

RATS

VERSION 11

USER'S GUIDE

RATS

VERSION 11

USER'S GUIDE

Estima

2717 Harrison St
Evanston, IL 60201

Orders, Inquiries
Technical Support
Fax:

+1 847-864-8772
+1 847-864-1910
+1 847-447-0174

Web:

www.estima.com

Sales:

sales@estima.com

Technical Support:

support@estima.com

© 2025 by Estima. All Rights Reserved.

No part of this book may be reproduced or transmitted in any form or by any means without the prior written permission of the copyright holder.

Estima
2717 Harrison St
Evanston, IL 60201

Published in the United States of America

Table of Contents

What's New in Version 10?	UG–ix
New Functions	UG–xi
New Instructions.....	UG–xi
Changes to Existing Instructions.....	UG–xii

1. Scalars, Matrices, and Functions **UG–1**

1.1 Working with Scalars.....	UG–2
1.2 Functions.....	UG–6
1.3 Getting Information from RATS	UG–8
1.4 The RATS Data Types.....	UG–10
1.4.1 Basic Data Types.....	UG–14
1.4.2 The Aggregate Data Types	UG–16
1.4.3 Using Arrays	UG–18
1.5 Data Series (The SERIES Type)	UG–19
1.6 Strings and Labels.....	UG–23
1.7 Matrix Calculations	UG–25
1.8 Calendar/Date Functions.....	UG–30
1.9 Creating Reports	UG–31

2. Linear Regression Models **UG–37**

2.1 The Standard Normal Linear Model	UG–38
2.2 Extensions to Linear Regression: A Framework.....	UG–39
2.3 Heteroscedasticity	UG–42
2.4 Serial Correlation.....	UG–45
2.5 Instrumental Variables and Two-Stage Least Squares	UG–48
2.6 Recursive Least Squares	UG–51
2.7 Systems Estimation.....	UG–53
2.8 Distributed Lags	UG–57
2.9 Information Criteria (IC).....	UG–60
2.10 Restricted Regressions	UG–62
2.11 Doing it Your Way: LINREG with CREATE	UG–64
2.11.1 Recomputing a Covariance Matrix.....	UG–66
2.12 Robust Estimation	UG–67

3. Hypothesis Testing **UG–69**

3.1 Introduction	UG–70
3.2 RATS Instructions.....	UG–71
3.3 Technical Information (Wald Tests)	UG–74
3.4 Technical Information (Lagrange Multiplier Tests).....	UG–75
3.5 Testing for Heteroscedasticity	UG–77
3.6 Serial Correlation.....	UG–81
3.7 Granger-Sims Causality/Exogeneity Tests.....	UG–84
3.8 Structural Stability Tests or Constancy Tests	UG–86

Table of Contents

3.9 Functional Form Tests	UG-93
3.10 Specification Tests	UG-95
3.11 Unit Root Tests	UG-100
3.12 Cointegration Tests	UG-104
3.12.1 CATS Cointegration Analysis Procedures	UG-107
4. Non-Linear Estimation	UG-109
4.1 General Principles and Problems	UG-110
4.2 Derivative-Based Methods	UG-113
4.3 Derivative-Free Methods	UG-117
4.4 Constrained Optimization	UG-122
4.5 Covariance Matrices	UG-125
4.6 Setting Up Your Model: NONLIN and FRML	UG-129
4.7 The START and ONLYIF Options	UG-135
4.8 Non-Linear Least Squares	UG-136
4.9 Method of Moments Estimators (Univariate)	UG-138
4.10 Non-Linear Systems Estimation	UG-142
4.11 More General Maximization	UG-146
4.12 Using FUNCTION	UG-149
4.13 Using FIND	UG-152
4.14 EM Algorithm	UG-153
4.15 Troubleshooting	UG-155
5. Introduction to Forecasting	UG-159
5.1 Introduction	UG-160
5.2 Producing Forecasts	UG-162
5.3 Generating and Using Results	UG-165
5.4 Forecast Performance	UG-169
5.5 Comparing Forecasts	UG-171
5.6 Troubleshooting	UG-172
6. Univariate Forecasting	UG-175
6.1 Time Series Methods	UG-176
6.2 Common Issues	UG-177
6.3 Exponential Smoothing	UG-179
6.4 Box-Jenkins Models	UG-181
6.4.1 A Worked Example	UG-182
6.4.2 The ARIMA Procedures	UG-192
6.4.3 Intervention Modeling	UG-200
6.4.4 ARMAX and RegARIMA Models	UG-203
6.5 Spectral Forecasting	UG-205

7. Vector Autoregressions	UG-207
7.1 Setting Up a VAR	UG-208
7.2 Estimation.....	UG-210
7.3 Data Analysis	UG-211
7.4 Hypothesis Tests	UG-212
7.5 Orthogonalization	UG-216
7.5.1 Cholesky Factorization	UG-217
7.5.2 Structural Decompositions.....	UG-218
7.5.3 Blanchard-Quah/Long-Run Restrictions.....	UG-223
7.5.4 Isolating a Single Shock	UG-226
7.5.5 Sign Restrictions.....	UG-227
7.5.6 Generalized Impulse Responses.....	UG-227
7.5.7 Examples.....	UG-228
7.5.8 Structural Residuals	UG-229
7.6 Using IMPULSE and ERRORS	UG-230
7.7 Historical Decomposition and Related Techniques	UG-239
7.7.1 Counterfactual Simulation	UG-241
7.7.2 Conditional Forecasts	UG-242
7.8 Cointegration and Error Correction Models	UG-247
7.9 Near-VAR's	UG-251
7.10 VAR's for Forecasting: The Bayesian Approach.....	UG-253
7.11 Differences When Using a Prior.....	UG-258
7.12 Choosing a Prior	UG-259
7.13 Selecting a Model.....	UG-261
7.14 Using the Kalman Filter	UG-264
7.14.1 Sequential Estimation	UG-265
7.14.2 Time-Varying Coefficients.....	UG-267
7.14.3 More General Time-Variation.....	UG-270
7.14.4 Dummy Observation Priors	UG-272
8. Simultaneous Equations	UG-273
8.1 Estimation.....	UG-274
8.2 Setting Up a Model for Forecasting.....	UG-279
8.3 Solving Models: Various Topics.....	UG-281
9. ARCH and GARCH Models	UG-285
9.1 ARCH and Related Models	UG-286
9.2 Data Preparation/Preliminaries.....	UG-288
9.3 Standard Univariate Models	UG-290
9.3.1 Exponential Models	UG-291
9.3.2 IGARCH.....	UG-291
9.3.3 Fat-Tailed Distributions	UG-292
9.3.4 Asymmetry.....	UG-293

Table of Contents

9.3.5 The Mean Model.....	UG-293
9.3.6 ARCH-X, GARCH-X Models	UG-294
9.3.7 Estimation	UG-295
9.3.8 Diagnostics	UG-298
9.3.9 Extensions (Using MAXIMIZE)	UG-299
9.3.10 Out-of-Sample Variance Forecasts.....	UG-301
9.4 Multivariate GARCH Models.....	UG-302
9.4.1 VEC form models	UG-303
9.4.2 Restricted Correlation Models (CC, DCC and others)	UG-305
9.4.3 Other Model Options	UG-308
9.4.4 Fetching the Residuals and Covariance Matrices.....	UG-310
9.4.5 The Mean Model.....	UG-312
9.4.6 Diagnostics	UG-315
9.4.7 Hypothesis Tests and Coefficient Interpretation	UG-316
9.4.8 Forecasting	UG-318
9.4.9 Extensions (Multivariate)	UG-319

10. State Space and DSGE Models UG-321

10.1 Dynamic Linear (State Space) Models.....	UG-322
10.2 Filtering and Smoothing	UG-323
10.3 Setting up the System	UG-325
10.4 The DLM Instruction	UG-330
10.5 Simulations.....	UG-332
10.6 Initialization.....	UG-334
10.7 Estimating Parameters	UG-336
10.8 DSGE: Setting Up and Solving Models.....	UG-343
10.8.1 Requirements	UG-345
10.8.2 Non-linear models	UG-346
10.8.3 Unit Roots.....	UG-347
10.8.4 DSGE: Applications	UG-348
10.8.5 DSGE: Estimation.....	UG-352

11. Thresholds, Breaks, Switching UG-355

11.1 Stability Testing and Structural Breaks.....	UG-356
11.2 Rolling Regressions	UG-358
11.3 A General Structure for Analyzing Breaks.....	UG-362
11.4 The Bai-Perron Algorithm	UG-367
11.5 Threshold Autoregression Models.....	UG-369
11.6 Unit Roots and Breaks	UG-376
11.7 Markov Switching Models.....	UG-379
11.7.1 Mixture Models	UG-379
11.7.2 Markov Chains	UG-381

12. Cross Section and Panel Data	UG-393
12.1 Overview	UG-394
12.2 Probit and Logit Models.....	UG-395
12.3 Censored and Truncated Samples	UG-403
12.4 Hazard Models	UG-410
12.5 Panel and Grouped Data.....	UG-413
12.5.1 Forming a Panel Data Set: The Instruction PFORM	UG-418
12.5.2 Panel Data Transformations: the Instruction PANEL	UG-419
12.6 Statistical Methods for Panel Data	UG-420
12.7 Fixed and Random Effects Estimators	UG-425
13. Other Models and Techniques	UG-431
13.1 Non-Parametric Regression and Density Estimation	UG-432
13.2 Linear and Quadratic Programming	UG-435
13.3 Neural Networks.....	UG-439
9. Spectral Analysis	UG-447
14.1 Complex-Valued Expressions.....	UG-448
14.2 Complex Series	UG-449
14.3 Complex-Valued Matrices and FRML's	UG-450
14.4 Fourier Transforms	UG-451
14.5 Preparation and Padding.....	UG-452
14.6 Getting Output.....	UG-454
14.7 Computing Spectra and Cross Spectra.....	UG-456
14.8 Frequency Domain Filtering	UG-460
14.9 Forecasting.....	UG-462
14.10 Fractional Differencing/Integration	UG-464
15. Programming Tools	UG-467
15.1 Program Control and Compile Mode.....	UG-468
15.2 Procedures and Functions	UG-470
15.2.1 Using Existing Procedures	UG-471
15.2.2 Writing Your Own Procedures and Functions	UG-473
15.2.3 Putting It All Together: An Example	UG-478
15.2.4 Regressor List Functions	UG-482
15.3 Repetitive Analyses: Cutting Your Workload	UG-483
15.3.1 Numbered Series and Arrays of Series	UG-483
15.3.2 HASH Aggregator	UG-486
15.3.3 LIST Aggregator	UG-487
15.4 Interactive Procedures	UG-489
15.5 Handling Output	UG-496

Table of Contents

16. Simulations, Bootstrapping, and Monte Carlo Techniques	UG-499
16.1 Bookkeeping.....	UG-500
16.2 Bootstrapping and Simulation Tools	UG-505
16.3 Rejection Method	UG-511
16.4 Standard Posteriors.....	UG-513
16.5 Monte Carlo Integration.....	UG-517
16.5.1 Monte Carlo Integration (VAR).....	UG-519
16.6 Importance Sampling	UG-523
16.7 Gibbs Sampler	UG-528
16.8 Metropolis-Hastings.....	UG-535
16.9 Bootstrapping and Resampling Methods	UG-540
Bibliography	UG-545
Index	UG-555

What's New in Version 11?

