %aic "bic = " %sbc "Q(signif)" *.### %qsignif
*
boxjenk(noprint,constant,ma=1,sar=1) m 1962:3 *
@regcorrs(title="(0,1,1)x(1,1,0)",dfc=%narma)
display "aic = " %aic "bic = " %sbc "Q(signif)" *.### %qsignif
*
* Do estimation with output for the preferred model
*
boxjenk(print,constant,ar=1,sma=1) m 1962:3 *
@regcorrs(title="(1,1,0)x(0,1,1)",dfc=%narma)
display "aic = " %aic "bic = " %sbc "Q(signif)" *.### %qsignif
*
@gmautofit(regular=1,seasonal=1,noconst,full,report) m

```

Example 2.6 Out-of-sample forecasts with ARIMA model

```

cal(q) 1960:1
all 2012:4
*
open data quarterly(2012).xls
data(org=obs,format=xls)

log ppi / ly
dif ly / dly

boxjenk(constant,ar=||1,3||,define=ar1_3) dly
ufore(equation=ar1_3,print) forecasts 2013:1 2014:4
ufore(equation=ar1_3,print,steps=8) forecasts
*
forecast(print,steps=8,from=2013:1) 1
# ar1_3 forecasts
*
* Forecasts of log PPI (not differences)
*
boxjenk(define=ar_alt,constant,ar=||1,3||,dif=1) ly
*
disp ar1_3 ar_alt
*
ufore(equation=ar_alt,print) forecasts 2013:1 2014:4
graph(footer="Actual and Forecasted Values of the log PPI") 2
# ly 2009:1 *
# forecasts

```

Example 2.7 Comparison of Forecasts

```

cal(q) 1960:1
all 2012:4
*
open data quarterly(2012).xls
data(org=obs,format=xls)
*
set ly      = log(ppi)
set dly     = ly-ly{1}
*
compute dataend=2012:4
compute baseend=dataend-50
*
set error1 baseend+1 * = 0.
set error2 baseend+1 * = 0.
do end=baseend,dataend-1
    boxjenk(constant,ar=||1,3||,define=ar1_3) dly * end
    boxjenk(constant,ar=1,ma=1,define=arma)    dly * end
    ufore(equation=ar1_3,steps=1) f1
    ufore(equation=arma,steps=1) f2
    compute error1(end+1)=dly(end+1)-f1(end+1)
    compute error2(end+1)=dly(end+1)-f2(end+1)
end do t
table / error1 error2
*
linreg dly
# constant f1
test(title="Test of Unbiasedness of AR(3) forecasts")
# 1 2
# 0 1
*
linreg dly
# constant f2
test(title="Test of Unbiasedness of ARMA forecasts")
# 1 2
# 0 1
*
* Granger-Newbold and Diebold-Mariano tests
*
@gnewbold dly f1 f2
*
corr(qstats,span=4) error1
corr(qstats,span=4) error2
*
@dmariano(lags=4,lwindow=newey) dly f1 f2

```

Example 2.8 Stability Analysis

```

cal(q) 1960:1
all 2012:4
*
open data quarterly(2012).xls
data(org=obs,format=xls)
*
set ly      = log(ppi)
set dly     = ly-ly{1}
*
set intercept = 0.
set ar1 = 0.
set sd0 = 0.
set sd1 = 0.
*
do end = 1980:1,2012:4
    boxjenk(constant,ar=||1,3||,noprint) dly * end
    com intercept(end) = %beta(1), ar1(end) = %beta(2)
    com sd0(end) = %stderrs(1) , sd1(end) = %stderrs(2)
    if %converged<>1
        dis "###DID NOT CONVERGE for " %datelabel(end)
end do end
*
set plus0 = intercept + 1.64*sd0
set minus0 = intercept - 1.64*sd0
set plus1 = ar1 + 1.64*sd1
set minus1 = ar1 - 1.64*sd1
*
spgraph(hfield=2,vfields=1,$
    footer="Coefficient Estimates with 90% Confidence Intervals")
    graph(header="The Estimated Mean") 3
    # intercept 1980:1 *
    # plus0      1980:1 * 2
    # minus0     1980:1 * 2
graph(header="The AR(1) Coefficient") 3
    # ar1        1980:1 *
    # plus1      1980:1 * 2
    # minus1     1980:1 * 2
spgraph(done)
*
* Alternatively, use a rolling window with 75 observations
*
compute width=75
do end = 1980:1,2012:4
    boxjenk(constant,ar=||1,3||,noprint) dly end-width+1 end
    com intercept(end) = %beta(1), ar1(end) = %beta(2)
    com sd0(end) = %stderrs(1) , sd1(end) = %stderrs(2)
    if %converged<>1
        dis "DID NOT CONVERGE for t = " %datelabel(end)
end do end
*
equation ar1_3 dly
# constant dly{1 3}

```

```
*  
rls(cohist=coeffs,sehist=serrs,equation=ar1_3) dly / resids  
set plus0 = coeffs(1) + 1.64*serrs(1)  
set minus0 = coeffs(1) - 1.64*serrs(1)  
set plus1 = coeffs(2) + 1.64*serrs(2)  
set minus1 = coeffs(2) - 1.64*serrs(2)
```

Non-linear Least Squares

It is well-known that many economic variables display asymmetric adjustment over the course of the business cycle. The recent financial crisis underlines the point that economic downturns can be far sharper than recoveries. Yet, the standard ARMA(p, q) model requires that all adjustment be symmetric, as it is linear in all lagged values of $\{y_t\}$ and $\{\varepsilon_t\}$. For example, in the AR(1) model $y_t = 0.5y_{t-1} + \varepsilon_t$, a one-unit shock to ε_t will induce y_t to increase by one unit, y_{t+1} to increase by 0.5 units, y_{t+2} to increase by 0.25 units, and so on. And a one-unit *decrease* in ε_t will induce y_t to decrease by one unit, y_{t+1} to decrease by 0.5 units, and y_{t+2} to decrease by 0.25 units. Doubling the magnitude of the shock doubles the magnitude of the change in y_t and in all subsequent values of the sequence. The point is that in a linear specification, such as the ARMA(p, q) model, it isn't possible to capture the types of asymmetries displayed by many time-series variables. As such, there is a large and growing literature on non-linear alternatives to the standard ARMA model. RATS allows you to estimate dynamic nonlinear models in a number of different ways including non-linear least squares, which is the subject of this chapter.

In general, non-linear estimation requires more work from you and more careful attention to detail than does linear regression. You really have to try very hard to construct an example where the **LINREG** instruction fails to get the “correct” answer, in the sense that it gives a result other than the coefficients which minimize the sum of squares to any reasonable level of precision—for linear regressions, most statistical packages agree to at least eight significant digits even on fairly “difficult” data. This doesn't mean that the results make *economic* sense, but you at least get results.

However, there are various “pathologies” which can affect non-linear models that never occur in linear ones.

Boundary Issues

Probabilities have to be in $[0, 1]$. Variances have to be non-negative. These are just two examples of possible non-linear parameters where the optimum might be at a boundary. If the optimum *is* at the boundary, the partial derivative doesn't have to be zero. Since the most straightforward method of optimizing a continuous function is to find a zero of the gradient, that won't work properly with the bounded spaces. In addition, the derivative may not even exist at the optimum if the function isn't definable in one direction or the other.

Unbounded parameter space

If the X matrix is full-rank, the sum of squares surface for linear least squares is globally concave and goes to infinity in every direction as the coefficients get very large. However, a non-linear function like $\exp(-x_t\beta)$ is bounded below by zero no matter how large β gets. It's possible for an optimum to be at $\beta = \inf$. This is often related to the boundary issue, as bounded parameters are sometimes mapped into an unbounded parameter space. For instance, σ^2 can be replaced with $\exp(\kappa)$ where the boundary $\sigma^2 = 0$ is now mapped to $\kappa = -\infty$.

Convergence issues

Assuming that X is full rank, least squares can be solved exactly with a single matrix calculation. Even if a non-linear least squares problem avoids the previous two issues, the minimizer can rarely be computed analytically with a finite number of calculations. Instead, the solution has to be approximated, and at some point we have to decide when we're "done". Because of its importance, we're devoting a full section to it (Section 3.4).

Lack of identification

A linear model *can* have identification issues—the dummy variable trap is a good example—but they are usually the result of an error in specifying the set of regressors. Generally, you can test for additional coefficients by “overfitting” a model (adding additional regressors) without any major computational problems. By contrast, there are whole classes of non-linear models where entire sets of parameters can, under certain circumstances, fail to be identified. In particular, various “switching” and “threshold” models can fail if you try to overfit by adding an extra (and unnecessary) regime. This will come up quite often in this chapter.

3.1 Nonlinear Least Squares

Suppose that you want to estimate the following model using nonlinear least squares:

$$y_t = \beta x_t^\gamma + \varepsilon_t \quad (3.1)$$

Since the disturbance term is additive, you cannot simply take the log of each side and estimate the equation using OLS.¹ However, nonlinear least squares allows you to estimate β and γ directly, finding the values of the parameters which minimize

$$\sum_{t=1}^T (y_t - \beta x_t^\gamma)^2 \quad (3.2)$$

¹If the model had the form $y_t = \beta x_t^\gamma \varepsilon_t$ where $\{\varepsilon_t\}$ was log-normal, it *would* be appropriate to estimate the regression in logs using **LINREG**.

The instruction which does the minimization is **NLLS** (for NonLinear Least Squares). However, before we use it, we must go through several preliminary steps. Instead of a linear function of explanatory variables, we need to allow for a general function of the data on the right-side of the equation as in (3.1). This is done using the instruction **FRML**.

However, before we can even define the explanatory **FRML**, we need to let RATS know the variable names that we are going to use in defining that, translating the math equation into a usable expression. That is done with the **NONLIN** instruction, which both defines the variables to RATS and also creates the parameter set to be used in estimation. And there is one *more* step before we can use **NLLS**—we need to give “guess values” to those parameters. That’s not necessary with a linear regression, where a single matrix calculation solves the minimization problem. Non-linear least squares requires a sequence of steps, each of which brings us closer to the minimizers, but we have to start that sequence somewhere. Sometimes the estimation process is almost completely unaffected by the guess values; in other cases, you will get nowhere without a very good starting point.

The obvious choices for the names of the two parameters would be **BETA** and **GAMMA**. So the first instruction would be

```
nonlin beta gamma
```

The following defines a **FRML** named **F1** with dependent variable **Y** and explanatory function $\text{BETA} * X^{\text{GAMMA}}$.

```
frml f1 y = beta*x^gamma
```

A **FRML** is a function of the (implied) entry variable **T**, so, for instance, **F1(100)** evaluates to $\text{BETA} * X(100)^{\text{GAMMA}}$. Note that if we had not done the **NONLIN** instruction first, the **FRML** instruction would have no idea what **BETA** and **GAMMA** were supposed to be. You would get the message:

```
## SX11. Identifier BETA is Not Recognizable. Incorrect Option Field or Parameter Order?
```

If you get a message like that while setting up a non-linear estimation, you probably either have a typo in the name, or you failed to create all the parameters before defining the **FRML**.

By default, RATS will use 0.0 for any non-linear parameter which isn’t otherwise initialized. Often, that’s OK; here, not so. If $\beta = 0$, γ has no effect on the function value. In this case, it turns out that RATS can fight through that² but it’s not a good strategy to ignore the need for guess values and hope that it

² β changes on the first iteration while γ doesn’t, after which, with a non-zero β , estimation proceeds normally. However, quite a few statistical programs would quit on the first iteration.

works. Since we won't be estimating this model, we'll just give β an arbitrary value, and start γ at 1.

```
compute beta=0.5, gamma=1.0
```

Finally the parameters are estimated with

```
nlls(frml=f1) y
```

How does **NLLS** work? The first thing it does is to see which entries can be used in estimation by evaluating the input `FRML` at each entry and seeing if it gets a (legal) value. **NLLS** has the standard `start` and `end` parameters and `SMPL` option to allow you to control the estimating range, but it also needs to test for problems itself. This is one place where bad guess values, or a generally bad setup might give a bad outcome. While it wouldn't happen here, it's possible to get the message:

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

which is telling you that the explanatory function has *no* entries at which it could be computed, generally due to either missing values working their way through the data set, or something like log or square root of a negative number being part of the formula at the guess values.

The next step (under the default method of estimation) is to take a first order Taylor series expansion of (3.1) with respect to the parameters.

$$y_t - \beta x_t^\gamma \approx x_t^\gamma (\beta^* - \beta) + (\beta x_t^\gamma \log x_t) (\gamma^* - \gamma) + \varepsilon_t^*$$

If we treat the unstarred β and γ as fixed, then this is in the form of a linear regression of the current residuals $y_t - \beta x_t^\gamma$ on the two derivative series x_t^γ and $\beta x_t^\gamma \log x_t$ to get $(\beta^* - \beta)$ and $(\gamma^* - \gamma)$. Going from (β, γ) to (β^*, γ^*) is called taking a *Gauss-Newton* step. This is repeated with the new expansion point, and the process continues until the change is small enough that the process is considered to be converged.

However, **NLLS** doesn't always take a full Gauss-Newton step. It's quite possible that on the first few Gauss-Newton steps, the sum of squares function actually increases on a full step. This is because the first order expansion may not yet be very accurate. While the G-N algorithm may actually work (and work well) despite taking steps that are "too large", **NLLS** instead will take a shorter step in the same direction as the full step so that the sum of squares decreases, adopting a slower but steadier approach to optimization.

Note that this first G-N step is where the $\beta = 0$ guess creates a problem since it zeroes out the derivative with respect to γ . What **NLLS** does with that is to

not try to solve for $\gamma^* - \gamma$ (since there's no useful information for that) and just solve for $\beta^* - \beta$. On the second step, β is no longer zero, so it's possible to move both parameters.

3.2 Using NLLS

We'll use the data set to do several examples of non-linear least squares using something similar to the simple model from the previous section. These are both in Example 3.1). The first of these will be a “nonsense” regression, to show how having a (theoretically) unidentified parameter can affect your estimation.

This first example will be a regression of inflation on its lag plus a power function on the lag of real GDP:

$$\pi_t = b_0 + b_1\pi_{t-1} + b_2y_{t-1}^\gamma + \varepsilon_t \quad (3.3)$$

There is no particular reason to believe that the *level* of GDP has any effect on inflation (the growth rate of GDP would be a different story), so we really wouldn't expect b_2 to be non-zero. But if b_2 is zero, γ isn't identified. However, this is only a theoretical lack of identification—in sample, b_2 won't be exactly zero, so we *will* be able to estimate γ , if not very well.

The second example (which will be discussed in detail in Section 3.3) will use the two interest rates, estimating an exponential rather than linear relationship:

$$LR_t = a_0 + a_1LR_{t-1} + a_2SR_{t-1}^\delta + \varepsilon_t$$

where LR is the “long” rate (one year) and SR the short (three month).

Step 1-Define parameter set

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

```
NONLIN parameter list
```

In most instances, the *parameter list* will be a simple list of the coefficients to be estimated, separated by spaces. For our first model (3.3), this (with the obvious choices for names) would be

```
nonlin b0 b1 b2 gamma
```

One **NONLIN** controls the parameter set until another **NONLIN** is executed, so we don't want to define parameter set for the second problem until we're done with the first. By the way, there is no reason that we couldn't have used the same set of parameter names in the second problem as the first—we're using different ones here for illustration, since the two models have nothing to do with each other. In practice, where the different models are probably much

more closely related, you would largely keep the same parameter set and make adjustments to it from one estimation to the next.

We'll see later that you can define `PARMSET` variables that save a parameter list like this so you can easily switch between sets of parameters.

Step 2-Define FRML

This defines the explanatory formula. The syntax for **FRML** (as it is generally used in non-linear least squares is):

```
frml (options) formula_name depvar = function(t)
```

where:

formula_name The name you choose to give to the formula

depvar Dependent variable

function(t) The explanatory function

For the first example, this would be

```
frml pi pi = b0+b1*pi{1}+b2*y{1}gamma
```

where the `PI` and `Y` series are assumed to have already been defined.

Step 3-Set Guess Values

An obvious set of guess values would be to take `B0`, `B1` and `B2` from a linear regression with `Y{1}` as the third variable, which gives the sum of squares minimizers for the case where `GAMMA` is 1. Could we get by with something less “accurate”? For this model, probably yes. For a more complicated model, possibly not. For a (non-linear) model to be useful in practice, it's important that there be a reasonable way to get guess values for the parameters using more basic models applied to the data plus “prior knowledge” from other datasets or similar models. A model which can only be estimated if fed the best from the results of dozens of attempts at guess values is unlikely to be useful in practice, since it can't be applied easily to different data sets.

For the first example, we would do this with:

```
linreg pi  
# constant pi{1} y{1}  
compute b0=%beta(1),b1=%beta(2),b2=%beta(3),gamma=1.0
```

Note that `%BETA` gets redefined by `NLLS`, so you need to get those as soon as possible after the `LINREG`.

Step 4-Estimate using NLLS

The syntax for **NLLS** is

```
NLLS (frml=formula name,...) depvar start end residuals
```

depvar Dependent variable used on the **FRML** instruction.

start end Range to estimate.

residuals Series to store the residuals. This is optional. %RESIDS is always defined by **NLLS**.

The principal options are:

METHOD=[GAUSS]/SIMPLEX/GENETIC

GAUSS is the (modified) Gauss-Newton algorithm described on page 66. SIMPLEX and GENETIC are slower optimizers which don't use the special structure of the non-linear least squares problem, but are more robust to bad guess values. Gauss-Newton tends to work fine, but these are available if you get convergence problems.

ITERATIONS=*maximum number of iterations to make* [100]

ROBUSTERRORS/[NOROBUSTERRORS]

As with **LINREG**, this option calculates a consistent estimate of the covariance matrix in the presence of heteroscedasticity.

NLLS defines most of the same internal variables as **LINREG** including %RSS, %BETA, %TSTATS and %NOBS. It also defines the internal variable %CONVERGED which is 1 if the estimation converged and otherwise is 0.

With the variable definitions (which need to be done at some point before the **FRML** instruction)

```
set pi  = 100.0*log(ppi/ppi{1})
set y   = .001*rgdp
```

we can estimate the non-linear least squares model with

```
nlls(frml=pi f) pi
```

which gives us

Nonlinear Least Squares - Estimation by Gauss-Newton				
Convergence in 61 Iterations. Final criterion was 0.0000067 <= 0.0000100				
Dependent Variable PI				
Quarterly Data From 1960:02 To 2012:04				
Usable Observations	210			
Degrees of Freedom	206			
Skipped/Missing (from 211)	1			
Centered R ²	0.3183981			
R-Bar ²	0.3084719			
Uncentered R ²	0.5576315			
Mean of Dependent Variable	0.8431326891			
Std Error of Dependent Variable	1.1492476234			
Standard Error of Estimate	0.9556932859			
Sum of Squared Residuals	188.15002928			
Regression F(3,206)	32.0764			
Significance Level of F	0.0000000			
Log Likelihood	-286.4410			
Durbin-Watson Statistic	2.1142			
Variable	Coeff	Std Error	T-Stat	Signif

1. B0	0.446055502	0.243426525	1.83240	0.06833506
2. B1	0.556482217	0.057868679	9.61629	0.00000000
3. B2	-0.000876438	0.017920471	-0.04891	0.96104072
4. GAMMA	2.072568472	7.640418224	0.27126	0.78645982

A few things to note about this:

1. This shows 1 Skipped/Missing Observation. If you look at the output for the **LINREG** used for the guess values (not shown here), you'll see that it gets the same 210 usable observations, but doesn't show any as being skipped. This is due to the difference in how the two instructions work if you let the program choose the estimation range. **LINREG** scans all the series involved in the regression to figure out the maximum usable range. **NLLS** initially restricts the range based upon the one input series that it knows about (the dependent variable), then tries to evaluate the **FRML** at the points in that range, knocking out of the sample any at which it can't evaluate it. It ends up actually using the range from 1960:3 to 2012:4, but accounts for it differently.
2. The second line in the output shows the actual iteration count. 61 is quite a few iterations for such a small model. That's mainly due to the problem figuring out the poorly-estimated power term.

The **GAMMA** is showing the signs of a parameter which isn't really identified. Even a one standard deviation range runs from roughly -5 to 10, most of which are rather nonsensical values.

You might ask why **Y** was defined as

```
set y = .001*rgdp
```

rather than just **RGDP** alone. For a linear regression, a re-scaling like that has no real effect on the calculation of the estimates—in effect, the data get standardized as part of the process of inverting the $X'X$. The only effect is

on how the estimates look when displayed: if we use RGDP alone, the linear regression would give

Linear Regression - Estimation by Least Squares				
Dependent Variable PI				
Quarterly Data From 1960:03 To 2012:04				
Usable Observations	210			
Degrees of Freedom	207			
Centered R ²	0.3179162			
R-Bar ²	0.3113260			
Uncentered R ²	0.5573187			
Mean of Dependent Variable	0.8431326891			
Std Error of Dependent Variable	1.1492476234			
Standard Error of Estimate	0.9537190610			
Sum of Squared Residuals	188.28306978			
Regression F(2,207)	48.2409			
Significance Level of F	0.0000000			
Log Likelihood	-286.5152			
Durbin-Watson Statistic	2.1147			
Variable	Coeff	Std Error	T-Stat	Signif

1. Constant	0.496236135	0.175785276	2.82297	0.00522240
2. PI{1}	0.557330554	0.057684919	9.66163	0.00000000
3. Y{1}	-0.000016091	0.000019699	-0.81684	0.41495922

A good practitioner would try to avoid reporting a regression like this which requires either scientific notation or a large number of digits in order to show the coefficient on $Y\{1\}$. However, everything about the regression matches *exactly* with and without the rescaling except for the display of that last coefficient and its standard error.

Proper scaling of the data makes a much greater difference in non-linear estimation for more reasons than just how the estimates look. In many cases, the only way to get proper behavior is to rescale the data (or sometimes reparameterize the whole model). Theoretically, if you could do all calculations to “infinite” precision, this wouldn’t be an issue. However, the standard at this point in statistical calculations is “double precision” (64-bit representation with about 15 significant digits) typically with intermediate calculations done to a somewhat higher precision by the microprocessor.³ If we look at the results from **NLLS** with the rescaled RGDP series, and think about what would happen if we used it *without* rescaling, y_{t-1}^* would be higher by a factor of more than 10^6 , and b_2 would correspondingly have to be divided by more than 10^6 , making it on the order of 10^{-9} . In the Gauss-Newton algorithm (or any iterative procedure), there’s always a question of when to stop—when should we consider that we’ve done the best that we reasonably can. When you have a parameter with a tiny scale like that, it’s hard to tell whether it’s small because it’s naturally small and small changes in it may still produce observable changes in the function value, or it’s small because it’s really (machine-)zero and small changes won’t have an effect. It’s not the scale of the data itself that’s the problem, but the scale of the parameters that result from the scale of the data.

³See this chapter’s *Tips and Tricks* (page 101) for more on computer arithmetic.

In this case, the **NLLS** on the data without scaling down RGDP doesn't converge at 100 iterations, but does if given more (`ITERS=200` on the **NLLS** is enough), and does give *roughly* the same results for everything other than `B2`. In other cases, you may never be able to get convergence without reworking the data a bit.

3.3 Restrictions: Testing and Imposing

We'll now do the second example of non-linear least squares:

$$LR_t = a_0 + a_1 LR_{t-1} + a_2 SR_{t-1}^\delta + \varepsilon_t \quad (3.4)$$

The setup here is basically the same as before with renaming of variables:

```
nonlin a0 a1 a2 delta
linreg tb1yr
# constant tb1yr{1} tb3mo{1}
frml ratef tb1yr = a0+a1*tb1yr{1}+a2*(tb3mo{1})^delta
compute a0=%beta(1),a1=%beta(2),a2=%beta(3),delta=1.0
nlls(frml=ratef) tb1yr
```

Nonlinear Least Squares - Estimation by Gauss-Newton				
Convergence in 11 Iterations. Final criterion was 0.0000089 <= 0.0000100				
Dependent Variable TB1YR				
Quarterly Data From 1960:01 To 2012:04				
Usable Observations	211			
Degrees of Freedom	207			
Skipped/Missing (from 212)	1			
Centered R ²	0.9445648			
R-Bar ²	0.9437614			
Uncentered R ²	0.9864386			
Mean of Dependent Variable	5.5835545024			
Std Error of Dependent Variable	3.1851074843			
Standard Error of Estimate	0.7553379512			
Sum of Squared Residuals	118.10083205			
Regression F(3,207)	1175.6969			
Significance Level of F	0.0000000			
Log Likelihood	-238.1723			
Durbin-Watson Statistic	1.5634			
Variable	Coeff	Std Error	T-Stat	Signif

1. A0	-0.083102618	0.233285762	-0.35623	0.72203360
2. A1	0.845299348	0.142798909	5.91951	0.00000001
3. A2	0.302138458	0.203001283	1.48836	0.13817851
4. DELTA	0.719742244	0.319257061	2.25443	0.02521634

The “significant” *t*-statistic on `DELTA` is somewhat misleading because it's a test for $\delta = 0$ and that's not a particularly interesting hypothesis. Of the hypothesis testing instructions from Section 2.2, you can't use **EXCLUDE**, since it uses “regressor lists” to input the variables to be tested, and **NLLS** works with parameter sets instead. **TEST** and **RESTRICT**, which use coefficient positions, are available, and there is a form of **SUMMARIZE** which can be used as well.

In this case, the most interesting hypothesis regarding δ would be whether it's equal to 1. We can use **TEST** for that—DELTA is coefficient 4, so the test would be done with

```
test(title="Test of linearity")
# 4
# 1.0
```

Test of linearity t(207)= -0.877844 or F(1,207)= 0.770609 with Significance Level 0.38104636

so we would conclude that the extra work for doing the power term doesn't seem to have helped much.

SUMMARIZE can also be applied to the results from non-linear least squares. It can be used to test non-linear functions of the parameters, and can also use the delta method (Appendix C) to compute asymptotic variances for non-linear functions. One potentially interesting question about the relationship between the short and long rates is whether a permanent increase in the short rate would lead to the same increase in the long rate. If δ were 1, the long-run effect would be

$$\frac{a_2}{1 - a_1}$$

We can use **SUMMARIZE** after the **LINREG** to estimate that and its standard error (note—this is at the *end* of the example program):

```
linreg tb1yr
# constant tb1yr{1} tb3mo{1}
summarize(title="Long-run effect using linear regression") $
%beta(3) / (1-%beta(2))
```

Long-run effect using linear regression			
Value	0.88137621	t-Statistic	3.63694
Standard Error	0.24234021	Signif Level	0.0003479

It's not significantly different from one, but the standard error is quite large.

With the non-linear model, the effect of a change in the short rate on the long rate is no longer independent of the value of **TB3MO**. The analogous calculation for the long run effect would now be:

$$\frac{a_2 \delta S R^{\delta-1}}{1 - a_1}$$

Since this depends upon the short rate, we can't come up with a single value, but instead will have a function of "test" values for the short rate. We can compute this function, together with upper and lower 2 standard error bounds using the following:

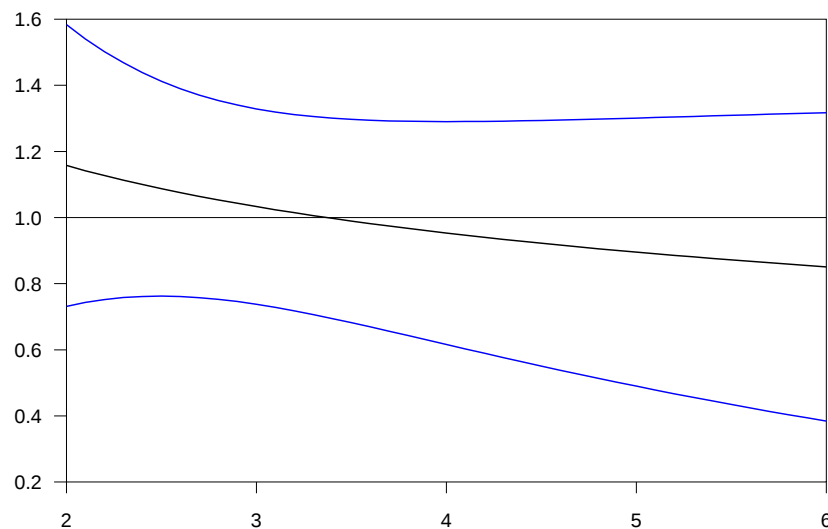


Figure 3.1: Long-run effect using non-linear regression

```

set testsr    1 100 = .1*t
set lreffect  1 100 = 0.0
set lower     1 100 = 0.0
set upper     1 100 = 0.0
*
do t=1,100
  summarize(noprint) $
    %beta(3)*%beta(4)*testsr(t)^(%beta(4)-1)/(1-%beta(2))
  compute lreffect(t)=%sumlc
  compute lower(t)=%sumlc-2.00*sqrt(%varlc)
  compute upper(t)=%sumlc+2.00*sqrt(%varlc)
end do t

```

TESTSR is a series of values from 0.1 to 10. The three other series are initialized to zero over the “grid range” so they can be filled in entry by entry inside the loop. **SUMMARIZE** computes %SUMLC as the estimate of the non-linear function and %VARLC as the estimated variance.

The following graphs (Figure 3.1) the function. The range is limited to values of SR between 2 and 6, as the function, particularly for under 2, grows rapidly, dominating the range of the graph.

```

scatter(smpl=testsr>=2.0.and.testsr<=6.0,style=lines,vgrid=1.0,$
  footer="Long-run effect using non-linear regression") 3
# testsr lreffect
# testsr lower / 2
# testsr upper / 2

```

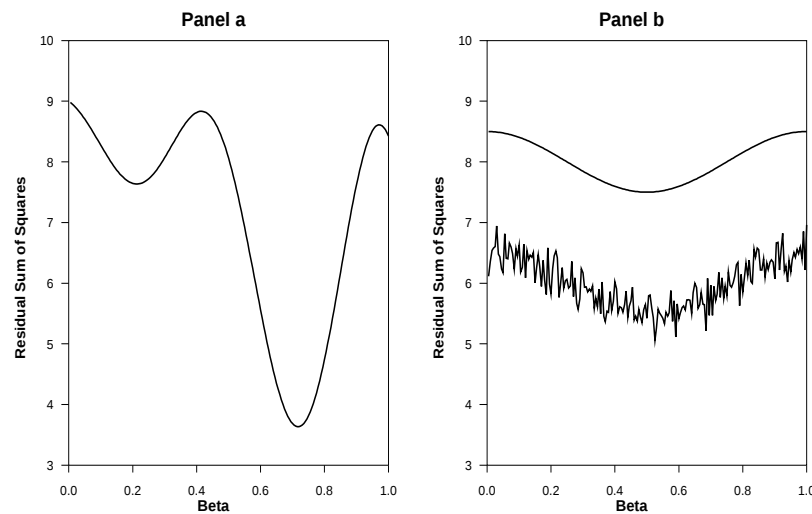


Figure 3.2: Sums of Squares Examples

3.4 Convergence and Convergence Criteria

Numerical optimization algorithms use iteration routines that cannot guarantee precise solutions for the estimated coefficients. Various types of “hill-climbing” methods (Gauss-Newton is an example, though it “climbs” only if you switch the sign of the objective function) 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.

To explain, suppose that the sum of squared residuals for various values of β can be depicted by Panel a of Figure 3.2. Obviously, a value of β of about 0.71 minimizes the sum of squared residuals. However, there is a *local* minimum at $\beta = 0.23$. If we started with a guess value less than .4 (where the function has the local *maximum*), it is likely that Gauss-Newton will find that local minimum instead. How can we know if we have the global rather than local maximum? In general, we can’t, unless we have strong knowledge about the (global) behavior of the function.⁴ For instance, even in this case, notice that the function is starting to turn down at the right edge of Panel a—there’s no way to be sure that the function doesn’t get even smaller out beyond $\beta = 1.0$.

The sum of squares surface shown in the figure is rather convex (has a strongly positive second derivative) so it is clear where the local minima occur. However, suppose that the surface was quite flat. As shown by the smooth line in Panel b, the sum of squared residuals is almost invariant to the value of β selected. In such circumstances, it might take many iterations for RATS to select a value of β within the default tolerance of 0.00001. There’s a practical limit to how

⁴For many functions, it’s possible to prove that the sum of squares surface is (quasi-)convex, which would mean that it can have just one local minimum.

accurately you can estimate the coefficients. Suppose you're trying to maximize $f(\beta)$ with respect to β , and assume for simplicity that β has just one element. If f has enough derivatives, we can approximate f with a two term Taylor series approximation as

$$f(\beta) \approx f(\beta_0) + f'(\beta_0)(\beta - \beta_0) + \frac{1}{2}f''(\beta_0)(\beta - \beta_0)^2$$

If we're very close to the optimum, $f'(\beta_0)$ will be very close to zero. So if

$$\frac{f''(\beta_0)(\beta - \beta_0)^2}{2f(\beta_0)} \quad (3.5)$$

is less than 10^{-15} (which is “machine-zero”), then on a standard computer, we can't tell the difference between $f(\beta)$ and $f(\beta_0)$. Since the difference in the β 's comes in as a *square*, in practice, we're limited to about 7 significant digits *at the most*, and there is rarely any need to try to push below the 5 significant digits that are the default—you are unlikely to ever report more digits than that, and the extra work won't really change the results in any meaningful way.

The two main controls for the Gauss-Newton algorithm are options on **NLLS**: the `ITERATIONS` option, which you can use to increase the number of iterations, and the `CVCRT` option, which can be used to tighten or loosen the convergence criterion. The default on `ITERATIONS` is 100, which is usually enough for well-behaved problems, but you might, on occasion need to increase it. The default on `CVCRT` is .00001, which, as we noted above, is a reasonable value in practice—you're unlikely to report even five significant digits in practice, and it's unlikely that you could get much of a better result if you put in a smaller value. There's also a `PMETHOD` option for using a different method at first (`PMETHOD` means Preliminary METHOD) before switching to Gauss-Newton, but it is rarely needed for **NLLS**. It *will* be important for models estimated with maximum likelihood.

Some secondary controls for non-linear estimation routines are included in the separate **NLPAR** instruction which is covered in this chapter's *Tips and Tricks* (page 102).

The non-linear least squares algorithm is based upon an assumption that the sum of squares surface is (at least locally) well-behaved. If you have the (for all practical purposes) non-differentiable function shown in the lower function in panel (b), **NLLS** is very unlikely to give good results. Even a brute-force “grid search” might fail to find the minimum unless it uses a *very* fine grid. This again shows the importance of having some idea of how the sum of squares surface looks.

3.5 ESTAR and LSTAR Models

The Logistic Smooth Transition Autoregressive (LSTAR) and Exponential Smooth Transition Autoregressive (ESTAR) models generalize the standard autoregressive model to allow for a varying degree of autoregressive decay, thus allowing for different dynamics for the up and down parts of cycles. The LSTAR model can be represented by:

$$y_t = \alpha_0 + \sum_{i=1}^p \alpha_i y_{t-i} + \theta \left[\beta_0 + \sum_{i=1}^p \beta_i y_{t-i} \right] + \varepsilon_t \quad (3.6)$$

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

In the limit, as $\gamma \rightarrow 0$, the LSTAR model becomes an $AR(p)$ model since θ is actually constant. For $0 < \gamma < \infty$, the value of θ changes with the value of y_{t-1} . Hence, θ acts as a weighting function so the degree of autoregressive decay depends on the value of y_{t-1} . As the value of $y_{t-1} \rightarrow -\infty$, $\theta \rightarrow 0$ so the behavior of y_t is given by $\alpha_0 + \alpha_1 y_{t-1} + \dots + \alpha_p y_{t-p} + \varepsilon_t$ (which we'll call the first branch). And, as $y_{t-1} \rightarrow +\infty$, $\theta \rightarrow 1$ so that the behavior of y_t is given by $(\alpha_0 + \beta_0) + (\alpha_1 + \beta_1)y_{t-1} + \dots + (\alpha_p + \beta_p)y_{t-p} + \varepsilon_t$ (which we'll call the second branch). The two branches can have different means and different dynamics, sometimes very different. For an LSTAR model, as y_{t-1} ranges from very small to very large values, θ goes from zero to unity, and you get a blend of the two branches. In particular, when y_{t-1} equals the centrality parameter c , the value of $\theta = 0.5$, and you get an average of the coefficients.

The ESTAR model is similar to the LSTAR model except θ has the form:

$$\theta = [1 - \exp(-\gamma(y_{t-1} - c)^2)] ; \gamma > 0$$

For the ESTAR model, $\theta = 0$ when $y_{t-1} = c$ and approaches unity as y_{t-1} approaches $\pm\infty$. The shape of θ is somewhat like an inverted bell—effectively it's a Normal density flipped outside down.

You can get a good sense of the nature of the LSTAR and ESTAR models by experimenting with the following code (Example 3.2). The first line sets up y to range from -0.5 to $+0.5$. In the second, c is given the value zero, with a γ of 10. We then compute the two shape functions:

```
set y 1 201 = (t-100)/201.
compute c=0.0,gamma=10.0
set lstar = ((1 + exp(-gamma*(y-c))))^-1
set estar = 1 - exp(-gamma*(y-c)^2)
```

The following graphs (Figure 3.3) the two transition functions:

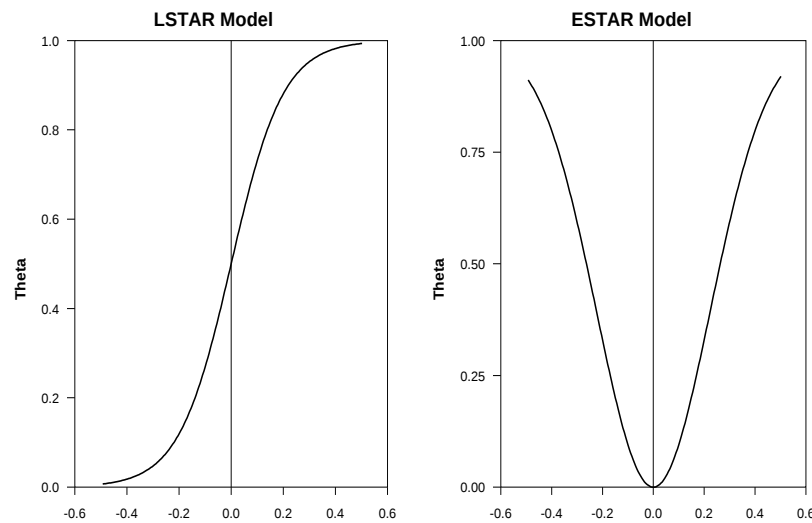


Figure 3.3: Shapes of STAR Transitions

```
spgraph(footer="Shapes of STAR Transitions",vfields=1,hfields=2)
  scatter(header="LSTAR Model",style=line,vlabels="Theta")
  # y lstar
  scatter(header="ESTAR Model",style=line,vlabels="Theta")
  # y estar
spgraph(done)
```

You should experiment by rerunning the program with different values of c and γ . You will find that increasing γ makes the transitions shorter and steeper; in the limit, as $\gamma \rightarrow \infty$, the LSTAR converges to a step function and the ESTAR to a (downwards) “spike”. Changing the value of the centrality parameter, c , changes the transition point for the LSTAR and the point of symmetry for the ESTAR.

We’ll first use generated data to illustrate the process of estimating a STAR model. We will do this because STAR models are a classic example of the 3rd word of advice: “Realize That Not All Models Work!” Suppose that there really is no “second branch” in (3.6), that is, the data are generated by simply:

$$y_t = \alpha_0 + \sum_{i=1}^p \alpha_i y_{t-i} + \varepsilon_t$$

What would happen if we tried to estimate a STAR (in particular, an LSTAR)? If $\beta_i = 0$ for all i , then the transition parameters don’t matter. If c is bigger than any data value for y , then a large value of γ will make θ effectively zero through the data set, so the β coefficients don’t matter. If c is *smaller* than any data value for y , a large value of γ will now make θ effectively one through the data set, so only $\alpha_i + \beta_i$ matters and not the individual values for α or β . So we have three completely different ways to get equivalent fits to the model.

However, what is likely to happen *in practice*?. Suppose the residual for the entry with the highest value of y_{t-1} in the data set is non-zero. Then, we can reduce the sum of squares by adding a dummy variable for that entry. Now we would never actually pick a data point and dummy it out without a good reason, but we have a non-linear model which can “generate” a dummy by particular choices for γ and c . If c is some value larger than the *second* highest value of y_{t-1} , and γ is very large, then, in this data set, θ will be a dummy for that one observation with the highest value of the threshold, and this will thus reduce the sum of squares. You can do something similar at the other end of the data set, isolating the *smallest* value for y_{t-1} . Thus the sum of squares function is likely to have two “local” modes with c on either end of the data set, and there will be no particularly good way to move between them. Neither generates an interesting transition model, and it will often be hard to get non-linear least squares to converge to either since it requires a very large value of γ .

Note, by the way, that this problem is much worse for an ESTAR model, where the transition function can (in effect) dummy *any* data point. You can’t just take a set of data and fit a STAR model to it and hope it will give reasonable results.

3.6 Estimating a STAR Model with NLLS

If c and γ were known (or at least treated as fixed), (3.6) would be linear—we would just have to construct the variables for the β terms by multiplying the (time-varying) θ by the lagged y . However, in practice, the transition parameters aren’t known, which makes this a non-linear least squares problem. To illustrate how to estimate this type of model, we first need to generate data with a STAR effect.

The following example is from Section 7.9 of Enders (2010). The first part of the program (Example 3.3) generates a simple LSTAR process containing 250 observations. The first three lines of code set the default series length to 250, seed the random number generator, and draw 350 pseudo-random numbers⁵ from a normal distribution with standard deviation one (and mean zero):

```
all 250
seed 2003
set eps 1 350 = %ran(1)
```

Next, we’ll create an LSTAR process with 350 observations. This uses the RATS function %LOGISTIC function, which computes the transition function without any chance of an overflow on the *exp* function.⁶ This uses $c = 5$ and $\gamma = 10$:

⁵The reason for 350 will be described shortly. See this chapter’s *Tips and Tricks* (page 105) for more on random number generation.

⁶ $\exp(z)$ will overflow when z is 710 or greater. The way that expressions are evaluated in RATS, $1.0/(1 + \exp(z))$ will be evaluated as NA for such a value of z . The %LOGISTIC function knows the behavior of the overall function, and so returns 0 for a case like that.

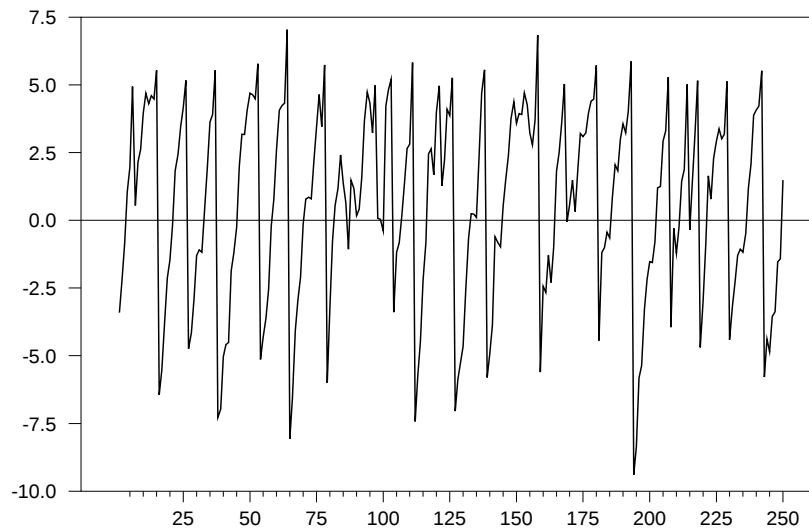


Figure 3.4: Simulated STAR Process

```
set(first=1.0) x 1 350 = $
    1.0+.9*x{1}+(-3.0-1.7*x{1})*%logistic(10.0*(x{1}-5.0),1.0)+eps
```

It is not obvious how to set the initial value of x . In such circumstances, a common practice is to generate a series longer than necessary and then discard the extra. Here we use 100 extra points—these are known as the *burn-in* period. There are several ways to handle this—you could start all the analysis at entry 101, but here we'll simply copy the data down to the desired entries with

```
set y 1 250 = x(t+100)
```

The time series graph of the series (Figure 3.4) is created using:

```
graph(footer="The Simulated LSTAR Process")
# y
```

The first branch of the LSTAR is $y_t = 1.0 + .9y_{t-1} + \varepsilon_t$. This has mean 10 and is strongly positively correlated. The second branch is created by adding the two processes—it is what you would get for values of y_{t-1} above 5,⁷ so it's $y_t = -2.0 - .8y_{t-1} + \varepsilon_t$. This has a mean of $-2/1.8 \approx -1.11$ and is strongly *negatively* correlated. As a result, the process generally moves steadily up under the control of the first branch until the value of y is greater than 5. In the next time period, it is likely to drop very sharply under control of the second branch, which will drive it back onto the first branch. So the process going up is slower than the one going down.

The **NONLIN** instruction and **FRML** definition are:

⁷Since γ is large, the transition is very short.

```

nonlin a0 a1 b0 b1 gamma c
frml lstar y = (a0+a1*y{1})+$
            (b0+b1*y{1})*%logistic(gamma*(y{1}-c),1.0)

```

Guess Values, Method One. Based at OLS

One way to get initial guesses is to estimate a linear model and use the coefficient estimates as the initial values. This starts as if we have only the first branch, zeroing out the change. For illustration, we'll start with $c = 0$ (roughly the middle of the data) and $\gamma = 5$.

```

linreg y
# constant y{1}
compute a0=%beta(1),a1=%beta(2),b0=0.0,b1=0.0
compute c=0.0,gamma=5.0
*
nlls(frml=lstar) y 2 250

```

Nonlinear Least Squares - Estimation by Gauss-Newton				
NO CONVERGENCE IN 100 ITERATIONS				
LAST CRITERION WAS 0.0185721				
Dependent Variable Y				
Usable Observations	249			
Degrees of Freedom	243			
Centered R ²	0.6130672			
R-Bar ²	0.6051056			
Uncentered R ²	0.6234211			
Mean of Dependent Variable	0.5876231693			
Std Error of Dependent Variable	3.5509791503			
Standard Error of Estimate	2.2314573881			
Sum of Squared Residuals	1209.9947042			
Log Likelihood	-550.1400			
Durbin-Watson Statistic	1.7369			
Variable	Coeff	Std Error	T-Stat	Signif

1. A0	2.74882	0.50706	5.42105	0.00000014
2. A1	1.04721	0.11905	8.79664	0.00000000
3. B0	-273.09372	11487.80317	-0.02377	0.98105359
4. B1	28.61805	1312.03609	0.02181	0.98261587
5. GAMMA	0.75187	0.43441	1.73079	0.08475907
6. C	8.92116	51.91215	0.17185	0.86369759

The results don't look at all sensible, but more important, when you look at the second and third lines of the output, it's clear that we don't *have* "results" in the first place. If we haven't gotten convergence, one question to ask is whether it's worth just increasing the iteration limit in hope that that will fix things. When you have a model like this that can have multiple modes, you probably want to stop and look at whether you will likely just be converging to a bad mode. Here, we have a value for c which is well above the maximum value for y in the data set (nearly 9 vs a maximum value of just above 7). With these values, the *largest* value that θ will take is about .2, so only a fraction of those (rather absurd-looking) values for B0 and B1 will apply. Despite this not being converged, and having very strange dynamics, the sum of squares is (much)

lower with these “estimates” than it is for the simple least squares model, 1209 here vs 2218 for OLS—note that the estimated first branch is somewhat similar to the one used in the actual DGP, and the effect of the θ times the second branch will be sharply negative for the values closest to c , which is the behavior we need.

In this case, the main problem with the guesses was the value of γ . Even though the guess of 5 is *smaller* than the true value, when combined with the wrong value of c , it works poorly because the function is almost non-differentiable with respect to c due to the sharp cutoff. If you go back and try with `GAMMA=1.0`, you’ll see that you get convergence to a reasonable set of estimates. Note, however, that you need to re-execute the **LINREG** and **COMPUTE** instructions to make that work: `%BETA` has been re-defined by **NLLS** so the **COMPUTE** instructions for `A0` and `A1` will no longer use the OLS values unless you re-do the **LINREG**.

The biggest problem in fitting STAR models is finding the threshold value c . As we mentioned, given γ and c , the model is linear in the other parameters, and given c , γ usually isn’t hard to estimate.

Guess Values, Method 2: Data-Determined

The initial values for c and γ above were basically just wild guesses. A more straightforward alternative is to use **STATISTICS** on the threshold and get guess values for c and γ off of that, for instance:

```
stats y
compute c=%mean,gamma=1.0/sqrt(%variance)
```

Using the reciprocal of the sample standard error⁸ starts with a rather “flat” transition function (most of the observed data will be a blend of the two branches rather than one or the other), which makes it easier for Gauss-Newton to find the optimal c . The following then treats `C` and `GAMMA` as fixed to get the corresponding coefficients of the two branches (in the first **NLLS**) then estimates all the parameters together:

```
nonlin a0 a1 b0 b1
nlls(frml=1star) y 2 250
nonlin a0 a1 b0 b1 gamma c
nlls(frml=1star) y 2 250
```

The first **NLLS** is actually linear, since `GAMMA` and `C` are fixed (not included in the **NONLIN**). If you look at the output from it (not shown), you’ll see that it converged in 2 iterations—the first moves to the minimizer, and the second tries to improve but can’t. The output from the **NLLS** with the full parameter set is:

⁸For a **ESTAR**, you would use `1.0/%variance` instead since the exponent depends upon the square of the data.

Nonlinear Least Squares - Estimation by Gauss-Newton				
Convergence in 57 Iterations. Final criterion was 0.0000018 <= 0.0000100				
Dependent Variable Y				
Usable Observations	249			
Degrees of Freedom	243			
Centered R ²	0.9201029			
R-Bar ²	0.9184590			
Uncentered R ²	0.9222409			
Mean of Dependent Variable	0.5876231693			
Std Error of Dependent Variable	3.5509791503			
Standard Error of Estimate	1.0139960256			
Sum of Squared Residuals	249.84966939			
Regression F(5,243)	559.6826			
Significance Level of F	0.0000000			
Log Likelihood	-353.7398			
Durbin-Watson Statistic	2.0482			
Variable	Coeff	Std Error	T-Stat	Signif

1. A0	1.017585644	0.068354840	14.88681	0.00000000
2. A1	0.917712225	0.020712881	44.30635	0.00000000
3. B0	-4.467787643	3.502956803	-1.27543	0.20337389
4. B1	-1.438183472	0.591133075	-2.43293	0.01569975
5. GAMMA	9.957050393	1.551367386	6.41824	0.00000000
6. C	5.002927739	0.019240934	260.01481	0.00000000

Note that this has quite accurately estimated A0, A1, GAMMA and C, but not so much B0 and B1. That's not that surprising since there are only about 20 data points out of the 250 where the B0 and B1 even matter.

Guess Values, Method 3: Grid Search

A grid search can sometimes be helpful if it's hard to fit a model from more "generic" guess values *and* there is a single parameter that is the main problem. A grid search over a full six-dimensional space (as we have here) is not really feasible, and here isn't even necessary since the model is linear given γ and c . We have three possible approaches that might make sense:

1. Fix γ and grid search over c .
2. Grid search over c , estimating γ by non-linear least squares for each.
3. Jointly search over a grid in c and γ .

The first would involve the least calculation since each test model is linear (γ is fixed in advance, and c is fixed for a given evaluation). The second would likely require the most calculation since it would be fully estimating a non-linear least squares model for each c . The third would be the hardest to set up since it requires a two-dimensional grid.

The simplest and most general way to handle the control of a grid search (for minimization) is the following "pseudo-code" (descriptive rather than actual):

```
compute bestvalue=%na
do over grid
  calculate value(thisgrid) to thisvalue
  if .not.%valid(bestvalue).or.bestvalue>thisvalue
    compute bestvalue=thisvalue,bestgrid=thisgrid
end grid
```

At the end of this, the grid point at which the calculation is smallest will be in “bestgrid”,⁹ and the value of the function there will be “bestvalue”. The **IF** statement will give a “true” condition if either the current bestvalue is %NA,¹⁰ or if thisvalue is smaller than the current bestvalue.

Although the grid can be an actual equally-spaced set of values, that isn’t required. For our purposes, the quickest way to create the grid is with the %SEQA function (meaning additive sequence), where %SEQA(start,increment,n) returns the VECTOR with n values start, start+increment, ..., start+increment*($n-1$). For the grids for c , we’ll use

```
stats(fractiles) y
compute ygrid=%seqa(%fract05, (%fract95-%fract05)/19,20)
```

which will do a 20 point grid ($n = 20$) over the range from 5%-ile to the 95%-ile of the data.¹¹

The loop over the grid values is done with

```
dofor c = ygrid
...
end dofor c
```

DOFOR is a more general looping instruction than **DO** (section 2.8.1)—while **DO** only loops under the control of “counter” (**INTEGER**) variables, **DOFOR** can loop over a “list” of anything. Usually (as here), the “list” is a **VECTOR** of some data type but it can be a list of items separated by spaces:

```
dofor c = 1.0 2.0 4.0 8.0 16.0
...
end dofor c
```

Each time through the loop, **DOFOR** just pulls the next value off the list or out of the **VECTOR**, sets the index variable (here C) equal to it, and re-executes the content of the loop. There *are* other ways to set this up—in general, there can be many equivalent ways to code a calculation like this. For instance, we could have done

```
stats(fractiles) y
do i=1,20
  compute c=%fract05+(i-1)*(%fract95-%fract05)/19
  ...
end do i
```

⁹If you want to maximize instead, simply change the <to > on the **IF**.

¹⁰.not.%valid(bestvalue) will be “true” if and only if bestvalue is missing, which will happen the first time through the loop.

¹¹**STATS (FRACTILES)** computes a standard set of quantiles of the data, for 1, 5, 10, 25, 50, 75, 90, 95 and 99 which are fetchable as variables named %FRACTnn.

There are two advantages of using the **DOFOR** setup:

1. It's clearer what the loop is doing.
2. The "controls" of the loop (start, end, number of values) are included in just a single instruction (the **COMPUTE** with the **%SEQA**), instead of two (the limit on the **DO** and the **COMPUTE C**).

The working code for the grid search for a fixed value of γ is:

```
stats(fractiles) y
compute gamma=2.0/sqrt(%variance)
compute ygrid=%seqa(%fract05, (%fract95-%fract05)/19,20)
nonlin a0 b1 b0 b1
compute bestrss=%na
dofor c = ygrid
  nlls(noprint, frml=1star) y 2 250
  if .not.%valid(bestrss).or.%rss<bestrss
    compute bestrss=%rss, bestc=c
end dofor c
```

This uses a **NONLIN** which doesn't include **C** or **GAMMA**, since those are being fixed on each evaluation. This restores to **C** the best of the grid values and estimates all parameters of the model:

```
disp "Guess Value used" bestc
*
compute c=bestc
nonlin a0 a1 b0 b1 gamma c
nlls(frml=1star) y 2 250
```

The setup for the grid search across **C** with **GAMMA** being estimated given **C** is similar. We'll include all the instructions, even though many are the same as before:

```
stats(fractiles) y
compute gamma0=2.0/sqrt(%variance)
compute ygrid=%seqa(%fract05, (%fract95-%fract05)/19,20)
nonlin a0 b1 b0 b1 gamma
compute bestrss=%na
dofor c = ygrid
  compute gamma=gamma0
  nlls(noprint, frml=1star) y 2 250
  if .not.%valid(bestrss).or.%rss<bestrss
    compute bestrss=%rss, bestc=c, bestgamma=gamma
end dofor c
```

This now includes **GAMMA** in the parameter set on the **NONLIN** and saves the value of **GAMMA** along with **C** when we find an improvement. Note that **GAMMA**

is restored to its original guess value each time through the loop—this avoids problems if (for instance) the first values of C have an optimal GAMMA which is large. If GAMMA isn't re-initialized, it will use the value it got as part of the previous estimation, which might be a problem.

We now have to restore the best values for both C and GAMMA :

```
disp "Guess values used" bestc "and" bestgamma
compute c=bestc,gamma=bestgamma
nonlin a0 a1 b0 b1 gamma c
nlls(frml=1star) y 2 250
```

The bivariate grid search requires that we also set up a grid for γ . We'll use the same grid for c . Since γ has to be positive, we'll use the `%EXP` function with `%SEQA` to generate a geometric sequence. `%EXP` takes the element-by-element *exp* of a matrix—here it will give us fractions ranging from .25 to 25 of the reciprocal of the inter-quartile range of the data.¹²

```
stats(fractiles) y
compute ygrid=%seqa(%fract05, (%fract95-%fract05)/19,20)
compute ggrid=%exp(%seqa(log(.25), .1*log(100),11)) / (%fract75-%fract25)
```

As with the first grid, we leave both GAMMA and C out of the parameter set. We nest the two `DOFOR` loops (the order here doesn't matter), and do the `NLLS` and the test for improvement inside the inner loop.

```
nonlin a0 b1 b0 b1
compute bestrss=%na
dofor c = ygrid
    dofor gamma = ggrid
        nlls(noprint,frml=1star) y 2 250
        if .not.%valid(bestrss).or.%rss<bestrss
            compute bestrss=%rss,bestc=c,bestgamma=gamma
        end dofor gamma
    end dofor c
```

As before, we restore the best values for both, and estimate the full set of parameters:

```
disp "Guess values used" bestc "and" bestgamma
compute c=bestc,gamma=bestgamma
nonlin a0 a1 b0 b1 gamma c
nlls(frml=1star) y 2 250
```

¹²The inter-quartile range is the distance between the 25%-ile and 75%-ile of a series, and is a (robust) alternative to the standard deviation for measuring dispersion since it isn't an sensitive to outliers.

Not surprisingly (since it's with constructed data), all three grid search methods end up with the same optimum as we got originally. That may not be the case with actual data, as we'll see in Section 3.8.

3.7 Smooth Transition Regression

In the models examined above, the threshold variable was a lag of the dependent variable. It's also possible to use a lag of the difference (known as a *momentum TAR model*), or possibly some other linear combination of lags (several period average for instance). Because the threshold in any of these cases is endogenous, the dynamics of the generated process can be quite complicated.

It's also possible to apply the same type of non-linear model to a situation where the threshold is *exogenous*. Such models are called Smooth Transition Regression (or STR) rather than STAR. One obvious case would be where the threshold variable is *time*—so the model has a structural break at some point but smoothly moves from one regime to the other, perhaps due to a gradual rollout or slow adoption of new technologies.

The following program simulates a series with an LSTR break such that:

$$y_t = 1 + 3/[1 + \exp(-0.075(t - 100))] + 0.5y_{t-1} + \varepsilon_t \quad (3.7)$$

Note that the centrality parameter is 100 and that $\gamma = 0.075$. Here, the break affects only the intercept term as the autoregressive parameter is always 0.5. Since the value of θ ranges from 0 to 1, the intercept is 1 for small values of t and is 4 for large values of t . With an autoregressive parameter of 0.5, the mean of the series is about 2 for small values of t and is about 8 for very large values of t . Nevertheless, with a smooth LSTR break, the average value of the series starts to slowly shift upward beginning around $t = 75$ and continues the upward shift until the process levels off at roughly $t = 125$.

The first part of the program (Example 3.4) sets the default length of a series to be 250 observations, seeds the random number generator and creates the *eps* series containing 250 pseudo-random numbers drawn from a normal distribution with a standard deviation of unity.

```
all 250
seed 2003
set eps = %ran(1)
```

The next two lines create the series for θ and the y_t . The resulting series is graphed, producing Figure 3.5:

```
set theta = 1/(1+exp(-.075*(t-100.)))
set(first=2.) y = 1 + 3*theta + 0.5*y{1} + eps
graph(footer="A Simulated LSTR Break") 1
# y
```

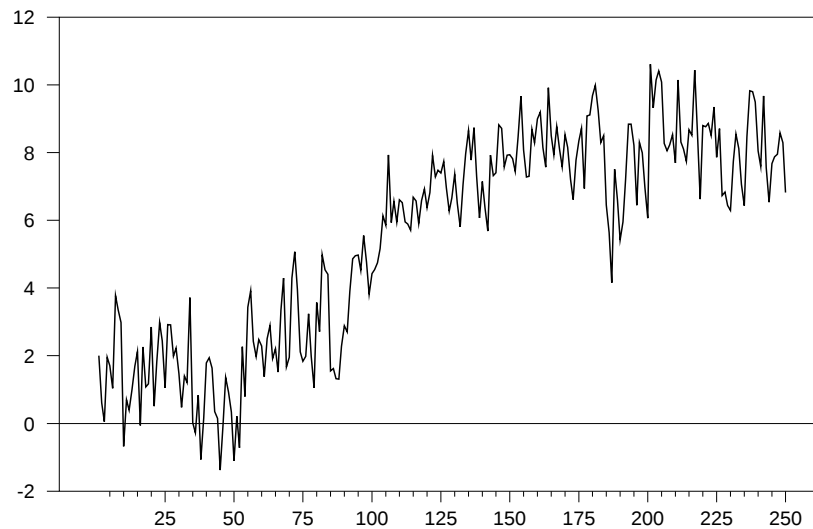


Figure 3.5: A Simulated LSTR Break

Note that it's easier to simulate this because the θ function can be generated separately from the y .

This type of model is much easier to handle than a STAR because the “break”, while not necessarily sharp, has an easily visible effect: here the series seems to have a clearly higher level at the end than at the beginning.

One way to proceed might be to estimate the series as a linear process and create guess values based upon the estimated coefficients. Since the intercept appears to be lower near the start of the data set, this makes the first branch somewhat lower than the linear estimate, and the second somewhat higher.¹³

```
linreg(noprint) y
# constant y{1}
compute a1=%beta(2), a0=%beta(1)-%stderrs(1), b0=2*%stderrs(1)
compute c=75.0, gamma=.25
```

The guess value for c appears at least reasonable given the graph, as 75 seems to be roughly the point where the data starts changing. This value of γ may be a bit too high (at $\gamma = .25$ about 90% of the transition will be over the range of $[c - 12, c + 12]$) but it appears to work in this case:

```
nlls(frml=1star, iterations=200) y
```

¹³Since the second branch intercept is the sum of a_0 and b_0 , b_0 is initialized to the guess at the difference between the two intercepts.

Nonlinear Least Squares - Estimation by Gauss-Newton				
Convergence in 9 Iterations. Final criterion was 0.0000029 <= 0.0000100				
Dependent Variable Y				
Usable Observations	249			
Degrees of Freedom	244			
Skipped/Missing (from 250)	1			
Centered R ²	0.8871302			
R-Bar ²	0.8852799			
Uncentered R ²	0.9725995			
Mean of Dependent Variable	5.4629432995			
Std Error of Dependent Variable	3.0993810584			
Standard Error of Estimate	1.0497715136			
Sum of Squared Residuals	268.89293630			
Regression F(4,244)	479.4457			
Significance Level of F	0.0000000			
Log Likelihood	-362.8848			
Durbin-Watson Statistic	1.9852			
Variable	Coeff	Std Error	T-Stat	Signif

1. A0	0.723962117	0.181964155	3.97860	0.00009144
2. A1	0.433037455	0.057838156	7.48705	0.00000000
3. B0	3.876373910	0.448189508	8.64896	0.00000000
4. GAMMA	0.065350537	0.012699962	5.14573	0.00000055
5. C	97.482532947	3.385817497	28.79143	0.00000000

The coefficient estimates are reasonably close to the actual values in the data generating process. As an exercise, you might want to experiment with different initial guesses. It turns out that, for this model, the results are quite robust to the choice of the initial conditions. Also, use the AIC and BIC to compare the fit of this model to that of a linear model and to a model estimated with a *sharp* structural break.

A grid search for a threshold based upon time is simpler than it is for a continuous variable, as you can just loop over the entries. The following does a preliminary grid search over the center 70% of the data (leaving out 15% at either end). It also saves the values of the sum of squares into the series `RSS` so we can graph it (Figure 3.6):

```

nonlin a0 a1 b0 gamma
set rss = %na
do time=38,213
    compute gamma=.25
    compute c=time
    nlls(frml=lstr,iterations=200,noprint) y
    if %converged==1
        compute rss(time)=%rss
end do ic

```

This is doing the second form of grid search, where γ is estimated for each test value of c . We don't need to save the best value for c because we can simply use **EXTREMUM** to find the best value, and entry at which it's achieved (%MINENT is defined by **EXTREMUM**):

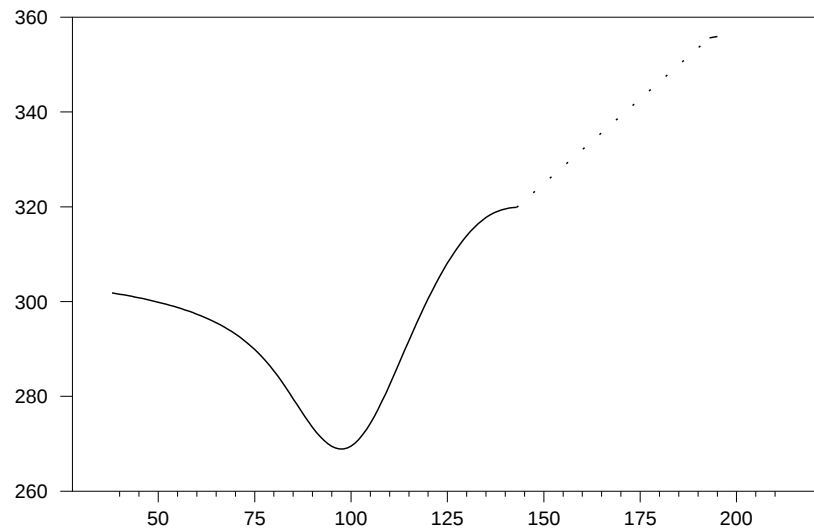


Figure 3.6: Sum of Squares for LSTR Break

```
extremum rss
compute c=%minent
compute gamma=.25
nonlin a0 a1 b0
nlls(frml=lstr,iterations=200) y
nonlin a0 a1 b0 gamma c
nlls(frml=lstr,iterations=200) y
```

This gives us the same results as we got with the empirical guess values. The graph of the sums of squares (as a function of time) is produced with:

```
graph(footer="Sum of Squares for LSTR Break") 1
# rss 38 213
```

Note that the model doesn't always converge for larger values, which is why the graphs has gaps. (Note the test for %CONVERGED in the grid search loop above).

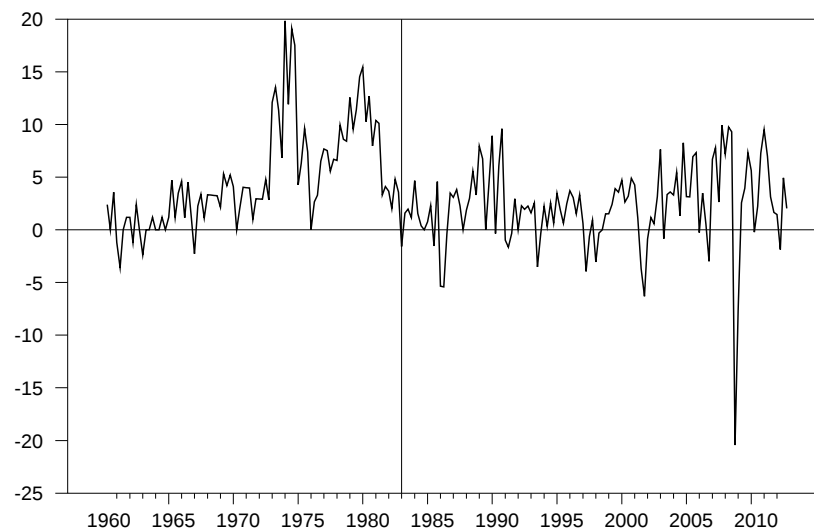


Figure 3.7: Annualized Inflation Rate (Measured by PPI)

3.8 An LSTAR Model for Inflation

Our previous examples used simulated data, so we knew what the “true” model was. We will now try to fit an LSTAR model to the U.S. inflation rate. The full program is Example 3.5. We can compute and graph (Figure 3.7) the (annualized) inflation rate with:

```
set pi = 400.0*log(ppi/ppi{1})
graph(footer="Annualized Inflation Rate (Measured by PPI)", $
  grid=(t==1983:1))
# pi
```

It’s fairly clear from looking at this that there is a difference in the process in the period from 1973 to 1980 compared with after that. During the 1970’s, inflation was often over 5% quarter after quarter; more recently, almost any time inflation exceeds 5% in a quarter, it is followed by a quick drop. While the U.S. monetary policy has never formally used “inflation targeting”, it certainly seems possible that the inflation series since the 1980’s might be well-described by a non-linear process similar to the one generated in Section 3.6.

We’ll focus on the sample period from 1983:1 on. In the simulated example, we knew that the process was an LSTAR with AR(1) branches—here we *don’t* know the form, or even know whether an LSTAR will even work. We can start by seeing if we can identify a AR model from the autocorrelation function (restricting the calculation to the desired sample, Figure 3.8):

```
@bident pi 1983:1 *
```

Since we’re picking a pure autoregression, the partial autocorrelations are the main statistic, and they would indicate either 1 or 4 lags. Since these statistics

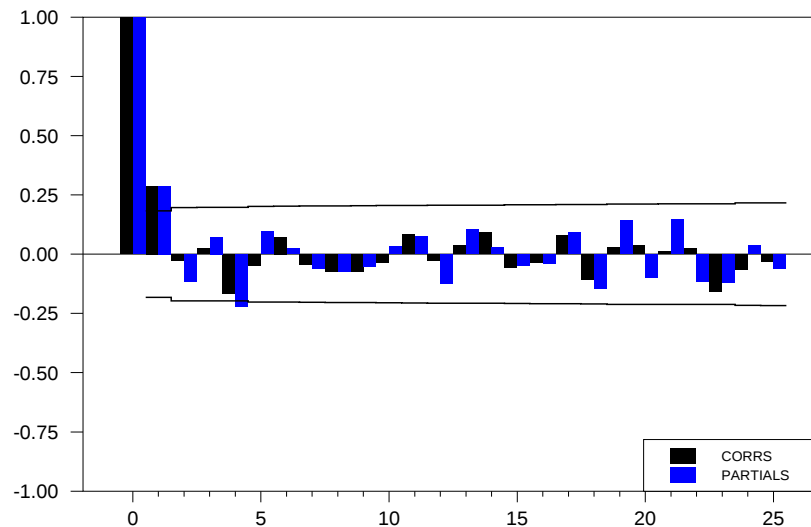


Figure 3.8: Correlations of Inflation Rate (1983-2012)

are being computed assuming a single model applies to the whole sample, it makes sense to work with the more general model to start, possibly reducing it if it appears to be necessary. Note that there is no reason the two branches must have the same form.

Now, we can estimate the base AR(4) model and look at the autocorrelations of the residuals (Figure 3.9):