Following is an overview of the improvements and new features added to RATS since the Version 10 *User's Guide* and *Reference Manual* were produced.

Version 10 continues the process of adding greatly to the amount of information in the on-line help. A full list of procedures, examples and paper replications is provided in that— the older PDF's in the distribution have been eliminated. Almost all the procedure descriptions now have a standard “reference manual” style description.

Documentation

With Version 11, we've completed the process of putting all the documentation into the on-line help. While we are still providing a more modest level of documentation in PDF form for those who want it, you are likely to find the on-line help to be easier to use since it has hot links to the many examples and procedures.

Projects (11.0)

This is a new file type which allows you to organize a set of calculations into a single easily portable “zip” file.

RGFZ format for graphs (11.0)

The standard format for graphs has been changed to a Zipped JSON format.

RRFZ format for reports (10.1)

Reports can now be saved in a format which allows them to be reloaded into RATS.

Preferences (10.1,11.0)

This now includes options for adjusting the size of the interface font for most (non-text-edit) windows, as well as adjusting all fonts sizes based upon Accessibility/Ease of Use settings from the operating system. (The old Editor tab is now split between Interface and Editor tabs).

Dark/High-Contrast Mode (11.0)

RATS now changes the color schemes for most windows (except graphs) based upon the choices for Dark Mode (in Macintosh and UNIX) and High-Contrast appearance in Windows.

What's New?

Text Edit Window (10.1)

Now includes line numbering, and the Edit menu includes Go To operations to navigate to a specific line number (either direct jump or relative to the current position). Also allows for “colorizing” RATS syntax elements.

CALENDAR scheme with irregular input dates (10.1)

CALENDAR with the **JULIAN** option can take an input series with coded date information to allow dates to be included in graphs, and allow date notation and date functions to refer to entries. There are several new functions to compute the date coding, plus the **JULIAN** option on **DATA** which can pull it off the date information on the file (if it's available). With 10.1, the Data (Other Formats) Wizard can create the required options.

JPEG graphics (10.1)

JPEG (Joint Photographic Expert Group) format is now available as an export format for graphics which can be used on the *File—Save As...* or *File—Export...* menu operations, or as a format in **GSAVE**.

URL's reading (10.1)

For files that read data or text, you can now use URL's (beginning with HTTP, HTTPS, FTP, SFTP or FILE) to access data directly from websites. (FTP, SFTP and FILE were added with 10.1). For instance,

```
cal (m) 1991:1
*
* Import German New Manufacturing Order Index directly
* from the Deutsche Bundesbank
*
open data "https://api.statistiken.bundesbank.de/rest/download/
BBDE1/M.DE.Y.AEA1.A2P300000.F.C.I15.L?format=csv&lang=en"
data (format=cdf,nolabels,org=col,skiplines=10,right=2) $
    1991:1 * neword
```

fetches a series from the Bundesbank's web site and gives it the name **NEWORD**. The end of the range is left open, so this will automatically expand the range based upon what's on the web site.

```
source http://estima.com/procedures/dfunit.src
```

pulls whatever is the current **@DFUNIT** procedure off the Estima web site.

Changes to Existing Instructions

CMOMENT instruction (11.0)

Now defines the `%SCALEV VECTOR` which provides the conversion factors used in converting a raw cross product matrix to correlations.

COPY instruction (11.0)

Now includes the `DFORMAT` option for formatting the dates.

DENSITY instruction (11.0)

Now includes the `SHUFFLE` option for bootstrapping.

ERRORS instruction (10.1, 11.0)

The input `MODEL` can include `FRML`'s if the `FRML`'s are of a simple linear form (10.1)

The `ACCUMULATE` option was added for accumulating results (impulse responses prior to computing the variances) for series which were in the model in differenced form (10.1)

`TITLE` option was added to allow you to add a title over the displayed forecasts (11.0)

FORECAST instruction (11.0)

`TITLE` option was added to allow you to add a title over the displayed forecasts.

HISTORY instruction (11.0)

`TITLE` option was added to allow you to add a title over the displayed data decomposition.

IMPULSE instruction (10.1,11.0)

The `ACCUMULATE` option was added for accumulating results for series which were in the model in differenced form (10.1).

`TITLE` option was added to allow you to add a title over the displayed responses (11.0)

What's New?

LINREG instruction (11.0)

With `CMOM` option, inherits `SHUFFLE` option from corresponding `CMOMENT` instruction.

The `HC1` option was added for adjusting robust error calculation for degrees of freedom.

NLLS instruction (11.0)

The `HC1` option was added for adjusting robust error calculation for degrees of freedom.

NPREG instruction (11.0)

Now includes the `SHUFFLE` option for bootstrapping.

PRINT instruction (11.0)

`TITLE` option was added to allow you to add a title over the displayed data.

SIMULATE instruction (11.0)

`TITLE` option was added to allow you to add a title over the displayed simulated data.

SSTATS instruction (11.0)

Now includes the `SHUFFLE` option for bootstrapping.

STEPS instruction (11.0)

`TITLE` option was added to allow you to add a title over the displayed forecasts.

UFORECAST instruction (11.0)

`TITLE` option was added to allow to add a title over the displayed forecasts.

New Instructions

OS instruction (11.0)

This is a new instruction for running external commands.

PLS instruction (11.0)

This is a new instruction for doing Penalized Least Squares (ridge regression or LASSO)

New Functions

The following functions have been added since the release of 10.0. Descriptions can be found in the Function Wizard (there's a separate category for "Recent Additions") and the Help.

Probability/random functions

%BINOMIALCDF, %BINOMIALK

Others

%IRFFLATTEN

1. Scalars, Matrices, and Functions

This chapter describes the many data types which RATS supports and the functions and operators which can be applied to them. In addition to simple integer and real scalars, RATS supports complex numbers, strings, and arrays of various configurations. In addition, certain objects which are important in the program, such as equations, sets of equations and formulas (MODELS), sets of parameters for non-linear estimation (PARMSETS), and tables of reports and other information (REPORTS) can be defined and manipulated. Scalar and matrix calculations allow you to extend the capabilities of RATS far beyond what is provided by the standard statistical instructions.

The COMPUTE Instruction

Variable Types and Operators

Date Expressions and Arithmetic

Using Functions

Strings and Labels

Arrays and Matrix Algebra

REPORTS

1.1 Working with Scalars

Overview

In the *Introduction*, we look at RATS primarily as a program to input, display, transform and analyze entire data series. This, however, is only part of what RATS can do. Much of the power and flexibility of RATS comes from its ability to work not just with entire data series, but with real- and integer-valued scalars and arrays, as well as other types of information, such as string and label variables.

We'll discuss arrays and matrix manipulations in Section 1.7. First, we'll look at scalar computations. These allow you to compute a wide variety of test statistics, set up loops and conditional instructions, and do difficult transformations which just can't be done with **SET**.

The Instruction **COMPUTE**

COMPUTE is the principal instruction for working with scalar and matrix variables (the main instruction for working with series is **SET**). **COMPUTE** takes the form:

```
compute variable name = expression
```

For example, to store the real value 4.5 in a variable called A, do

```
compute a = 4.5
```

COMPUTE supports the standard numerical operators (page Int–40 in the *Introduction*). The following adds the values of four entries of the series **IP**, saves the result of the expression in the variable **SUM**, and then uses **SET** to create a benchmarked series (**IP95**). Note that you refer to an element of a series (or array) by putting the date or element reference in parentheses immediately following the variable name:

```
compute sum = ip(1995:1)+ip(1995:2)+ip(1995:3)+ip(1995:4)
set ip95 = 400*ip/sum
```

You can evaluate multiple expressions with a single **COMPUTE** instruction by separating the *variable=expression* fields with commas (blank spaces before and after the commas are optional):

```
compute temp1=1.0, temp2=2.0, temp3=3.0
```

Data Types and **COMPUTE**

Every variable you use in RATS has a *data type* which determines what kind of information it can hold. The variables constructed with **DATA** and **SET** instructions are examples of the data type **SERIES**. Once the type of a variable has been set, it cannot be changed (except by deleting it from the symbol table, which should only be done very carefully).

RATS has a wide variety of data types, which we discuss in detail in Section 1.4. We are interested here principally in two of them: **INTEGER** and **REAL**.

Chapter 1: Scalars, Matrices, and Functions

INTEGER	whole numbers, which are used primarily as counters, subscripts and entry numbers or dates. The SET index T is an INTEGER.
REAL	general real values, used for data, sums, test statistics, etc. The variables A and SUM in the examples above are type REAL.

When you first use a variable name in a **COMPUTE**, RATS needs to assign it a data type. You can do this in advance by using the instruction **DECLARE** to create the variable (see Section 1.4), but that is rarely necessary for scalars. Instead, you can usually let RATS determine what type of variable you need based upon the context. In the example above, TEMP1, TEMP2 and TEMP3 will all be REAL variables because the expressions are real-valued (1.0 with a decimal point is real, 1 without it is integer).

You can also do an explicit type assignment in the **COMPUTE** by inserting the desired type in brackets [...].

Instruction:

```
compute a = 35.0
compute b = 125
compute [real] c = 26
compute [integer] d = 99
```

Variable type:

```
REAL, as expression is real.
INTEGER, as expression is integer
REAL
INTEGER
```

RATS will convert an integer expression to a real value if the variable is type REAL. It will *not*, however, convert decimal values to integers automatically—you have to use the **FIX** function for that. For instance,

```
compute frac=slot-fix(slot)
```

makes FRAC equal to the fractional part of the variable SLOT.