```
linreg pi 1983:1 *
# constant pi{1 to 4}
@regcorrs(qstats, footer="Residuals from AR(4) Model")
```

Linear Regression - Estimation by Least Squares				
Dependent Variable PI				
Quarterly Data From 1983:01 To 2012:04				
Usable Observations		120		
Degrees of Freedom		115		
Centered R ²		0.1453374		
R-Bar ²		0.1156100		
Uncentered R ²		0.3440071		
Mean of Dependent Variable		2.2002048532		
Std Error of Dependent Variable		4.0148003022		
Standard Error of Estimate		3.7755989989		
Sum of Squared Residuals		1639.3419971		
Regression F(4,115)		4.8890		
Significance Level of F		0.0011203		
Log Likelihood		-327.1461		
Durbin-Watson Statistic		1.9443		
Variable	Coeff	Std Error	T-Stat	Signif

1. Constant	2.009014916	0.474728227	4.23193	0.00004681
2. PI{1}	0.345155837	0.090901340	3.79704	0.00023547
3. PI{2}	-0.180008354	0.095737234	-1.88023	0.06260632
4. PI{3}	0.151718031	0.096356361	1.57455	0.11810720
5. PI{4}	-0.227600168	0.091450487	-2.48878	0.01424990

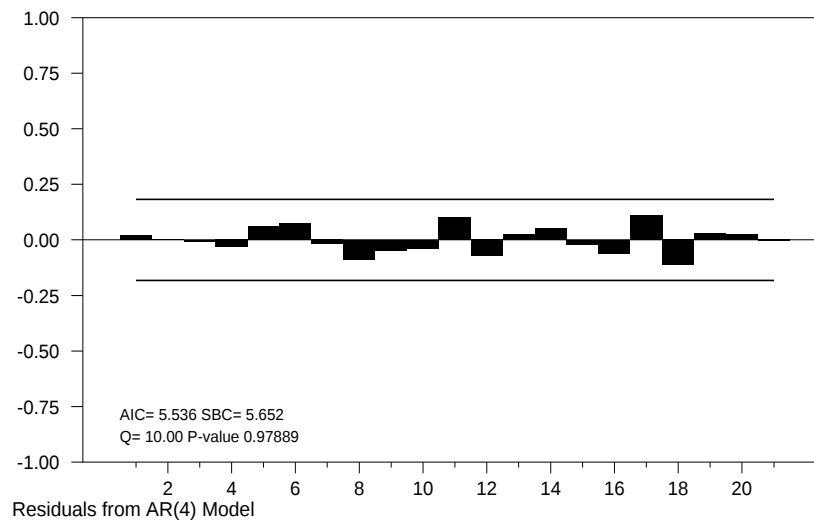


Figure 3.9: Residuals from AR(4) Model

Everything seems fine. Maybe we could look at re-estimating dropping the 3rd lag, but otherwise this looks fine. However, there are some other tests that we can apply that could pick up more subtle failures in the model. One of these, which looks specifically at an autoregression and tests for the possibility of STAR behavior is **STARTEST**. The testing procedure is from Terasvirta (1994). A similar test for more general smooth transition behavior is provided by **REGSTRTEST**.

The two main options on **STARTEST** are **P**=number of lags and **D**=delay (lag) in the threshold. It's recommended that you try several values for **D**—here, we'll do 1, 2 and 3:

```
@startest (p=4,d=1) pi 1983:1 *
@startest (p=4,d=2) pi 1983:1 *
@startest (p=4,d=3) pi 1983:1 *
```

The results for **D=2** are the most significant, which is the method used to choose the best delay:

Test for STAR in series PI		
AR length	4	
Delay	2	
Test	F-stat	Signif
Linearity	3.6535512	0.0001
H01	5.3764713	0.0005
H02	1.5748128	0.1863
H03	3.2926785	0.0139
H12	3.5313261	0.0012

These provide a series of LM tests for the correlation of the standard AR residuals with various non-linear functions on the data (products of the regressors with powers of the threshold variable). You should be very careful in interpreting the results of this (and any other) LM diagnostic test. If the data show

STAR behavior, we would expect that this test would pick it up; you might want to check this with the data from Example 3.3. However, a significant test result doesn't necessarily mean that STAR behavior is present—it indicates that there is evidence of *some* type of non-linearity not captured by the simple AR. However, it is also possible for this test to be “fooled” by outliers using the mechanism described on page 79.

The different test statistics are for different set of powers (from 1 to 3) of the threshold in the interaction terms. The “Linearity” statistic is a joint test of all of them. The combination of results points to a LSTAR rather than an ESTAR as the more likely model—some type of STAR behavior is possible given the rejection of linearity, but if H03 (the 3rd power) were insignificant, it would point towards an ESTAR which, because of symmetry, wouldn't have 3rd power contributions.

Because the model now is bigger, and may be subject to change, we'll introduce a more flexible way to handle it:

```
linreg pi 1983:1 *
# constant pi{1 to 4}
frml(lastreg,vector=b1) phi1f
frml(lastreg,vector=b2) phi2f
```

This estimates the standard AR(4) model, then defines two FRMLs, called PHI1F and PHI2F, each with the form of that last regression (the LASTREG option). The PHI1F formula will use the VECTOR B1 for its coefficients and PHI2F will use B2. Thus, given values for the B1 coefficients, PHI1F(T) will evaluate

$$b1(1) + b1(2) * \pi(t-1) + b1(3) * \pi(t-2) + b1(4) * \pi(t-3) + b1(5) * \pi(t-4)$$

The advantage of this is clear: we can change the entire setup of the model by changing that **LINREG**.

We'll now split the parameter set into two parts: STARPAMS, which will be just the γ and c , and REGPARMS, which will have the two “B” VECTORS. This is done using **NONLIN** with the **PARMSET** option.

```
nonlin(parmset=starpams) gamma c
nonlin(parmset=regpams) b1 b2
```

A **PARMSET** is a convenient way to organize a set (or subset) of non-linear parameters. If you “add” **PARMSET**'s with (for instance) STARPAMS+REGPARMS, you combine them into a larger set. Any of the estimation instructions which allow for general non-linear parameters (such as **NLLS**) have a **PARMSET** option so you can put in a **PARMSET** that you've created.

We can put together the final calculation for the LSTAR explanatory model with

```
frml glstar = %logistic(gamma*(pi{2}-c),1.0)
frml star pi = g=glstar,phi1f+g*phi2f
```

This does the same general type of calculation as the single FRML used in Example 3.3, but has broken it up into more manageable pieces.

The following will do the estimation using the “data-determined” method of guess values (page 82):

```
stats pi 1983:1 *
compute c=%mean,gamma=1.0/sqrt(%variance)
*
nlls(parmset=regparms,frml=star,noprint) pi 1983:1 *
nlls(parmset=regparms+starparms,frml=star,print) pi 1983:1 *
```

This uses the PARMSET option on NLLS and the ability to “add” PARMSETS to simplify the two-step estimation process; the first NLLS holds c and γ fixed (since they aren’t in REGPARMS), estimating only the regression coefficients, while the second NLLS does the whole model.

Unfortunately, the results are not promising. The model doesn’t converge and the coefficients look like:

1. B1 (1)	0.85	1.50	0.56531	0.57303574
2. B1 (2)	0.47	0.13	3.48749	0.00070638
3. B1 (3)	-0.24	0.18	-1.31836	0.19017384
4. B1 (4)	0.19	0.12	1.66112	0.09958973
5. B1 (5)	-0.03	0.19	-0.13401	0.89364362
6. B2 (1)	16687.40	37452754.62	4.45559e-004	0.99964532
7. B2 (2)	-503.54	1125697.16	-4.47310e-004	0.99964392
8. B2 (3)	-250.03	559047.27	-4.47245e-004	0.99964398
9. B2 (4)	-1326.00	2958199.04	-4.48246e-004	0.99964318
10. B2 (5)	-1459.13	3260045.06	-4.47579e-004	0.99964371
11. GAMMA	0.38	0.47	0.81112	0.41908326
12. C	27.94	5961.68	0.00469	0.99626886

The values for B2 are nonsensical, as is C, which is much higher than the largest observed value in the data range. How is it possible for this to give a lower sum of squares than a non-threshold model (which it does—even not converged, it’s 1309 vs 1639 for the simple AR(4))? With the large value of c , the θ function is only barely larger than 0 for *any* of the data points—the largest is roughly .001 at 2008:2. Clearly, however, that tiny fraction applied to those very large B2 values improves the fit (quite a bit) at some of the data points.

We can see if the more complicated preliminary grid search helps, though this seems unlikely given the results above (where c drifted off outside the data range). To use the combined PARMSET’s, we need to define a new one (called GAMMAONLY) which leaves C out, since we want to peg that at each pass through the grid:

```

stats(fractiles) pi 1983:1 *
*
nonlin(parmset=gammaonly) gamma
*
compute bestrss=%na
do for c = %sega(%fract10, (%fract90-%fract10)/19,20)
    compute gamma=1.0/sqrt(%variance)
    nlls(parmset=regparms, frml=star, noprint) pi 1983:1 *
    nlls(parmset=regparms+gammaonly, frml=star, noprint) pi 1983:1 *
    if .not.%valid(bestrss).or.%rss<bestrss
        compute bestrss=%rss, bestc=c, bestgamma=gamma
end do for
*
disp "Grid choices" bestc bestgamma

```

This picks the following which (as might be expected) has c at the upper bound of the grid.

```

Grid choices          7.28375          0.33748

```

As before, the following would estimate the model given those guess values. Since we now want to estimate both c and γ , we use `STARPARMS` rather than `GAMMAONLY`.

```

compute c=bestc, gamma=bestgamma
nlls(parmset=regparms, frml=star, noprint) pi 1983:1 *
nlls(parmset=regparms+starparms, frml=star, print) pi 1983:1 *

```

The outcome, however, is basically the same as before.

When a model fails like this, it's helpful to know what conditions might cause it. With any model, you need to check for major structural breaks, but STAR models are particularly susceptible to outliers so that's a first check. If we take a look at the residuals from that last (unconverged) non-linear model and the original linear regression (Figure 3.10):

```

set nllsresids = %resids
linreg pi 1983:1 *
# constant pi{1 to 4}
set olsresids = %resids
graph(footer="Comparison of STAR and OLS Residuals", $
    key=upleft, klabels=||"STAR", "OLS"||) 2
# nllsresids
# olsresids

```

The 2008:4 residual on the linear model is about 7 standard deviations. Since 2008:4 is preceded by three relatively high values for π , the STAR model is able to combine that with a “trigger” using the second period lag in the threshold to

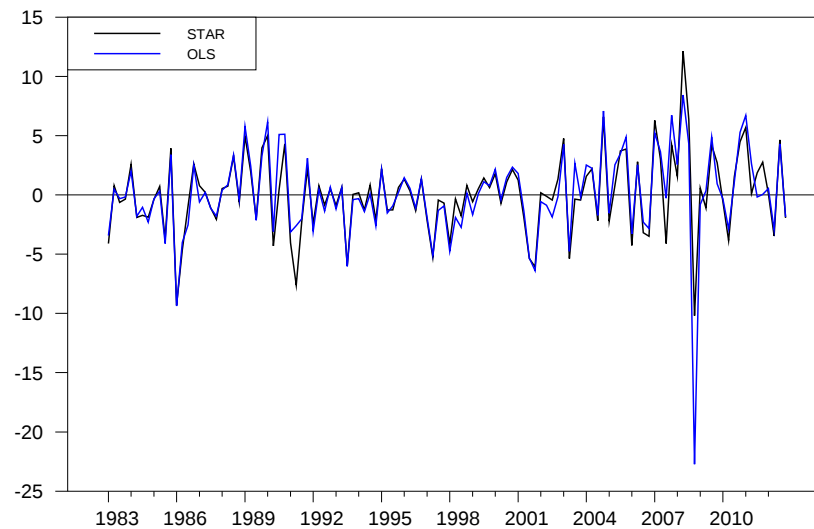


Figure 3.10: Comparison of STAR and OLS Residuals

dramatically reduce the residual at that point, at the cost of worse fits following a few of the other large values. The change in the sum of squares due to the better “fit” at that one point for the STAR more than covers the difference between the sums of squares over the whole 120 data points, so the fit is actually, on net, worse on the other 119 entries.

If we revisit the test for STAR, but apply it to the data set only through 2007:4, we get a very different result:

```
@startest (p=4,d=2) pi 1983:1 2007:4
```

Test for STAR in series PI		
AR length	4	
Delay	2	
Test	F-stat	Signif
Linearity	1.1955332	0.3001
H01	2.4000773	0.0557
H02	0.4187428	0.7947
H03	0.8483300	0.4986
H12	1.3787493	0.2171

Our conclusion is that a STAR process doesn’t seem to be required to explain the behavior of the inflation rate over that period. Since the autoregressive representation has relatively little persistence, it’s quite possible that the apparent sharp drops are simply the result of the natural behavior of an AR process with low serial correlation. If there *is* some systematic non-linearity, it isn’t explained by a STAR model.

3.9 Functions with Recursive Definitions

An alternative model which shows non-linear adjustment is the *bilinear* model. A simple case of this is

$$y_t = \alpha y_{t-1} + \beta \varepsilon_{t-1} + \gamma y_{t-1} \varepsilon_{t-1} + \varepsilon_t \quad (3.8)$$

If γ were zero, this would be a standard ARMA(1,1) model. The “bilinear” part of this is the last term, which is linear in y_{t-1} given ε_{t-1} and linear in ε_{t-1} given y_{t-1} . For this model

$$\frac{\partial y_t}{\partial \varepsilon_{t-1}} = \alpha + \beta + \gamma y_{t-1} + \gamma \varepsilon_{t-1}$$

which takes into account the fact that y_{t-1} moves with ε_{t-1} . Note that when γ is zero, the derivative doesn’t depend upon the past value(s) of y or the size of ε_{t-1} —that’s what the process being “linear” means. With γ non-zero, it depends upon both.

What complicates this is that two of the regressors depend upon the unobservable ε_{t-1} which has a *recursive* definition: we can’t compute the residual ε_t without ε_{t-1} (and the values for α , β and γ) and we can’t compute ε_{t-1} without ε_{t-2} , etc. This is also true for ARMA models with MA terms, but because that’s such a standard type of model, the recursion is handled internally by the **BOXJENK** instruction. For a non-standard recursive model, you’ll have to write the **FRML** carefully to make sure that it is calculated the way you want.

One obvious problem is that the recursion has to start somewhere. If we begin at $t = 2$ (which is the first data point where y_{t-1} is available), what can we use for ε_{t-1} ? Since it’s a residual, the most obvious choice is 0. The following gets us started:

```
dec series eps
clear(zeros) eps
```

The series **EPS** will be used for the generated time series of residuals.

In Example 3.6, we’ll again use the inflation rate over the period from 1983 on as the dependent variable:

```
set pi = 400.0*log(ppi/ppi{1})
```

To the three parameters in (3.8), we’ll add an intercept to allow for a non-zero mean.¹⁴ We’ll call that **C0**. So the **NONLIN** instruction to declare the non-linear parameters is

```
nonlin c0 alpha beta gamma
```

¹⁴Note, however, that the bilinear term $y_{t-1}\varepsilon_{t-1}$ has a non-zero expectation. A bilinear term where the y is dated *before* the ε will have zero expected value.

The following is probably the simplest way to do the FRML:

```
frml bilinear pi = z=c0+alpha*pi{1}+beta*eps{1}+gamma*pi{1}*eps{1}, $
    eps=pi-z, z
```

In the end, the FRML needs to provide the explanatory part of the equation, but we need that to compute the current residual for use at the next data point.¹⁵ Thus the three-step calculation, generating into the (REAL) variable Z the systematic part, computing the current value of EPS using the dependent variable and the just-computed value of Z, then bringing back the Z to use as the return value of the FRML. If we didn't add that , Z to the end return value would be the last thing computed, which would be EPS.

Obvious guess values here would be either the results from an AR(1) model (β and γ both zero) or from an ARMA(1,1) model (only γ zero). We'll use the ARMA. To do this, all we have to do is peg GAMMA to 0 and let NLLS handle it:

```
nonlin c0 alpha beta gamma=0.0
nlls(frml=bilinear) pi 1983:1 *
```

We then relax the restrictions to get our model:

```
nonlin c0 alpha beta gamma
nlls(frml=bilinear, iters=500) pi 1983:1 *
```

The before and after estimates (cut down to the key values) are

Sum of Squared Residuals		1676.4146107			
Variable	Coeff	Std Error	T-Stat	Signif	

1. C0	3.063121504	0.713047291	4.29582	0.00003615	
2. ALPHA	-0.410666169	0.171062737	-2.40068	0.01794100	
3. BETA	0.755033543	0.125563348	6.01317	0.00000002	

Sum of Squared Residuals		1622.0602194			
Variable	Coeff	Std Error	T-Stat	Signif	

1. C0	3.170708712	0.695829116	4.55673	0.00001293	
2. ALPHA	-0.561753920	0.102134138	-5.50016	0.00000023	
3. BETA	0.926950151	0.043157032	21.47854	0.00000000	
4. GAMMA	0.007104857	0.003221804	2.20524	0.02940923	

The bilinear model fits somewhat better than the ARMA.¹⁶ From the t -statistic, the bilinear term is significant at standard levels, but just barely. However, the sign on γ would appear to be “wrong” to explain non-linearity due to some form of inflation targeting—a positive value of γ means that large residuals

¹⁵%RESIDS doesn't get defined until the end of the calculations, so you can't use it here.

¹⁶If you use **BOXJENK** to estimate the ARMA model, you'll get effectively the same coefficients for the AR and MA, but a different constant because the “constant” in the model used by **BOXJENK** is the process mean, not the intercept in a reduced form equation.

(of either sign) tend to produce higher values for π in the next period. If we re-estimate ending at 2008:2 (just before the big negative outlier), we get

	Variable	Coeff	Std Error	T-Stat	Signif

1.	C0	2.957252096	0.735882030	4.01865	0.00011494
2.	ALPHA	-0.213433697	0.264220271	-0.80779	0.42116963
3.	BETA	0.597679361	0.217233787	2.75132	0.00707134
4.	GAMMA	-0.030825660	0.021977365	-1.40261	0.16389334

so the positive γ seems to have been created by that outlier. You might ask why we didn't just "SMPL" the outlier out of the data set (with the option `SMPL=T<>2008:3`). With a recursively-defined function, you can't really do that. If you try to use `SMPL` this way, it will cut the estimation off at the end of 2008:2 anyway—since `EPS` doesn't get defined for 2008:3, it's not available to calculate the model at 2008:4, so `EPS` can't be computed for 2008:4, and, extending this, the remainder of the sample can't be computed. You could "dummy out" the data point by adding the dummy variable created with

```
set dummy = t==2008:4
```

to the model.¹⁷ What that does, however, is forcibly make `EPS` equal to zero at 2008:3, which might cause problems explaining the data in the periods immediately after this. Since it's near the end of the data set, simply cutting the sample before the big outlier is probably the best approach, and the results would show that, again, we haven't come up with a good way to describe the inflation rate with a nonlinear time series model.

¹⁷As an exercise, you might want to try to incorporate that.

3.10 Tips and Tricks

3.10.1 Understanding Computer Arithmetic

The standard representation for “real” numbers in statistical computations for roughly the past 40 years has been “double precision”. This uses 64 bits to represent the value. The older “single precision” is 32 bits. Before the introduction of specialized “floating point processors”, both the single and double precision floating point calculations had to be done using blocks of shorter integers, similar to the way that children are taught to multiply two four-digit numbers. If you compare the amount of work required to multiply two two-digit numbers by hand vs the amount required to do two four-digit numbers, you can see that single precision calculations were quite a bit faster, and were sometimes chosen for that reason. However, particularly when used with time series data, they were often inadequate—even a relatively short univariate autoregression on (say) log GDP wouldn’t be computed accurately at single precision.

There are three principal problems which are produced due to the way computers do arithmetic: *overflow*, *underflow* and *loss of precision*. The standard now for double precision number is called IEEE 754. This uses 1 sign bit, 11 exponent bits and 53 “significand” bits, with a lead 1 bit assumed, so 52 are actually needed.¹⁸ The value is represented in scientific notation as (in binary, though we’ll use standard notation)

$$(sign) \times 1.xxxxxxxxxxxxxxxxx \times 10^{power}$$

The range of the power is from -308 to +308. An *overflow* is a calculation which makes the power larger than +308, so there’s no valid double precision representation. When this happens, RATS and most other software will treat the result as “infinity”.¹⁹ An *underflow* is a calculation which makes the power less than -308. Here the standard behavior is to call the result zero.

Most statistical calculations are able to steer clear of over- and underflow, typically by using logs rather than multiplying—the main cause of these conditions is multiplying many large or small numbers together, so adding logs avoids that. However, there are situations (particularly in Bayesian methods) where the actual probabilities or density functions are needed, and you may have to exercise some care in doing calculations in those cases.

A more serious problem is *loss of precision*. What’s the value of

$$1.000000000000000001 - 1.0$$

If you’re a computer doing double-precision arithmetic, it’s 0 because it doesn’t have the 17 significant digits available to give different representations to the

¹⁸The number 0 is represented by all zero bits.

¹⁹The IEEE standard has special codings for infinity and other “denormals” such as NaN (not a number) for a result such as $\sqrt{-1}$.

two values on either side of the $-$. When you subtract two large and almost equal numbers, you may lose almost all the precision in the result. Take the following situation—this is (to 10 digits) the cross product matrix of two lags of GDP from the data set. If we want to do a linear regression with those two lags as the regressors, we would need to invert this matrix.

14872.33134	14782.32952
14782.32952	14693.68392

You can compute the determinant of this²⁰ using all ten digits, and the same rounding to just five with

```
disp 14872.33134*14693.68392-14782.32952^2
disp 14872.*14694.-14782.^2
```

The results are *very* different:

12069.82561
21644.00000

Even though the input values in the second case were accurate to five digits, the end result isn't even correct to a *single* digit. And this is just a 2×2 case.

Linear regressions in RATS and most other software are set up to avoid these types of problems where possible. For instance, specialized inversion routines are used. While theoretically, the sum of squared residuals could be computed as

$$\mathbf{y}'\mathbf{y} - \mathbf{y}'\mathbf{X}\hat{\beta}$$

that's "subtracting two big numbers to create a small one" that can cause precision problems. Instead, RATS computes the individual residuals and uses those to compute the sum of squares.

It's much easier to work around potential problems when the calculations are as well-structured as a linear regression. It's much harder to do this with non-linear ones since it's not always obvious where the difficulties may lie. This is why it's important to understand the usefulness of minor changes to the model discussed in Section 3.4.

3.10.2 The instruction NLPAR

The most important controls for non-linear estimation are the `ITERS`, `CVCRIT` and `METHOD` options on the estimation instructions. There are, however, quite

²⁰RATS uses a different method for doing the inversion.

a few other controls which can be used for particularly troublesome estimation problems. Rather than add these (rarely used) tweaks in each estimation instruction, they are handled by the separate instruction **NLPAR**.

One option on **NLPAR** that changes fundamentally how convergence is determined is

`CRITERION=[COEFFICIENTS]/VALUE`

Convergence occurs if the change in the `COEFFICIENTS` or `VALUE` is less than the number specified by the `CVCRT` on the estimation instruction.

In most cases, we want the coefficients to be well-estimated, as that's usually the main interest. The default is thus `CRITERION=COEFFICIENTS`, so the process doesn't converge until the change in each individual coefficient is small. However, in some cases, you don't need that—you just need to be reasonably close to the optimum. Using **NLPAR(CRIT=VALUE)** changes the convergence test to look only at the function value, and not the coefficients, and is usually met more easily. When might that be reasonable? Perhaps you're only doing some type of grid search and need only a reasonable approximation to the optimum. In the examples we did in this chapter, that wouldn't be necessary, but a grid search with a much more complicated function might take hours. Cutting that by a 1/3 by using a looser fit might be noticeable. In some cases, you may have a function where an optimum at a boundary as described on page 64 might be possible. In that case, changes of a parameter on the order of 1000's might have no noticeable effect on the function value so convergence on coefficient values would *never* be met. However, a better approach in a case like that is to either re-parameterize the model so the (new) parameter is bounded rather than unbounded, or to simply peg it at a large value of the proper sign (since it should be clear the optimum is at one of the infinities) and estimate the rest of the model.

Some other options control the *sub-iteration* process using by **NLLS** and most other non-linear estimation instructions. As we describe on page 66, **NLLS** does not always take a full Gauss-Newton step, since a full step might actually increase the sum of squares. Instead, it moves in the same direction as the G-N step, often taking a full step, but sometimes a shorter one. It's looking for a point where the new sum of squares function is lower than the current one, and where certain other criteria are met. The process of choosing how far to go in the selected direction is known as sub-iteration. Gauss-Newton is usually well enough behaved that adjustments to this aren't necessary—there are other types of optimization algorithms for which this isn't true. The main option in this category is `EXACTLINESEARCH`. By default, the subiteration process is generally to start with a full step—if that doesn't work (sum of squares increases), take a half-step, test that, take a quarter step, etc. until a point is found at which the sum of squares is better than it was. With `EXACTLINESEARCH`, instead of this simple process of searching for a place where the sum of squares

is better, it searches along the direction for the place where it's best. That sounds like a good idea, but in fact, is usually just a waste of calculations—since it's not finding the optimum of the function itself (just the optimum in one direction from a certain point), the added time required rarely pays off. It can sometimes be helpful in big models with dozens of parameters, but almost never makes sense with small ones.

One option which was added with RATS version 8.1 is the `DERIVES` option:

`DERIVES=[FIRST]/SECOND/FOURTH`

NLLS and most other `FRML`-based estimation instructions use numerical methods when they need derivatives. While some functions have analytical derivatives, many don't, or the analytical derivatives are too complicated to be calculated feasibly.²¹ An approximation to the derivatives of ε_t with respect to θ at θ_0 require that we compute ε_t at θ_0 and at nearby points. These are often quite accurate, but sometimes might not be. The proper amount by which to “perturb” θ isn't known, and smaller isn't necessarily better—too small a change and the calculation might run into the loss of precision problem from Section 3.10.1. The default is `DERIVES=FIRST`, which does the simple arc-derivative calculation:

$$f'(\theta) \approx \frac{f(\theta + h) - f(\theta)}{h}$$

The h is chosen differently for each parameter based upon the information known about it at the time. This requires one extra full function evaluation per parameter to get the partial derivatives. With `DERIVES=SECOND`, the calculation is done using

$$f'(\theta) \approx \frac{f(\theta + h) - f(\theta - h)}{2h}$$

This requires *two* extra function evaluations per parameter, thus doubling the time required for computing the derivatives. However, it's more accurate as it eliminates a “second order” term. It thus can be done with a slightly larger value of h , which makes it less likely that precision issues will come up.

`DERIVES=FOURTH` is a four-term approximation which is still more accurate at the cost of four times the calculation versus the simple numerical derivatives.

Again, in most cases, changing this won't help, but it's available for problems where it appears that the accuracy of the derivatives seems to be an issue.

²¹For instance, the derivatives of ε_t in the bilinear model are a function of the derivatives at all preceding time periods.

3.10.3 The instruction SEED

The purpose of seeding the random number generator is to ensure that you generate the same data from when you do simulations. After all, computers are not capable of generating truly random numbers—any sequence generated is actually a deterministic sequence. If you are aware of the algorithm used to generate the sequence, all values of the sequence can be calculated by the outside observer. What computers generate are known as “pseudo-random” numbers in the sense that the sequence is deterministic if you know the algorithm but otherwise are indistinguishable from those obtained from independent draws from a prespecified probability distribution.

In this manual, we use **SEED** in all the instructions which generate data so you will get exactly the same data that we do. RATS uses a “portable” random number generator which is designed to generate identical sequences on any computer given the seed. If you don’t use **SEED**, the seed for the random number generator is initialized using date and time when the program is executed, so it will be different each time and thus will generate a completely different set of data.

Example 3.1 Simple nonlinear regressions

These the simple non-linear least squares regressions from Sections 3.2 and 3.3.

```
cal(q) 1960:1
all 2012:4
open data quarterly(2012).xls
data(org=obs,format=xls)
*
set pi = 100.0*log(ppi/ppi{1})
set y = .001*rgdp
*
* Power function with GDP in inflation equation
*
nonlin b0 b1 b2 gamma
frml pif pi = b0+b1*pi{1}+b2*y{1}^gamma
linreg pi
# constant pi{1} y{1}
compute b0=%beta(1),b1=%beta(2),b2=%beta(3),gamma=1.0
nlls(frml=pif) pi
*
* Power function with interest rates
*
nonlin a0 a1 a2 delta
linreg tblyr
# constant tblyr{1} tb3mo{1}
frml ratef tblyr = a0+a1*tblyr{1}+a2*(tb3mo{1})^delta
compute a0=%beta(1),a1=%beta(2),a2=%beta(3),delta=1.0
nlls(frml=ratef) tblyr
*
test(title="Test of linearity")
# 4
# 1.0
*
set testsr 1 100 = .1*t
set lreffect 1 100 = 0.0
set lower 1 100 = 0.0
set upper 1 100 = 0.0
*
do t=1,100
  summarize(noprint) $
    %beta(3)*%beta(4)*testsr(t)^(%beta(4)-1)/(1-%beta(2))
  compute lreffect(t)=%sumlc
  compute lower(t)=%sumlc-2.00*sqrt(%varlc)
  compute upper(t)=%sumlc+2.00*sqrt(%varlc)
end do t
*
scatter(smpl=testsr>=2.0.and.testsr<=6.0,style=lines,vgrid=1.0,$
  footer="Long-run effect using non-linear regression") 3
# testsr lreffect
# testsr lower / 2
# testsr upper / 2
```

```
linreg tb1yr
# constant tb1yr{1} tb3mo{1}
summarize(title="Long-run effect using linear regression") $
  %beta(3)/(1-%beta(2))
```

Example 3.2 Sample STAR Transition Functions

These generate and graph the sample transition functions described in Section 3.5.

```
set y 1 201 = (t-100)/201.
compute c=0.0,gamma=10.0
set lstar = ((1 + exp(-gamma*(y-c))))^-1
set estar = 1 - exp(-gamma*(y-c)^2)
spgraph(footer="Shapes of STAR Transitions",vfields=1,hfields=2)
  scatter(header="LSTAR Model",style=line,vlabels="Theta")
  # y lstar
  scatter(header="ESTAR Model",style=line,vlabels="Theta")
  # y estar
spgraph(done)
```

Example 3.3 STAR Model with Generated Data

This estimates the LSTAR model with generated data from Section 3.6.

```

all 250
seed 2003
set eps 1 350 = %ran(1)
set(first=1.0) x 1 350 = $
    1.0+.9*x{1}+(-3.0-1.7*x{1})*%logistic(10.0*(x{1}-5.0),1.0)+eps
*
* Shift final 250 observations down
*
set y 1 250 = x(t+100)
*
graph(footer="The Simulated LSTAR Process")
# y
*
nonlin a0 a1 b0 b1 gamma c
frml lstar y = (a0+a1*y{1})+$
    (b0+b1*y{1})*%logistic(gamma*(y{1}-c),1.0)
*
* Guess values based upon linear regression
*
linreg y
# constant y{1}
compute a0=%beta(1),a1=%beta(2),b0=0.0,b1=0.0
compute c=0.0,gamma=5.0
*
nlls(frml=lstar) y 2 250
*
* Guess values for C and GAMMA from sample statistics with regressions
* from NLLS with those fixed.
*
stats y
compute c=%mean,gamma=1.0/sqrt(%variance)
nonlin a0 a1 b0 b1
nlls(frml=lstar) y 2 250
nonlin a0 a1 b0 b1 gamma c
nlls(frml=lstar) y 2 250
*
* Guess value for C from grid search with fixed value of GAMMA
*
stats(fractiles) y
compute gamma=2.0/sqrt(%variance)
compute ygrid=%sega(%fract05, (%fract95-%fract05)/19,20)
nonlin a0 b1 b0 b1
compute bestrss=%na
do for c = ygrid
    nlls(noprint,frml=lstar) y 2 250
    if .not.%valid(bestrss).or.%rss<bestrss
        compute bestrss=%rss,bestc=c
end do for c
*

```

```

disp "Guess Value used" bestc
*
compute c=bestc
nonlin a0 a1 b0 b1 gamma c
nlls(frml=1star) y 2 250
*
* Guess value for C from grid search with GAMMA estimated separately
*
stats(fractiles) y
compute gamma0=2.0/sqrt(%variance)
compute ygrid=%seqa(%fract05, (%fract95-%fract05)/19,20)
nonlin a0 b1 b0 b1 gamma
compute bestrss=%na
do for c = ygrid
    compute gamma=gamma0
    nlls(noprint, frml=1star) y 2 250
    if .not.%valid(bestrss).or.%rss<bestrss
        compute bestrss=%rss, bestc=c, bestgamma=gamma
end do for c
*
disp "Guess values used" bestc "and" bestgamma
*
compute c=bestc, gamma=bestgamma
nonlin a0 a1 b0 b1 gamma c
nlls(frml=1star) y 2 250
*
* Guess values for C and GAMMA from bivariate grid search
*
stats(fractiles) y
compute ygrid=%seqa(%fract05, (%fract95-%fract05)/19,20)
compute ggrid=%exp(%seqa(log(.25), .1*log(100), 11))/(%fract75-%fract25)
nonlin a0 b1 b0 b1
compute bestrss=%na
do for c = ygrid
    do for gamma = ggrid
        nlls(noprint, frml=1star) y 2 250
        if .not.%valid(bestrss).or.%rss<bestrss
            compute bestrss=%rss, bestc=c, bestgamma=gamma
        end do for gamma
    end do for c
end do for c
*
disp "Guess values used" bestc "and" bestgamma
compute c=bestc, gamma=bestgamma
nonlin a0 a1 b0 b1 gamma c
nlls(frml=1star) y 2 250

```

Example 3.4 Smooth Transition Break

This an example of a AR model with a smooth transition break in the mean from Section 3.7.

```

cal(q) 1960:1
all 2012:4
open data quarterly(2012).xls
data(org=obs,format=xls)
*
set pi = 400.0*log(ppi/ppi{1})
graph(footer="Annualized Inflation Rate (Measured by PPI)", $
    grid=(t==1983:1))
# pi
*
@bident pi 1983:1 *
*
linreg pi 1983:1 *
# constant pi{1 to 4}
*
@regcorrs(qstats,footer="Residuals from AR(4) Model")
*
@startest(p=4,d=1) pi 1983:1 *
@startest(p=4,d=2) pi 1983:1 *
@startest(p=4,d=3) pi 1983:1 *
*
linreg pi 1983:1 *
# constant pi{1 to 4}
frml(lastreg,vector=b1) philf
frml(lastreg,vector=b2) phi2f
*
nonlin(parmset=starparms) gamma c
nonlin(parmset=regparms) b1 b2
*
frml glstar = %logistic(gamma*(pi{2}-c),1.0)
frml star pi = g=glstar,philf+g*phi2f
*
stats pi 1983:1 *
compute c=%mean,gamma=1.0/sqrt(%variance)
*
nlls(parmset=regparms,frml=star,noprint) pi 1983:1 *
nlls(parmset=regparms+starparms,frml=star,print) pi 1983:1 *
*
stats(fractiles) pi 1983:1 *
*
nonlin(parmset=gammaonly) gamma
*
compute bestrss=%na
do for c = %sega(%fract10, (%fract90-%fract10)/19,20)
    compute gamma=1.0/sqrt(%variance)
    nlls(parmset=regparms,frml=star,noprint) pi 1983:1 *
    nlls(parmset=regparms+gammaonly,frml=star,noprint) pi 1983:1 *
    if .not.%valid(bestrss).or.%rss<bestrss

```

```

        compute bestrss=%rss,bestc=c,bestgamma=gamma
end dofor
*
disp "Grid choices" bestc bestgamma
*
compute c=bestc,gamma=bestgamma
nlls (parmset=regparms,frml=star,noprint) pi 1983:1 *
nlls (parmset=regparms+starparms,frml=star,print) pi 1983:1 *
*
* Compare residuals
*
set nllsresids = %resids
linreg pi 1983:1 *
# constant pi{1 to 4}
set olsresids = %resids
graph(footer="Comparison of STAR and OLS Residuals",$
      key=upleft,klabels=| "STAR", "OLS" | ) 2
# nllsresids
# olsresids
*
* Test just with the data through 2007:4
*
@startest (p=4,d=2) pi 1983:1 2007:4

```

Example 3.5 LSTAR Model for Inflation

This attempts to fit an LSTAR model to the U.S. inflation rate. This is described in detail in section 3.8.

```

cal(q) 1960:1
all 2012:4
open data quarterly(2012).xls
data(org=obs,format=xls)
*
set pi = 400.0*log(ppi/ppi{1})
graph(footer="Annualized Inflation Rate (Measured by PPI)", $
    grid=(t==1983:1))
# pi
*
@bident pi 1983:1 *
*
linreg pi 1983:1 *
# constant pi{1 to 4}
*
@regcorrs(qstats,footer="Residuals from AR(4) Model")
*
@startest(p=4,d=1) pi 1983:1 *
@startest(p=4,d=2) pi 1983:1 *
@startest(p=4,d=3) pi 1983:1 *
*
linreg pi 1983:1 *
# constant pi{1 to 4}
frml(lastreg,vector=b1) philf
frml(lastreg,vector=b2) phi2f
*
nonlin(parmset=starparms) gamma c
nonlin(parmset=regparms) b1 b2
*
frml glstar = %logistic(gamma*(pi{2}-c),1.0)
frml star pi = g=glstar,philf+g*phi2f
*
stats pi 1983:1 *
compute c=%mean,gamma=1.0/sqrt(%variance)
*
nlls(parmset=regparms,frml=star,noprint) pi 1983:1 *
nlls(parmset=regparms+starparms,frml=star,print) pi 1983:1 *
*
stats(fractiles) pi 1983:1 *
*
nonlin(parmset=gammaonly) gamma
*
compute bestrss=%na
do for c = %sega(%fract10, (%fract90-%fract10)/19,20)
    compute gamma=1.0/sqrt(%variance)
    nlls(parmset=regparms,frml=star,noprint) pi 1983:1 *
    nlls(parmset=regparms+gammaonly,frml=star,noprint) pi 1983:1 *
    if .not.%valid(bestrss).or.%rss<bestrss

```

```

        compute bestrss=%rss,bestc=c,bestgamma=gamma
end dofor
*
disp "Grid choices" bestc bestgamma
*
compute c=bestc,gamma=bestgamma
nlls (parmset=regparms,frml=star,noprint) pi 1983:1 *
nlls (parmset=regparms+starparms,frml=star,print) pi 1983:1 *
*
* Compare residuals
*
set nllsresids = %resids
linreg pi 1983:1 *
# constant pi{1 to 4}
set olsresids = %resids
graph(footer="Comparison of STAR and OLS Residuals",$
      key=upleft,klabels=|"STAR","OLS"|) 2
# nllsresids
# olsresids
*
* Test just with the data through 2007:4
*
@startest(p=4,d=2) pi 1983:1 2007:4
*
set threshvar = pi{1}+pi{2}
linreg pi 1983:1 *
# constant pi{1 to 4}
@regstrtest(threshold=threshvar) 1983:1 *
frml glstar = %logistic(gamma*(threshvar-c),1.0)
*
stats threshvar 1983:1 *
compute c=%mean,gamma=1.0/sqrt(%variance)
*
nlls (parmset=regparms,frml=star,noprint) pi 1983:1 *
nlls (parmset=regparms+starparms,frml=star,print) pi 1983:1 *

```

Example 3.6 Bilinear Model

This is an example of a bilinear model from Section 3.9.

```

cal(q) 1960:1
all 2012:4
open data quarterly(2012).xls
data(org=obs, format=xls)
*
set pi = 400.0*log(ppi/ppi{1})
*
dec series eps
clear(zeros) eps
*
nonlin c0 alpha beta gamma
frml bilinear pi = z=c0+alpha*pi{1}+beta*eps{1}+gamma*pi{1}*eps{1},$
    eps=pi-z, z
*
* Estimate ARMA model
*
nonlin c0 alpha beta gamma=0.0
nlls(frml=bilinear) pi 1983:1 *
*
* Estimate bilinear model
*
nonlin c0 alpha beta gamma
nlls(frml=bilinear) pi 1983:1 *
*
* Estimate with truncated sample
*
nlls(frml=bilinear) pi 1983:1 2008:2

```

Maximum Likelihood Estimation

Suppose you wanted to estimate parameters (β and σ) in the process:

$$y_t = X_t\beta + \varepsilon_t; \varepsilon_t \sim N(0, \sigma^2) \quad (4.1)$$

The obvious way to do this is least squares, which can be done using the **LINREG** instruction. An alternative approach to parameter estimation is maximum likelihood. The following derivation can be found in any elementary econometrics text: the likelihood for entry t is:

$$\frac{1}{\sqrt{2\pi}} (\sigma^2)^{-1/2} \exp\left(-\frac{1}{2\sigma^2} (y_t - X_t\beta)^2\right)$$

If the entries are independent, then the log likelihood for the full sample is:

$$\sum_t -\frac{1}{2} \log(2\pi) - \frac{1}{2} \log \sigma^2 - \frac{1}{2\sigma^2} (y_t - X_t\beta)^2 = -\frac{T}{2} \log(2\pi) - \frac{T}{2} \log \sigma^2 - \frac{1}{2\sigma^2} \sum_t (y_t - X_t\beta)^2$$

By inspection, the maximizer of this for β is the minimizer of the sum of squares:

$$\sum_t (y_t - X_t\beta)^2$$

regardless of the value of σ^2 . The first order condition for maximization over σ^2 is

$$-\frac{T}{2\sigma^2} + \frac{1}{2\sigma^4} \sum_t (y_t - X_t\beta)^2 = 0 \Rightarrow \sigma^2 = \frac{1}{T} \sum_t (y_t - X_t\beta)^2$$

Thus, for model (4.1), the maximum likelihood and least squares give identical estimates. This is specific to the assumption that the residuals are Normal. Suppose, instead, that the residuals were assumed to be t with ν degrees of freedom to allow for heavier tails in the error process. The likelihood for y_t is now:

$$K (\sigma^2(\nu - 2)/\nu)^{-1/2} (1 + (y_t - X_t\beta)^2 / (\sigma^2/(\nu - 2)))^{-(\nu+1)/2} \quad (4.2)$$

where the (rather complicated) integrating constant K depends upon ν . This is parameterized with σ^2 as the variance of the ε process. The only term in the full sample log likelihood which depends upon β is:

$$-\frac{(\nu + 1)}{2} \sum_t \log (1 + (y_t - X_t\beta)^2 / (\sigma^2/(\nu - 2)))$$

Unlike the Normal, you can't optimize β separately from σ^2 , and there are no sufficient statistics for the mean and variance, so the log likelihood for a $\{\beta, \sigma^2\}$ combination can only be computed by summing over the full data set.

There are many other types of models for which the log likelihood can't be simplified beyond the sum of the log likelihood elements, even if the error process which is basically Normal—to simplify to least squares or weighted least squares, the variance has to be constant or at least a function that depends upon exogenous variables but not free parameters.

There are some specialized instructions for estimating particular models by maximum likelihood, such as **DDV** for probit, **LDV** for tobit and related models, **BOXJENK** for ARIMA and related models, **GARCH** for specific types of ARCH and GARCH models. However, the instruction which is most flexible and thus most commonly used is **MAXIMIZE**.

4.1 The MAXIMIZE instruction

MAXIMIZE requires that you define a **FRML** which evaluates at entry t the function $f_t(y_t, x_t, \beta)$ where the log likelihood takes the form

$$F(\beta) \equiv \log L(Y|X, \beta) = \sum_{t=1}^T f_t(y_t, x_t, \beta) \quad (4.3)$$

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

Now **MAXIMIZE** doesn't know whether the **FRML** that you provided is, in fact, the log likelihood element for t . It will try to maximize (4.3) no matter what you are actually calculating. And, in most cases, it will give the proper optimized values for β . F being the actual log likelihood matters to the interpretation of the output: what are displayed as the "standard errors" and "t-statistics" are that only if F is the log likelihood, or at least the log likelihood up to an additive constant which doesn't depend upon β . However, the **ROBUSTERRO** option can be used to compute an asymptotically valid covariance matrix if F isn't the true log likelihood.

```
MAXIMIZE(options) frml start end
```

where

frml A previously defined formula

start end The range of the series to use in the estimation

The key options for our purposes are:

```

METHOD=BHHH/[BFGS]/SIMPLEX/GENETIC/EVALUATE
PMETHOD=BHHH/BFGS/[SIMPLEX]/GENETIC
ITERATIONS=limit on iterations
PITERATIONS=limit on preliminary iterations
CVCRIT=convergence limit [0.00001]

ROBUSTERRORS/[NOROBUSTERRORS]

```

The set-up of a maximum likelihood estimation is very similar to that of **NLLS**. The essential difference is that the **FRML** instruction defines the log likelihood that you want to maximize. Otherwise the steps to perform maximum likelihood estimation are just like those of nonlinear least squares:

1. Define the parameters to be estimated using **NONLIN** instruction.
2. Define the log likelihood for observation t using a **FRML** instruction.
3. Set the initial values of the parameters using the **COMPUTE** command.
4. Use the **MAXIMIZE** instruction to maximize the sum across time of the formula in Step 2.

How does **MAXIMIZE** solve the optimization of (4.3)? At a minimum, F must be a continuous function of β . The **SIMPLEX** and **GENETIC** choices for the **METHOD** (and **PMETHOD**) options require nothing more than that.

Of course, the same types issues that arise with **NLLS** (page 63) can occur with **MAXIMIZE**. Be sure to use good initial guesses and ensure that you are finding the global maximum.

The two main algorithms used for optimization are **BFGS** and **SIMPLEX**. These are described in greater detail in this chapter's *Tips and Tricks* (Section 4.4). These are often used together, with **SIMPLEX** used for preliminary iterations (**PMETHOD**=**SIMPLEX**, **PITERATIONS**=*number of preliminary iterations*) and **BFGS** (the default choice for **METHOD**) used to actually estimate the model. **SIMPLEX** is slower, and, because it doesn't assume differentiability, can't provide estimates of the standard errors, and thus is rarely used as the *main* estimation method. However, it is much less sensitive to bad guess values, and so is handy for getting the estimation process started.

If you have a program written for an older version of RATS, you might see two otherwise identical **MAXIMIZE** instructions at some point, one with **METHOD**=**SIMPLEX**, the second with **METHOD**=**BFGS**. Since version 6, **MAXIMIZE** has had the **PMETHOD** option to allow the single instruction to use the two methods sequentially, so you should take advantage of that. However, note that many optimization problems don't *need* preliminary simplex iterations.

As a simple example (Example 4.1), we'll do the same power function on the interest rates as in Section 3.3, but we'll allow for t distributed rather than

Normal errors. It's helpful to start out with the Normal model, which can be estimated with **NLLS** as before:

```
nonlin a0 a1 a2 delta
linreg tb1yr
# constant tb1yr{1} tb3mo{1}
frml ratef tb1yr = a0+a1*tb1yr{1}+a2*(tb3mo{1})^delta
compute a0=%beta(1),a1=%beta(2),a2=%beta(3),delta=1.0
nlls(frml=ratef) tb1yr 2 *
```

How do we extend this to allow for t errors? First, as we saw above, we can't simply "concentrate" out the variance. So we need to add two parameters, σ^2 and ν . For convenience, we'll separate the original and new parameters into two **PARMSET**'s.

```
nonlin(parmset=baseparms) a0 a1 a2 delta
nonlin(parmset=tparms) sigsq nu
```

We can take the guess value for **SIGSQ** from the non-linear least squares and start with **NU** at a fairly high value, so that we will be approximately at the same location as we would have with Normal residuals:

```
compute sigsq=%seesq,nu=20.0
```

Now, we need to define a **FRML** which evaluates the log likelihood for the t rather than simply the right-hand-side of an equation. Although one *could* write out the density for the t as in (4.2), it's simpler (and faster) to use the existing **%LOGTDENSITY** function. This takes (in order) the variance, residual and degrees of freedom as its arguments. Since we already have the existing **RATEF** formula to evaluate the right-side expression, this is fairly simple:

```
frml logl = %logtdensity(sigsq,tb1yr-ratef,nu)
```

Note that all **RATS** built-in density and log density functions like **%LOGTDENSITY** and **%LOGDENSITY** include *all* integrating constants, not just the ones that depend upon the parameters. What **RATS** reports as the log likelihood is indeed the full log likelihood given the model and parameters. If you try to replicate published results, you may find that the reported log likelihoods are quite different than you get from **RATS** even if the coefficients are effectively identical. If that's the case, it's usually because the authors of the original work left out some constants that had no effect on the results otherwise.

The optimization can be done with:

```
maximize(parmset=baseparms+tparms,itors=300) logl 2 *
```

We ended up increasing the number of iterations, as the model hadn't converged in 100. The original output (with the default 100 iterations) was

MAXIMIZE - Estimation by BFGS				
NO CONVERGENCE IN 100 ITERATIONS				
LAST CRITERION WAS 0.0183646				
Quarterly Data From 1960:02 To 2012:04				
Usable Observations		211		
Function Value		-211.5020		
Variable	Coeff	Std Error	T-Stat	Signif

1. A0	-0.387364457	2.750515178	-0.14083	0.88800154
2. A1	0.986460486	0.028992266	34.02495	0.00000000
3. A2	0.440294321	2.735910770	0.16093	0.87214732
4. DELTA	0.067177518	0.545349075	0.12318	0.90196250
5. SIGSQ	0.983706526	0.801170498	1.22784	0.21950828
6. NU	2.451540091	0.514078793	4.76880	0.00000185

A common error that users make is to ignore those warnings about lack of convergence. *They're in all upper case for a reason.* This is an easy one to fix by increasing the iteration count.

One thing to note about **MAXIMIZE** and the BFGS algorithm is that if the model fails to converge in 100 iterations, and you simply select the **MAXIMIZE** again and re-execute, you will get roughly the same parameter estimates as if you allowed for 200 (or more) in the first go, but *not* the same standard errors. The BFGS estimate of the (inverse) Hessian is dependent upon the path taken to reach the optimum. If you re-execute the **MAXIMIZE**, the BFGS Hessian is re-initialized as a diagonal matrix. If the optimization was interrupted by the iteration limit when nearly converged, the information to update the Hessian is rather weak; the changes in both the gradient and the parameter vectors are small. In this case, doing 100 iterations, then another 100 gives:

MAXIMIZE - Estimation by BFGS				
Convergence in 11 Iterations. Final criterion was 0.0000095 <= 0.0000100				
Quarterly Data From 1960:02 To 2012:04				
Usable Observations		211		
Function Value		-211.5020		
Variable	Coeff	Std Error	T-Stat	Signif

1. A0	-0.387252416	0.050604638	-7.65251	0.00000000
2. A1	0.986378426	0.025989119	37.95351	0.00000000
3. A2	0.440550069	0.061171096	7.20193	0.00000000
4. DELTA	0.067414052	0.100235707	0.67256	0.50123027
5. SIGSQ	0.984391447	0.275353108	3.57501	0.00035021
6. NU	2.450853777	0.179995863	13.61617	0.00000000

If we do the same estimation starting with enough iterations to converge we get:

MAXIMIZE - Estimation by BFGS				
Convergence in 130 Iterations. Final criterion was 0.0000000 <= 0.0000100				
Quarterly Data From 1960:02 To 2012:04				
Usable Observations		211		
Function Value		-211.5019		
Variable	Coeff	Std Error	T-Stat	Signif

1. A0	-0.433719170	1.787260243	-0.24267	0.80825900
2. A1	0.986521104	0.031013227	31.80969	0.00000000
3. A2	0.486938369	1.756506306	0.27722	0.78161132
4. DELTA	0.060438285	0.279017340	0.21661	0.82851134
5. SIGSQ	0.984881664	0.785715851	1.25348	0.21002989
6. NU	2.450541777	0.503477442	4.86723	0.00000113

The estimation with the two sequential **MAXIMIZE**'s wasn't really even able to move off the results from the first successfully because the initial "diagonal" Hessian has the shape of the function completely wrong. Note that even though the coefficients appear to be quite different in the last two outputs, the function value itself, what we're trying to maximize, is almost identical, so the likelihood surface is *very* flat. That's reflected in the high standard errors in the second estimator which had enough iterations.

Of the various algorithms used by RATS for non-linear estimation, BFGS is the only one with the property that the covariance matrix is dependent upon the path used by the optimization algorithm. Since it is heavily used, not just by RATS, but by other software, this is a characteristic of the algorithm about which you need to be careful.

MAXIMIZE defines both %LOGL and %FUNCVAL as the value of the function at the final set of parameters. Assuming that you've set up the FRML to include all integrating constants (as mentioned above, %LOGTDENSITY does this), and you've estimated the model over the same range, then the log likelihood for **MAXIMIZE** and a simpler **NLLS** are comparable. If we want to compare the results from the t with the Normal, we need to be careful about the parameter count. While it would appear that the t has six free parameters, and the Normal has only four, that's not including the variance in the latter. The variance is estimated by **NLLS**, but in a second step given the other parameters rather than being included directly. So the t model adds just 1. The Normal is a special case of the t with $\nu = \infty$. If we ignore the fact that the restriction is on the boundary¹ we can get a likelihood ratio statistic for the Normal vs the t with $2(-211.5019 - -238.1723) = 53.3408$, which is clearly way out in the tails for one degree of freedom.

We can test the linearity hypothesis $\delta = 1$ with a Wald test either by

```
test(title="Test for linearity")
# 4
# 1.0
```

¹which violates one of the assumptions governing the most straightforward proof of the asymptotics of the likelihood ratio test.

since `DELTA` is the fourth parameter in the combined parameter set, or, with RATS 8.2 or later

```
summarize (parmset=baseparms+tparms, $
          title="Test for linearity") delta=1
```

These produce the identical results (the **TEST** in “squared” form):

Test for linearity			
Chi-Squared(1)= 11.339352 with Significance Level 0.00075882			
Test for linearity			
Value	-0.9395617	t-Statistic	-3.36740
Standard Error	0.2790173	Signif Level	0.0007588

We can also do a likelihood ratio test for $\delta = 1$ relatively easily: we just add a third `PARMSET` that pegs δ to the hypothesized value. We save the original log likelihood and re-estimate the model with the restriction:

```
nonlin (parmset=pegs) delta=1.0
compute loglunr=%logl
*
maximize (parmset=baseparms+tparms+pegs) logl 2 *
cdf (title="LR Test for delta=1") chisqr 2*(%logl-loglunr) 1
```

This gives us the rather remarkable result:

LR Test for delta=1	
Chi-Squared(1)=	0.576804 with Significance Level 0.44756761

which conflicts with the Wald test. Now unlike linear restrictions on linear models, there is no theorem which says that the Wald and Likelihood Ratio tests should be identical, but these aren’t even close. Apparently, the likelihood surface is even flatter than even the rather wide standard errors in the **MAXIMIZE** output suggest. Now, we knew from the beginning that this was likely a flawed model, and the results here would confirm that it’s not reliable for any real inference. Again, not all models work. The fact that you get converged estimates doesn’t help if the model itself isn’t good.

4.2 ARCH and GARCH Models

RATS includes a **GARCH** instruction that is capable of estimating many standard types of univariate and multivariate GARCH models. However, there are even more forms of GARCH that *aren’t* covered by the built-in **GARCH**, and these require **MAXIMIZE**. Thus, it makes sense to examine the process of estimating a simple GARCH model using the more general instruction, since extensions usually start with the more basic forms.

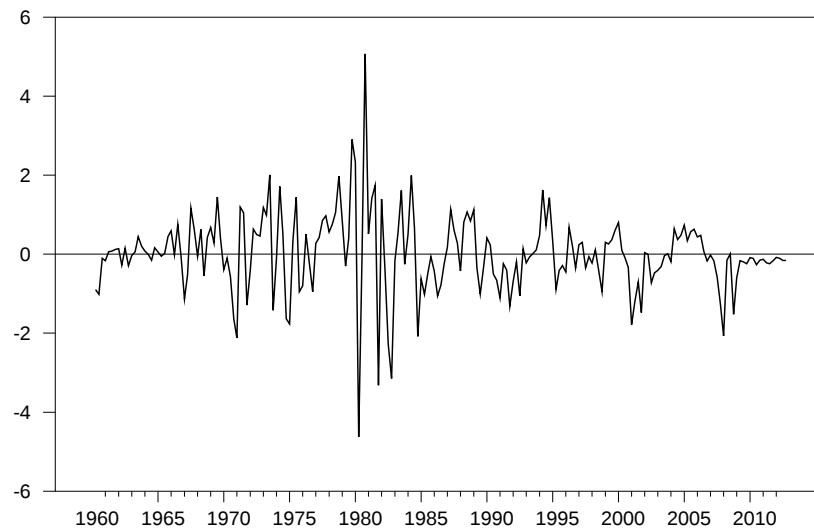


Figure 4.1: Standardized Residuals from Linear Regression

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

$$y_t = x_t\beta + \varepsilon_t \quad (4.4)$$

$$\varepsilon_t = v_t \sqrt{\alpha_0 + \alpha_1 \varepsilon_{t-1}^2} \quad (4.5)$$

where v_t is a zero-mean normal *i.i.d.* variable. Under these assumptions,

$$E_{t-1}\varepsilon_t = 0 \quad (4.6)$$

$$E_{t-1}\varepsilon_t^2 \equiv h_t = \alpha_0 + \alpha_1 \varepsilon_{t-1}^2 \quad (4.7)$$

We'll look at the regression model

$$LR_t = \beta_0 + \beta_1 LR_{t-1} + \beta_2 SR_{t-1} + \varepsilon_t$$

which is the model from Section 3.3 without the power on the SR term. In Example 4.2, we start with OLS estimates:

```
linreg tb1yr
# constant tb1yr{1} tb3mo{1}
```

We can compute and graph (Figure 4.1) the standardized residuals:

```
set stdu = %resids/sqrt(%seesq)
graph(footer="Standardized Residuals from Linear Regression")
# stdu
```

We see that we have a clear problem that there are several *very* large residuals (greater than 4 standard errors), with more that are larger than 2 than would be expected in just 200 data points. That, by itself, could be an indication that a fatter tailed distribution than the Normal might be appropriate, but, what

suggests the need for something more complicated is that most of the largest residuals (of various signs) are grouped together in a relatively short range of data. While the residuals might not be showing serial correlation (which is a measure of *linear* association), they don't appear to be independent—instead, the *squares* of the residuals appear to be serially correlated. A Lagrange multiplier (LM) test for ARCH disturbances was proposed by Engle (1982). After you have estimated the most appropriate model for y_t , save the residuals, then create the square of the residuals and regress these squared residuals on a constant and on m lagged values of the squared residuals. In our case, if $m = 4$:

```
set u    = %resids
set usq = u^2
linreg usq
# constant usq{1 to 4}
cdf(title="Test for ARCH") chisqr %trsquared 4
```

If there are no ARCH or GARCH effects, this regression will have little explanatory power so the coefficient of determination (the usual R^2) 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 m degrees of freedom. We can use the variance %TRSQUARED computed by **LINREG** as the test statistic. This turns out to be very significant:

Test for ARCH Chi-Squared(4)=	55.352434 with Significance Level 0.00000000
----------------------------------	--

so we would conclude that the residuals aren't (conditionally) homoscedastic, which is the null, and strongly suggests the presence of ARCH or GARCH effects.² Since this is a heavily-used test, there's a standard **@ARCHTEST** procedure to do it:

```
@archtest(lags=4, form=lm) u
```

which produces the same result as before. Note that the input to **@ARCHTEST** is the residual itself, *not* its square. Note also that we saved the original residuals from the **LINREG** into a separate series since the standard %RESIDS will be overwritten by the auxiliary regression.

Since our test seems to show the residuals show "ARCH" behavior, how do we adjust our estimation to allow for that? We can write $f(y_t|y_{t-1}, \dots, y_1)$ using

$$\begin{aligned}y_t &\sim N(x_t\beta, h_t) \\h_t &= a_0 + a_1\varepsilon_{t-1}^2 \\ \varepsilon_t &= y_t - x_t\beta\end{aligned}$$

²Again, it's important to understand what a test like this means. We haven't determined that there are, in fact, ARCH or GARCH effects, or (more particularly) that any specific ARCH or GARCH model is appropriate, just that the large residuals are clustering in a way that isn't compatible with a simple model (the null hypothesis) where the sizes of residuals are independent across time.

since ε_{t-1} (and thus h_t) is a function of data only through $t-1$. Thus, we can get the full sample likelihood by using the standard “trick” of writing $f(y_1, \dots, y_T)$ as $f(y_1)f(y_2|y_1) \dots f(y_T|y_{T-1}, \dots, y_1)$. In logs, this converts to a sum so

$$\log f(y_1, \dots, y_T) = \sum_{t=1}^T \log f_N(\varepsilon_t | h_t)$$

where $f_N(x|\sigma^2)$ is the Normal density function at x with mean 0 and variance σ^2 . The simplest way to compute $\log f_N(\text{eps}, h)$ in RATS is with the function `%LOGDENSITY(h, eps)` (note the parameter order).

We *could* directly write out the log likelihood at t by substituting everything out and getting

$$\log f(y_t | y_{t-1}, \dots) = f_N(y_t - x_t\beta | a_0 + a_1(y_{t-1} - x_{t-1}\beta)^2)$$

However, aside from the fact that writing a complicated formula like that accurately isn’t easy, it also evaluates ε_t twice, once at t , when it’s the residual and once at $t+1$ when it would be the lagged residual needed for computing h_{t+1} . The extra time required for computing the same value twice isn’t a major problem here, though it could be in other cases; the biggest problem is the lack of flexibility—if we want to change the “mean” function for the process, we’ll have to make the same change twice. It would be better if we could change it just once.

We’ll start by using **NONLIN** instructions to declare the five parameters: the three parameters in the regression model and two in the ARCH process. By splitting these up, we make it easier to change one part of the model separately from the other.

```
nonlin(parmset=meanparms) b0 b1 b2
nonlin(parmset=archparms) a0 a1
```

We’ll define the log likelihood in three parts:

```
frml efrml = tblyr-(b0+b1*tblyr{1}+b2*tb3mo{1})
frml hfrml = a0+a1*efrml{1}^2
frml logl = %logdensity(hfrml,efrml)
```

The “mean model” appears only in the `EFRML`, the “variance model” only in the `HFRML`, and the `LOGL FRML` really doesn’t need to know how *either* of those is computed. This is good programming practice—if different parts of a model can be changed independently of each other, try to build the model that way from the start.

We now need guess values for the parameters. We can’t simply allow the default 0’s, because if `A0` and `A1` are zero, `HFRML` is zero, so the function value is undefined—if the function value is undefined at the guess values, there’s really

no good way off of it. The first thing that **MAXIMIZE** does is to compute the formula to see which data points can be used. Without better guesses (for A_0 and A_1), the answer to that is none of them. You'll get the message:

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

In this case, there are fairly obvious guess values for the regression parameters as the coefficients from the **LINREG**. One obvious choice for A_0 and A_1 are the residual variance from the **LINREG** and 0, respectively, which means that the model starts out from the linear regression with fixed variance. Since 0 is a boundary value, however, it might be better to start with a positive value for A_1 and adjust A_0 to match the sample variance, thus:

```
linreg tb1yr
# constant tb1yr{1} tb3mo{1}
*
compute b0=%beta(1),b1=%beta(2),b2=%beta(3)
compute a0=%seesq/(1-.5),a1=.5
```

The instruction for maximizing the log likelihood is:

```
maximize(parmset=meanparms+archparms) logl 3 *
```

Note that we lose two observations: one for the lag in the mean model so ε_t isn't defined until 2 and one additional observation due to the lag ε_{t-1} in the ARCH specification, so we start at entry 3.

```
MAXIMIZE - Estimation by BFGS
Convergence in 19 Iterations. Final criterion was 0.0000078 <= 0.0000100
Quarterly Data From 1960:03 To 2012:04
Usable Observations 210
Function Value -226.8267
```

Variable	Coeff	Std Error	T-Stat	Signif

1. B0	0.091504296	0.092871014	0.98528	0.32448474
2. B1	1.010540358	0.173973445	5.80859	0.00000001
3. B2	-0.033379484	0.184926188	-0.18050	0.85675875
4. A0	0.386146972	0.047680337	8.09866	0.00000000
5. A1	0.350387587	0.122202087	2.86728	0.00414017

If we want to compare the ARCH model with the linear regression, we need to use the same sample range. The AIC for the ARCH model can be computed with

```
compute aicarch=-2.0*%logl+2.0*%nreg
```

while the same for the OLS is done with:

```
linreg tb1yr %regstart() %regend()
# constant tb1yr{1} tb3mo{1}
compute aicols=-2.0*%logl+2.0*(%nreg+1)
```

using `%REGSTART()` and `%REGENE()` to ensure we use the same range on the **LINREG** as we did on the **MAXIMIZE**. As with the earlier example of the t vs Normal, we need to count the variance as a separate parameter for OLS since the ARCH is explicitly modeling the variance. A comparison of the AIC values shows a clear edge to the ARCH:

```
disp "AIC-ARCH" @15 * .### aicarch
disp "AIC-OLS" @15 * .### aicols
```

```
AIC-ARCH      463.653
AIC-OLS       483.993
```

The same model can be estimated using the built-in **GARCH** instruction with:

```
garch(reg,q=1) / tb1yr
# constant tb1yr{1} tb3mo{1}
```

which is obviously much simpler than going through the setup for using **MAXIMIZE**. In general, where a built-in instruction is available for a model, it's a good idea to use it.

4.3 Using FRMLs from Linear Equations

Many users concentrate so much on condition (4.7) that they forget about (4.6)—that the residuals are also supposed to be serially uncorrelated. Although Engle's original ARCH paper used the inflation rate in the U.K. as its example, for years most empirical work using the more flexible GARCH model (Bollerslev (1986)) applied it to returns on financial assets where lack of serial correlation could almost be taken as a given. However, if you're applying an ARCH technique to some other type of data (such as macroeconomic data as we are here), it's important to also try to get the "mean model" correct.

One problem with the program above is that it keeps repeating the same linear equation specification. The first one is for illustration only, but there's a **LINREG** for guess values, a **LINREG** for the AIC comparison, the regression relation is coded into the **EFRML** formula, and the regressor list is repeated again on the **GARCH** instruction. We would be much better off if we could define the mean model once and have that used all the way through.

We saw this idea in Chapter 3, page 94. Here, we'll define both an **EQUATION**, and a **FRML** based upon the single linear specification. Right up at the top of Example 4.3, we'll do:

```
linreg tb1yr
# constant tb1yr{1} tb3mo{1}
equation(lastreg) meaneq
frml(lastreg,vector=beta) meanfrml
compute rstart=%regstart(),rend=%regend()
```

This

1. Defines the `EQUATION MEANEQ` with the form (and coefficients) taken from the regression.
2. Defines the `FRML MEANFRML` with the form taken from the regression, using `BETA(1)`, `BETA(2)` and `BETA(3)` for the three coefficients, with those three given the values from the regression.
3. Defines `RSTART` and `REND` as the estimation range of the regression.

The third of these is useful because if we add lags to the model, the estimation range will change, and `RSTART` will change automatically. We can then use `RSTART` to get the proper start entry on the **MAXIMIZE** instruction.

In addition to using a more flexible setup for the mean model, we'll use a GARCH model rather than the simpler ARCH. In a GARCH(1,1) model, the variance evolves as

$$h_t = c + a\varepsilon_{t-1}^2 + bh_{t-1} \quad (4.8)$$

This gives a smoother evolution to the variance than is possible with the simpler ARCH, and, in practice, works so much better that the ARCH is now rarely used. One major difference in programming is that, unlike the ARCH, the variance isn't computable using only the data and parameters: how do we compute h_1 (or more specifically h at the first entry in the estimation range)? Unlike the case of a moving average model, zero isn't an obvious "pre-sample" value, since the expected value of a variance isn't zero. The model can be solved to get a "long-run" value for the variance of

$$h_\infty = c/(1 - a - b)$$

except that won't exist if $a + b \geq 1$. And unlike an ARMA model, there is no "stationary" distribution for a GARCH process which can be used for doing full information maximum likelihood. In short, there is no single obvious log likelihood value given the data and the parameters—different programs will come up with somewhat different results given different choices for the pre-sample values for h . With a large data set, the differences are generally quite minor, but with a shorter one (and the roughly 200 data points in our data set is quite short for a GARCH model), they could be more substantial.

The RATS **GARCH** instruction uses the common choice of the estimate from a fixed variance model (that is, the linear regression) and that's what we'll show here. In order to avoid dropping data points to handle the ε_{t-1}^2 term, we will also use the same pre-sample value for that.

Because h now requires a lagged value of itself, we can't simply write out a `FRML` for (4.8). Instead, we have to create a separate series for the variances, and have the formulas use that for lags and reset that as its computed. The following will get us started: these create three series, one for the variances, one for the residuals and one for the squared residuals.