If there is a conflict between the type of the variable and the result of the expression, RATS will give you an error message. For example:

```
compute [integer] a = 1000.5
```

will produce the error:

```
SX22. Expected type INTEGER, Got REAL instead
```

Date Expressions and Entry Numbers

Dates, series entry numbers, and array elements are all treated as integers in RATS. In the example below, START and END are INTEGER variables.

```
cal(m) 1960:1
compute start=1965:1, end=2017:2
allocate end
```

START and END are set to the entry numbers associated with the dates 1965:1 and 2017:2 *given the current CALENDAR setting* (entries 61 and 686 in this case). If you change the **CALENDAR** later in the program for some reason, the values of START and END will *not* change, so those entry numbers may no longer refer to the same dates.

Chapter 1: Scalars, Matrices, and Functions

An important advantage of storing dates as integer entry numbers is the ability to do arithmetic with dates. For example, suppose you have set the following **CALENDAR**:

```
calendar(m) 1990:1
```

This maps the date 1990:1 (January, 1990) to entry number 1. The following computations are all legal, and produce the results indicated:

```
compute start = 1990:1          start = 1
compute end = start + 23        end = 24
compute lags = 4
compute startreg = start+lags    startreg = 5
```

This makes it easy to do rolling regressions and similar loops:

```
compute start = 1990:1, end = 2016:12
do i=0,11
  linreg y start+i end+i
  # x1 x2 x3
end
```

This does twelve separate regressions, over the ranges 1990:1 through 2016:12, 1990:2 through 2017:1, and so on.

Displaying Values of Variables and Expressions

The instruction **DISPLAY** allows you to express results in a very flexible way: you can include formatting information and descriptive strings.

```
display "Breusch-Pagan Test" cdstat "P-Value" #.### signif
```

produces output like

```
Breusch-Pagan Test      3.45000 P-Value 0.063
```

Examples

```
linreg longrate
# constant shortrate{0 to 24}
compute akaike = -2.0*%logl+%nreg*2.0
compute schwarz = -2.0*%logl+%nreg*log(%nobs)
display "Akaike Crit. = " akaike "Schwarz Crit. = " schwarz
```

computes Akaike and Schwarz criteria for a distributed lag regression (Section 2.9). This uses “reserved” variables set by the **LINREG** instruction. You can use the *View—Standard Variables Wizard*, or the Help for a complete list of these. See Section 1.3 for more information on reserved variables.

DISPLAY can even do computations on the fly. Note that you must not use any blank spaces between parts of an expression, as otherwise **DISPLAY** will try to parse the separate pieces of the expression as separate items to be displayed. For example:

```
display "Significance level" #.#### (sigcount+1)/(ndraws+1)

display "90% confidence interval" $
    (sampmean+(testmean-%fract95)) (sampmean+(testmean-%fract05))

display "LR Test of Overidentification"
display "Chi-Square(" ### degrees ") =" * teststat $
    "Signif. Level =" #.##### %chisqr(teststat,degrees)
```

This last example will produce output like:

```
LR Test of Overidentification
Chi-Square( 5 ) =          8.56000 Signif. Level = 0.1279546
```

The ### strings are *picture codes*, which show how you want a number to be displayed. A picture clause stays in effect (on a single **DISPLAY** instruction) until you use a new one. Each # character shows one character position, so #.### requests one digit left of the decimal and three digits right. *.### requests three digits right and as many digits left as are needed. * by itself requests a standard format which generally shows five digits right of the decimal.

The Assignment Operator (=)

The arithmetic and logical operators supported by RATS were introduced in Section 1.4.9 of the *Introduction*, and are described in more detail in Appendix A of the *Reference Manual*. Here we will take a special look at the assignment operator.

In an expression, the = operator stores the value of the expression to its right into the variable immediately to its left (the target variable). By embedding = operators within an expression, you can accomplish several tasks at once. Some general points about the assignment operator:

- Unlike the other operators, in the absence of parentheses RATS does the *right-most* = operation first.
- = has the lowest precedence of the operators so, in the absence of parentheses calculations on the right-side will be done first.

Note: a single = symbol is the *assignment* operator. Use == (two equal signs) to *test* for equality.

```
compute y(1)=y(2)=y(3)=0.0
set zhat = (x2(t)=z2/y2) , x2 * v2/y2      X2, V2, Y2, and Z2 are series
```

This second example at entry T first computes $X_2(T)$, then uses that to compute the value of $ZHAT$, which will be $X_2(T) * Y_2(T) / Y_2(T)$.

1.2 Functions

RATS provides over 300 built-in functions which can be used in **COMPUTE** expressions, **SET** instructions, and formulas; these can do everything from simple matrix operations, to generating random draws, to specialized programming operations. See page Int–41 of the *Introduction* for a general introduction to using functions. You can even define your own functions using the **FUNCTION** instruction (see pages UG–149 and UG–470).

We'll provide a quick overview of the various categories of functions and mention a few of the most important examples here. For a complete listing and description of all the functions available in RATS, see the Help. Here are just a few examples using functions in **COMPUTE** instructions:

compute sqx = sqrt(x)	<i>Takes the square root of x</i>
compute loga = log(a)	<i>Takes the natural log of a</i>
compute random = %ran(1.0)	<i>Generates a random number from a Normal distribution with a standard deviation of 1.0.</i>

Note that most function names in RATS begin with the “%” character, but a few of the most common do not—largely for backwards compatibility with RATS code written for older versions of the programs.

With so many functions, it's easy to forget the name or the arguments for one that you want. RATS provides a very helpful wizard for this. You can get to this three ways: the *View—Standard Functions* operation, the *Paste Function* toolbar item, and *Insert Function* on the contextual menu. However you start it, it opens a dialog which lists functions alphabetically, or by category. Clicking on a function name displays the syntax and a short description. If you were just looking for a reminder of the name, you can Cancel the dialog. If you click on OK, you open a second dialog box allowing you to enter arguments for the function and then paste it into the input window. You can use the “Syntax Check” button to verify that your syntax is correct before clicking OK.

The categories are listed below. Note that many functions are in more than one.

Common Mathematical

These include such common functions as LOG for natural logs, ABS for absolute values, and SQRT for taking square roots, as well as %IF for conditional expression evaluation, %ROUND for rounding numbers, and more.

Trigonometric

The standard trig functions are available (COS, SIN, and TAN), as well as the hyperbolic (%COSH and %SINH) and inverse variations.

Matrix

This is, by far, the largest category, and we discuss them in greater detail in Section 1.7. The two most common are INV and TR (inverse and transpose). it includes functions for sorting (%SORT, %SORTC), extracting submatrices (%XDIA, %XSUBMAT), and factoring and decomposing (%DECOMP, %SVDECOMP).

Probability (Density/CDF)

This includes distribution and density functions for a large number of standard distributions, such as Normal, Student's t , chi-squared, F. There are inverse distributions functions for many of these as well. The Normal family, for instance, includes %DENSITY (standard Normal density), %CDF (standard Normal CDF), %INVNORMAL (inverse standard Normal) and %ZTEST (two-tailed standard Normal test).

Random Number Generation (Chapter 16)

This includes %RAN (draws standard Normals), %UNIFORM (uniform random numbers), %RANGAMMA and %RANBETA for random gammas and betas. It also includes “convenience functions” which do draws from particular types of posterior distributions.

String, Label, and Output Formatting (Section 1.6)

Functions like %LEFT, %MID, and %RIGHT extract substrings from larger string variables, while %S and %L are useful for automating repetitive operations involving series.

Date, Entry Number Functions (Section 1.8)

These functions can be very useful in automating handling of dates and sample ranges, and performing transformations. For example, you can use %YEAR, %MONTH, and %DAY to get the year, month, or day, respectively, of a given entry number. The %INDIV and %PERIOD functions are useful for creating panel-data dummy variables.

Financial Functions

RATS provides %ANNUITY and %PAYMENT functions for financial computations.

Complex Numbers and Matrices (Chapter 14)

This is an extensive collection of functions for complex-valued computations, such as complex conjugate (%CONJG), and functions for getting the real and imaginary parts of a complex number (%REAL and %IMAG).

Equation and Regression

These are functions for working with the EQUATION data type, regression lists and tables.

Model

These are functions for working with the MODEL data type.

Polynomial

Functions for working with polynomials.

Utility

These are functions which don't fit neatly into one of the other groups.

Bayesian Methods (Chapter 16)

These are functions useful for implementing Bayesian techniques, including some of the random number generation functions and some numerical integration functions.

1.3 Getting Information from RATS

Variables Defined by RATS Instructions

You can access directly most of the information displayed by the RATS statistical instructions. For example, **LINREG** computes variables such as %RSS (residual sum of squares), %NDF (number of degrees of freedom) and %DURBIN (Durbin-Watson). You can display these values, save them in other variables, or use them in calculations.

All of the reserved variable names (except for the integer subscripts T, I, and J) begin with the % symbol, to help distinguish them from user-defined variables. You cannot define your own variables using any of these reserved names.

You can use the *View—Standard Variables* operation or the *Insert Standard Variable* operation on the contextual menu to see a complete, categorized listing of these variables. The Help has an alphabetized list.

Examples

This computes an *F*-statistic using %RSS and %SEESQ (Standard Error of Estimate Squared). The first **COMPUTE** instruction saves the %RSS value into a variable called RSSRESTR. The second **COMPUTE** takes RSSRESTR and the %RSS and %SEESQ from the second regression to compute the *F*-statistic. (Note that this is just an illustration—RATS has built-in instructions for doing tests like this. See Chapter 3 for details).

```
linreg pcexp
# constant pcaid
compute rssrestr=%rss
```

```
linreg pcexp
# constant pcaid pcinc density
compute fstat=(rssrestr-%rss)/(2*%seesq)
```

Remember that each time you use a statistical instruction (**LINREG**, **STATISTICS**, etc.), the newly-computed values will replace the old ones. We needed to preserve the %RSS from the first regression, so we had to save it somewhere *before* doing the second regression. If we didn't think ahead, we would have had to type the value back into the FSTAT formula.

You *could* just punch the numbers from the output into a calculator. However, RATS does the computations at full precision, which may make a substantial difference if, say, RSSRESTR and the second %RSS agree to four or five significant digits.

Note also that with RATS, you *have* a powerful calculator all set to go. For example:

```
? (1.34759-.33153)*46/(2*.33153)
```

displays the result of the calculation. (?) can be used in place of the full instruction name for **DISPLAY**). This has the added feature of being 100% reproducible, unlike computations typed into an external calculator, which are often impossible to verify without going through the entire process again.