```

set h = %seesq
set u = %resids
set uu = %seesq

```

H and UU are initialized to the fixed variance from the **LINREG**; however, the only entries for which that matters are the pre-sample ones, as all others will be rewritten as part of the function evaluation.

The two parameter sets can be defined with

```

nonlin(parmset=meanparms) beta
nonlin(parmset=garchparms) c a b

```

and the log likelihood formula is again defined in three parts:

```

frml efrml = tbyr-meanfrml
frml hfrml = c+a*uu{1}+b*h{1}
frml logl = h=hfrml,u=efrml,uu=u^2,%logdensity(h,u)

```

Unlike Example 4.2, this now has the more flexible method of handling the mean model, so if we change the initial **LINREG**, the whole model will change.

If we look at the **LOGL** formula piece by piece we see that it does the following (in order) when evaluating entry T:

1. Evaluates and saves $H(T)$ using **HFRML**. This uses $UU(T-1)$ and $HH(T-1)$. When T is the first entry in the estimation range, those will be the values we put into the H and UU series by the **SET** instructions earlier
2. Evaluates and saves (into $U(T)$) the residual using **EFRML**
3. Saves the square of $U(T)$ into $UU(T)$
4. Finally, evaluates the log likelihood for T.

Why do we create a series for UU rather than simply squaring U when needed? It's all for that one pre-sample value—there is no residual available to be squared to get $U(T-1)^2$.

Why do we evaluate H first rather than U? In this model, it doesn't matter, but if you allow for an "M" effect (see Engle, Lilien, and Robins (1987)) you must compute current H first so it will be available for computing the residual. It's *never* wrong to compute H before E.

We now need guess values. The mean model is already done—the **FRML(LASTREG)** copies the **LINREG** coefficients into **BETA**. The following is a reasonable set of start values for the GARCH parameters:

```

compute a=.1,b=.6,c=%seesq/(1-a-b)

```

In practice, if there is a GARCH effect, the coefficient on the lagged variance (what we're calling B) tends to be larger than the one on the lagged squared residual (A).

We can estimate the model with

```
maximize (parmset=meanparms+garchparms, $
    pmethod=simplex, pitters=10) logl rstart rend
compute aicgarch=-2.0*%logl+2.0*%nreg
```

This model doesn't converge properly without the simplex iterations—if you want to test this, change it to `PITTERS=0` and re-do the program starting at the beginning (so you get the original guess values).

The results from estimation with the combination of simplex and BFGS are

MAXIMIZE - Estimation by BFGS				
Convergence in 27 Iterations. Final criterion was 0.0000009 <= 0.0000100				
Quarterly Data From 1960:02 To 2012:04				
Usable Observations	211			
Function Value	-187.4168			
Variable	Coeff	Std Error	T-Stat	Signif

1. BETA(1)	0.0368121628	0.0486053825	0.75737	0.44882940
2. BETA(2)	0.9346725646	0.1594455856	5.86202	0.00000000
3. BETA(3)	0.0584863698	0.1776252153	0.32927	0.74195283
4. C	0.0147378717	0.0105093858	1.40235	0.16080978
5. A	0.5084325110	0.1922289141	2.64493	0.00817072
6. B	0.5779414855	0.0930391861	6.21181	0.00000000

This is *much* better than the earlier ARCH model. This uses an extra data point (211 rather than 210) and has one extra parameter, but the gap in log likelihood between the ARCH and GARCH is huge. As before, we can do an AIC comparison with OLS using

```
linreg(equation=meaneq) * %regstart() %regend()
compute aicols=-2.0*%logl+2.0*(%nreg+1)
*
disp "AIC-GARCH" @15 * .### aicgarch
disp "AIC-OLS" @15 * .### aicols
```

This leaves no doubt as to which model the data prefer:

```
AIC-GARCH      384.730
AIC-OLS        483.993
```

We can do tests for the adequacy of the model using the standardized residuals and the squared standardized residuals:³

³The `DFC` options are 1 for the residuals because we have one lag of the dependent variable in the mean model, and 2 for the squared residuals because we have two lagged coefficients in the GARCH variance model.

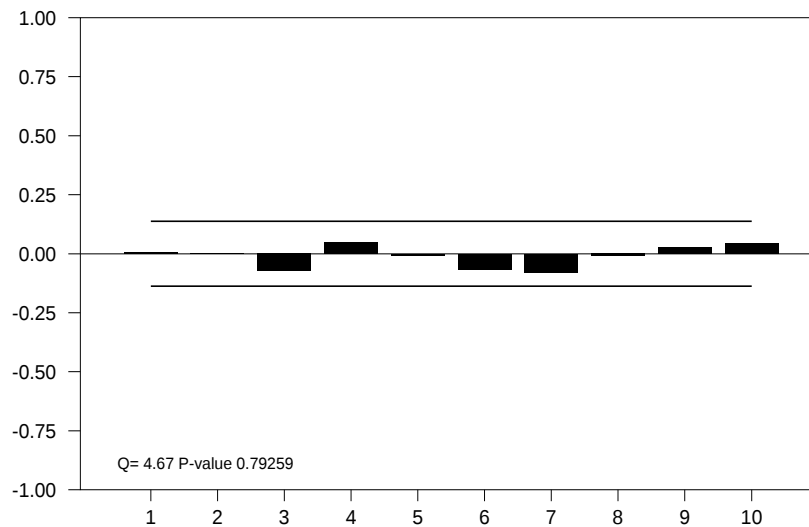


Figure 4.2: Standardized Squared Residuals

```
set stdu    = u/sqrt(h)
set stdusq  = stdu^2
*
@regcorrs(number=10,dfc=1,nocrits,qstat,$
  title="Standardized Residuals") stdu
@regcorrs(number=10,dfc=2,nocrits,qstat,$
  title="Standardized Squared Residuals") stdusq
```

The second of these (Figure 4.2) shows what we would like to see:

This is a test of any remaining ARCH or GARCH. If we applied this to the ARCH model, we wouldn't get such a comforting result—a significant Q suggests that the variance model isn't adequate, and the simple ARCH isn't.

The first test, on the standardized residuals themselves, gives us Figure 4.3 which shows a problem with serial correlation, that is, that (4.6) doesn't appear to be true. Note that we can't really test the non-standardized residuals as we did with ARMA models because those calculations will be strongly influenced by a relatively small number of data points where the variance is high. After standardizing by the GARCH estimate of the standard deviation (Figure 4.4), we get a first lag autocorrelation which is very significant and a Q with a very significant p -value.

This suggests that we didn't allow enough lags in the mean model. If you go back and change the model to

```
linreg tb1yr
# constant tb1yr{1 2} tb3mo{1 2}
```

and re-run, you'll find that the standardized residuals are *much* closer to white noise.

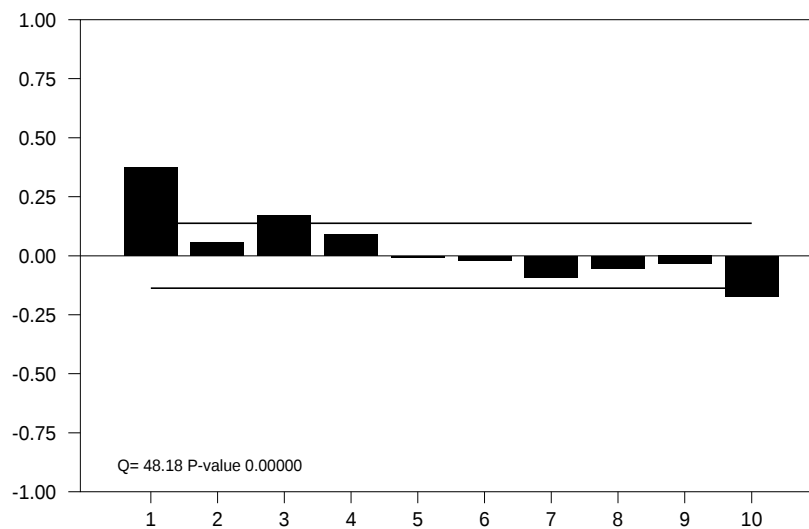


Figure 4.3: Standardized Residuals from GARCH

The GARCH estimates of the standard deviations (from the model with just one lag) can be created with

```
set hstddev = sqrt(h)
graph(footer="Standard Deviations from GARCH Model")
# hstddev
```

It usually works better to graph standard deviations rather than variances because of scale—the larger variances are so much higher than the low ones that you get very little detail except for the zone where the variance is high.

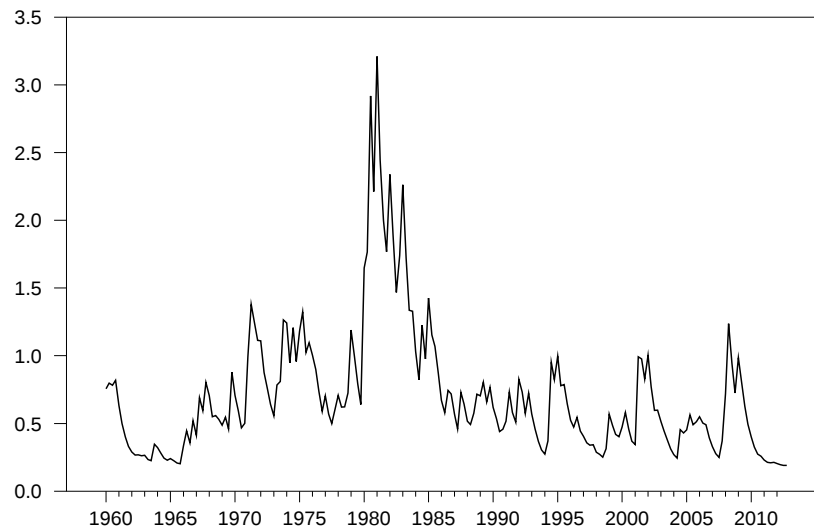


Figure 4.4: Standard Deviations from GARCH Model

4.4 Tips and Tricks

4.4.1 The Simplex Algorithm

The simplex algorithm can be applied to an optimization problem assuming nothing more than continuity of the objective function. In order to operate with so little restriction on the behavior of the function, the simplex algorithm is more a collection of “rules” that seem to work in practice than a formal algorithm.

Instead of trying to “climb” the hill directly, it instead crawls upward primarily by determining which directions *aren’t* correct. In a K -dimensional space, it uses a set of $K + 1$ vertices, thus, for instance, the vertices of a triangle in 2-space. At a pass through the algorithm, a replacement is sought for the *worst* of the vertices. An obvious guess (if we’re still trying to move up) is that the function will be better if we go through the face opposite the worst point. The test is whether that new point is better than the one we’re trying to replace, *not* that it’s better than all the other points. If we have an improvement over the worst point, that old one is removed from the simplex collection and the new one added. If the new point is worse, then it seems likely that the optimum may already be surrounded, so a test point is chosen in the interior. This process of replacing the worst of the vertices continues until the entire simplex has shrunk down so the difference between the best and worst vertices satisfies the convergence criterion.

As an example, suppose that we are trying to maximize $f(x, y) = -(x + 4y^2)$. The optimum is $(0, 0)$ by inspection, but suppose that we only have a “black box” which returns the value given an (x, y) combination. If our guess value is $(1, 2)$, we need to construct a triangle with that as one of the vertices. Suppose

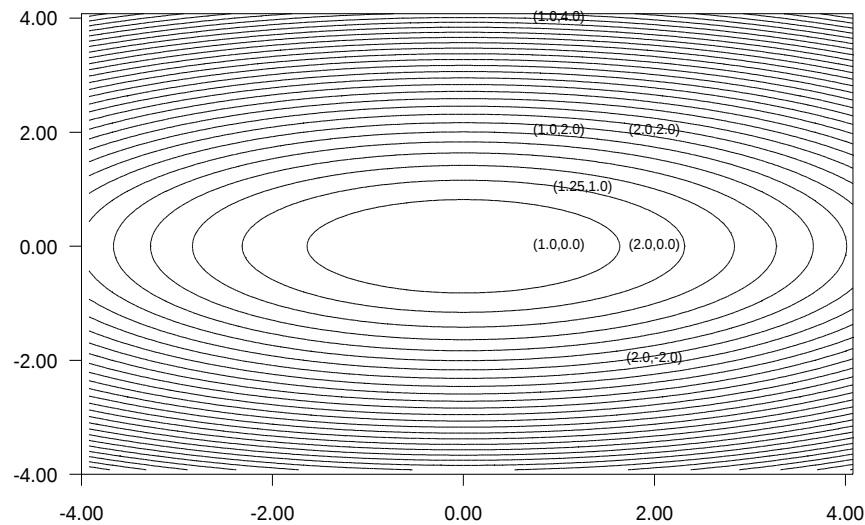


Figure 4.5: Simplex Algorithm in Action

that our three initial points are $(1, 2)$, $(2, 2)$ and $(1, 4)$.⁴ The function values at the three points are -17, -20 and -65 respectively, so the vertex to be replaced is $(1, 4)$. The test point is the reflection of $(1, 4)$ through the midpoint of the side formed by the two other points, which is $(2, 0)$.⁵ The value there is -4, so we accept it and eliminate $(1, 4)$. Now the worst of the three is $(2, 2)$. The new test point is $(1 + 2 - 2, 2 + 0 - 2)$ or $(1, 0)$ where the function value is -1, again, better than the point being replaced. Our three vertices are now $(1, 0)$, $(2, 0)$ and $(1, 2)$. The test replacement for $(1, 2)$ is $(2, -2)$ where the function is -20, thus worse than at $(1, 2)$. So instead, we try the interior point halfway between the vertex being replaced and the center of the side opposite: $(1.25, 1.00)$. The function there is -5.5625, thus better than the -17 at $(1, 2)$, though not better than the best one so far. Figure 4.5 shows the contours of the function, and the points evaluated. Note that it's replaced all three of the original vertices.

Unlike more direct “climbing” methods, this has to drag all $K + 1$ vertices up the hill, rather than a single point, so it's less efficient for well-behaved problems. However, even with functions that are twice-continuously differentiable, the simplex algorithm can be useful as a preliminary optimization method, as it can work even if the likelihood surface has the “wrong” curvature at the guess values (that is, it's not concave). In effect, the preliminary simplex iterations help to refine the guess values. To get preliminary simplex iterations, use the combination of the options `PMETHOD=SIMPLEX` and `PITERS=number of preliminary “iterations”`. What counts as an iteration for this is $2K$ simplex moves, which roughly equalizes the number of function evaluations in an actual iteration with other methods.

⁴RATS will actually start with a much tighter cluster.

⁵In two dimensions, the coordinates are found by summing the two kept points and subtracting the coordinates of the one being rejected, thus the test has $x = 1 + 2 - 1$ and $y = 2 + 2 - 4$.

We often see users overdo the number of preliminary iterations. Usually 5 to 10 is enough, and rarely will you need more than 20. `PMETHOD=SIMPLEX, PITER=200` does quite a bit of calculation and won't really get you much farther along than the same with `PITER=20`.

4.4.2 BFGS and Hill-Climbing Methods

The BFGS algorithm⁶ is the workhorse optimization algorithm for general maximum likelihood, not just in RATS but in many other statistical packages, as it works quite well in a broad range of applications. BFGS requires that the function being optimized be twice-continuously differentiable which will be the case for most log likelihoods that you will encounter. The function will have a second order Taylor expansion around θ_0 :

$$f(\theta) \approx f(\theta_0) + f'(\theta_0) \bullet (\theta - \theta_0) + \frac{1}{2}(\theta - \theta_0)' f''(\theta_0)(\theta - \theta_0)$$

If f were quadratic, then $f''(\theta_0)$ would be a constant (negative definite) matrix Q , and the optimum could be found directly by solving

$$\theta = \theta_0 - Q^{-1} f'(\theta_0)$$

no matter what start values we have for θ_0 . If we start near the optimum, then the same calculation should be at least close to finding the top even if the function is only locally quadratic. There are two main problems in practice:

1. For a non-quadratic function, what happens if we're not near the optimum?
2. It may not be easy to compute $f''(\theta_0)$.

The gradient can usually be computed fairly accurately by numerical methods. Numerical second derivatives are much less accurate and require quite a bit of extra calculation—you can compute the gradient in K space with $K+1$ function evaluations (a base plus a slightly perturbed value in each direction), but the second derivative requires an extra $K(K+1)/2$ to fill up a $K \times K$ symmetrical array.

The key result for the BFGS algorithm is that you can get an increasingly accurate estimate of the Hessian (f'') without ever computing it directly by seeing how the gradient changes from one iteration to the next. The precise result is that if the function is actually quadratic and you do K iterations of BFGS with exact line searches at each stage, then at the end, you will have built Q exactly (and thus will have also found the optimum).

In practice, the function isn't globally quadratic, and we generally don't (for efficiency) do "exact" line searches, so the algorithm will not converge exactly in K iterations and the final f'' will only be approximate. However, with rare

⁶For Broyden, Fletcher, Goldfarb and Shanno, the creators of the algorithm.

exceptions, if the function is, indeed, twice continuously differentiable and has a negative definite Hessian (no singularities) at the local maximum on the starting “hill”, then BFGS will find its way to the top of that hill.

To illustrate how BFGS works, we’ll use the same example as simplex, maximizing $-(x + 4y^2)$ starting at $(1, 2)$ where the function value is -17. BFGS builds an approximation to $\mathbf{G} \equiv [-f''(\theta_0)]^{-1}$ which (ideally) will be a positive definite matrix. For illustration, we’ll start with $\mathbf{G} = \mathbf{I}$. The gradient at $(1, 2)$ is $\mathbf{g} = (-2, -16)$. The direction of search on the first iteration is $\mathbf{d} = \mathbf{G}\mathbf{g}$, so the line being searched is $\theta(\lambda) = \theta_0 + \lambda\mathbf{d}$. The directional derivative along this line is $\mathbf{d} \bullet \mathbf{g} = \mathbf{g}'\mathbf{G}\mathbf{g}$, which must be positive since $\mathbf{G}$ is positive definite, so there must be some positive value of λ (possibly very small) at which $f(\theta(\lambda)) > f(\theta_0)$. We need to find such a value. Because $\mathbf{G}$ isn’t giving us a good estimate of the curvature (yet), a “full” step ($\lambda = 1$) doesn’t work well: that would take us to $(1, -14)$ where the function value is -785. The RATS hill-climbing procedure then tries $\lambda = .5$ (still doesn’t help), then $\lambda = .25$, which puts us at $(.5, -2.0)$ where the function value is -16.25. This is better than -17, so it would appear that we are done. However, the directional derivative is 260. That means that with $\lambda = .25$, we would expect to see the function increase by *much* more than simply -17 to -16.25—the small arc-derivative to the new point fairly strongly indicates that $\lambda = .25$ is still too long a step, so it’s on the other side of the maximum in the chosen direction. So the next test is with $\lambda = .125$ or $(.75, 0.0)$ where the function value is -.5625. Now, we’re improving by 16.4375 over a distance of .125; the arc-derivative is 131.25 which is a (very) good ratio to 260.⁷ So our first iteration takes us to $(.75, 0.0)$. This used 4 subiterations—the function evaluations along the chosen line. It’s very common for early iterations of BFGS to require this many, or sometimes more, subiterations since the early $\mathbf{G}$ isn’t very precise. With well-behaved problems, the number of subiterations usually drops fairly quickly to 1 or perhaps 2 on each iteration.

BFGS then updates the estimate of $\mathbf{G}$ using the actual and predicted values for the gradient at the new position. In this case, the result is

$$\mathbf{G} = \begin{bmatrix} .5 & 0 \\ 0 & .125 \end{bmatrix}$$

which is exactly correct. Ordinarily, the first update wouldn’t hit exactly, but because the Hessian is diagonal and we started with a diagonal matrix, it works here. Given that we have the correct $\mathbf{G}$, the next iteration finds the optimum.⁸

You can see why preliminary simplex iterations are often handy—even with a truly quadratic function, the first iteration on the hill-climbing algorithm

⁷A ratio of .5 is what we would get if we found the *exact* maximum for λ given that this is a true quadratic.

⁸Actually, almost. Because the gradient is computed numerically, none of the calculations done above are exact.

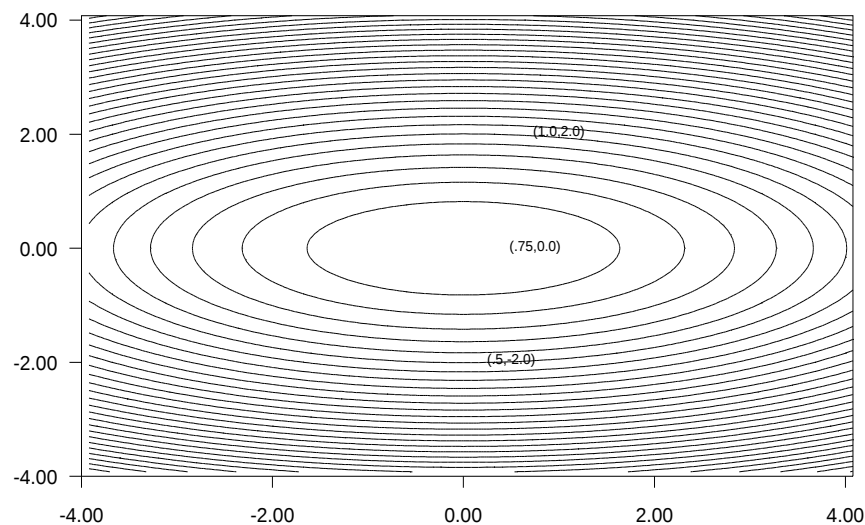


Figure 4.6: BFGS in Action

tries some *very* wrong parameter vectors. In Figure 4.6, we can't even show the first two subiterations on the graph—the first one would be three pages down. In this case, the wildly incorrect vectors won't be a problem, but with some functions which aren't globally defined (because of the presence of logs or square roots as part of the calculations, or because of explosive behavior for an iterative calculation), it may be necessary to check for conditions that would invalidate the value. A function evaluation which requires log or square root of a negative number will naturally result in a missing value the way that RATS does evaluations—it's the explosive recursions, which might occur in GARCH or ARMA models, that may require special care.

4.4.3 The CDF instruction and Standard Distribution Functions

We used the **CDF** instruction earlier to compute and display the significance level of the test for ARCH. It supports the four most commonly used test distributions: the Normal, t , F and χ^2 . The syntax is:

```
CDF(option) distribution statistic degree1 degree2
```

where

distribution Choose the desired distribution: FTEST for F , TTEST for t , CHISQ for χ^2 or NORMAL for (standard) 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 (main) option is `TITLE=descriptive title`. One of the examples was:

```
cdf(title="LR Test for delta=1") chisqr 2*(%logl-loglunr) 1
```

which produces

<pre>LR Test for delta=1 Chi-Squared(1)= 0.576804 with Significance Level 0.44756761</pre>

CDF sets the variables `%CDSTAT` as the test statistic and `%SIGNIF` as the significance level.

CDF is very handy if output formatted as above is fine. If you need something else (for instance, to insert the information onto a graph or into a report), you can use a built-in function to compute the same significance level:

Normal	<code>%ZTEST(z)</code> returns the two-tailed significance level of z as a Normal(0,1).
t	<code>%TTEST(t, nu)</code> returns the two-tailed significance level of t as a t_ν .
F	<code>%FTEST(F, num, den)</code> returns the significance level of F as an F with num numerator degrees of freedom and den denominator degrees of freedom.
χ^2	<code>%CHISQR(x, nu)</code> returns the significance level of x as a χ^2 with nu degrees of freedom.

An alternative to using **CDF** would be something like:

```
compute deltatest=2*(%logl-loglunr)
compute deltasignif=%chisqr(deltatest,1)
disp "Test delta=1" *.### deltatest "with signif" *.### deltasignif
```

Example 4.1 Likelihood maximization

```

cal(q) 1960:1
all 2012:4
open data quarterly(2012).xls
data(org=obs,format=xls)
*
set pi = 100.0*log(ppi/ppi{1})
set y = .001*rgdp
*
* Power function with interest rates
*
* Non-linear least squares (maximum likelihood assuming Normal residuals)
*
nonlin a0 a1 a2 delta
linreg tblyr
# constant tblyr{1} tb3mo{1}
frml ratef tblyr = a0+a1*tblyr{1}+a2*(tb3mo{1})^delta
compute a0=%beta(1),a1=%beta(2),a2=%beta(3),delta=1.0
nlls(frml=ratef) tblyr 2 *
*
* Maximum likelihood assuming t residuals. This requires adding the
* variance and the degrees of freedom to the parameter set.
*
nonlin(parmset=baseparms) a0 a1 a2 delta
nonlin(parmset=tparms) sigsq nu
*
* This starts with sigsq as the estimate from NLLS with a relatively
* high value of nu.
*
compute sigsq=%seesq,nu=20.0
frml logl = %logtdensity(sigsq,tblyr-ratef,nu)
maximize(parmset=baseparms+tparms,itors=300) logl 2 *
*
* Test delta=1
*
test(title="Test for linearity")
# 4
# 1.0
*
* This is available in RATS 8.2 or later
*
summarize(parmset=baseparms+tparms,title="Test for linearity") delta-1
*
* Add a constraint that delta is 1.
*
nonlin(parmset=pegs) delta=1.0
*
* Save the unrestricted log likelihood
*
compute loglunr=%logl
*
maximize(parmset=baseparms+tparms+pegs) logl 2 *
cdf(title="LR Test for delta=1") chisqr 2*(%logl-loglunr) 1

```

Example 4.2 ARCH Model, Estimated with MAXIMIZE

```

cal(q) 1960:1
all 2012:4
open data quarterly(2012).xls
data(org=obs,format=xls)
*
linreg tb1yr
# constant tb1yr{1} tb3mo{1}
*
* Graph standardized residuals
*
set stdu = %resids/sqrt(%seesq)
graph(footer="Standardized Residuals from Linear Regression")
# stdu
*
* Test for ARCH using auxiliary regression
*
set u = %resids
set usq = u^2
linreg usq
# constant usq{1 to 4}
cdf(title="Test for ARCH") chisqr %trsquared 4
*
* Test for ARCH using @ARCHTEST
*
@archtest(lags=4,form=lm) u
*
* Define the parameters
*
nonlin(parmset=meanparms) b0 b1 b2
nonlin(parmset=archparms) a0 a1
*
* Define log likelihood FRML in three parts:
*
frml efrml = tb1yr-(b0+b1*tb1yr{1}+b2*tb3mo{1})
frml hfrml = a0+a1*efrml{1}^2
frml logl = %logdensity(hfrml,efrml)
*
* Guess values based upon linear regression
*
linreg tb1yr
# constant tb1yr{1} tb3mo{1}
compute b0=%beta(1),b1=%beta(2),b2=%beta(3)
compute a0=%seesq/(1-.5),a1=.5
*
* Estimate the ARCH model
*
maximize logl 3 *
compute aicarch=-2.0*%logl+2.0*%nreg
*
* Estimate the linear regression over the same range
*
linreg tb1yr 3 *

```

```
# constant tb1yr{1} tb3mo{1}
compute aicols=-2.0*%logl+2.0*(%nreg+1)
*
* Compare AIC's with proper counting of parameters
*
disp "AIC-ARCH" @15 *.### aicarch
disp "AIC-OLS" @15 *.### aicols
*
* Same model done with GARCH
*
garch(reg,q=1) / tb1yr
# constant tb1yr{1} tb3mo{1}
```

Example 4.3 GARCH Model with Flexible Mean Model

```
cal(q) 1960:1
all 2012:4
open data quarterly(2012).xls
data(org=obs,format=xls)
*
linreg tb1yr
# constant tb1yr{1} tb3mo{1}
equation(lastreg) meaneq
frml(lastreg,vector=beta) meanfrml
compute rstart=%regstart(),rend=%regend()
*
set h = %seesq
set u = %resids
set uu = %seesq
*
* Define the parameters
*
nonlin(parmset=meanparms) beta
nonlin(parmset=garchparms) c a b
*
* Define log likelihood FRML in three parts:
*
frml efrml = tb1yr-meanfrml
frml hfrml = c+a*uu{1}+b*h{1}
frml logl = h=hfrml,u=efrml,uu=u^2,%logdensity(h,u)
*
* The BETA's are already done as part of FRML(LASTREG)
*
compute a=.1,b=.6,c=%seesq/(1-a-b)
*
* Estimate the GARCH model
*
maximize(parmset=meanparms+garchparms,$
  pmethod=simplex,piters=10) logl rstart+1 rend
compute aicgarch=-2.0*%logl+2.0*%nreg
*
linreg(equation=meaneq) * %regstart() %regend()
```

```
compute aicols=-2.0*%logl+2.0*(%nreg+1)
*
* Compare AIC's with proper counting of parameters
*
disp "AIC-GARCH" @15 *.*## aicgarch
disp "AIC-OLS" @15 *.*## aicols
*
* Test for serial correlation in the standardized residuals
*
set stdu = u/sqrt(h)
set stdusq = stdu^2
*
@regcorrs(number=10,dfc=1,nocrits,qstat,$
  title="Standardized Residuals") stdu
@regcorrs(number=10,dfc=2,nocrits,qstat,$
  title="Standardized Squared Residuals") stdusq
*
set hstddev = sqrt(h)
graph(footer="Standard Deviations from GARCH Model")
# hstddev
```

Standard Programming Structures

We’ve already seen some (relatively) simple examples of using the programming features of RATS using the **DO** and **DOFOR** loops. In this chapter, we’ll look in greater detail at the program control structures in RATS, emphasizing the ones that tend to be common, in some form, to most programming languages. These are the **DO** loop, **IF** and **ELSE** blocks, and **WHILE** and **UNTIL** loops. We’ll cover **DOFOR**, which is very useful but not as standard, in the next chapter.

5.1 Interpreters and Compilers

Except for the examples with loops, most of what we’ve seen has been RATS as an *interpreted* language, which means that it executes each instruction immediately after it is processed. This is often very handy as it allows you to experiment with different ways of handling a model and you get immediate feedback.

However, let’s take a line out of a **DO** loop in an earlier program:

```
compute lreffect(t)=%sumlc
```

This really doesn’t do much—it takes the real value `%SUMLC` and puts it into an entry of the series `LREFFECT`. What does the interpreter have to do before this happens? The main steps are that it takes the first three characters on the line (“com”) converts them to upper case, looks that up in a table of instructions and determines that it’s a **COMPUTE** instruction. It then has to isolate the “token” `LREFFECT`, look that up in a symbol table, recognize it’s a **SERIES**, look up `T`, recognize that it’s an **INTEGER** variable, look up `%SUMLC`, recognize that it’s a **REAL**, and determine that it can put all those together into a sensible instruction. At that point, it can actually do the assignment of the value of `%SUMLC` to `LREFFECT(T)`. If you got the impression that it takes a lot more time to turn the character string “compute lreffect(t)=%sumlc” into something usable than it does to actually do what it requests, you’re right. Now neither takes very long in an absolute sense—the total time required might be 50 microseconds. However, if you had to do that millions of times as part of a calculation, it could matter.

A pure interpreted language (which RATS is *not*) has to go through something like that process each time an instruction is executed. That has certain advantages as you can quite literally alter instructions right up to the time that

they are executed. There's a cost in time, however, so it's most useful when the instructions, when executed, tend to do a lot of calculation. For instance, if instead of a simple assignment above, we were doing a **LINREG**, the amount of time the interpreter requires might go up by a factor of three, while the amount of work done by the instruction would likely go up by many thousands, so interpretation wouldn't be as significant a part of the calculation time.

Instructions in complex RATS programs are often a mix of simple (such as **COMPUTE** on scalars) and complicated (**LINREG**, **MAXIMIZE**) instructions. For efficiency, when you do a loop or some other programming structure RATS uses a *compiler*, which does the interpretation once and saves the generated code so it can be executed with relatively little time added to what is needed to execute the requested calculations. This requires advanced planning on your part as RATS doesn't actually do anything (other than parse the instructions) until you're done with *all* the instructions for the loop. As a general rule, if you are doing any type of loop, you are best off entering the instructions "off-line". If you type the following into your input window while it's in "ready" or "on-line" mode:

```
do i=1,10
  dsp i
```

(the last line is a typo, it should have been "disp i"), you'll get an error message that it expected an instruction. At this point, the attempt to compile the loop is aborted; you can't just correct the spelling and continue on.

Instead, the better approach is to make the window "local" or "off-line" before you even start putting in the loop code. Click the  icon or type <Control>+L, then type (or paste) in the following (*with* the mistake on the second line):

```
do i=1,10
  dsp i
end do i
```

Now click on the  or type <Control>+L to put the window back into "ready" mode, select the three lines and hit the <Enter> key or click the  icon. You'll get the "Expected Instruction" error.¹ Now, however, you can just fix the second line to read `disp i`, select the three lines again, hit <Enter> and you get the expected

¹If you select the three lines and hit <Enter> without switching back to "ready" mode, you'll delete the three lines since in "local" mode the <Enter> is just a standard editing keystroke. If you do that by accident, just Undo and make sure you switch to "ready".

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

Now let's look at

```
disp "Before loop"  
do i=1,10  
  disp i  
end do i  
disp "After loop"
```

As we've written this, the first and last **DISPLAY** instructions are done in interpreted mode, while the middle three lines are done in compiler mode. Interpreted mode is the “natural” state for RATS, so it needs one of a few special instructions to put it into compiler mode. **DO** is one of those. Once it's in compiler mode, RATS needs another signal that it should leave compiler mode and execute the code that it has just generated. How that is done will be different depending upon what instruction put it into compiler mode. In the case of **DO**, it's a matching **END**. (The **DO I** after **END** are actually treated as comment, so that's for information only.) If you have nested **DO** loops such as

```
do i=1,4  
  do j=1,3  
    disp i j  
  end do j  
end do i
```

the outer **DO** puts RATS into compiler mode. RATS then keeps track of the “level” of the compiler structures, so the second **DO** raises that to level two. The matching **END** for the **DO J** drops the level to one so it's still in compile mode. The level doesn't drop to zero until after the matching **END** for the outer **DO I** at which point RATS exits compiler mode and executes the double loop. This may seem obvious from the indenting, but the indenting is only to make it easier to read for humans—RATS ignores the lead blanks when processing the instructions. If you have a long or complicated program segment that either is switching to interpreter mode before you expect or not leaving compiler model when you think it should, you probably have some problem with the structure levels. It's much easier to find those types of errors if you try to keep the instructions properly indented (see page 6) to show the levels.

5.2 DO Loops

As illustrated in Section 2.8.1, the **DO** loop is a very simple way to automate many of your repetitive programming tasks. It's by far the most common program control structure. The most common **DO** loop will look like:

```
DO i=1,n
    instructions to execute
end do i
```

There are slight differences in how **DO** loops function in different languages, so we'll go through this step by step and point out where you need to be careful if you are trying to translate a program from a different language.

1. The variable i is given the start value (here 1)
2. RATS determines how many passes through the loop are required to run the index from the start value (here 1) to the end value (here n). If the end value is less than the start value, the number of passes is zero, and control passes immediately to the instruction after the **END DO i** . In some languages (though not many), the loop instructions are always executed at least once.
3. The instructions are executed with the current value of i .
4. If the number of passes computed in step 2 has been reached, the loop is exited and control passes to the instruction after the **END DO i** . This is where different languages can differ quite a bit as we'll point out in more detail.
5. i is incremented by 1 and the pass count is incremented by 1.
6. Repeat steps 3, 4 and 5 until the pass count is reached in step 4.

Some programming languages test for the loop exit differently—at the end of the loop, they first increment i , then test it against N . If $i > n$, the loop is exited. As a result, once the loop is done, i will be equal to $n + 1$ (assuming n was bigger than 0 in the first place). The way RATS handles this, at loop exit i will be equal to the value on the final trip through the loop. This is a subtle difference and we'll see that it can matter.

What happens if you change the value of i inside the loop? While this is legal in RATS (in some programming languages it isn't), it isn't a good idea, as the results will probably not be what you expect. If you execute:

```
do i=1,10
    compute i=i+3
    disp i
end do i
```

the output is

```

4
8
12
16
20
24
28
32
36
40

```

and on loop exit, `I` will be equal to 40. As it says in steps 2 and 4, the **DO** loop operates by determining how many passes are required right up front, and then does that number, regardless of what happens to the value of the index. If you need a loop where the increment can change from pass to pass, or the end value might change, use a **WHILE** or **UNTIL** loop instead (section 5.4).

More generally, the **DO** loop has the form:

```

DO integer_variable=n1,n2,increment
  instructions to execute
end do integer_variable

```

The variables `I` and `J` are pre-defined in RATS as **INTEGER** variables and are by far the most common loop index variables. You can introduce a new variable name for the “integer_variable” and it will be defined as an **INTEGER** type.