Scalar Values from Series—Using SSTATS

Often, we need to distill a series down into one or more scalar statistics. While there are some specialized instructions for this (**STATISTICS** and **EXTREMUM**, for instance), the instruction generally used for this type of “query” is **SSTATS**. It has several advantages over the other instructions:

1. It can do calculations on several series or expressions in parallel.
2. It can do types of statistics (such as the product) that can’t be done using the standard instructions.
3. It can be applied directly to data as they are computed, while **STATISTICS** and **EXTREMUM** require you to create an actual series to analyze.

By default, **SSTATS** computes the sum of the values of an expression. You use the notation **>>** to save the results into a real-valued variable. So, for instance,

```
sstats nbeg nend log(series)>>logy
```

will make **LOGY** equal to the sum of the logs of **SERIES** over the range from **NBEG** to **NEND**, and

```
sstats 6 36 residcols^2>>olssqr residexp^2>>expsqr $  
      residpar^2>>parsqr
```

will make **OLSSQR** equal to the sum of squares of **RESIDSOLS** over the range 6 to 36. Similarly, **EXPSQR** is created from the squares of **RESIDSEXP**, and **PARSQR** from the squares of **RESIDSPAR**.

```
sstats (mean) 1 100000 (stats<target)>>pvalue
```

STATS<TARGET is 1 if that is true and 0 otherwise, so the mean of this calculation will be the fraction of the 100,000 entries for which **STATS<TARGET**. This is put into the variable **PVALUE**.

```
sstats (min) 2 ntp extract-extract{1}>>minvalue
```

computes the minimum value of the period to period changes of the series **EXTRACT**.

1.4 The RATS Data Types

Much of the computational power of RATS comes from the wealth of data types which allow you to work with data in the most efficient or convenient form. For instance:

- RATS treats data series differently from simple arrays of numbers, by keeping track of defined entries and allowing varying entry ranges.
- RATS differentiates between integer and real computations. This allows programs which do extensive looping to run faster because integer calculations are much more efficient.
- RATS can perform matrix arithmetic, with the matrices treated as individual “variables.”

The Data Types

There are two categories of variable types in RATS: basic and aggregate.

- Most of the *basic* data types store a single value or object, such as a single real or integer number. We also include some special data types (MODEL, PARMSET, and REPORT) in this category. Although these can contain multiple objects, they have more in common with the basic types than with the aggregate types.
- The *aggregate* data types store *several* data objects in a single variable. For example, a RECTANGULAR array of integers, or a SERIES of real numbers.

These are the basic and aggregate data types available in RATS:

Basic Data Types

INTEGER	a single integer number
REAL	a single real number
COMPLEX	a single complex number
LABEL	a single string of up to 16 characters
STRING	a single string of up to 255 characters
EQUATION	a linear equation
MODEL	a group of FRMLs or EQUATIONS
PARMSET	a description of free parameters for non-linear estimation
REPORT	an object containing a table of information.

Aggregate Data Types

VECTOR	a one-dimensional array
RECTANGULAR	a general two-dimensional array
SYMMETRIC	a symmetric two-dimensional array
PACKED	a lower triangular square array (zeros above diagonal)
SERIES	a “time” series
FRML	a coded function of time
HASH	a one-dimensional store of data indexed by strings
LIST	a one-dimensional store of data whose length can be adjusted

Denoting Data Types

Where you need to refer to a basic data type, you need to use three or more letters of the data type name from the previous page. For instance, you can use EQU to abbreviate EQUATION or COMP for COMPLEX. When defining an aggregate data type, you specify both the type of the aggregate variable itself and the type of information contained in it using the construction:

```
type[type[type[...]]]
```

Examples are VECTOR[INT] (read VECTOR of INTEGER), RECT[COMPLEX] (RECTANGULAR of COMPLEX), VECTOR[SYMM[REAL]], (VECTOR of SYMMETRIC of REAL).

Creating Variables

You can create variables of any of the types using the **DECLARE** instruction, at least in the sense that you can define its type to RATS and have space set aside for it. A variable, however, is not of much use until you give it a value.

When you store a value into a variable, such as with a **SET** or **COMPUTE** instruction, RATS is usually able to determine what type is needed, so you do not need an explicit **DECLARE**. In cases where you *need* to use **DECLARE**, or want to use it so that you can keep better track of the variables used in a program, use the following syntax:

```
declare   type   name(s)
```

For example, the first of these declares three real scalar variables, and the second a ten-element VECTOR of reals. (VECTOR is shorthand for VECTOR[REAL]).

```
declare real a b c  
declare vector x(10)
```

and this declares a RECTANGULAR array, where each element is a SERIES.

```
declare rect[series] impulses(nvar,nvar)
```

DECLARE defines *global* variables, which will be accessible anywhere in the remainder of your program or session. You can use **LOCAL** to define local variables visible only in the **PROCEDURE** or **FUNCTION** where they are defined (see page UG-475).

Restrictions on Variable Names

Any variable name must satisfy the following restrictions:

- It must begin with a letter, % or _. % is used mainly for names defined by RATS itself. Our standard practice is to use a prefix of “%%” for variables defined in RATS procedures, with just one % for those from the built-in instructions.
- It must consist only of letters, \$, %, or _.
- It can be up to sixteen characters in length. You can use more than sixteen characters, but only the first sixteen determine which variable you are using.

Chapter 1: Scalars, Matrices, and Functions

Instructions That Set Variables

These are the main instructions used to set and manipulate the data types (other than **SERIES**—many different RATS instructions can create and use series):

COMPUTE	assigns values to scalar and array variables.
DECLARE	declares one or more (global) variables. In many cases, you do not need to declare variables before using them, but doing so at the beginning of a program may make it easier to read and modify.
DIMENSION	sets the dimensions of an array. You can also provide array dimensions when you DECLARE them.
EQUATION	defines a linear equation (an EQUATION variable).
EWISE	sets all the elements of an array as a function of the row and column subscripts. You must DECLARE and dimension an array before using it in an EWISE instruction.
FRML	defines a formula using a function of entry numbers (a FRML variable). This can be a non-linear equation.
GROUP	“groups” formulas (FRMLS) and/or equations (EQUATIONS) into a description of a model (a MODEL variable).
GSET	sets elements of a SERIES with elements other than reals.
LOCAL	declares one or more local variables.
NONLIN	creates or modifies a PARMSET for non-linear estimation.

Input/Output Instructions

DISPLAY	displays various types of variables, strings and expressions.
REPORT	creates a spreadsheet-style report using a sequence of instructions. This can be very helpful in (accurately) extracting results for publication.
WRITE	displays the contents of arrays or scalars of integer, real, complex, label or string information.
INPUT, READ	read numeric or character information into arrays or scalars.
ENTER	reads information from supplementary cards into arrays or scalars of integer, real, complex, label or string information.
MEDIT	displays arrays in “spreadsheet” windows for input or output.
QUERY	gets scalar information from the user through a dialog box.
DBOX	generates more sophisticated user-defined dialog boxes.

Because **INPUT**, **READ**, **ENTER**, **MEDIT** and **QUERY** can process many types of information, you must **DECLARE** any variables which you read using them.

Other Instructions

Many of the instructions in RATS set one or more variables automatically. These are “reserved” variables, meaning that RATS uses a pre-determined set of variable names. For example, most estimation instructions automatically save estimated coefficients in a VECTOR called %BETA.

Many instructions also offer options or parameters you can use to store results into user-defined variables. For example, the RESULTS option on the **IMPULSE** instruction saves computed impulse responses into a RECTANGULAR array of SERIES.

Reading and Writing Data Files

Most of your work with reading and writing data files will involve SERIES variables. We describe the relevant tools, including the Data Wizards and the **DATA** and **COPY** instructions, in the *Introduction*.

To read data into other types of variables (arrays, scalars, strings, etc.), you can use the **READ** and **INPUT** instructions. Both of these normally accept input from the input window (i.e. information you type directly into your program), but you can use the UNIT and FORMAT options to read data from an external file instead. **READ** accepts data in quite a few formats, both text and spreadsheet.

For writing these types of variables, you can use **WRITE** or **DISPLAY**. These normally send output to the output window, but you can use the UNIT option to re-direct the output to a file. **WRITE** also offers a FORMAT option which supports a wide variety of formats, including Excel, text, HTML, and TeX. **DISPLAY** has more flexibility for showing the information (allowing you to easily add labels), but can only write text files.

You can also export arrays by using **MEDIT** with the WINDOW option to display the array and then doing *File–Export...* More generally, you can use **REPORT** instructions (Section 1.9) to generate a report (which can include a mix of data types) and either write the report directly to a file, or display it in a window which can be written to a file using *File–Export...*

Listing Variables

View–User Variables... displays a dialog box listing all the variables *you* have defined. You can also bring that up using *Insert User Variable* from the contextual menu. You can preview the contents of a variable, and paste the variable name into the text window (helpful when constructing expressions). The *View–Standard Variables...* operation is similar, but displays reserved variables defined by RATS—the contextual menu equivalent is *Insert Standard Variable*. The *View–All Symbols* operation displays a list of *all* of the variables in memory. You can double-click on most types of arrays to view the array in a window (equivalent to doing an **MEDIT** instruction).

1.4.1 Basic Data Types

INTEGER stores single whole numbers. These can range in value up to approximately (plus and minus) two billion. The **SET** and **FRML** subscript **T** and the **EWISE** subscripts **I** and **J** are **INTEGERS**. **DO** loop index variables also must be **INTEGERS**. Dates are also stored internally as, and can be referenced using, integer entry numbers. **INTEGER** variables are usually set using **COMPUTE**:

```
compute start=1947:1, end=2018:2
compute k = %nobs-10
```

REAL stores a single floating point number. Reals are stored as double-precision, and on PCs, Macintoshes and most workstations, a real value can be as small (in absolute value) as 10^{-308} and as large as 10^{308} . Real variables are important if you go beyond the simple RATS commands to do your own programming. Virtually all calculations of test statistics, data values, sums, etc. are done with real-valued data. **REALS** are usually set with **COMPUTE**:

```
compute fstat = (rssr-rssu)/(q*%seesq)
compute [real] c = 10
compute d = 11.5
```

COMPLEX stores two floating point values, one each for the real and imaginary parts of a complex number. They usually are set by **COMPUTE**, though the **COMPLEX** type is relatively unimportant itself. You will usually work with complex numbers through complex series. Most of the uses of complex-valued data is in frequency domain analysis—see Chapter 14.