```

do k=p,1,-1
  ...
end do k

```

will loop p times through the controlled instructions (as long as p is 1 or larger) with `K` taking the value p to start and being decreased by 1 each pass through the loop.

In Example 5.1, we’ll use the **DO** loop to analyze the possibility that log GDP (Figure 5.1) has a “broken trend”.

To the eye, it has an approximately linear trend, but with substantial deviations. If we regress on just constant and trend:

```

set trend = t
*
linreg loggdp
# constant trend

```

we get a Durbin-Watson of .035. The residuals are very strongly serially correlated and any attempt to model the trend will have to take that into account. A “broken” trend could take several forms, but what we’ll look at here is a “joined” trend, where the linear trend rate has one value through some point in time, and another after, but the level of the process doesn’t change at the join point. That can be handled by a linear function which looks like

$$\alpha + \beta t + \gamma \max(t - t_0, 0) \quad (5.1)$$

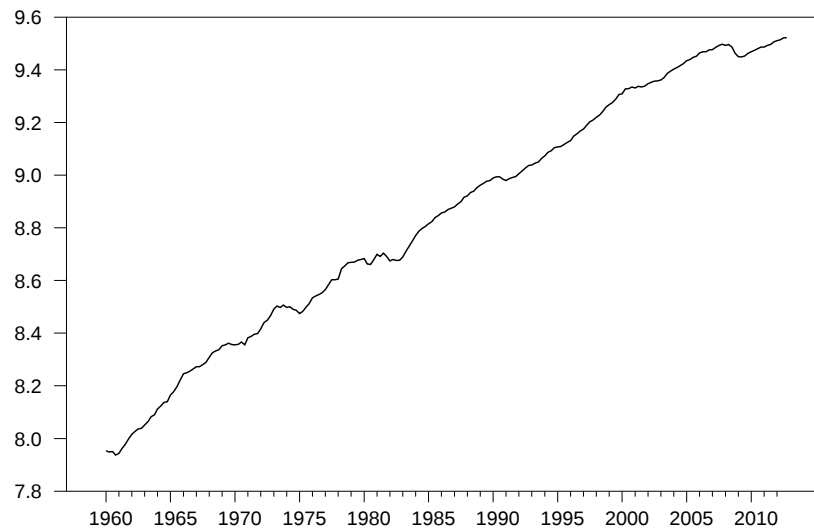


Figure 5.1: Log U.S. Real GDP

This will grow at the rate β up to time t_0 , then at $\beta + \gamma$ after t_0 .

There are two basic ways to model this while allowing for serial correlation. The simpler way to do this is add lags of the dependent variable to (5.1):

$$y_t = \alpha + \beta t + \gamma \max(t - t_0, 0) + \varphi_1 y_{t-1} + \dots + \varphi_p y_{t-p} + u_t \quad (5.2)$$

This makes the break what is known as an *innovational outlier*. In this setup, β and γ are *not* structural parameters describing the trend rate of the process—the trend rate of y prior to t_0 can be solved out as

$$\frac{\beta}{(1 - \varphi_1 - \dots - \varphi_p)}$$

If you look at what happens at $t_0 + 1$, none of the lagged y terms have yet been affected by the trend change, so the first period after the break, the process goes up by an extra γ compared to the process without the break. At $t_0 + 2$, the y_{t-1} has increased by an extra γ , and the γ term itself will now be $\gamma \times 2$, so the overall effect from including the break term is $\gamma \times 2 + \gamma \times \varphi_1$ and so on. Notice how the break works itself into the system gradually as the lag terms reach the break location.

If we allow for two autoregressive lags, we can compute the sums of squares for different break locations using the following loop:

```
set rssio = %na
do t0=1965:1,2007:4
    set btrend = %max(t-t0,0)
    linreg(noprint) loggdp
    # constant trend btrend loggdp{1 2}
    compute rssio(t0)=%rss
end do t0
```

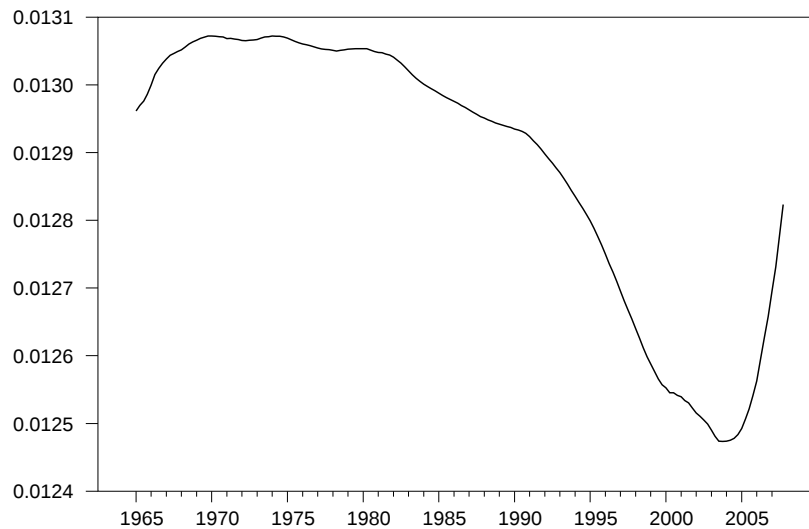


Figure 5.2: RSS for Broken Trend, Innovational Outlier

The **LINREG** is just a straightforward translation of the formula (5.2) given the value of T_0 . All we had to do was “throw a loop around” this. Why is this running only from 1965:1 to 2007:4? With any analysis where you’re looking for some type of break in a model, you want to exclude breaks near the ends. Because we are searching for breaks in the trend, it makes sense to require enough data points in each branch to properly determine a trend rather than just a cycle. Here we make sure there are at least five years of data both before and after the change date.

The series **RSSIO** is initialized to %NA since the series doesn’t exist outside the range of the **DO** loop. Inside the loop, we do the regression and save the sum of squared residuals into the T_0 entry in **RSSIO**.

We can graph (Figure 5.2) the sum of squares with

```
graph(footer="RSS for Broken Trend, Innovational Outlier")
# rssio
```

We can find where the minimum was attained using

```
ext(noprint) rssio
disp "Minimum at" %datelabel(%minent) %minimum
```

Minimum at 2003:04	0.01247
--------------------	---------

$$\begin{aligned} y_t &= \alpha + \beta t + \gamma \max(t - t_0, 0) + z_t \\ z_t &= \varphi_1 z_{t-1} + \dots + \varphi_p z_{t-p} + u_t \end{aligned} \quad (5.3)$$

The more complicated type of model has

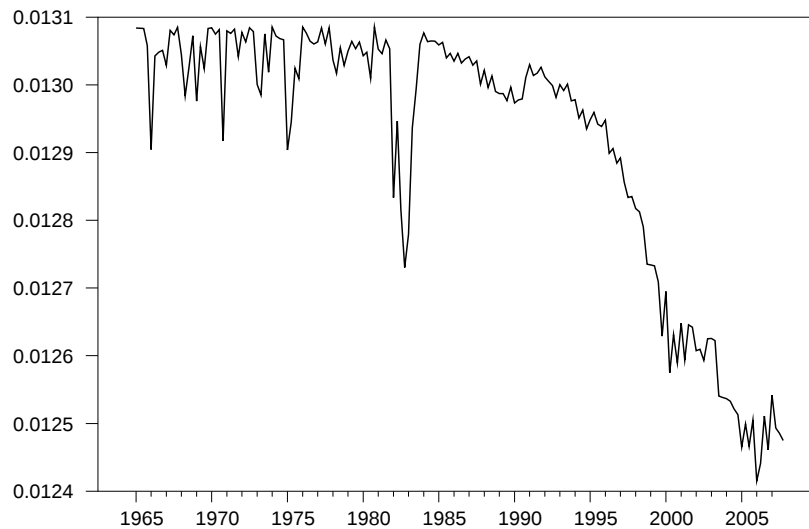


Figure 5.3: RSS for Broken Trend, Additive Outlier

that is, y is described as a broken trend plus an AR(p) noise term. If it were not for the broken trend terms, (5.2) and (5.3) would be equivalent models (with the coefficients on the deterministics mapping to each other) if you work through the expansions. However, with the broken trend, they aren't. (5.3) has what is known as an *additive outlier*. Here α , β and γ are structural parameters, with $\beta + \gamma$ being the trend rate of the y process starting *immediately* at $t_0 + 1$.

You can't estimate (5.3) using **LINREG**—a mean + AR or ARMA noise is done using **BOXJENK** with the **REGRESSORS** option. The loop is almost identical other than the substitution of the main instruction:

```
set rssao = %na
do t0=1965:1,2007:4
    set btrend = %max(t-t0,0)
    boxjenk(regressors,ar=2,noprint) loggdp
    # constant trend btrend
    compute rssao(t0)=%rss
end do t0
graph(footer="RSS for Broken Trend, Additive Outlier")
# rssao
*
ext(noprint) rssao
disp "Minimum at" %datelabel(%minent) %minimum
```

Perhaps not too surprisingly, the sum of squares (Figure 5.3) is quite a bit more volatile than it is for innovational model since changes to the trend rate hit immediately.

It's important to note that the F statistic for either model in comparison with a non-breaking trend model has a non-standard distribution if you search for

the best break point. It's beyond the scope of this book to deal with the theory behind that.

5.3 IF and ELSE Blocks

There are many instances in which we want to perform a set of instructions only if a particular condition is met. The most common way to do this is to use an **IF** or **IF-ELSE** block. We already saw a very simple example of this on page 49 where we checked for whether a **BOXJENK** estimation converged and displayed a message when it failed.

The basic structure of an **IF** block is:

```
IF condition {
    block of statements executed if condition is "true"
}
```

while an **IF-ELSE** block is:

```
IF condition {
    block of statements executed if condition is "true"
}
ELSE {
    block of statements executed if condition is "false"
}
```

What form does the *condition* take? It can be any expression that is non-zero when you want "true" and zero when you want "false". This is usually built using the following standard relational operators for comparing expressions A and B. (Each has two equivalent representations).

A==B or A.EQ.B	Equality
A<>B or A.NE.B	Inequality
A>B or A.GT.B	Greater than
A<B or A.LT.B	Less than
A>=B or A.GE.B	Greater than or equal to
A<=B or A.LE.B	Less than or equal to

Note well that the test for equality is done with == (two =), not just a single =. A=B assigns the value of B to the variable or array element A.

You can create compound conditions using "and", "or" and "not" with

```
condition 1.AND.condition 2
condition 1.OR.condition 2
.NOT.condition
```

It's important to note that some programming languages have constructions like this that are used in transforming data. That is not done in RATS—use **SET** with **%IF** or a relational operator instead. For instance, this is how *not* to create a dummy variable in RATS that's 1 when real GDP is above potential:

```
* This is not RATS code
if rgdp>potent
    set boom = 1
else
    set boom = 0
```

Instead, you would use simply

```
set boom = rgdp>potent
```

SET has an implied loop over the entries—the **IF-ELSE** does not. Something like the **IF-ELSE** code could work by inserting it inside a loop. Here it would be a bad idea since seven lines can be replaced with one, but there are other situations where it would be superior to using a **SET** if the two branch calculations were sufficiently complicated. In this case, the code would be something like:

```
set boom = 0.0
do t=1,%allocend()
    if rgdp(t)>potent(t)
        compute boom(t)=1
    else
        compute boom(t)=0
end do t
```

You may have noticed that we didn't use { and } around the instructions controlled by the **IF** and the **ELSE** in this last example. By default, **IF** and **ELSE** control just one line. If you need to execute more than one line, you need to enclose the controlled lines in braces. It never *hurts* to add the braces, but they aren't necessary in the simplest case.

As we've seen before on page 49, you can have an **IF** without an **ELSE**. **IF-ELSE** can be used if there are two alternative calculations. If you have *three* or more (mutually exclusive) cases, you can string together a set of **IF**'s and **ELSE**'s. For illustration:

```

do rep=1,100
  compute r=%ran(1.0)
  if r<-2.0
    disp "Big Negative value" r
  else
    if r<2.0
      disp "Between -2 and 2" r
    else
      disp "Big Positive value" r
  end do rep

```

Each pass through the loop, this draws a $N(0,1)$ random number. If $r < -2$, the first condition is met, and the message about the “big negative value” is displayed. If $r \geq -2$, since the first **IF** condition fails, we do the first **ELSE**. That immediately goes into a second **IF**. If $r < 2$, the second **IF** condition is met, and the “between -2 and 2” message gets displayed. We know that $r \geq -2$ as well since we’re in the **ELSE** from the first **IF**. Finally, if $r \geq 2$, we get down to the final **ELSE** clause and display the “big positive value” message.

Note that you can do the same type of multiple branch calculation within a **SET** instruction using nested **%IF** functions. For instance

```

set u = %ran(1.0)
set range = %if(u<-2,-1,%if(u<2,0,1))

```

will create **RANGE** as a series which has $range_t = -1$ if $u_t < -2$, $range_t = 0$ if $u_t \geq -2$ and $u_t < 2$ and $range_t = +1$ if $u_t \geq 2$.

As a more concrete example, we’ll look at lag length selection again in Example 5.2. We’ll find the best AIC autoregression on the change in log real RGDP. As we mentioned on Chapter 2 (page 23), when you use an information criterion, it’s important to run the regressions over the same range or you’ll bias the results in one direction or the other. Before, we used the range parameters on **LINREG** to enforce that. We’ll show an alternative which both makes sure that the range is the same, and also is more efficient computationally. The **CMOMENT** instruction generates a cross product matrix of a set of data. After that, the **LINREG** instruction with a **CMOMENT** option will run a regression using the cross product information (and range) from the **CMOMENT** instruction.

The **CMOMENT** instruction needs to include both the explanatory variables and the dependent variable(s) from all the regressions that will be run using it. In this case, that means lags from 0 (for the dependent variable) to 12 of the **DLRGDP** plus the **CONSTANT**.

```

set dlr GDP = log(rgdp)-log(rgdp{1})
*
cmom
# dlr GDP{0 to 12} constant

```

The following does the regressions and picks out the minimum AIC lag length:

```
do lags=0,12
  if lags==0 {
    linreg(noprint,cmom) dlrgdp
    # constant
    compute aic = -2.0*%logl + %nreg*2
    compute bestlag=lags,bestaic=aic
  }
  else {
    linreg(noprint,cmom) dlrgdp
    # constant dlrgdp{1 to lags}
    compute aic = -2.0*%logl + %nreg*2
    if (aic < bestaic)
      compute bestlag=lags,bestaic=aic
  }
end do lags
```

The instruction block controlled by the **IF** runs the regression on the **CONSTANT** only and saves the AIC into the **BESTAIC** variable. These lines are executed only when **LAGS** is equal to 0. Since that's the first pass through the loop, we know that the model is the best that we have seen to that point. If **LAGS** is non-zero, we execute the instruction block controlled by the **ELSE**. This estimates the model with the current number of **LAGS**, computes the AIC and compares it to whatever is now in **BESTAIC**. If the new AIC is smaller, we replace both **BESTLAG** with the current number of lags and **BESTAIC** with the current value for AIC. Note that we used a second (rather simple) **IF** inside the instruction block controlled by the main **ELSE**—structures can be nested as deeply as you need, though once you get above five levels it can be very hard to keep track of which is controlling what. More advanced structures called **PROCEDURES** and **FUNCTIONS** are often handy for removing parts of a very involved calculation into a separate subblock of code to make the program flow easier to follow and the whole program easier to maintain.

5.4 WHILE and UNTIL Loops

The **DO** loop is appropriate if you know exactly how many passes 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 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. This is known as a *general-to-specific* model selection process. You can do this with the help of a **WHILE** or **UNTIL** instruction.

The syntax for a **WHILE** block is:

```
WHILE condition {
    block of statements executed as long as condition is “true”
}
```

The syntax for an **UNTIL** block is:

```
UNTIL condition {
    block of statements executed until condition is “true”
}
```

As part of Example 5.3, we’ll first use **WHILE** to do the lag selection as described above, picking a lag length no larger than 12 for the growth rate in the deflator (which we’ll call `DLDEFLATOR`). A possible way to write this is:

```
set dldeflator = log(deflator)-log(deflator{1})
*
compute lags=13,signif=1.00
while signif>.05 {
    compute lags=lags-1
    linreg(noprint) dldeflator
    # constant dldeflator{1 to lags}
        compute signif=%ttest(%tstats(%nreg),%ndf)
        disp "Significance of lag" lags "=" signif
    }
end while
```

which will give us

Significance of lag 12 =	0.14249
Significance of lag 11 =	0.73725
Significance of lag 10 =	0.48749
Significance of lag 9 =	0.78494
Significance of lag 8 =	0.43323
Significance of lag 7 =	0.28264
Significance of lag 6 =	0.98133
Significance of lag 5 =	0.04629

The first time through the loop, the variable `SIGNIF` is compared to 0.05. Since `SIGNIF` was initialized to be larger than 0.05, all of the the instructions within the block are executed. So `LAGS` is decreased from 13 to 12 and a 12 lag AR is estimated on `DLDEFLATOR`. `%TSTATS(%NREG)` is the *t*-statistic on the final coefficient in the regression; we compute its two-tailed significance level with `%TTEST` using as the degrees of freedom for the *t* the variable `%NDF` that’s set by the `LINREG`. Note that both `%NREG` and `%NDF` will change (automatically) with the number of lags in the regression—you don’t have to figure them out yourself. For illustration, this now displays the number of lags and the significance level. In a working program, you probably wouldn’t do that, but it’s a good idea to put something like that in until you’re sure you have the loop correct.

We’re now at the end of the block so control loops up to the **WHILE** check at the top. With `LAGS=12` on the first pass, the significance is .14249 so the **WHILE**

condition is still true. Thus we start a second pass through, decreasing lags to 11 and redoing the calculation. This repeats until `LAGS` is 5. This gives `SIGNIF=.04629` so when we loop to the top, the **WHILE** condition finally fails. Control passes to the first instruction after the controlled block of instructions, which means we're done with the whole compiled subprogram.²

You may have noticed a problem with the **WHILE** loop—what happens if *none* of the final coefficients is ever significant? That's certainly possible if the series is white noise. Most loops of this nature need a "safeguard" against running forever in case the condition isn't met. Here, the loop won't run forever, but it *will* get to the point where the regression uses lags from "1 to 0". RATS actually interprets that the way it would be intended here, which is to use no lags at all. However, the *t*-test would then be on the `CONSTANT` (since it would be the last and only coefficient), rather than a lag.

One possibility would be to change the condition to

```
while signif>.05.and.lags>1
```

which will prevent it from running the regression with no lags. However, it won't give us the right answer for the number of lags, because we can't tell (based upon this condition alone) whether `LAGS` is 1 when we drop out of the loop because lag 1 was significant, or whether it was because lags 1 *wasn't* significant, and we triggered the second clause. If it's the latter, we want the report to be `LAGS=0`.

In most such cases, the secondary condition to break out of the loop is best done with a separate **BREAK** instruction, controlled by an **IF**. **BREAK** does exactly what it sounds like it would do—breaks out of the current (most inner) loop. Here, we insert the test right after `LAGS` is reduced. You can check that this *does* get the result correct. If we get to lag 1 and it's significant, we break the loop based upon the **WHILE** condition while `LAGS` is still 1. If we get to lag 1 and it's not significant, we break the loop when `LAGS` is reduced to zero, which is what we want.

```
compute lags=13,signif=1.00
while signif>.05 {
  compute lags=lags-1
  if lags==0
    break
  linreg(noprint) dldeflator
  # constant dldeflator{1 to lags}
    compute signif=%ttest(%tstats(%nreg),%ndf)
    disp "Significance of lag" lags "=" signif
}
end while
```

²The **END WHILE** is the signal that you want to exit compile mode. It's only needed if the **WHILE** instruction isn't already inside some other compiled structure.

Not surprisingly, there is more than one way to do this. **BREAK** also can be applied to **DO** and **DOFOR** loops. Instead of using **WHILE**, we could use a **DO** loop which counts backwards through the lags and break out of it when we get a significant coefficient. A first go at this would be something like:

```
do lags=12,1,-1
    linreg(noprint) dldeflator
    # constant dldeflator{1 to lags}
    compute signif=%ttest(%tstats(%nreg),%ndf)
    disp "Significance of lag" lags "=" signif
    if signif<.05
        break
end do lags
```

This will give exactly the same results as before. The only problem is again with the case where none of the lags is significant. It's not that the loop runs forever, since it will quit after the pass where **LAGS** is 1 as we planned. It's just that, because of the way that the **DO** loop runs (section 5.2), the value of **LAGS** will be 1 if the only significant lag is 1, and will also be 1 if none of the lags are significant—on the normal exit from the loop, the value of the index is the value from the last pass through.

An alternative which gets *all* cases correct is:

```
compute p=0
do lags=12,1,-1
    linreg(noprint) dldeflator
    # constant dldeflator{1 to lags}
    compute signif=%ttest(%tstats(%nreg),%ndf)
    disp "Significance of lag" lags "=" signif
    if signif<.05 {
        compute p=lags
        break
    }
end do lags
disp "Number of lags chosen =" p
```

Instead of using the loop index **LAGS** to represent the chosen number of lags, this uses the separate variable **P**. This is originally set to 0 and is only reset if and only if we hit a significant lag.

WHILE and **UNTIL** are similar, but there are two differences, one minor and one more important:

1. The condition for **WHILE** is “true” if the loop is to continue, while for **UNTIL** it is “true” if the loop is to terminate.
2. The body of the **UNTIL** loop is always executed at least once, as the test is done at the end of a pass; **WHILE** tests at the top and so could drop out without ever executing the body.

The same analysis done using an **UNTIL** loop is:

```
compute lags=13,signif=1.00
until signif<.05 {
    compute lags=lags-1
    if lags==0
        break
    linreg(noprint) dldeflator
    # constant dldeflator{1 to lags}
    compute signif=%ttest(%tstats(%nreg),%ndf)
    disp "Significance of lag" lags "=" signif
}
end until
```

Which of these is best to use? All of them get the job done correctly and are roughly the same number of lines so it's largely a matter of taste. Automatic lag selection is extremely common in modern econometrics, particularly in unit root and cointegration testing, so this shows up in quite a few RATS procedures. For several reasons, we generally end up using the **DO** loop, as the coding is a bit clearer, plus, the added step of saving the chosen lags into a separate variable isn't (in practice) really an added step, since that will almost always be done anyway so the selected number of lags can be used later.

After the **UNTIL** or **WHILE** examples, we can estimate the chosen regression with:

```
compute p=lags
linreg(title="Least Squares with Automatic Lag Selection") $
    dldeflator
# constant dldeflator{1 to p}
```

We would do the same after the **DO**, but without the `compute p=lags`.

Linear Regression - Estimation by Least Squares with Automatic Lag Selection				
Dependent Variable DLDEFLATOR				
Quarterly Data From 1961:03 To 2012:04				
Usable Observations		206		
Degrees of Freedom		200		
Centered R ²		0.7980032		
R-Bar ²		0.7929533		
Uncentered R ²		0.9376799		
Mean of Dependent Variable		0.0088412547		
Std Error of Dependent Variable		0.0059200109		
Standard Error of Estimate		0.0026937464		
Sum of Squared Residuals		0.0014512539		
Regression F(5,200)		158.0229		
Significance Level of F		0.0000000		
Log Likelihood		929.6086		
Durbin-Watson Statistic		1.9964		
Variable	Coeff	Std Error	T-Stat	Signif

1. Constant	0.000685461	0.000353780	1.93754	0.05408801
2. DLDEFLATOR{1}	0.576388906	0.070224827	8.20777	0.00000000
3. DLDEFLATOR{2}	0.166608793	0.080443169	2.07114	0.03962957
4. DLDEFLATOR{3}	0.112942556	0.080816573	1.39752	0.16380637
5. DLDEFLATOR{4}	0.210004031	0.080784979	2.59954	0.01003079
6. DLDEFLATOR{5}	-0.142165754	0.070898975	-2.00519	0.04629106

Note that, while the final lag in this regression is significant at .05, lag 3 isn't. It's *possible* to do a more involved lag pruning to get rid of any other apparently insignificant lags using, for instance, stepwise regression with **STWISE**. However, that's almost never done in practice—you use an automatic procedure to select just the length, not the full set of lags.

5.5 Estimating a Threshold Autoregression

To provide another example of the topics in this chapter, we will estimate a threshold autoregression. The threshold autoregressive (TAR) model has become popular as it allows for different degrees of autoregressive decay. Consider a two-regime version of the threshold TAR developed by Tong (1983):

$$y_t = I_t \left[\alpha_0 + \sum_{i=1}^p \alpha_i y_{t-i} \right] + (1 - I_t) \left[\beta_0 + \sum_{i=1}^p \beta_i y_{t-i} \right] + \varepsilon_t \quad (5.4)$$

where

$$I_t = \begin{cases} 1 & \text{if } y_{t-1} \geq \tau \\ 0 & \text{if } y_{t-1} < \tau \end{cases} \quad (5.5)$$

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.

How is this different from the STAR models in Section 3.5? The TAR is the limit as $\gamma \rightarrow \infty$ in the LSTAR model. It seems like there might not be much point to the TAR when it's a special case of the STAR, but it's a special case that (as

we saw) isn't well-handled by non-linear least squares because the objective function isn't differentiable (at the limit) and, in fact, isn't even continuous.

The nice thing about the STAR is that, *if* it is a good explanation of the data with a finite value of γ , it can be estimated successfully by **NLLS**; however, if it requires an infinite value of γ , or if there is no threshold effect, the estimation fails completely. On the other hand, the sum of squares (or log likelihood) of the TAR model is easily computed given τ (just two standard least squares regressions over the two branches), but is discontinuous in τ itself. The only way to estimate τ is with a grid search over the observed values of y_{t-1} .

Example 5.4 illustrates the estimation of a TAR model for the growth rate of the money supply. The first part of the program reads in the data set and constructs the variable `gm2` using:

```
set gm2 = log(m2) - log(m2{1})
```

The next line in the program estimates the `gm2` series as an $\text{AR}(\{1,3\})$ process.

```
linreg gm2
# constant gm2{1 3}
```

If you experiment a bit, you will see that the $\text{AR}(\{1,3\})$ specification is quite reasonable. If you are going to estimate a TAR model, it is standard to start with a parsimonious linear specification. First, suppose that we want the value of the threshold τ to equal the sample mean (0.016788). This might be the case if we were certain that greater than average money growth behaved differently from below average growth. Also, suppose you knew the delay factor used to set the heaviside indicator was 2.³

We can create the indicator I_t (called `PLUS`) using

```
stats gm2
compute tau=%mean
set plus = gm2{2}>=tau
```

We cannot use the symbol `I` (since `I`, along with `J` and `T` are reserved integer variables) to represent the indicator, so we use the label `PLUS`. For each possible entry in the data set, the **SET** instruction compares gm_{t-2} to the value in `TAU`. If gm_{t-2} is greater than `TAU`, the value of $plus_t$ is equal to 1, otherwise it's zero.

Next, we create $(1 - I_t)$ as the series `MINUS` using:

```
set minus = 1 - plus
```

There are two ways to estimate the model: you can do two separate estimations with **LINREG** using the `SMPL=PLUS` option for one and `SMPL=MINUS` for

³We actually experimented to find the best delay.

the other, adding the two sums of squared residuals to get the full model sum of squares, or you can generate “dummied-out” versions of the regressors for the two periods and do a single **LINREG**. We’ll first show a “brute force” implementation of the second of the two by creating the variables $I_t gm2_{t-1}$, $I_t gm2_{t-3}$, $(1 - I_t) gm2_{t-1}$ and $(1 - I_t) gm2_{t-3}$:

```
set y1_plus  = plus*gm2{1}
set y3_plus  = plus*gm2{3}
set y1_minus = minus*gm2{1}
set y3_minus = minus*gm2{3}
```

Now we can estimate the regression using:

```
linreg gm2
# plus y1_plus y3_plus minus y1_minus y3_minus
```

Linear Regression - Estimation by Least Squares				
Dependent Variable GM2				
Quarterly Data From 1961:01 To 2012:04				
Usable Observations	208			
Degrees of Freedom	202			
Centered R ²	0.4665048			
R-Bar ²	0.4532994			
Uncentered R ²	0.8914579			
Mean of Dependent Variable	0.0168372348			
Std Error of Dependent Variable	0.0085299382			
Standard Error of Estimate	0.0063069683			
Sum of Squared Residuals	0.0080351254			
Regression F(5,202)	35.3270			
Significance Level of F	0.0000000			
Log Likelihood	761.6537			
Durbin-Watson Statistic	1.9522			
Variable	Coeff	Std Error	T-Stat	Signif

1. PLUS	0.0049868654	0.0023541720	2.11831	0.03537110
2. Y1_PLUS	0.5062254697	0.0799787393	6.32950	0.00000000
3. Y3_PLUS	0.1542573375	0.0790118150	1.95233	0.05228060
4. MINUS	0.0017519286	0.0014967659	1.17048	0.24318846
5. Y1_MINUS	0.8237612657	0.0958011189	8.59866	0.00000000
6. Y3_MINUS	0.2164646725	0.1009068421	2.14519	0.03313113

At this point, you might want perform the standard diagnostic checks and perhaps eliminate **MINUS** coefficient since its t -statistic is quite low. However, our goal here is to illustrate programming techniques, not to obtain the best fitting TAR model for money growth.

5.5.1 Estimating the Threshold

One problem with the above model is that the threshold may not be known. 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:

$$y^1 < y^2 < y^3 \dots < y^T \quad (5.6)$$

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 to ensure an adequate number of observations on each side of the threshold.

Note that this is quite a different form of grid search than we saw in Chapter 3 (page 83). Because the objective function there was continuous, each different grid value likely would produce a different value of the objective—we can only hope that the grid isn't too coarse to miss the minimum. Here, however, the objective function is discontinuous and we know *exactly* at which points it can change. Thus, the grid search that we're conducting here, over the observed values of the threshold, is guaranteed to find the minimum. It is, however, a bit harder to set up. The following two lines copy the threshold series into a new series called `TAUS` and sorts it (in increasing order, which is the default for the `ORDER` instruction).

```
set taus = gm2{2}
order taus
```

We now need to figure out which entries of `TAUS` we can use, given that we want to eliminate 15% at each end. The quickest and most flexible way to do that is to use the `INQUIRE` instruction to figure out what the defined range of `TAUS` is. `INQUIRE` is described in greater detail in this chapter's *Tips and Tricks* section (page 169).