A **COMPLEX** is displayed as a pair of real numbers separated by a comma and enclosed in parentheses. For instance: $(1.00343, 3.53455)$, where the first number is the real part and the second is the imaginary part.

LABEL is a character variable up to 16 characters long, which is, of course, the length of RATS variable and series names. You can set a **LABEL** with a **COMPUTE** instruction. **INPUT**, **READ** and **ENTER** are also common.

STRING is a character variable which can have up to 255 characters. They are often used for graph headers or file names. You can set them with:

- **COMPUTE**
- **DISPLAY (STORE=STRING variable)**
- **INPUT**, **READ**, **ENTER**, **DBOX** or **QUERY**. You must **DECLARE** a variable to be type **STRING** before you can do this.

EQUATION is a description of a linear relationship. These are used in computing forecasts and impulse response functions. Equation coefficients can be estimated using instructions like **LINREG** and **SUR**. You can create **EQUATIONS** using the **EQUATION** instruction, or with the **DEFINE** options of instructions such as **LINREG**, **BOXJENK** and **AR1**. You can edit or display an existing equation using **MODIFY**.

An **EQUATION** can be defined and referenced by name, or by its sequence number.

MODEL is a collection of **FRMLS**, **EQUATIONS** and related information, used for systems simulation. The **GROUP** instruction defines a **MODEL** by specifying which formulas and equations to include in the model. **MODELS** can also be created as a description of a **VAR** with **SYSTEM** and its subcommands.

MODELS can be combined using the “+” operator.

PARMSET is a specialized data type which consists of a list of parameters for non-linear estimation and constraints on them. **PARMSETS** are constructed and maintained using the instruction **NONLIN**. **PARMSETS** can be combined using the standard “+” operator. See page UG–129 for more on the creation and use of these.

REPORT is another special data type used to hold tables of information. These can be generated by the user, using **REPORT** instructions, and can also generated automatically by many **RATS** instructions. See Section 1.9.

1.4.2 The Aggregate Data Types

You can make an aggregate of any data type. However, the individual elements of an aggregate must be of the same data type—RATS does not allow you to define general structures containing a mix of types. Note that there are no aggregators with three or more dimensions—you can achieve the same thing by nesting one- and two-dimensional types, such as `VECT[RECT]`, which will have, in effect, three dimensions, one on the outer (the `VECTOR`) and two on the inner (the `RECTANGULAR`).

With any of these, the aggregator name by itself means elements that are type `REAL`.

VECTOR	One-dimensional arrays. You refer to the elements of a <code>VECTOR</code> using the notation <code>name(subscript)</code> . Subscripts start at 1. <code>INDEX</code> can be used as shorthand for <code>VECT[INTEGER]</code> and <code>CVECTOR</code> for <code>VECT[COMPLEX]</code> .
RECTANGULAR	General two-dimensional arrays. You refer to their entries with the notation <code>name(row, column)</code> . The subscripts start at 1 in each dimension. <code>RECTANGULAR</code> arrays are stored by columns, that is, in an $M \times N$ array, the first M elements are the first column, the next M are the second column, etc. (This matters only for a few specialized operations). <code>IMATRIX</code> can be used as shorthand for <code>RECT[INTEGER]</code> and <code>CMATRIX</code> for <code>RECT[COMPLEX]</code> .
SYMMETRIC	Symmetric two-dimensional arrays. RATS stores only the elements located on and below the diagonal, which saves approximately half the space. RATS uses more efficient routines for computing inverses, determinants and eigenvalues than are available for <code>RECTANGULAR</code> arrays. You refer to their entries by <code>name(row, column)</code> . The subscripts start at 1 in each dimension. <code>SYMMETRIC</code> arrays are stored by the rows of the lower triangle. The subscripts of the elements are, in order, (1,1), (2,1), (2,2), (3,1),....
PACKED	Packed lower triangular two-dimensional arrays. This is similar to <code>SYMMETRIC</code> except the above diagonal is zero, not the mirror image of the lower triangle.
SERIES	<code>SERIES</code> are one-dimensional arrays of objects, but they have more structure than a simple array (<code>VECTOR</code>): their entries do not have to run from 1 to N and they have “defined” and “undefined” ranges.

FRML

A **FRML** is a coded description of a (possibly) non-linear relationship. It is a function of the time subscript **T**. **FRML**s are used for non-linear estimation and general model simulation. You can define **FRML**'s which evaluate to types other than **REAL**, but you have to declare their type before using a **FRML** to define them.

See Section 4.6 for more on **FRML**'s.

HASH

A **HASH** is similar to a **VECTOR**, except that the elements are indexed by strings (called “keys”) rather than by numbers. It can't be used directly in calculations the way a **VECTOR** can, but is instead designed to make it simpler to organize information.

See Section 15.3.2 for more on **HASH**'s.

LIST

A **LIST** is similar to a **VECTOR**, except that you can change the number of elements easily. “Adding” to a **LIST** appends an element or set of elements to it. As with the **HASH**, its main use is for organizing information when you may not know in advance how many items need to be saved.

See Section 15.3.3 for more on **LIST**'s.

You will usually have to **DECLARE** arrays to make sure that you get the output in the form you want. In a matrix expression (page UG–25), the specific type of result is usually ambiguous: **VECTORS** and **SYMMETRICS**, after all, are really special cases of two-dimensional arrays. We use the special types for efficiency.

1.4.3 Using Arrays

Real-valued arrays are useful directly in matrix computations (page UG–25). Arrays of all types are useful for “bookkeeping” when we are examining a possibly varying number of items. The ability to construct arrays of **SERIES** is particularly helpful.

For example, several of the example programs and procedures included with RATS that deal with Vector Autoregressions (such as **IMPULSES.RPF**) use **VECTORS** and **RECTANGULARS** of **SERIES** to organize the calculations for large numbers of series.

You can specify array dimensions either as part of the **DECLARE** instruction, or with a separate **DIMENSION** instruction. If you have an array of arrays, you must dimension the element arrays individually.

Arrays Within Lists

Whenever RATS expects a “list” of objects of some type, it will accept any array of that type as well. This is very useful for writing general procedures which can handle problems for varying sizes.

For instance, the instruction **KFSET** needs a list of **SYMMETRIC** arrays. Instead of

```
kfset xxx1 xxx2 xxx3
```

when you need three arrays and

```
kfset xxx1 xxx2 xxx3 xxx4
```

when you need four, you can use

```
compute nvar=4 (or 3)  
declare vector[symmetric] xxx(nvar)  
kfset xxx
```

The arrays will be **XXX(1), . . . , XXX(NVAR)**.

We use this extensively for supplementary cards within procedures. The entries on a supplementary card for **LINREG**, for instance, can be put in a **VECTOR[INTEGER]** to be recalled as a whole. RATS also provides a set of functions for manipulation such “regression lists”.

For instance, if you have set up a **VECTOR[INTEGER]** called **REGLIST**, the instruction

```
linreg y / resids  
# reglist
```

regresses **Y** on the regressors listed in **REGLIST** and

```
linreg resids  
# reglist resids{1 to lag}
```

uses the **REGLIST** regressors plus lags of **RESIDS**.

1.5 Data Series (The SERIES Type)

Most of the RATS commands use data series. Because of the special nature of time series data, we have distinguished between a data series and a simple one-dimensional array of numbers.

When you use a new name, say in a **DATA** or **SET** instruction, you are creating a variable of type **SERIES**. Internally, a **SERIES** consists of some bookkeeping information, such as the defined length and a pointer to the data.

When you use a series, you are actually specifying not just the series itself, but also the range of data to use. This may not always be apparent, due to the use of default ranges and **SMPL** settings.

Series and the Workspace

When you do an **ALLOCATE** instruction, or do some other instruction which first creates a data series, you are setting the number of entries in the *standard workspace*. This is the length that RATS assumes you mean when you create further data series where the number of elements isn't clear—most commonly when you do a **SET** without the *start* and *end* parameters. The standard workspace length isn't a limit on how large a data series can get. For instance, if your program starts with

```
cal(q) 1960:1  
allocate 2017:4
```

you can create a series with 10000 data points using

```
set sims 1 10000 = %ran(1.0)  
stats sims
```

you just have to give an explicit range on the **SET**. Instructions which are driven by (potentially) complicated expressions such as **SET** and **SSTATS** will assume the default workspace length unless you override it. Instructions which take only series themselves as inputs (**STATISTICS** and **LINREG**, for instance) work off the defined length of the series they are given.

Series Numbers vs. Names

Every data series has a *sequence number* or *handle* as well as a name. This is a remnant of early versions of RATS, but we have retained it because series handles can still be useful in some cases. In general, though, you will probably find it easier to use arrays of series or the **%S** function, described on the next page, for handling these kinds of repetitive tasks. Nonetheless, it is helpful to understand how series handles work.

The first series created in a RATS session or program is series number 1, the second is number 2, etc. Suppose your program begins with these lines:

Chapter 1: Scalars, Matrices, and Functions

```
calendar(q) 1980:1
data(format=rats) 1980:1 2017:4 gdp wages m1 price
```

GDP is series 1, WAGES series 2, M1 series 3 and PRICE series 4.

Anywhere RATS expects a series name, you can also use an integer value or **INTEGER** variable. When RATS expects a set of series, you can supply a list of integers, a set of **INTEGER** variables, or even **VECTORS** of **INTEGERS**. For example:

```
print / 1 2
```

prints series 1 and 2 (GDP and WAGES using the above example). If you execute the instruction

```
print / gdp to price
```

RATS interprets “GDP TO PRICE” as all series from 1 (GDP) to 4 (PRICE). Notice that the series numbers are based purely upon the order in which the series were created.

If you want to check the number associated with a series name, just use **COMPUTE** to set an integer equal to the series handle, then **DISPLAY** it:

```
compute sernum = gdp
?sernum
```

Be Careful With Series Numbers!

Using series numbers in places where RATS always expects a series type variable is easy. For example, *series* on a **STATS** instruction expects a **SERIES** type so

```
do i=1,3
  statistics i
end do
```

produces statistics for series numbers 1, 2, and 3 as you would expect.

The same is true for any *series* parameter, such as on **SET** instructions. For example, if X is series 1 and Y is series 2, the following instructions are identical:

```
set y = 100 + .3*x
set 2 = 100 + .3*x
```

However, the same is *not* true on the *right* hand side of a **SET**, **FRML**, or **COMPUTE** instruction, where RATS accepts a variety of variable and expression types. So:

```
set 2 = 1
```

simply sets entries of series 2 equal to the constant value 1, *not* the values of series number 1. If you want RATS to treat an integer expression as a series number in such cases, you need to indicate this explicitly, using either lag notation (**SET** or **FRML**) or the type modifier **[SERIES]** (on **SET**, **FRML**, **COMPUTE**, **DISPLAY**, etc.). For example:

```
set y = 1{0}
set y = ([series]1)
```

To refer to a specific entry of a series referenced by number, use the type modifier in parentheses, followed immediately by the entry number in another set of parens:

```
compute test= ([series]1) (3)
```

Arrays of Series and the %S Function

RATS also allows you to set up arrays of series, which can be very helpful for automating tasks involving many series, or for simply keeping track of a large number of series, particularly when those series are related to each other in some manner. As a simple example, suppose you want to generate series containing the natural logs of a list of other series. You can do something like:

```
declare vector[series] loggdp(10)  
do for s = usdgd to ukgd  
    set loggdp(%do for pass) = log(s{0})  
end do for
```

The %S function provides another handy way to refer to or create series. You supply a LABEL variable or expression as the argument for the function. If a series with that label exists, the function returns the number of that series. Otherwise, it creates a new series, using the supplied label as the series name. So the above example could also be handled using:

```
do for serlist = usdgd to ukgd  
    set %s("log"+%l(serlist)) = log(serlist{0})  
end do
```

The %L function returns the label of series, so we use it to get the label of the current GDP series, and prefix that with the text “LOG” to create the name for our new series. For example, the first series will be LOGUSGDP. Because the SERLIST index variable only contains the number of the current series, we use the lag notation inside the LOG function so that we get the log of the associated series, not the log of that integer value.

See page UG–484 for more on using these features for automating repetitive tasks.

More on Creating Series

SERIES variables are almost always set from context (or using %S as shown above), but there are other ways to create them:

- The *series* parameter on **ALLOCATE** creates a block of series numbered 1 to *series* (see page UG–483 for examples). The first series created after the **ALLOCATE** is *series*+1. Note that you may see programs which (incorrectly) use a **ALLOCATE** instruction with two “date” parameters. While that almost certainly isn’t the correct use of the first parameter in **ALLOCATE**, it generally will create no problems as all it does is to set aside some “numbered” series which will never get used in the remainder of the program.
- The **SCRATCH** and **CLEAR** instructions both create new series. **SCRATCH** creates a block of consecutively numbered series. **CLEAR** creates series from a list of names or numbers.
- **DECLARE SERIES** *name(s)* can also be used to create series. Unlike the other methods listed above, the declared series are not automatically defined over any range (they have length zero). You won’t be able to refer to a particular element of the series, or set an element using **COMPUTE**, until the series has been defined over the appropriate range, such as by setting it with a **SET** or **CLEAR** instruction.

1.6 Strings and Labels

String and label variables can be used for graph headers, for labeling output, or for picture codes. Anywhere that RATS expects an *undelimited* string (such as for the file name on an **OPEN** instruction or a picture code on **DISPLAY**), you can use a **LABEL** or **STRING** variable prefixed with the **&** symbol. For instance, **OPEN DATA &FNAME** will open the file whose name is stored in **FNAME**.

You can read strings or labels from an external file using the instruction **READ**. You can create string variables using the instruction **DISPLAY** with the **STORE** option. You can also create them with **COMPUTE**.

Operators

The **+** operator, when applied to two strings, concatenates them. If there is a space at the end of the first string or the beginning of the second, it will be retained in the combined string. If not, no blanks will be inserted and the two strings will run together. If you need spacing, you may need to add it, for instance, with **s1+" "+s2**. If there is more than one blank between the two, the extras will be removed, leaving just one blank as spacing.

You can also use the **+** operator to add (the character representation of) an integer to the *end* of a string. For instance **"ABC"+I** will produce the string **ABC13** if **I=13**.

String Functions

Below are most of the functions available for working with strings and labels.

%LEFT (S, n)	Returns the leftmost <i>n</i> characters from the string <i>S</i> .
%RIGHT (S, n)	Returns the rightmost <i>n</i> characters from the string <i>S</i> .
%MID (S, m, n)	Returns <i>n</i> characters from the middle of string <i>S</i> .
%VALUE (S)	Returns the value of a numeric string as a real number.
%STRING (n)	Returns a string with the integer <i>n</i> in character form.
%STRVAL (x, f)	Returns a real value as a formatted string.
%STRLEN (S)	Returns the length of a string.
%STRLOWER (S)	Returns a string in all lower-case characters
%STRUPPER (S)	Returns a string in all upper-case characters
%STRREP (S, n)	Returns a string which repeats the substring <i>S</i> <i>n</i> times.
%DATELABEL (t)	Returns date label of entry, for the current CALENDAR .
%DATEANDTIME ()	Returns the current date and time.
%L (S)	Returns the label for series <i>S</i>
%LABEL (v)	Returns the label of the variable, array or series <i>v</i> .
%S (L)	Returns a reference for a series with label <i>L</i> .
%STRCMP (S1, S2)	Compares two strings.
%STRCMPNC (S1, S2)	Compares two strings without regard to case.
%STRESCAPE (S, C, E)	Returns a string with certain characters “escaped”.
%STRFIND (S, F)	Returns a string for a substring.
%STRMATCH (S, P)	Tests a string for a match with a pattern.
%STRTRIM (S)	Returns a string with spaces removed from start and end.

Chapter 1: Scalars, Matrices, and Functions

Examples

```
compute header="Transformations of "+%1(series)
```

creates in HEADER a string such as Transformations of FYGM3

```
compute dlabel="Using data from "+%datelabel(start)+$  
" to "+%datelabel(end)
```

creates in DLABEL a string describing the range from START to END using date strings.

```
disp(store=keylabel) %1(series)+"\\ " *.### value
```

creates in KEYLABEL a string such as GDP\\7.391. (The \\ is used in several situations to show a line break).

```
declare string fname  
compute fname=%1(i)+".XLS"  
open copy &fname  
copy(format=xls,org=col,dates) / i
```

creates an Excel spreadsheet whose name is formed by adding “.XLS” to the name of series I, and which has the data for series I. Note that if you don’t put the & in front of FNAME, the file name is literally “FNAME”, not the string that FNAME holds.

1.7 Matrix Calculations

The following pages cover the instructions and some of the functions available for evaluating matrix expressions. Most of these apply only to real-valued arrays.

Accessible Arrays

Among the variables which RATS makes available to you are several arrays. The most important of these are:

Variable	Type	Description
%BETA	VECTOR	The vector of coefficients from the most recent regression.
%STDERRS	VECTOR	The vector of standard errors from the most recent regression.
%TSTATS	VECTOR	The vector of t -statistics from the most recent regression.
%XX	SYMMETRIC	The estimated covariance matrix for most instructions, or the $(\mathbf{X}'\mathbf{X})^{-1}$ matrix for single equation regressions without ROBUSTERRORS.
%CMOM	SYMMETRIC	The cross-moment matrix of regressors, defined by CMOMENT or MCOV .

There is a major difference between the scalar variables defined by RATS instructions, such as %RSS, and the arrays: *you are permitted to change the internal values of these arrays*. You cannot do this with the scalars. For instance, the command **COMPUTE %XX(2,1)=5.0** changes element (2,1) of the program's own internal copy of the %XX matrix, which is used by the hypothesis testing instructions.

Matrix Instructions

You can create matrices in a variety of ways. You *could* set individual elements of an array with a whole set of **COMPUTE** instructions. However, RATS also offers several instructions (including **COMPUTE**) which can create and set matrices as a whole. Use these wherever possible, because they are much faster than brute force methods.

These are instructions which exist primarily to do matrix calculations. There are many others (such as **LINREG**) that create matrices as a side-effect of another calculation.

COMPUTE	can set entire matrices using algebraic formulas and special matrix functions, as in COMPUTE XXMATRIX = TR(X)*X . Most of this section shows how to use COMPUTE for matrices.
EWISE	sets matrices using element-wise functions of the subscripts I and J . For instance, EWISE SIGMA(I,J)=.9^(I-J)
CMOM	computes a cross product or correlation matrix of a set of variables.

Chapter 1: Scalars, Matrices, and Functions

MCOV	computes a long-run variance matrix for a set of variables.
EIGEN	computes eigen decompositions of $N \times N$ arrays.
QZ	computes a generalized Schur decomposition of a pair of matrices.
MAKE	creates an array from a collection of data series.
FMATRIX	creates a “filter matrix”: a matrix with repeating patterns which shift over one column per row.

In-Line Matrices

You can create an array by listing its entries explicitly in an expression. For example:

```
compute [vect] a = || 1.0, 2.0, 3.0 ||
```

defines A as a 3-element VECTOR and

```
declare symmetric s
compute s = || c11 | c21,c22 | c31,c32,c33 ||
```

defines S as a 3×3 SYMMETRIC. The `||...||` delimits the entire array, `|` separates rows of the array, and the commas separate elements within a row. You can use either constants or expressions for the elements.

Variable Types

You can mix arrays, series, and scalar variables in matrix expressions, subject to:

Arrays	New arrays created in an expression will be of type RECTANGULAR by default, unless you DECLARE the array prior to using it or include an explicit type assignment before the matrix name. (See the examples above). You do not need to dimension arrays set in the course of the expression, except when you use functions like %RAN, %UNIFORM, %RANGAMMA, %RANCHISQR, %RANBETA, %CONST and %MSCALAR, which fill an entire array with values.
Series	COMPUTE treats data series as column vectors. It is a good idea to use a SMPL instruction to indicate the range to use. If you don't, RATS uses the defined range of the series. You cannot set a series directly using matrix operations.
Scalars	Several of the matrix functions return scalar values, and you can use $A * x$ or $x * A$ to multiply all elements of an array A by a scalar x, A / x to divide all elements of A by x, $A + x$ to add x to every element and $A - x$ to subtract x from every element. If you have a calculation which returns a 1×1 matrix and you want to treat the result as a scalar, use the %SCALAR function.

Operators

COMPUTE supports the following operators in matrix expressions. **A** and **B** can be array variables or expressions of the proper form.