```
inquire(series=taus) tstart tend
compute tlow=tstart+fix(%nobs*.15),thigh=tend-fix(%nobs*.15)
```

`TSTART` will be the first defined entry of `TAUS` (here 4 because `GM2` starts at 2 and the threshold has a delay of 2), so `TLOW` will be 15% of the way into the data set from the lowest value and `THIGH` 15% of the way in from the highest. Note that this is not 15% of the gap in the values between the highest and lowest, but 15% of the entry count. If there are many values at (for instance) the low end, we could be starting at a value not much above the minimum, but that's OK since we are excluding these largely so that we don't run regressions with almost no data. The `FIX` function is needed because the entry numbers are integer-valued and `%NOBS*.15` is real—`FIX(x)` rounds x down to the first integer below it.

The search can be done with:

```

compute rssbest=%na
do itau=tlow,thigh
  compute tau=taus(itau)
  set plus  = gm2{2}>=tau
  set minus = 1 - plus
  *
  set y1_plus  = plus*gm2{1}
  set y3_plus  = plus*gm2{3}
  set y1_minus = minus*gm2{1}
  set y3_minus = minus*gm2{3}
  linreg(noprint) gm2
  # plus y1_plus y3_plus minus y1_minus y3_minus
  if .not.%valid(rssbest).or.%rss<rssbest
    compute rssbest=%rss,taubest=tau
end do itau

```

Once the program exits the loop, we can display the consistent estimate of the threshold with

```

disp "We have found the attractor"
disp "Threshold=" taubest

```

<pre> We have found the attractor Threshold= 0.01660 </pre>
--

Finally, we can estimate the TAR model with the consistent estimate of the threshold using

```

compute tau=taubest
set plus  = gm2{2}>=tau
set minus = 1 - plus
*
set y1_plus  = plus*gm2{1}
set y3_plus  = plus*gm2{3}
set y1_minus = minus*gm2{1}
set y3_minus = minus*gm2{3}
linreg(title="Threshold autoregression") gm2
# plus y1_plus y3_plus minus y1_minus y3_minus

```

Linear Regression - Estimation by Threshold autoregression				
Dependent Variable GM2				
Quarterly Data From 1961:01 To 2012:04				
Usable Observations		208		
Degrees of Freedom		202		
Centered R ²		0.4687938		
R-Bar ²		0.4556451		
Uncentered R ²		0.8919236		
Mean of Dependent Variable		0.0168372348		
Std Error of Dependent Variable		0.0085299382		
Standard Error of Estimate		0.0062934232		
Sum of Squared Residuals		0.0080006495		
Regression F(5,202)		35.6533		
Significance Level of F		0.0000000		
Log Likelihood		762.1009		
Durbin-Watson Statistic		1.9391		
Variable	Coeff	Std Error	T-Stat	Signif

1. PLUS	0.0058913925	0.0023302839	2.52819	0.01223032
2. Y1_PLUS	0.4745687514	0.0791112032	5.99876	0.00000001
3. Y3_PLUS	0.1502105166	0.0787585904	1.90723	0.05790997
4. MINUS	0.0015219242	0.0015023522	1.01303	0.31225885
5. Y1_MINUS	0.8625034171	0.0969960051	8.89215	0.00000000
6. Y3_MINUS	0.1885968485	0.1012876103	1.86199	0.06405639

5.5.2 Improving the Program

The program described in Section 5.5.1 is rather crude. It works, but far too much of it is hard-coded for a specific example. For instance, it uses the variable GM2 almost 20 times and the threshold delay of 2 is repeated four times. We can also enhance the program by creating a series of the sums of squared residuals for different values of τ , so we can see how sensitive the objective is to the threshold value. The revised program is Example 5.5.

One question you might have is whether we should have planned ahead for this when we originally wrote the program. How you handle it will generally depend upon how comfortable you are with the more flexible coding that we will be doing. One problem with trying to start with the “improved” program is that the most complicated part of this isn’t making the specification more flexible—it’s getting the coding for finding the optimal threshold correct. If you try to do two things at once:

1. work out and debug the optimal threshold code
2. write a program easily adapted to other data

you might have a hard time getting either one correct. Again, that will depend upon your skill level with RATS programming. However, the graph of the sums of squares is definitely something that good programming practice would tell you to wait on—that’s easy to add once everything else is done.

The first thing we will do differently is to add a `DEFINE` option to the initial `LINREG`:

```
linreg(define=baseeq) gm2
# constant gm2{1 3}
compute rssols=%rss
```

This defines `BASEEQ` as an `EQUATION` data type, which keeps track of (among other things) the form of the equation and the dependent variable. The last line above also saves the sum of squared residuals from the least squares estimation.

To allow for greater flexibility in setting the threshold variable and delay, we can do the following:

```
set threshvar = gm2
compute d=2
```

From this point on, if we use `THRESHVAR{D}` whenever we need the threshold expression, then we can change the threshold by changing just these two lines.

As we described earlier, there are two ways to estimate the threshold regression. The method from the previous section was to create dummied-out regressors and do a combined `LINREG`. The alternative is to run two `LINREG`'s over the “plus” and “minus” samples. While we’ll show later how to create the dummies more flexibly, it’s much easier to do the two sample regression.

We’ll skip over the estimation with τ at the mean and jump straight into the code for finding the optimal threshold. You’ll see two differences with the set up code:

```
clear taus
set taus = threshvar{d}
order taus
inquire(series=taus) tstart tend
*
compute tlow=tstart+fix(%nobs*.15),thigh=tend-fix(%nobs*.15)
```

First, we added a `CLEAR` instruction for the `TAUS` series. That will allow us to change the threshold variable or delay without having to worry whether `TAUS` still has left-over values from the previous analysis.⁴ Second, the `SET TAUS` now uses `THRESHVAR{D}` rather than hard-coded values from the example.

We also need to add one more instruction to initialize a series for the sums of squares as they are generated:

```
set rsstau = %na
```

⁴You could also avoid any problems like this by doing *File-Clear Memory* menu item or by clicking on the  toolbar button before re-running the program with any changes, but the `CLEAR` instruction will work whether or not you do that.

Note that **CLEAR RSSTAU** would also work fine. This sets up the series **RSSTAU** and sets all values to **NA**—the only data points which will have non-missing values will be the ones where we estimate a threshold regression. The simplified loop for finding the attractor is:

```
compute rssbest=rssols
do itau=tlow,thigh
  compute tau=taus(itau)
  set plus = threshvar{d}>=tau
  linreg(noprint,equation=baseeq,smpl=plus)
  compute rssplus=%rss
  linreg(noprint,equation=baseeq,smpl=.not.plus)
  compute %rss=%rss+rssplus
  compute rsstau(itau)=%rss
  if %rss<rssbest
    compute rssbest=%rss,taubest=tau
end do itau
```

What's different here? First, the test for whether a new sum of squares is the best that we've seen is simplified a bit by starting with **RSSBEST** equal to **RSSOLS**. Since *all* models with a break have to be at least as good as the same model with no breaks, we know that this will be replaced right away. In the previous coding, we started with **RSSBEST=%NA**, which then required testing **RSSBEST** for **%VALID**. Since we have available a value which we're computing anyway that we know is finite but bigger than the optimal value, we might as well use it.

Second, we're using **THRESHVAR{D}** rather than the specific **GM2{2}**. Third, the sums of squares for the threshold regression is computing using:

```
linreg(noprint,equation=baseeq,smpl=plus)
compute rssplus=%rss
linreg(noprint,equation=baseeq,smpl=.not.plus)
compute %rss=%rss+rssplus
```

The first **LINREG** runs the regression over the sample where **PLUS** is non-zero (in this case, non-zero always means “one”), and the second runs it over the remainder of the sample (**.NOT.PLUS** is “true” wherever **PLUS** is non-zero). **%RSS** will be equal to the sum of the **%RSS** values from the two regressions.

Finally

```
compute rsstau(itau)=%rss
```

saves the value of **%RSS** into the entry of **RSSTAU** that corresponds to the current value of **TAU** being examined. Note that since **TAUS** is a sorted copy, these don't represent the original time period of the data, but we're doing a **SCATTER** plot, so all that matters is that **RSSTAU** matches up with **TAUS**.

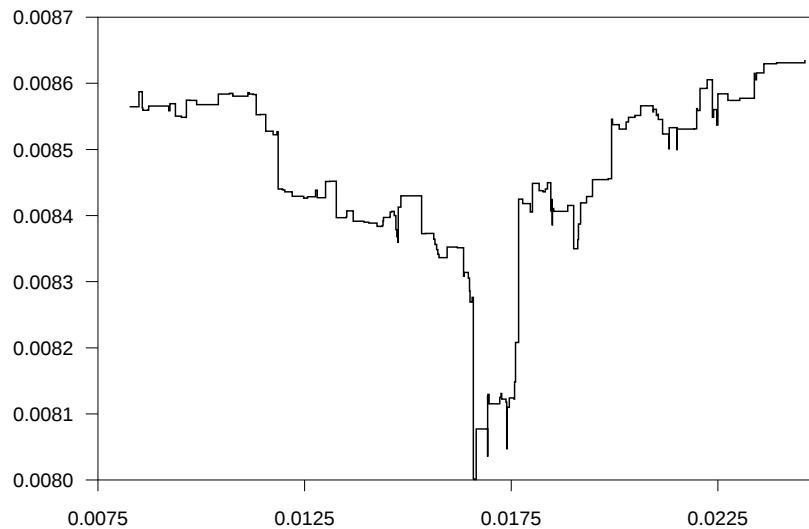


Figure 5.4: Threshold Values vs Sums of Squares

Not surprisingly, this produces the same result as the cruder coding. We add the graph (Figure 5.4) using:

```
scatter(footer="Threshold Values vs Sums of Squares",style=step)
# taus rsstau
```

Note that this uses `STYLE=STEP` rather than `STYLE=LINE`. `STYLE=STEP` gives a graph of function as it *should* look, which is a step function between the observed values for the threshold.

The process for generating the final regression with the dummied-out break variables uses some more advanced programming features which will be covered in the next chapter. This will give you a taste of some of the special capabilities that RATS has, particular for dealing with time series.

As before, we need to create a `PLUS` series which is the dummy for the “above” branch:

```
compute tau=taubest
set plus = threshvar{d}>=tau
```

The number of “dummied-out” series that we need is $2 \times$ the number of regressors in the base model. That is most conveniently done by creating a `RECTANGULAR` matrix of `SERIES` with the dimensions we need. You’ll notice that this next code segment uses a specialized set of functions for pulling information out of the saved equation. See this Chapter’s *Tips and Tricks* page 170 for a more complete description of those.

```

dec rect[series] expand(%eqnsize(baseeq),2)
do i=1,%eqnsize(baseeq)
    set expand(i,1) = %eqnxvector(baseeq,t)(i)*plus
    set expand(i,2) = %eqnxvector(baseeq,t)(i)*(1-plus)
    labels expand(i,1) expand(i,2)
    # "PLUS_" + %eqnreglabels(baseeq)(i) $
    "MINUS_" + %eqnreglabels(baseeq)(i)
end do i

```

EXPAND(I,1) is the “plus” branch and EXPAND(i,2) is the “minus” branch for each of the regressors. The **LABELS** instruction is then used to give more informative output labels to those two series. The + operator, when applied to strings, does concatenation, so this will create labels which are PLUS_CONSTANT and MINUS_CONSTANT when the regressor’s standard label is CONSTANT, PLUS_GM{1} and MINUS_GM{1} when the regressor’s standard label is GM{1}, etc.

We run the final regression with the optimal threshold using:

```

linreg(title="Threshold Regression") %eqndepvar(baseeq)
# expand

```

which gives us

Linear Regression - Estimation by Threshold Regression				
Dependent Variable GM2				
Quarterly Data From 1961:01 To 2012:04				
Usable Observations		208		
Degrees of Freedom		202		
Centered R ²		0.4687938		
R-Bar ²		0.4556451		
Uncentered R ²		0.8919236		
Mean of Dependent Variable		0.0168372348		
Std Error of Dependent Variable		0.0085299382		
Standard Error of Estimate		0.0062934232		
Sum of Squared Residuals		0.0080006495		
Regression F(5,202)		35.6533		
Significance Level of F		0.0000000		
Log Likelihood		762.1009		
Durbin-Watson Statistic		1.9391		
Variable	Coeff	Std Error	T-Stat	Signif

1. PLUS_Constant	0.0058913925	0.0023302839	2.52819	0.01223032
2. PLUS_GM2{1}	0.4745687514	0.0791112032	5.99876	0.00000001
3. PLUS_GM2{3}	0.1502105166	0.0787585904	1.90723	0.05790997
4. MINUS_Constant	0.0015219242	0.0015023522	1.01303	0.31225885
5. MINUS_GM2{1}	0.8625034171	0.0969960051	8.89215	0.00000000
6. MINUS_GM2{3}	0.1885968485	0.1012876103	1.86199	0.06405639

You can experiment with different sets of lags in the AR and different delay values and see how this is able to adapt to them.

As we said in the Preface, you should try not to “reinvent the wheel”. We’ve shown a program to estimate a threshold autoregression, but there already exist several procedures which may be able to do what you need. The

@THRESHTEST procedure both estimates a general threshold regression (not just an autoregression) and can compute bootstrapped significance levels. **@TAR** estimates a threshold autoregression including a test for the best threshold delay. For specific applications, there are **@EndersGranger** and **@EndersSiklos** which do threshold unit root and cointegration tests respectively.

5.6 Tips and Tricks

The Instruction **INQUIRE**

If you look at the code for many of the popular RATS procedures (such as **DFUNIT.SRC**), you'll see that one of the first executable instructions is an **INQUIRE**. If you do an instruction like **LINREG** or **STATISTICS**, RATS will automatically determine the maximum range given the series involved. However, for instance, you need to run a **DO** loop over a range of entries, you need to find out in advance the specific range that's available. That's what **INQUIRE** is designed to do.

```
INQUIRE(options) value1<<p1 value2<<p2
# list of variables in regression format (only with
REGLIST)
```

The **<<p1** and **<<p2** are only used in procedures, so we'll discuss them later. Thus, we're looking at the basic instruction being

```
INQUIRE(options) value1 value2
# list of variables in regression format (only with
REGLIST)
```

In the example in this chapter, we used:

```
inquire(series=taus) tstart tend
```

which makes **TSTART** equal to the first entry of **TAUS** which isn't an NA, and **TEND** equal to the *last* valid entry.

If you need the limit of a set of series, use the **REGRESSORLIST** (which we usually shorten to **REGLIST**) and list the variables, which can include lag/lead fields on a supplementary line. For instance, to determine the largest estimation range for the model used in Example 5.4, we would do

```
inquire(reglist) rstart rend
# gm2 constant gm2{1 3}
```

Note that you need to include the dependent variable as well—if you don't, **REND** will actually be one entry past the end of the data since entry $T + 1$ is valid for the lags.

There's also an `EQUATION` option which can be used to determine the maximum range permitted by the variables (both dependent and explanatory) in an `EQUATION`. For instance,

```
inquire(equation=baseeq) estart eend
```

If it's important to identify missing values within the data range, you can add the `VALID` option to any of those. For instance,

```
inquire(valid=esmpl,equation=baseeq) estart eend
```

would define `ESTART` and `EEND` as the outer common limits of the variables in `BASEEQ` with `ESMPL` created as a dummy variable with 1's in the entries which are valid across all those variables and 0's in the entries which aren't. In this example, since there are no missing values inside a series, `ESMPL` would just be all 1's between `ESTART` and `END`.

EQUATION functions

We defined an `EQUATION` early in Example 5.5 to save the base specification that we extended with breaks. There is a whole set of functions which can be used to take information out of (or, less often, put information into) an `EQUATION`. All of these have names starting with `%EQN`. Here, we used the rather simple `%EQNSIZE(eqn)` which returns the size (number of explanatory variables) of the equation. The two more important functions used in the example are `%EQNXVECTOR` and `%EQNREGLABELS`.

`%EQNXVECTOR(eqn,t)` returns the `VECTOR` of explanatory variables for equation `eqn` at entry `t`. An `eqn` of 0 can be used to mean the last regression run. The instructions

```
set expand(i,1) = %eqnxvector(baseeq,t) (i)*plus  
set expand(i,2) = %eqnxvector(baseeq,t) (i)*(1-plus)
```

are inside a loop over the time subscript `T`. The `%EQXVECTOR(baseeq,t)` pulls out the vector of explanatory variables for `BASEEQ` at `T`, which (in this case) means $[1, gm2_{t-1}, gm2_{t-3}]$. The further `(I)` subscript takes one of those three elements out.

`%EQNREGLABELS(eqn)` returns a `VECTOR` of `STRINGS` which are the "regressor labels" used in standard regression output, combining the variable name and (if used) lag number, such as `GM2{1}` and `GM2{3}` for the lags of `GM2`. Again, we use subscript `I` applied to the result of that to pull out the string that we need.

There are several related functions which can also be handy. In all cases, `eqn` is either an equation name, or 0 for the last estimated (linear) regression. These also evaluate at a specific entry `T`.

- `%EQNPRJ (eqn, t)` evaluates the fitted value $X_t\beta$ for the current set of coefficients for the equation.
- `%EQNVALUE (eqn, t, beta)` evaluates $X_t\beta$ for an input set of coefficients.
- `%EQNRESID (eqn, t)` evaluates the residual $y_t - X_t\beta$ for the current set of coefficients, where y_t is the dependent variable of the equation.
- `%EQNRVALUE (eqn, t)` evaluates the residual $y_t - X_t\beta$ for an input set of coefficients.

Example 5.1 Illustration of DO loop

```

open data quarterly(2012).xls
cal(q) 1960:1
allocate 2012:4
data(org=obs, format=xls)
*
set loggdp = log(rgdp)
*
graph(footer="U.S. Real GDP")
# loggdp
*
set trend = t
*
linreg loggdp
# constant trend
*
set rssio = %na
do t0=1965:1,2007:4
    set btrend = %max(t-t0,0)
    linreg(noprint) loggdp
    # constant trend btrend loggdp{1 2}
    compute rssio(t0)=%rss
end do t0
*
graph(footer="RSS for Broken Trend, Innovational Outlier")
# rssio
ext(noprint) rssio
disp "Minimum at" %datelabel(%minent) %minimum
*
set rssao = %na
do t0=1965:1,2007:4
    set btrend = %max(t-t0,0)
    boxjenk(regressors,ar=2,noprint) loggdp
    # constant trend btrend
    compute rssao(t0)=%rss
end do t0
graph(footer="RSS for Broken Trend, Additive Outlier")
# rssao
*
ext(noprint) rssao
disp "Minimum at" %datelabel(%minent) %minimum

```

Example 5.2 Illustration of IF/ELSE

```

open data quarterly(2012).xls
cal(q) 1960:1
allocate 2012:4
data(org=obs, format=xls)
*
set dlrgdp = log(rgdp)-log(rgdp{1})
*
cmom

```

```

# dlrgdp{0 to 12} constant
*
do lags=0,12
  if lags==0 {
    linreg(noprint,cmom) dlrgdp
    # constant
    compute aic = -2.0*%logl + %nreg*2
    compute bestlag=lags,bestaic=aic
  }
  else {
    linreg(noprint,cmom) dlrgdp
    # constant dlrgdp{1 to lags}
    compute aic = -2.0*%logl + %nreg*2
    if (aic < bestaic)
      compute bestlag=lags,bestaic=aic
  }
end do lags
*
disp "Minimum AIC lag" bestlag

```

Example 5.3 Illustration of WHILE and UNTIL

```

open data quarterly(2012).xls
cal(q) 1960:1
allocate 2012:4
data(org=obs,format=xls)
*
set dldeflator = log(deflator)-log(deflator{1})
*
* Cut lags until the last one is significant
*
compute lags=13,signif=1.00
while signif>.05 {
  compute lags=lags-1
  linreg(noprint) dldeflator
  # constant dldeflator{1 to lags}
  compute signif=%ttest(%tstats(%nreg),%ndf)
  disp "Significance of lag" lags "=" signif
}
end while
*
disp "Chosen number of lags" lags
*
* Same thing with safeguard for the number of lags
*
compute lags=13,signif=1.00
while signif>.05 {
  compute lags=lags-1
  if lags==0
    break
  linreg(noprint) dldeflator
  # constant dldeflator{1 to lags}
  compute signif=%ttest(%tstats(%nreg),%ndf)
}

```

```

    disp "Significance of lag" lags "=" signif
}
end while
*
* Same thing done using a DO loop
*
compute p=0
do lags=12,1,-1
    linreg(noprint) dldeflator
    # constant dldeflator{1 to lags}
    compute signif=%ttest(%tstats(%nreg),%ndf)
    disp "Significance of lag" lags "=" signif
    if signif<.05 {
        compute p=lags
        break
    }
end do lags
disp "Number of lags chosen =" p
*
* Same thing done using an UNTIL loop
*
compute lags=13,signif=1.00
until signif<.05 {
    compute lags=lags-1
    if lags==0
        break
    linreg(noprint) dldeflator
    # constant dldeflator{1 to lags}
    compute signif=%ttest(%tstats(%nreg),%ndf)
    disp "Significance of lag" lags "=" signif
}
end until
compute p=lags
*
* Redo regression with chosen number of lags
*
linreg(title="Least Squares with Automatic Lag Selection") dldeflator
# constant dldeflator{1 to p}

```

Example 5.4 Threshold Autoregression, Brute Force

```

open data quarterly(2012).xls
cal(q) 1960:1
allocate 2012:4
data(org=obs,format=xls)
*
set gm2 = log(m2) - log(m2{1})
*
linreg gm2
# constant gm2{1 3}
*
stats gm2
compute tau=%mean

```

```

set plus = gm2{2}>=tau
set minus = 1 - plus
*
set y1_plus = plus*gm2{1}
set y3_plus = plus*gm2{3}
set y1_minus = minus*gm2{1}
set y3_minus = minus*gm2{3}
*
linreg gm2
# plus y1_plus y3_plus minus y1_minus y3_minus
*
* Create the empirical grid for the threshold values
*
set taus = gm2{2}
order taus
inquire(series=taus) tstart tend
*
* These are the lowest and highest entry numbers in <<taus>> that we
* will try, discarding 15% at either end.
*
compute tlow=tstart+fix(%nobs*.15),thigh=tend-fix(%nobs*.15)
*
compute rssbest=%na
do itau=tlow,thigh
    compute tau=taus(itau)
    set plus = gm2{2}>=tau
    set minus = 1 - plus
    *
    set y1_plus = plus*gm2{1}
    set y3_plus = plus*gm2{3}
    set y1_minus = minus*gm2{1}
    set y3_minus = minus*gm2{3}
    linreg(noprint) gm2
    # plus y1_plus y3_plus minus y1_minus y3_minus
    if .not.%valid(rssbest).or.%rss<rssbest
        compute rssbest=%rss,taubest=tau
end do itau
disp "We have found the attractor"
disp "Threshold=" taubest
*
* Re-estimate the model at the best values
*
compute tau=taubest
set plus = gm2{2}>=tau
set minus = 1 - plus
*
set y1_plus = plus*gm2{1}
set y3_plus = plus*gm2{3}
set y1_minus = minus*gm2{1}
set y3_minus = minus*gm2{3}
linreg(title="Threshold autoregression") gm2
# plus y1_plus y3_plus minus y1_minus y3_minus

```

Example 5.5 Threshold Autoregression, More Flexible Coding

```

open data quarterly(2012).xls
cal(q) 1960:1
allocate 2012:4
data(org=obs,format=xls)
*
set gm2 = log(m2) - log(m2{1})
*
linreg(define=baseeq) gm2
# constant gm2{1 3}
compute rssols=%rss
*
set threshvar = gm2
compute d=2
*
* Create the empirical grid for the threshold values
*
clear taus
set taus = threshvar{d}
order taus
inquire(series=taus) tstart tend
*
* These are the lowest and highest entry numbers in <<taus>> that we
* will try, discarding 15% at either end.
*
compute tlow=tstart+fix(%nobs*.15),thigh=tend-fix(%nobs*.15)
*
set rsstau = %na
*
compute rssbest=rssols
do itau=tlow,thigh
    compute tau=taus(itau)
    set plus = threshvar{d}>=tau
    linreg(noprint,equation=baseeq,smpl=plus)
    compute rssplus=%rss
    linreg(noprint,equation=baseeq,smpl=.not.plus)
    compute %rss=%rss+rssplus
    compute rsstau(itau)=%rss
    if %rss<rssbest
        compute rssbest=%rss,taubest=tau
end do itau
disp "We have found the attractor"
disp "Threshold=" taubest
*
scatter(footer="Threshold Values vs Sums of Squares",style=step)
# taus rsstau
*
* Re-estimate the model at the best values
*
compute tau=taubest
set plus = threshvar{d}>=tau
*
dec rect[series] expand(%eqnsize(baseeq),2)

```

```
do i=1,%eqnsize(baseeq)
    set expand(i,1) = %eqnxvector(baseeq,t)(i)*plus
    set expand(i,2) = %eqnxvector(baseeq,t)(i)*(1-plus)
    labels expand(i,1) expand(i,2)
    # "PLUS_" + %eqnreglabels(baseeq)(i) "MINUS_" + %eqnreglabels(baseeq)(i)
end do i
*
linreg(title="Threshold Regression") %eqndepvar(baseeq)
# expand
```

SERIES and Dates

6.1 SERIES and the workspace

The following is the top of the data file that we're using

DATE	Tb3mo	Tb1yr	RGDP	Potent	Deflator	M2	PPI	Curr
1960Q1	3.87	4.57	2845.3	2824.2	18.521	298.7	33.2	31.8
1960Q2	2.99	3.87	2832.0	2851.2	18.579	301.1	33.4	31.9
1960Q3	2.36	3.07	2836.6	2878.7	18.648	306.5	33.4	32.2
1960Q4	2.31	2.99	2800.2	2906.7	18.700	310.9	33.7	32.6

If we do the following:

```
open data quarterly(2012) .xls
cal(q) 1960:1
allocate 2012:4
data(org=obs, format=xls)
```

we create a series workspace with a standard length of 212 entries, which is 2012:4 given the quarterly calendar starting in 1960:1. At this point, it has eight series, in order, TB3MO, TB1YR, RGDP, POTENT, DEFLATOR, M2, PPI and CURR.

What does it mean for the workspace to have a standard length of 212 entries?
If we do the following

```
set sims = %ran(1.0)
stats sims
```

you'll see that SIMS is defined as 212 data points (the other statistics will differ because of randomness):

Statistics on Series SIMS			
Quarterly Data From 1960:01 To 2012:04			
Observations	212		
Sample Mean	0.069595	Variance	0.825391
Standard Error	0.908510	SE of Sample Mean	0.062397
t-Statistic (Mean=0)	1.115358	Signif Level (Mean=0)	0.265966
Skewness	-0.189381	Signif Level (Sk=0)	0.263675
Kurtosis (excess)	0.559017	Signif Level (Ku=0)	0.102232
Jarque-Bera	4.027656	Signif Level (JB=0)	0.133477

However, if you do

```
set sims 1 10000 = %ran(1.0)
stats sims
```

you'll get something like

Statistics on Series SIMS			
Quarterly Data From 1960:01 To 4459:04			
Observations	10000		
Sample Mean	-0.006149	Variance	0.980794
Standard Error	0.990350	SE of Sample Mean	0.009904
t-Statistic (Mean=0)	-0.620923	Signif Level (Mean=0)	0.534664
Skewness	-0.008913	Signif Level (Sk=0)	0.716008
Kurtosis (excess)	0.004250	Signif Level (Ku=0)	0.930898
Jarque-Bera	0.139914	Signif Level (JB=0)	0.932434

so `SIMS` now has 10000 data points. Thus the workspace length isn't a limit—it simply sets the standard length which is used if no other information is available. In general, that means only a few situations where this comes into play, typically on **SET** instructions. Because the expression on the right side of a **SET** could be quite complicated, RATS doesn't try to work out the range over which it could be computed, so, if there is no *end* parameter on the **SET**, it uses the standard length.

Note that you can lengthen a series easily, as we did here, changing `SIMS` from 212 to 10000 data points. A new **SET** on a series doesn't destroy the information that's already there. For instance, if you now repeat

```
set sims = %ran(1.0)
stats sims
```

you will replace the first 212 data points (default length), leaving everything from 213 to 10000 as it was.

What if you *want to* erase the old information in a series? You can do a **CLEAR** instruction. That replaces the current content of the series (as many as you list on the instruction) with NA's. If you now do

```
clear sims
set sims = %ran(1.0)
stats sims
```

you'll again see just 212 entries in the statistics.

What happens when you do a **SET** instruction involving lags?

```
set pi = 100.0*log(ppi/ppi{1})
```

Again, the target range for the **SET** is the standard 1 to 212. However, because `PPI{1}` isn't defined when `T=1`, the result for `PI` is an NA for entry 1. There is no effective difference between a series created from 1 to 212 with an NA in entry 1 and another which is defined only from 2 to 212, which is why we

suggest that you *not* try to adjust the ranges on **SET** to allow for lags—just let RATS handle it automatically.

As an example of the use of a series which is intentionally longer than the standard length, we'll generate draws of series of $N(0, 1)$ variates and see how well the Jarque-Bera test statistic compares to its asymptotic chi-squared distribution. We'll tack this onto the end of Example 6.1. First, we set the number of replications of the experiments and the length of the sampled series:

```
compute ndraws=10000
compute nob  =500
```

Now, we'll clear out the series of simulations so we don't get any unwanted data from before. We also zero out the series which will get the J-B statistics, making sure we extend it to the **NDRAWS** entries that we will need.

```
clear sims
set jbstats 1 ndraws = 0.0
```

The simulations and calculations of the Jarque-Bera statistics are done with:

```
do try=1,ndraws
  set sims 1 nob  = %ran(1.0)
  stats(noprint) sims
  compute jbstats(try)=%jbstat
end do try
```

At this point, we have 10000 (**NDRAWS** actually, since we wrote this to change that easily) samples of Jarque-Bera test statistics from independent standard Normal samples of length 500. The J-B statistic asymptotically has a χ^2_2 distribution. The following uses the **SSTATS** instruction (page 187) to evaluate the percentage of the draws that exceed the 5% and 1% critical values for the χ^2_2 . We would hope this would be close to .05 and .01 respectively.

```
compute crit05=%invchisqr(.05,2),crit01=%invchisqr(.01,2)
sstats(mean) 1 ndraws (jbstats>crit05)>>sim05 $
                  (jbstats>crit01)>>sim01

disp "JB Statistic"
disp "Rejections at .05" sim05 "at .01" sim01
```

The results will change because of the randomness, but they tend to be fairly similar to:

JB Statistic		
Rejections at .05	0.05510 at .01	0.01840

which would indicate that the empirical distribution is close, but, in practice, the tails are a bit thicker than the χ^2_2 .

6.2 SERIES and their integer handles

If you do the following

```
print 1970:1 1972:4 2 4 5
```

you'll get

ENTRY	TB1YR	POTENT	DEFLATOR
1970:01	7.55	4215.3	23.915
1970:02	7.45	4254.2	24.247
1970:03	6.94	4292.7	24.438
1970:04	5.65	4330.7	24.752
1971:01	4.05	4368.0	25.126
1971:02	4.99	4404.8	25.455
1971:03	5.75	4441.6	25.711
1971:04	4.73	4478.6	25.918
1972:01	4.41	4516.4	26.319
1972:02	4.84	4554.5	26.475
1972:03	5.15	4593.4	26.731
1972:04	5.44	4633.2	27.083

This is because each series created has an integer “handle” which is assigned in the order in which they are created. So TB3MO is number 1, TB1YR is number 2, etc. Handles are especially useful together with the **DOFOR** instruction. We first saw the **DOFOR** loop instruction on page 84 to handle a loop over a set of real values. In practice, it's probably more commonly used to loop over data series. The following, for instance, does a custom set of basic statistics on the raw data:¹

```
report(action=define)
report(atrow=1,atcol=1,align=center) "Series" "Mean" "Std Dev" $
    "Skew" "Kurtosis" "LB(Q)"
do for s = tb3mo to curr
    stats(noprint) s
    corr(noprint,qstats,number=12) s
    report(row=new,atcol=1) %l(s) %mean sqrt(%variance) %skewness $
        %kurtosis %qstat
end do for s
report(action=format,width=10)
report(action=show)
```

The first pass through the **DOFOR** loop, *s* (which is an **INTEGER**) is 1. The **STATISTICS** and **CORRELATE** instruction, and the **%L** function (which returns the label of a series) all understand that when they see an **INTEGER** where they expect a **SERIES** they should interpret it as the handle to a **SERIES**. The output that we get is:

¹This is for illustration—it would make little sense to offer the (excess) kurtosis and *Q* statistic on raw data like this.

Series	Mean	Std Dev	Skew	Kurtosis	LB(Q)
TB3MO	5.0325	2.9934	0.7154	1.0936	1448.0391
TB1YR	5.5788	3.1783	0.6662	0.8233	1595.7395
RGDP	7664.7505	3390.6523	0.3447	-1.2172	2602.4349
POTENT	7764.8722	3511.5367	0.3922	-1.0878	2609.8333
DEFLATOR	61.5296	31.5948	0.0578	-1.3610	2610.9023
M2	3136.8420	2648.8435	0.9134	-0.1237	2565.5939
PPI	99.9745	49.1331	0.0554	-1.1648	2587.5536
CURR	327.9118	309.0194	0.9423	-0.3364	2570.6722

The one place where it's harder for RATS to distinguish between an `INTEGER` as integer, and `INTEGER` as a series handle is in a `SET` or `FRML` expression. Here, the six series other than the two interest rates would often be studied in growth rates. With just six series, the easiest way to create their associated growth rates might be to just do six `SET` instructions. However, in other cases, you might have many more series than this which need transformation. The following shows how to do that:

```

do for s = rgdp to curr
    set %s("gr_" + %l(s)) = 100.0 * log(s{0} / s{1})
end do for s

```

`%S` maps a string expression to a `SERIES` (which can be new or existing). In this case, it will be a new series, which will be named `GR_RGDP` when `S` is representing `RGDP`, `GR_POTENT` when `S` is representing `POTENT`, etc. In the `SET` expression, you will note that we use `S{0}` and `S{1}` to represent the current and lagged value for `S`. The `S{1}` isn't a surprise, since that's how you represent a lag in a formula. If `S` were the name of an actual `SERIES`, rather than a `INTEGER` handle for `SERIES`, we could just use `S` by itself rather than `S{0}` to select the current value. However, since it's an `INTEGER`, `S` by itself means just the value of the handle (for instance, 3 for `RGDP`, 8 for `CURR`). Thus, the use of the `S{0}` notation, which means the current (0 lag) value for the series represented by `S`.

After we've generated the growth rates, we can make a (this time standard) statistical table of those series using:

```
table(picture="*.####") / %slke("gr_*")
```

which produces

Series	Obs	Mean	Std Error	Minimum	Maximum
GR_RGDP	211	0.7437	0.8625	-2.3276	3.8589
GR_POTENT	211	0.7755	0.1751	0.3491	1.1109
GR_DEFLATOR	211	0.8699	0.5921	-0.1956	2.9479
GR_M2	211	1.6788	0.8493	-0.2955	5.3601
GR_PPI	211	0.8420	1.1466	-5.1092	4.9597
GR_CURR	211	1.6991	1.0640	-1.7034	6.4793

`%SLIKE(string_exp)` returns a `VECTOR` of `INTEGER` series handles for the series whose labels match the string expression, where you can use `*` (match

any number of characters) and ? (match any one character) for wild cards. In this case, it will return a list of all the series which start with GR_.

A somewhat more complicated example of the use of **DOFOR** with series lists is the following, which does a regression of GDP growth on its own lags plus lags of the “real rate of interest”, where we try all four possible combinations of price indexes and interest rates for the real rate. This uses nested **DOFOR** loops, the outer one over the price index and the inner over the interest rate. Note again, how **RATE{0}** and **PRICE{0}** are used in the **SET** instruction to get the current value of the **RATE** and **PRICE** series.