A*B	Multiplication. A or B may be scalar (if both are, the result is scalar).
A+B, A-B	Addition and subtraction. If A and B are matrices, they must have compatible dimensions. B can be scalar; if it is, it is added to or subtracted from each element of A .
A/B	Division. B must be a scalar. A can be an array or scalar. If it is an array, each element of A is divided by B .
A^n or A**n	Exponentiation. If A is an array, it must be square and n must be an integer (positive or negative). This does repeated multiplications of A (or its inverse) with itself.
A.*B, A./B	Elementwise multiplication and division, respectively. A and B must both be arrays of the same dimensions.
A.^x	Elementwise exponentiation, taking each element of A to the x power. x must be real.
A~B	Does a horizontal concatenation of two arrays (columns of A to the left of columns of B). A and B must have the same number of rows.
A~~B	Does vertical concatenation of two arrays (rows of A above rows of B). A and B must have the same number of columns.
A~\B	Diagonal concatenation of two arrays. The result is block diagonal with A in the top left and B in the bottom right. There are no restrictions on the dimensions.
A=B	Assignment. A must be an array, scalar variable or series or array element. • If A is an array, you do not need to dimension it (except when you use the functions %RAN, %UNIFORM, %RANGAMMA, %RANCHISQR, %RANBETA, %CONST or %MSCALAR, all of which take scalars as arguments). The result B may not be scalar. • If A is scalar, you do not need to declare it; a new name will be created as type REAL. B may not be an array: use the %SCALAR function to turn an array B into a scalar.

Matrix Functions

There are well over 100 matrix functions. A full list is provided in the Help. Some of these are “convenience” functions for calculations that can be done relatively easily using loops; others (such as inversions and decompositions) would require hundreds or even thousands of individual calculations to duplicate. Because these are done using the high-level language in which RATS is written (C++) they will be faster (often many times faster) than the equivalent computation done using scalar calculations in RATS. However, until you get into calculations with very large matrices (such as 500 x 500 or larger) or a large number of simulations involving large matrices (say 100 x 100), you shouldn’t worry too much about speed if it comes at the cost of requiring hard to understand and hard to correct code.

It’s important to note that many calculations which would be done using matrix calculations in “math packages” are done much more efficiently (and much more clearly) in RATS using series-based instructions. For instance, the following is an example of the setting up a matrix to estimate an autoregression, done in one of the math packages:

```
Y=y(:,p:t);
for i=1:p-1
    Y=[Y; y(:,p-i:t-i)];
end;
X=[ones(1,t-p); Y(:,1:t-p)];
Y=Y(:,2:t-p+1);
```

which involves making copies of shifted data and concatenating matrices to create the “X” matrix. Aside from being clumsy, it’s also very easy to make errors with this if you try to do the regression using only a subset of the original data. With RATS, you just do this with

```
linreg y
# constant y{1 to p}
```

and can instantly change the range of the regression by just putting in *start* and *end* parameters. Instructions like **SET** and **SSTATS** can often take the place of entire blocks of element-by-element calculations.

If you truly need to create a matrix out of data series information, use the **MAKE** instruction. This is the RATS equivalent to the block of six math package instructions:

```
make x
# constant y{1 to p}
```

If you need to draw random numbers, pick the function that best matches what you need. Use %RANCHISQR, for instance, if you need random chi-squares rather than drawing normals and computing their sum of squares. Use %RANMVKRON to draw from a $N(0, \Sigma \otimes Z)$ rather than drawing a vector of independent normals and multiplying it by a Kroneker product.

Examples of Matrix Calculations

If $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} 1 & -2 \\ -4 & 8 \end{bmatrix}$

```
compute horiz=a~b
compute vert =a~~b
compute diag =a~\b
compute emult=a.*b
compute add =a+6.0
```

return:

$$\text{HORIZ} = \begin{bmatrix} 1 & 2 & 1 & -2 \\ 3 & 4 & -4 & 8 \end{bmatrix}, \text{VERT} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 1 & -2 \\ -4 & 8 \end{bmatrix}, \text{DIAG} = \begin{bmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 1 & -2 \\ 0 & 0 & -4 & 8 \end{bmatrix}$$

$$\text{EMULT} = \begin{bmatrix} 1 & -4 \\ -12 & 32 \end{bmatrix}, \text{ADD} = \begin{bmatrix} 7 & 8 \\ 9 & 10 \end{bmatrix}$$

```
compute logdet = log(%det(vcv))
```

LOGDET is equal to the log of the determinant of VCV.

```
compute convcrit = %dot(sigmainv,s)
```

CONVCRT is the dot product of SIGMAINV with S.

```
declare rect ddiag
compute ddiag=%diag(||4.5,5.6,6.7||)
```

produces

$$\text{DDIAG} = \begin{bmatrix} 4.5 & 0.0 & 0.0 \\ 0.0 & 5.6 & 0.0 \\ 0.0 & 0.0 & 6.7 \end{bmatrix}$$

```
make x
# constant x1 x2 x3
make y
# y
compute beta=inv(tr(x)*x)*tr(x)*y
```

calculates the coefficient vector for a regression using matrix operations. (Again, not recommended except for illustration).

1.8 Calendar/Date Functions

RATS includes a perpetual calendar which is valid for dates from 1600 on. There are quite a few functions which allow you to make use of this. The first group of these analyze the date information for entries (E argument) in the current **CALENDAR** setting. A second group looks directly at year:month:day (Y,M,D arguments). Some of the functions take a day of the week as an argument, or return day of the week information. This is coded as 1=Monday to 7=Sunday.

%YEAR (E)	Year of entry
%MONTH (E)	Month of entry
%DAY (E)	Day of month of entry
%TODAY ()	Entry for today's date
%WEEKDAY (E)	Day of week of entry
%PERIOD (E)	Period number (panel data) or period within year
%JULIAN (E)	Days from Jan 1, 0001 to entry
%TRADEDAY (E, DOW)	Trading days for day of week in an entry
%DAYCOUNT (E)	Number of days in an entry
%DOW (Y, M, D)	Day of week of Y:M:D.
%EASTER (Y)	Dates after March 22 for (Western) Easter in year Y.
%FLOATINGDATE (Y, M, DOW, D)	Day of month for floating holiday which is a particular (D) day of the week within a month.
%CLOSESTDATE (Y, M, D, DOW)	The observance day of the month which is the closest DOW to Y:M:D.
%CLOSESTWEEKDAY (Y, M, D)	The observance day of the month which is the closest weekday (Monday-Friday) for Y:M:D.
%YMDAYCOUNT (Y, M)	Number of days in the month Y:M.
%YMDDOW (Y, M, D)	Day of the week of Y:M:D.
%JULIANFROMYMD (Y, M, D)	Days from Jan 1, 0001 to Y:M:D.
%JULIANFROMCODED (X, CODE)	Days from Jan 1, 0001 to a numerically coded date.

Examples

```
cal(d) 1990:1:2
set wednesday = %weekday(t)==3
```

creates a dummy variable for the Wednesdays (weekday #3) in a daily data set.

```
cal(m) 1979:1
set d11 = %if(%month(t)==11, 30-%floatingdate(%year(t), 11, 4, 4), 0.0)
```

creates a series which, for the Novembers in a monthly data set (where %month(t)==11), has the number of days in November after U.S. Thanksgiving, which falls on the 4th (final argument in %FLOATINGDATE) Thursday (3rd argument) in November (2nd argument).

1.9 Creating Reports

One of the advantages of using RATS is that you can do all the calculations within the program, allowing you to reproduce results exactly, and eliminating the error-prone transfer of numbers from computer output to paper or a calculator. By using the instruction **REPORT**, you can take this a step farther and create full tables ready, or nearly ready, to be copied directly into a word processor. This again has the advantage of allowing you to avoid typing numbers.

You can do much of what **REPORT** can do by an ingenious use of the **DISPLAY** instruction, but **REPORT** is better designed to create the types of tables commonly used in reporting statistical results.

Much of the output generated by RATS itself, including tables of regression and estimation results, are also saved automatically as **REPORT** objects, which you can access (and then copy-and-paste or export) using the *Window* menu.

Displaying and Recalling Reports

Every report you create using **REPORT**, or by using a **WINDOW** option on instructions like **PRINT**, is stored internally and can be redisplayed in a window using the *Window—Report Windows* operation. Reports generated automatically by RATS, such as the tables of results produced by many instructions, are also accessible this way. Many of those instructions offer **TITLE** options for labelling output, and RATS will use any title you supply as the name for the report in the *Report Windows* list.

Exporting Reports

You can get information from a report into a file or another program in several ways:

- For reports you create using the **REPORT** instruction, you can use the **UNIT** and **FORMAT** options along with the **ACTION=SHOW** option to write the results directly to a file in any of several formats, including Excel, TeX, and HTML.
- You can use *Edit-Copy* to copy the report (either from the Output Window or from a Report Window) and paste it into another application. (Report Windows also have *Edit-Copy as TeX* which copies a TeX table into the clipboard). If you copy from a Report Window into a spreadsheet program, RATS will copy data at full numeric precision. Otherwise, (for text or TeX), the data will be copied as it appears on the screen in RATS. **REPORT** allows you to control the number of decimals used in the output.
- For a report displayed in a Report Window, you can use *File-Export...* to export the report to a file, with the same choices of format.

The latter two are also available for any of the output that RATS itself generates. As noted above, just use *Window—Report Windows* to display the desired report, and then either copy and paste or use *File-Export...* to get the information out.

Chapter 1: Scalars, Matrices, and Functions

Using REPORT

A report is designed as a table anchored at row 1 and column 1. This is similar to a spreadsheet, except both are numbered, rather than columns being lettered. And, like a spreadsheet, it's effectively unlimited in size in each direction. You fill in entries by specifying the row and column at which information is to be placed. This can be done in whatever order you find convenient.

You will need to use at least three **REPORT** instructions to create a report:

1. **REPORT (ACTION=DEFINE)** initializes the report.
2. **REPORT (ACTION=MODIFY)** adds information to the report. (You actually don't need to indicate ACTION=MODIFY, since that's the default).
3. **REPORT (ACTION=SHOW)** indicates that the report is done.

You can also use ACTION=FORMAT to format the numerical data, and ACTION=SORT to sort the data based on the values in one of the columns.

The `USE=reportname` option allows you to work with multiple reports. For example, this allows you to add information to two different reports inside a loop, and then display both after the loop is completed.