```
set gdpgrow = 400.0*log(rgdp/rgdp{1})
do for price = deflator ppi
  do for rate = tb3mo tb1yr
    set realrate = rate{0}-400.0*log(price{0}/price{1})
    disp "Real Rate using" %1(rate) "and" %1(price)
    linreg gdpgrow
    # constant gdpgrow{1 to 4} realrate{1 to 4}
    exclude
    # realrate{1 to 4}
  end do for rate
end do for price
```

6.3 Series Names and Series Labels

The names of the eight original data series came off the data file. Suppose you don't like those; perhaps **POTENT** and **CURR** aren't descriptive enough, or you would prefer to use a common naming convention for (for instance) the GDP, price and money series. For a spreadsheet data file like we have here (similarly for a labeled text file), you can change the names on the way *into* RATS by suppressing the file labels. This is done with the combination of the **TOP=2** and **NOLABELS** options. **TOP=2** tells **DATA** to start processing information beginning with row 2 (thus skipping the top row with the labels) and **NOLABELS** tells **DATA** that there are no usable labels on the file, and that it is to use the labels off the **DATA** instruction. You, of course, have to be very careful with the order that you list the names on **DATA**, since that will be the only source of identification of the series. An example would be:

```
data(format=xls,org=columns,top=2,nolabels) / r3mo r1yr y yp $
pdef m2 ppi mcurr
```

This method of re-labeling is possible for only certain types of data files. For others (RATS, FRED, Haver, FAME and others that do “random access” for series), you have to request a series by the name under which it is stored in the database. Relabeling the series after the fact can be done using either the **LABELS** or the **EQV** (short for “EQuiValence”) instruction.

LABELS re-defines the “output label” of a series. This can be any string up to 16 characters. The output label is used for a bit more than just output, as the **%L** and **%SLIKE** functions both work off the labels. An example of **LABELS** (using the original program with the original data file names) is:

```
labels rgdp potent
# "Y" "YP"
```

If we now do

```
stats rgdp
```

the output will read

Statistics on Series Y			
Quarterly Data From 1960:01 To 2012:04			
Observations	212		
Sample Mean	7664.750472	Variance	11496522.844881
Standard Error	3390.652274	SE of Sample Mean	232.870954
t-Statistic (Mean=0)	32.914154	Signif Level (Mean=0)	0.000000
Skewness	0.344714	Signif Level (Sk=0)	0.041897
Kurtosis (excess)	-1.217166	Signif Level (Ku=0)	0.000374
Jarque-Bera	17.285112	Signif Level (JB=0)	0.000176

Note that (for output) **RGDP** is now re-labeled as **Y**. However, we still had to use **RGDP** on the **STATISTICS** instruction.

EQV goes farther than this by both defining a new output label *and* a new name for the series which you can use directly in instructions. This has a slightly different (simpler) instruction syntax than **LABELS** because **EQV** can only take legal variable names, not general string expressions. The following shows a use of **EQV**:

```
eqv tb3mo tb1yr
    r3mo r1yr
set spread = r1yr-r3mo
```

Note that after the **EQV** instruction, we can use **R1YR** and **R3MO** in expressions to refer to the two series. (You can also still use **TB1YR** and **TB3MO** if you want).

6.4 Dates as Integers

When you write a date expression like 2012:4, you are actually using an operator which takes the pair of numbers (2012 and 4) and uses the current **CALENDAR** scheme to convert that to an entry number (in this case 212). The numbers could be replaced with variables or expressions:

```
compute endyear=2012,endqtr=4
compute end=endyear:endqtr
```

Print out the four values of **RGDP** from 1970:1 through 1970:4 using

```
print 1970:1 1970:4 rgdp
```

ENTRY	RGDP
1970:01	4252.9
1970:02	4260.7
1970:03	4298.6
1970:04	4253.0

Now try using:

```
print(nodates) 1970:1 1970:4 rgdp
```

ENTRY	RGDP
41	4252.9
42	4260.7
43	4298.6
44	4253.0

As **RGDP** is stored, it has 212 entries which are numbered from 1 to 212. The association of the entry number 41 with 1970:1 is based upon the current **CALENDAR** setting. If you change the **CALENDAR**,² *the data don't move*, all that changes is the association of a data point with a particular date. For illustration, if we now do

```
cal(m) 1980:1
print 41 44 rgdp
```

ENTRY	RGDP
1983:05	4252.9
1983:06	4260.7
1983:07	4298.6
1983:08	4253.0

Note that the data in entries 41 through 44 haven't changed, but the dates now associated with those entries have. *The time to change data frequencies is when you read the data in the first place, not later on.*

When you operate with the date functions, note that the calculation “wraps” the way you would expect. With the **CALENDAR** reset, let's try

```
disp %datelabel(2000:5)
```

This will display

2001:01

RATS has quite a large collection of date-related functions. Some of these are relatively straightforward functions that let you decompose the dates of entries. We've already used **%DATELABEL** to display the standard label of an entry. Others are **%YEAR** and **%PERIOD**. For instance,

²Which you should only do if you understand *exactly* what is happening

```
do time=1970:3,1971:4
    disp %datelabel(time) %year(time) %period(time)
end do time
```

will generate

```
1970:03 1970 3
1970:04 1970 4
1971:01 1971 1
1971:02 1971 2
1971:03 1971 3
1971:04 1971 4
```

If (for some reason) we needed a dummy variable for quarter 4 for years 1980 to 1989, we could generate that with

```
set d80_89q4 = %year(t)>=1980.and.%year(t)<=1989.and.%period(t)==4
```

There are quite a few others which are based upon a perpetual calendar—these tend to be more interesting with monthly data and particularly with sector or firm level data. But,

```
do time=1970:3,1971:4
    disp %datelabel(time) %daycount(time) %tradeday(time,5)
end do time
```

displays the number of days in the quarter, and the number of Fridays (week-day number 5, as the date functions number them):

```
1970:03 92 13
1970:04 92 13
1971:01 90 13
1971:02 91 13
1971:03 92 13
1971:04 92 14
```

If you do calculations that include a date field, the `:` operator takes precedence over other arithmetic operations (`+` and `-` being the only ones likely to ever be used). Thus,

```
disp 1980:1+1 1+1980:1
```

maps both expressions to entry 82, which is 1 period after 1980:1. However, even though that's how they are interpreted, it would be easier to read if you add parentheses, so $(1980:1)+1$ and $1+(1980:1)$ would be preferred.

On page 148, we checked for a broken trend in the GDP series. There, we ran the loop over possible break points from 1965:1 to 2007:4 to exclude breaks in the 20 observations at either end. We could also have let RATS figure out the limits for the loop using:

```
do t0=(1960:1)+20, (2012:4)-20
    ...
```

Of course, a better way to handle this would be

```
compute nobreak=20
do t0=(1960:1)+nobreak, (2012:4)-nobreak
```

so it would be more obvious what the point of the “20” is, and also to make it easier to change if we need to.

6.5 Tips and Tricks

The Instruction **SSTATS**

SSTATS is a handy instruction which can be used to compute the sum (or mean or maximum, etc.) of one or more general expressions. Since it accepts a formula, you don’t have to take the extra step of generating a separate series with the needed values.

It can be used to answer some (apparently) quite complicated questions. For instance,

```
sstats (min, smpl=peak+trough) start1 endl t>>tp0
```

gives `tp0` as the smallest entry for which either the series `peak` or `trough` (both dummies) is “true”.

```
sstats 1 nobs p1*y>>plys p1>>pls p2*y>>p2ys p2>>p2s
```

computes four parallel sums. Without the **SSTATS**, this would require about eight separate instructions.

```
sstats / date<>date{1}>>daycount
```

computes the number of days in a data set with intra-day data. `date<>date1` is 1 when the value of `date(t)` is different from `date(t-1)` and 0 if it’s the same. So the **SSTATS** is summing the number of changes in the date series.

In Example 6.1, we use the following:

```
sstats (mean) 1 10000 (jbstats>crit05)>>sim05 (jbstats>crit01)>>sim01
```

`JBSTATS>CRIT05` is 1 if `JBSTATS(t)` is bigger than the .05 critical value and 0 if it isn’t. When we request the mean for this calculation, we are getting the fraction of draws which exceed `CRIT05`. Similarly, the parallel calculation of `JBSTATS>CRIT01` is computing the fraction that exceed the .01 critical value.

Example 6.1 Series and Workspace Length

This demonstrates the effect of the standard workspace length versus extended lengths.

```
open data quarterly(2012).xls
cal(q) 1960:1
allocate 2012:4
data(org=obs,format=xls)
*
* Generate random data over the standard range
*
set sims = %ran(1.0)
stats sims
*
* Generate random data over longer range
*
set sims 1 10000 = %ran(1.0)
stats sims
*
* Replace data over standard range
*
set sims = %ran(1.0)
stats sims
*
* Clear information and generate data only over standard range
*
clear sims
set sims = %ran(1.0)
stats sims
*
* Example of use of "overlong" series
*
compute ndraws=10000
compute nob =500
*
clear sims
set jbstats 1 ndraws = 0.0
do try=1,ndraws
    set sims 1 nob = %ran(1.0)
    stats(noprint) sims
    compute jbstats(try)=jbstat
end do try
*
compute crit05=%invchisqr(.05,2),crit01=%invchisqr(.01,2)
sstats(mean) 1 ndraws (jbstats>crit05)>>sim05 $
                    (jbstats>crit01)>>sim01
disp "JB Statistic"
disp "Rejections at .05" sim05 "at .01" sim01
```

Example 6.2 Series handles and DOFOR

This demonstrates the use of the **DOFOR** instruction with lists of series.

```

open data quarterly(2012).xls
cal(q) 1960:1
allocate 2012:4
data(org=obs, format=xls)
*
* Create custom table of statistics
*
report(action=define)
report(atrow=1, atcol=1, align=center) "Series" "Mean" "Std Dev" $
      "Skew" "Kurtosis" "LB(Q)"
do for s = tb3mo to curr
  stats(noprint) s
  corr(noprint, qstats, number=12) s
  report(row=new, atcol=1) %1(s) %mean sqrt(%variance) $
    %skewness %kurtosis %qstat
end do for s
report(action=format, picture="*.###")
report(action=show)
*
* Create growth rates for non-interest rates
*
do for s = rgdp to curr
  set %s("gr_"+%1(s)) = 100.0*log(s{0}/s{1})
end do for s
*
table(picture="*.####") / %slike("gr_*")
*
* Do regression with generated real rates of interest
*
set gdpgrow = 400.0*log(rgdp/rgdp{1})
do for price = deflator ppi
  do for rate = tb3mo tb1yr
    set realrate = rate{0}-400.0*log(price{0}/price{1})
    disp "Real Rate using" %1(rate) "and" %1(price)
    linreg gdpgrow
    # constant gdpgrow{1 to 4} realrate{1 to 4}
    exclude
    # realrate{1 to 4}
  end do for rate
end do for price
*
* Relabel series
*
labels rgdp potent
# "Y" "YP"
*
stats rgdp
*
eqv tb3mo tb1yr

```

```

    r3mo r1yr
set spread = r1yr-r3mo

```

Example 6.3 Date calculations and functions

```

open data quarterly(2012).xls
cal(q) 1960:1
*
* Using expression for date
*
compute endyear=2012,endqtr=4
compute end=endyear:endqtr
*
allocate end
data(org=obs,format=xls)
*
print 1970:1 1970:4 rgdp
print(nodates) 1970:1 1970:4 rgdp
*
* For illustration (this is dangerous!)
*
cal(m) 1980:1
print 41 44 rgdp
*
* Reset the calendar
*
cal(q) 1960:1
*
disp %datelabel(2000:5)
*
do time=1970:3,1971:4
    disp %datelabel(time) %year(time) %period(time)
end do time
*
* Create quarter 4 dummy for 1980-1989
*
set d80_89q4 = %year(t)>=1980.and.%year(t)<=1989.and.%period(t)==4
*
do time=1970:3,1971:4
    disp %datelabel(time) %daycount(time) %tradeday(time,5)
end do time
*
disp 1980:1+1 1+1980:1

```

Probability Distributions

A.1 Univariate Normal

Parameters	Mean (μ), Variance (σ^2)
Kernel	$\sigma^{-1} \exp \left(-\frac{(x - \mu)^2}{2\sigma^2} \right)$
Support	$(-\infty, \infty)$
Mean	μ
Variance	σ^2
Main uses	Prior, exact and approximate posteriors for parameters with unlimited ranges.
Density Function	<code>%DENSITY(x)</code> is the non-logged standard Normal density. More generally, <code>%LOGDENSITY(variance,u)</code> . Use <code>%LOGDENSITY(sigmasq,x-mu)</code> to compute $\log f(x \mu, \sigma^2)$.
CDF	<code>%CDF(x)</code> is the standard Normal CDF. To get $F(x \mu, \sigma^2)$, use <code>%CDF((x-mu)/sigma)</code>
Draws	<code>%RAN(s)</code> draws one or more (depending upon the target) independent $N(0, s^2)$. <code>%RANMAT(m,n)</code> draws a matrix of independent $N(0, 1)$.

A.2 Univariate Student (*t*)

Parameters	Mean (μ), Variance of underlying Normal (σ^2) or of the distribution itself (s^2), Degrees of freedom (ν)
Kernel	$(1 + (x - \mu)^2 / (\sigma^2 \nu))^{-(\nu+1)/2}$ or $(1 + (x - \mu)^2 / (s^2(\nu - 2)))^{-(\nu+1)/2}$
Support	$(-\infty, \infty)$
Mean	μ
Variance	$\sigma^2 \nu / (\nu - 2)$ or s^2
Main uses	Prior, exact and approximate posteriors for parameters with unlimited ranges.
Density Function	<code>%TDENSITY(x, nu)</code> is the (non-logged) density function for a standard ($\mu = 0, \sigma^2 = 1$) <i>t</i> . <code>%LOGTDENSITY(ssquared, u, nu)</code> is the log density based upon the s^2 parameterization. Use <code>%LOGTDENSITY(ssquared, x-mu, nu)</code> to compute $\log f(x \mu, s^2, \nu)$ and <code>%LOGTDENSITYSTD(sigmasq, x-mu, nu)</code> to compute $\log f(x \mu, \sigma^2, \nu)$. ¹
CDF	<code>%TCDF(x, nu)</code> is the CDF for a standard <i>t</i> .
Draws	<code>%RANT(nu)</code> draws one or more (depending upon the target) standard <i>t</i> 's with independent numerators and a <i>common</i> denominator. To get a draw from a <i>t</i> density with variance <code>ssquared</code> and <code>nu</code> degrees of freedom, use <code>%RANT(nu)*sqrt(ssquared*(nu-2.)/nu)</code> .
Notes	With $\nu = 1$, this is a Cauchy (no mean or variance); with $\nu \leq 2$, the variance doesn't exist. $\nu \rightarrow \infty$ tends towards a Normal.

¹`%LOGTDENSITYSTD` and `%TCDF` were added with RATS 7.3. Before that, use `%LOGTDENSITY(sigmasq*nu/(nu-2), x-mu, nu)` and `%TCDFNC(x, nu, 0.0)`.

A.3 Chi-Squared Distribution

Parameters	Degrees of freedom (ν).
Kernel	$x^{(\nu-2)/2} \exp(-x/2)$
Range	$[0, \infty)$
Mean	ν
Variance	2ν
Main uses	Prior, exact and approximate posterior for the precision (reciprocal of variance) of residuals or other shocks in a model
Density function	<code>%CHISQRDENSITY(x, nu)</code>
Tail Probability	<code>%CHISQR(x, nu)</code>
Random Draws	<code>%RANCHISQR(nu)</code> draws one or more (depending upon the target) independent chi-squareds with NU degrees of freedom.

A.4 Gamma Distribution

Parameters	shape (a) and scale (b), alternatively, degrees of freedom (ν) and mean (μ). The RATS functions use the first of these. The relationship between them is $a = \nu/2$ and $b = \frac{2\mu}{\nu}$. The chi-squared distribution with ν degrees of freedom is a special case with $\mu = \nu$.
Kernel	$x^{a-1} \exp\left(-\frac{x}{b}\right)$ or $x^{(\nu/2)-1} \exp\left(-\frac{x\nu}{2\mu}\right)$
Range	$[0, \infty)$
Mean	ba or μ
Variance	b^2a or $\frac{2\mu^2}{\nu}$
Main uses	Prior, exact and approximate posterior for the precision (reciprocal of variance) of residuals or other shocks in a model
Density function	<code>%LOGGAMMADENSITY(x, a, b)</code> . Built-in with RATS 7.2. Available as procedure otherwise. For the $\{\nu, \mu\}$ parameterization, use <code>%LOGGAMMADENSITY(x, .5*nu, 2.0*mu/nu)</code>
Random Draws	<code>%RANGAMMA(a)</code> draws one or more (depending upon the target) independent Gammas with unit scale factor. Use <code>b*%RANGAMMA(nu)</code> to get a draw from $Gamma(a, b)$. If you are using the $\{\nu, \mu\}$ parameterization, use <code>2.0*mu*%RANGAMMA(.5*nu)/nu</code> . You can also use <code>mu*%RANCHISQR(nu)/nu</code> .
Moment Matching	<code>%GammaParms(mean, sd)</code> (external function) returns the 2-vector of parameters ((a, b) parameterization) for a gamma with the given mean and standard deviation.

A.5 Multivariate Normal

Parameters	Mean (μ), Covariance matrix (Σ) or precision (H)
Kernel	$ \Sigma ^{-1/2} \exp \left(-\frac{1}{2} (x - \mu)' \Sigma^{-1} (x - \mu) \right)$ or $ \mathbf{H} ^{1/2} \exp \left(-\frac{1}{2} (x - \mu)' \mathbf{H} (x - \mu) \right)$
Support	$\mathbb{R}^n$
Mean	μ
Variance	Σ or H^{-1}
Main uses	Prior, exact and approximate posteriors for a collection of parameters with unlimited ranges.
Density Function	<code>%LOGDENSITY(sigma,u)</code> . To compute $\log f(x \mu, \Sigma)$ use <code>%LOGDENSITY(sigma,x-mu)</code> . (The same function works for univariate and multivariate Normals).
Draws	<code>%RANMAT(m,n)</code> draws a matrix of independent $N(0,1)$. <code>%RANMVNORMAL(F)</code> draws an n -vector from a $N(0, FF')$, where F is any factor of the covariance matrix. This setup is used (rather than taking the covariance matrix itself as the input) so you can do the factor just once if it's fixed across a set of draws. To get a single draw from a $N(\mu, \Sigma)$, use <code>MU+%RANMVNORMAL(%DECOMP(SIGMA))</code> <code>%RANMVPOST</code> , <code>%RANMVPOSTCMOM</code> , <code>%RANMVKRON</code> and <code>%RANMVKRONCMOM</code> are specialized functions which draw multivariate Normals with calculations of the mean and covariance matrix from other matrices.

Quasi-Maximum Likelihood Estimations (QMLE)

The main source for results on QMLE is White (1994). Unfortunately, the book is so technical as to be almost unreadable. We'll try to translate the main results as best we can.

Suppose that $\{x_t\}, t = 1, \dots, \infty$ is a stochastic process and suppose that we have observed a finite piece of this $\{x_1, \dots, x_T\}$ and that the true (unknown) log joint density of this can be written

$$\sum_{t=1}^T \log g_t(x_t, \dots, x_1)$$

This is generally no problem for either cross section data (where independence may be a reasonable assumption) or time series models where the data can be thought of as being generated sequentially. Some panel data likelihoods will not, however, be representable in this form.

A (log) quasi likelihood for the data is a collection of density functions indexed by a set of parameters θ of the form

$$\sum_{t=1}^T \log f_t(x_t, \dots, x_1; \theta)$$

which it is hoped will include a reasonable approximation to the true density. In practice, this will be the log likelihood for a mathematically convenient representation of the data such as joint Normal. The QMLE is the (or more technically, a, since there might be non-uniqueness) $\hat{\theta}$ which maximizes the log quasi-likelihood.

Under the standard types of assumptions which would be used for actual maximum likelihood estimation, $\hat{\theta}$ proves to be consistent and asymptotically Normal, where the asymptotic distribution is given by $\sqrt{T}(\hat{\theta} - \theta) \xrightarrow{d} N(0, A^{-1}BA^{-1})$, where A is approximated by

$$A_T = \frac{1}{T} \sum_{t=1}^T \frac{\partial^2 \log f_t}{\partial \theta \partial \theta'}$$

and B by (if there is no serial correlation in the gradients)

$$B_T = \frac{1}{T} \sum_{t=1}^T \left(\frac{\partial \log f_t}{\partial \theta} \right)' \left(\frac{\partial \log f_t}{\partial \theta} \right) \quad (\text{B.1})$$

with the derivatives evaluated at $\hat{\theta}$.¹ Serial correlation in the gradients is handled by a Newey-West type calculation in (B.1). This is the standard “sandwich” estimator for the covariance matrix. For instance, if $\log f_t = -(x_t - z_t\theta)^2$, (with z_t treated as exogenous), then

$$\frac{\partial \log f_t}{\partial \theta} = 2(x_t - \theta z_t) z_t'$$

and

$$\frac{\partial^2 \log f_t}{\partial \theta \partial \theta'} = -2z_t' z_t$$

and the asymptotic covariance matrix of $\hat{\theta}$ is

$$\left(\sum z_t' z_t \right)^{-1} \left(\sum z_t' u_t^2 z_t \right) \left(\sum z_t' z_t \right)^{-1}$$

the standard Eicker-White robust covariance matrix for least squares. Notice that, when you compute the covariance matrix this way, you can be somewhat sloppy with the constant multipliers in the log quasi likelihood—if this were the actual likelihood for a Normal, $\log f_t$ would have a $\frac{1}{2\sigma^2}$ multiplier, but that would just cancel out of the calculation since it gets squared in the center factor and inverted in the two ends.

This is very nice, but what is the θ_0 to which this is converging? After all, nothing above actually required that the f_t even approximate g_t well, much less include it as a member. It turns out that this is the value which minimizes the Kullback-Liebler Information Criterion (KLIC) discrepancy between f and g which is (suppressing various subscripts) the expected value (over the density g) of $\log(g/f)$. The KLIC has the properties that it’s non-negative and is equal to zero only if $f = g$ (almost everywhere), so the QMLE will at least asymptotically come up with the member of the family which is closest (in the KLIC sense) to the truth.

Again, closest might not be close. However, in practice, we’re typically less interested in the complete density function of the data than in some aspects of it, particularly moments. A general result is that if f is an appropriate selection from the linear exponential family, then the QMLE will provide asymptotically valid estimates of the parameters in a conditional expectation. The linear exponential family are those for which the density takes the form

$$\log f(x; \theta) = a(\theta) + b(x) + \theta' t(x) \quad (\text{B.2})$$

This is a very convenient family because the interaction between the parameters and the data is severely limited.² This family includes the Normal, gamma (chi-squared and exponential are special cases), Weibull and beta distributions

¹The formal statement of this requires pre-multiplying the left side by a matrix square root of $AB^{-1}A$ and having the target covariance matrix be the identity.

²The exponential family in general has $d(\theta)$ entering into that final term, though if d is invertible, it’s possible to reparameterize to convert a general exponential to the linear form.

among continuous distributions and binomial, Poisson and geometric among discrete ones. It *does not* include the logistic, t , F , Cauchy and uniform.

For example, suppose that we have “count” data—that is, the observable data are nonnegative integers (number of patents, number of children, number of job offers, etc.). Suppose that we posit that the expected value takes the form $E(y_t|w_t) = \exp(w_t\theta)$. The Poisson is a density in the exponential family which has the correct support for the underlying process (that it, it has a positive density only for the non-negative integers). Its probability distribution (as a function of its single parameter λ) is defined by $P(x; \lambda) = \frac{\exp(-\lambda)\lambda^x}{x!}$. If we define $\omega = \log(\lambda)$, this is linear exponential family with $a(\omega) = -\exp(\omega)$, $b(x) = \log x!$, $t(x) = x$. There’s a very good chance that the Poisson will not be the correct distribution for the data because the Poisson has the property that both its mean and its variance are λ . Despite that, the Poisson QMLE, which maximizes $\sum -\exp(w_t\theta) + x_t(w_t\theta)$, will give consistent, asymptotically Normal estimates of θ .

It can also be shown that, under reasonably general conditions, if the “model” provides a set of moment conditions (depending upon some parameters) that match up with QMLE first order conditions from a linear exponential family, then the QMLE provides consistent estimates of the parameters in the moment conditions.

Delta method

The *delta method* is used to estimate the variance of a non-linear function of a set of already estimated parameters. The basic result is that if θ are the parameters and we have

$$\sqrt{T} \left(\hat{\theta} - \theta \right) \xrightarrow{d} N(0, \Sigma_{\theta}) \quad (\text{C.1})$$

and if $f(\theta)$ is continuously differentiable, then, by using a first order Taylor expansion

$$\left(f(\hat{\theta}) - f(\theta) \right) \approx f'(\theta) \left(\hat{\theta} - \theta \right)$$

Reintroducing the $\sqrt{T}$ scale factors and taking limits gives

$$\sqrt{T} \left(f(\hat{\theta}) - f(\theta) \right) \xrightarrow{d} N \left(0, f'(\theta) \Sigma_{\theta} f'(\theta)' \right)$$

In practice, this means that if we have

$$\hat{\theta} \approx N(\theta, \mathbf{A}) \quad (\text{C.2})$$

then

$$f \left(\hat{\theta} \right) \approx N \left(f(\theta), f'(\hat{\theta}) \mathbf{A} f'(\hat{\theta})' \right) \quad (\text{C.3})$$

(C.1) is the type of formal statement required, since the $\mathbf{A}$ in (C.2) collapses to zero as $T \rightarrow \infty$. It's also key that (C.1) implies that $\hat{\theta} \xrightarrow{p} \theta$, so $f'(\hat{\theta}) \xrightarrow{p} f'(\theta)$ allowing us to replace the unobservable $f'(\theta)$ with the estimated form in (C.3). So the point estimate of the function is the function of the point estimate, at least as the center of the asymptotic distribution. If $\hat{\theta}$ is unbiased for θ , then it's almost certain that $f(\hat{\theta})$ will *not* be unbiased for $f(\theta)$ so this is *not* a statement about expected values.

To compute the asymptotic distribution, it's necessary to compute the partial derivatives of f . For scalar functions of the parameters estimated using a RATS instruction, that can usually be most easily done using the instruction **SUMMARIZE**.

Central Limit Theorems with Dependent Data

The simplest form of Central Limit Theorem (CLT) assumes a sequence of i.i.d. random variables with finite variance. Under those conditions, regardless of the shape of the distributions (anything from 0-1 Bernoullis to fat-tailed variables with infinite fourth moments),

$$\sqrt{T}(\bar{x} - \mu) \xrightarrow{d} N(0, \sigma^2) \quad (\text{D.1})$$

Those were extended to allow independent, but non-identically distributed, random variables as long as there was some control on the tail behavior and the relative variances to prevent a small percentage of the summands from dominating the result. The assumption of independence serves two purposes:

1. It makes it much easier to prove the result, since it's relatively easy to work with characteristic functions of independent random variables.
2. Independence helps to restrict the influence of each element.

In time series analysis, independence is too strong an assumption. However, it's still possible to construct CLT's with weaker assumptions as long as the influence of any small number of elements is properly controlled.

One type of useful weakening of independence is to assume a sequence is a *martingale difference sequence* (m.d.s.). $\{u_t\}$ is an m.d.s. if

$$E(u_t | u_{t-1}, u_{t-2}, \dots) = 0$$

It's called this because a martingale is a sequence which satisfies

$$E(x_t | x_{t-1}, x_{t-2}, \dots) = x_{t-1}$$

so, by the Law of Iterated Expectations (conditioning first on a superset)

$$\begin{aligned} E(x_t - x_{t-1} | x_{t-1} - x_{t-2}, x_{t-2} - x_{t-3}, \dots) &= \\ E(E(x_t - x_{t-1} | x_{t-1}, x_{t-2}, \dots) | x_{t-1} - x_{t-2}, x_{t-2} - x_{t-3}, \dots) &= 0 \end{aligned}$$

thus the first difference of a martingale is an m.d.s. An i.i.d. mean zero process is trivially an m.d.s. A non-trivial example is $u_t = \varepsilon_t \varepsilon_{t-1}$, where ε_t is an i.i.d. mean zero process. u_t isn't independent of u_{t-1} because they share a ε_{t-1} factor; as a result, the *variances* of u_t and u_{t-1} will tend to move together.

The ergodic martingale CLT states that if u_t is a stationary ergodic m.d.s. and $Eu_t^2 = \sigma^2$, then

$$\frac{1}{\sqrt{T}} \sum_{t=1}^T u_t \xrightarrow{d} N(0, \sigma^2)$$

We can write this (somewhat informally) as

$$\frac{1}{\sqrt{T}} \sum_{t=1}^T u_t \xrightarrow{d} N(0, Eu_t^2)$$

and very informally, this is used as

$$\sum_t u_t \approx N\left(0, \sum_t u_t^2\right) \quad (\text{D.2})$$

This is the form that is useful when we have serially uncorrelated (though not necessarily serial independent) summands. However, it won't handle serial correlation. A basic CLT which can be applied more generally is the following: if

$$x_t = \sum_{s=0}^q c_s \varepsilon_{t-s} \quad (\text{D.3})$$

where ε_t has assumptions which generate a standard $N(0, \sigma^2)$, then

$$\frac{1}{\sqrt{T}} \sum_{t=1}^T x_t \xrightarrow{d} N\left(0, \left(\sum c_s\right)^2 \sigma^2\right) \quad (\text{D.4})$$

If we write $x_t = C(L)\varepsilon_t$, then we can write $C(1) = \sum_{s=0}^q c_s$, so the limiting distribution can be written $C(1)^2 \sigma^2$. This is known as the long-run variance of x : if ε_t were subject to a permanent shift generated by a random variable with variance σ^2 , the variance that would produce in x_t is $C(1)^2 \sigma^2$.

The somewhat informal restatement of this is

$$\frac{1}{\sqrt{T}} \sum_{t=1}^T x_t \xrightarrow{d} N(0, lvar(x))$$

where $lvar(x)$ is the long-run variance of the x process, and in practice we use

$$\sum_t x_t \approx N\left(0, \sum_t \sum_{l=-L}^L w_l x_t x'_{t-l}\right) \quad (\text{D.5})$$

where the variance in the target distribution uses some feasible estimator for the long-run variance (such as Newey-West).

The approximating covariance matrix in (D.2) can be computed using the instruction **CMOMENT** (applied to u), or with **MCOV** without any **LAG** options, and

that in (D.5) can be computed using **MCOV** using `LAG` and `LWINDOW` options. Note that both these are written using sums (not means) on both sides. That tends to be the most convenient form in practice—when you try to translate a result from the literature, you need to make sure that you get the factors of T correct.

Bibliography

- BOLLERSLEV, T. (1986): "Generalized Autoregressive Conditional Heteroskedasticity," *Journal of Econometrics*, 31(3), 307–327.
- CHAN, K. (1993): "Consistency and Limiting Distribution of the Least Squares Estimator of a Threshold Autoregressive Models," *Annals of Statistics*, 21, 520–533.
- DIEBOLD, F. X., AND R. S. MARIANO (1995): "Comparing Predictive Accuracy," *Journal of Business and Economic Statistics*, 13, 253–263.
- ENDERS, W. (2010): *Applied Econometric Time Series*. Wiley, 3rd edn.
- ENGLE, R. F. (1982): "Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation," *Econometrica*, 50(4), 987–1007.
- ENGLE, R. F., AND C. W. J. GRANGER (1987): "Co-integration and Error Correction: Representation, Estimation, and Testing," *Econometrica*, 55, 251–276.
- ENGLE, R. F., D. M. LILIEN, AND R. P. ROBINS (1987): "Estimating Time Varying Risk Premia in the Term Structure: The Arch-M Model," *Econometrica*, 55(2), 391–407.
- GRANGER, C. W. J., AND P. NEWBOLD (1973): "Some Comments on the Evaluation of Economic Forecasts," *Applied Economics*, 5, 35–47.
- HYLLEBERG, S., R. F. ENGLE, C. W. J. GRANGER, AND B. S. YOO (1990): "Seasonal Integration and Cointegration," *Journal of Econometrics*, 44, 215–238.
- TERASVIRTA, T. (1994): "Specification, Estimation and Evaluation of Smooth Transition Autoregressive Models," *Journal of American Statistical Association*, 89(425), 208–218.
- TONG, H. (1983): *Threshold Models in Nonlinear Time Series Analysis*. New York: Springer Verlag.
- WHITE, H. (1980): "A Heteroskedasticity-Consistent Covariance Matrix Estimator and a Direct Test for Heteroskedasticity," *Econometrica*, 48, 817–838.
- (1994): *Estimation, Inference and Specification Analysis*. Cambridge: Cambridge University Press.

Index

- Additive outlier, 150
- AIC, 23
- Akaike Information Criteria, 23
- %ALLOCEND function, 45
- @ARCHTEST procedure, 124
- Bayesian Information Criterion, 23
- BFGS algorithm, 135
- BIC, 23
- Bilinear model, 98
- @BJAUTOFIT procedure, 33, 36
- @BJDIFF procedure, 38
- @BJIDENT procedure, 27
- BOXJENK instruction, 28
 - DEFINE option, 42
- BREAK instruction, 156
- Burn-in, 80
- CDF instruction, 137
- %CDSTAT variable, 16, 138
- Chan, K., 161
- Chi-squared distribution, 194
- %CHISQR function, 138
- CLEAR instruction, 179
- CMOMENT instruction, 153
- Compiled language, 144
- COMPUTE instruction
 - for series elements, 45
- %CONVERGED variable, 28, 69
- CORRELATE instruction, 14
- DATA instruction
 - NOLABELS option, 183
 - TOP option, 183
- Delta method, 200
- Diebold, F., 48
- @DMARIANO procedure, 48
- DO instruction, 34
- DOFOR instruction, 84
- Double precision, 101
- @EGTEST procedure, 25
- Enders, W., 26, 79
- @EndersGranger procedure, 169
- @EndersSiklos procedure, 169
- Engle, R., 22, 38, 124
- %EQNPRJ function, 171
- %EQNREGLABELS function, 170
- %EQNRESID function, 171
- %EQNRVALUE function, 171
- %EQNSIZE function, 170
- %EQNVALUE function, 171
- %EQNXVECTOR function, 170
- EQV instruction, 183
- ESTAR model, 77
- EXCLUDE instruction, 16
- %EXP function, 86
- EXTREMUM instruction, 89
- FIX function, 162
- FORECAST instruction, 43
- %FRACTnn variables, 84
- FRML instruction, 65, 68
 - LASTREG option, 94
- %FTEST function, 138
- %FUNCVAL variable, 121
- Gamma distribution, 195
- Gauss-Newton algorithm, 66
- @GMAUTOFIT procedure, 40
- @GNEWBOLD procedure, 47
- Granger, C.W.J., 22, 38, 47
- GRAPH instruction, 10
 - NODATES option, 22
 - NUMBER option, 22
- Grid search, 83
- @HEGY procedure, 38
- Hylleberg, S., 38
- Identification
 - lack of, 64
- Innovational outlier, 148
- INQUIRE instruction, 169

- Inter-quartile range, 86
- Interpreted language, 143
- %INVNORMAL function, 49
- LABELS** instruction, 183
- lag length tests, 1
- Least squares
 - nonlinear, 64
- LINREG** instruction, 12
 - DEFINE option, 164
- Ljung-Box Q statistic, 15
- %LOGDENSITY function, 119, 196
- %LOGISTIC function, 79
- %LOGL variable, 121
- %LOGTDENSITY function, 119, 193
- %LOGTDENSITYSTD function, 193
- Loss of precision, 101
- LSTAR model, 77
- Mariano, R., 48
- Martingale difference sequence, 201
- MAXIMIZE** instruction, 117
- %MINENT variable, 89
- Multivariate Normal distribution, 196
- %NARMA variable, 31
- %NDFTEST variable, 16
- Newbold, P., 47
- NLLS** instruction, 65, 69
 - PARMSET option, 95
- NLPAR** instruction, 102
- NONLIN** instruction, 65, 67
- Nonlinear least squares, 64
- Normal distribution, 192
- Numerical derivatives, 104
- ORDER** instruction, 162
- Outlier
 - additive, 150
 - innovational, 148
- Overflow, 101
- PARMSET data type, 94
- PITERS option, 134
- PMETHOD option, 134
- Precision
 - double, 101
 - loss of, 101
- Probability distributions
 - chi-squared, 194
 - gamma, 195
 - multivariate normal, 196
 - normal, 192
 - t, 193
- Q statistic, 15
- %RANMAT function, 196
- %RANMVNORMAL function, 196
- %RANT function, 193
- Recursive formula, 98
- Recursive residuals, 51
- @REGCORRS procedure, 32
- @REGCRITS procedure, 25
- %REGSTART function, 23
- @REGSTRTEST procedure, 93
- %RESIDS series, 20, 69
- RESTRICT** instruction, 17
- RLS** instruction, 51
- %S function, 182
- SBC, 23
- Schwarz Bayesian Criterion, 23
- SEED** instruction, 105
- %SEQA function, 84
- Series
 - output label of, 184
- %SIGNIF variable, 16, 138
- Simplex algorithm, 133
- %SLIKE function, 182
- Smooth Transition Regression, 87
- SPGRAPH** instruction, 11
- SSTATS** instruction, 187
- STAR model, 77
 - problems with outliers, 96
- @STARTEST procedure, 93
- STATISTICS** instruction
 - FRACTILES option, 84
- STR model, 87
- %SUMLC variable, 17
- SUMMARIZE** instruction, 16

t distribution, 193
TAR model, 159
@TAR procedure, 169
%TCDF function, 193
%TDENSITY function, 193
Terasvirta, T., 93
TEST instruction, 17, 73
Threshold autoregression, 159
@THRESHTEST procedure, 169
Tong, H., 159
%TTEST function, 138

UFORECAST instruction, 42
Underflow, 101
UNTIL instruction, 157

VAR, 1
%VARLC variable, 17
Vector autoregression, 1

White, H., 20, 197

Yoo, B.S., 38

%ZTEST function, 138