ACTION=DEFINE

There are two options you can use with ACTION=DEFINE: HLABELS defines header labels, and VLABELS defines row labels. Both are vectors of strings. While the header labels are considered separate from the data, the VLABELS are just the first column of data, and can be altered later by using ACTION=MODIFY on column one.

ACTION=MODIFY

ACTION=MODIFY fills one or more cells at a row and column position in the table that you indicate. There are several ways to control the row and column positioning; you choose each separately:

1. You can do it directly by using ATROW=*row* and ATCOL=*column*.
2. You can open a new row or column with ROW=NEW or COLUMN=NEW.
3. You can use the same row or column as the last **REPORT** instruction with ROW=CURRENT or COLUMN=CURRENT.
4. (For ROW only) You can use ROW=FIND with STRING=*search string* to look for a match in column one. If no match is found, a new row is opened.

Beginning at the specified position, the expressions on the **REPORT** instruction are evaluated and, by default, placed into the cell positions running across the row. If you instead want the information inserted running down a column, use the option FILLBY=COLUMNS. Data in a RECTANGULAR or SYMMETRIC array will be inserted both in rows and columns in the same order as they appear in the array itself.

You can also insert another (already completed) **REPORT** which will extend from the specified position for as many rows and columns as are included in the **REPORT**.

The following is a simple example of the use of **REPORT**: it runs autoregressions with lags 1 to 12, and displays a table showing the lag number in column one and the R^2 in the second column.

```
report(action=define,hlabels=||"Lags","R^2"||)
do lags=1,12
    linreg(noprint) lgdp
    # constant lgdp{1 to lags}
    report(row=new,atcol=1) lags %rsquared
end do lags
report(action=show)
```

You can also add some commonly used tags to the cells using the **SPECIAL** option. The choices are **PARENS**, **BRACKETS**, **ONESTAR**, **TWOSTAR** and **THREESTAR**. **PARENS** encloses the cell in parentheses, as is often done for standard errors or t -statistics, while **BRACKETS** uses [and]. **ONESTAR** attaches a single *, **TWOSTAR** adds ** and **THREESTAR** adds ***. You can do custom tags (for use in TeX) with the **PREFIX** and **SUFFIX** options.

The **ALIGN=LEFT/CENTER/RIGHT/DECIMAL** option lets you set the justification of text strings. The **SPAN** and **TOCOL** options allow you to define cells that span columns.

ACTION=MODIFY,REGRESSION

With **ACTION=MODIFY**, there is a special option which is designed specifically to handle the typical types of reports generated from several regressions. This is the **REGRESSION** option. For example,

```
report(action=define)
linreg foodcons
# constant prretail dispinc trend
report(regression)
linreg foodcons
# constant prretail dispinc{0 1}
report(regression)
report(action=show)
```

This produces the output shown below:

Constant	105.093801 (5.687192)	90.277042 (6.085338)
PRRETAIL	-0.325062 (0.071043)	-0.149396 (0.067939)
DISPINC	0.315237 (0.032067)	0.199063 (0.044460)
TREND	-0.224430 (0.061217)	
DISPINC{1}		0.063716

Chapter 1: Scalars, Matrices, and Functions

(0.051319)

You use this after running a regression. It adds a new column and inserts the coefficients and (by default) the standard error for each of the regressors into it. The default behavior is shown above: it searches the first column for a match with each of the explanatory variables. If a match is found, the information is put into that row. If there is no match, it adds a new row and puts the regressor's label in it, so that if a future regression includes that regressor as well, **REPORT** will find it and put the coefficient from the new regression in that row. Here, both regressions have `Constant`, `PRRETAIL` and `DISPINC`. The first includes `TREND` but not `DISPINC{1}`, and the second `DISPINC{1}` but not `TREND`.

If you'd rather have the table arranged so that the regressors in the same position within the regression are to be considered "parallel", you can add the option `ARRANGE=POSITION`. With that, you also need the `ATROW` option to fix the row for the first regressor. This is necessary because you might very well want to include some summary statistics in addition to the regression coefficients, and those should go at the top to allow room for the table to grow at the bottom.

Your choices for the extra information to be included with the coefficient are specified with the `EXTRA` option. The default for that is `EXTRA=STDERRS`. You can also choose `TSTATS`, `BOTH` and `NEITHER`. If you choose `BOTH`, each regressor will get three lines in the table, while if `EXTRA=NONE`, each will get just one.

ACTION=FORMAT

This sets the numerical format for a selected set of cells in the table. You can either set a specific format using a picture code supplied via the `PICTURE` option, or set a column width with the `WIDTH` option and letting **REPORT** figure out the best format to display the covered data in the width you choose.

By default, this applies to the entire table; to format only a subset of the entries, use the options `ATROW` and `TOROW` to fix the row range and `ATCOL` and `TOCOL` to set the column range. If you do `ATROW` without a `TOROW`, the range will be all rows from the set row on, and similarly for `ATCOL` without `TOCOL`.

For instance, in the last example, if we just wanted three digits right of the decimal, we would insert the instruction

```
report (action=format,picture="*.###")
```

after the regressions, but before the `ACTION=SHOW`. Note that this only affects the formatting used in displaying data. The data themselves are maintained at full precision.

You can also use the `TAG` and `SPECIAL` options to flag the maximum or minimum values with stars or parentheses, and the `ALIGN=DECIMAL` option to decimal-align the values.

ACTION=SORT

Use ACTION=SORT with BYCOL=*column number* to sort the report based on the values in the specified column. You can use the ATROW and TOROW options to limit the sorting to a range of rows, rather than to the entire report.

ACTION=SHOW

REPORT (ACTION=SHOW) displays the constructed table. By default, the output will be directed to the output unit (usually the output window).

You can also display the report in a separate spreadsheet-style report window by including the option WINDOW=*title of window*.

Examples

This makes some improvements on the two regression example from before. It includes the R^2 and Durbin-Watson for each regression, formats the numbers to three decimal places, and displays the output in a window.

Note that the R^2 and Durbin-Watson values are added after the command REPORT (REGRESSORS). This is done because REPORT (REGRESSORS) adds a column, so by doing that first, you can just use COLUMN=CURRENT to place the summary statistics.

```
report(action=define)
report(atrow=1,atcol=1) "R^2"
report(atrow=2,atcol=1) "DW"

linreg foodcons
# constant prretail dispinc trend
report(regressors)
report(atrow=1,column=current) %rsquared
report(atrow=2,column=current) %durbin

linreg foodcons
# constant prretail dispinc{0 1}
report(regressors)
report(atrow=1,column=current) %rsquared
report(atrow=2,column=current) %durbin
report(action=format,picture="*.###")
report(action=show>window="FOODCONS Regressions")
report(action=show)
```

The example on the next page reproduces Table 9.1 from Greene (2012), which shows a regression with standard errors computed using four different formulas. The header labels are the five regressor labels. Note the empty label at the start of that, since the first column has the row labels. The first row has the sample means of three of the variables. The second has the coefficient, the third the OLS standard errors, the fourth the t -statistics. All of these are inserted easily because the %BETA, %STDERRS

Chapter 1: Scalars, Matrices, and Functions

and %TSTATS vectors have the information for all five regressors. Standard errors are computed using three other methods, and each one gets added into a new row. Finally, each of the columns is given a separate format.

```
REPORT(action=define, $
  hlables=| | "", "Constant", "Age", "OwnRent", "Income", "Incomesq" | |)

linreg(smpl=posexp) avgexp
# constant age ownrent income incomesq
stats(noprint) age
compute agemean=%mean
stats(noprint) ownrent
compute ownmean=%mean
stats(noprint) income
compute incmean=%mean

REPORT(atcol=1,atrow=1) "Sample Mean" "" agemean ownmean incmean
REPORT(atcol=1,atrow=2) "Coefficient" %beta
REPORT(atcol=1,atrow=3) "Standard Error" %stderrs
REPORT(atcol=1,atrow=4) "t-ratio" %tstats

linreg(smpl=posexp,robusterrors) avgexp
# constant age ownrent income incomesq

REPORT(atcol=1,atrow=5) "White S.E." %stderrs

compute %xx=%xx*(float(%nobs)/%ndf)
linreg(create,lastreg,form=chisquared)

REPORT(atcol=1,atrow=6) "D. and M. (1)" %stderrs

linreg(smpl=posexp) avgexp / resids
# constant age ownrent income incomesq
prj(xvx=xvx)
set resdm = resids/sqrt(1-xvx)
mcov(lastreg) / resdm
compute %xx=%xx*%cmom*%xx
linreg(create,lastreg,form=chisquared)

REPORT(atcol=1,atrow=7) "D. and M. (2)" %stderrs

REPORT(action=format,atcol=2,picture="*.##")
REPORT(action=format,atcol=3,picture="*.####")
REPORT(action=format,atcol=4,picture="*.####")
REPORT(action=format,atcol=5,picture="*.####")
REPORT(action=format,atcol=6,picture="*.####")

REPORT(action=show>window="Table 9.1")
```

2. Linear Regression Models

Regression techniques lie at the heart of econometrics. This chapter covers a wide variety of methods for estimating linear models. Included are discussions of heteroscedasticity, serial correlation and instrumental variables.

This chapter focuses upon methods for estimating linear models. Chapter 3 describes testing procedures for examining your specifications more carefully, and Chapter 4 discusses non-linear models.

Linear Regressions

Extensions to the Basic Model

Heteroscedasticity

Serial Correlation

Instruments and 2SLS

Recursive Least Squares

Systems Estimation (SUR)

Distributed Lags

Information Criteria

Restricted Regressions

Robust Estimation

2.1 The Standard Normal Linear Model

The first multiple regression model examined in almost any statistics text takes the form

$$(1) \quad y_i = X_i\beta + u_i, \text{ with}$$

(2) u_i assumed to be i.i.d. Normal with mean 0 and constant variance σ^2 .

Assumption (2) is usually made conditional upon the full $\mathbf{X}$ matrix, or, even stronger, $\mathbf{X}$ is assumed to be independent of u entirely. Under those assumptions, least squares has all kinds of optimal properties: it's the best linear unbiased estimator (BLUE), it's the maximum likelihood estimator. Under very weak assumptions, it is consistent, and the least squares estimator is Normally distributed, even in finite samples.

The LINREG Instruction

The primary RATS instruction for estimating linear regressions is **LINREG**. In its simplest form, it estimates the standard regression model described above. We discuss **LINREG** in some detail in Chapter 1 of the *Introduction*. Most of this chapter is devoted to cases where some of the assumptions described above do not hold. Many of these cases can be handled using **LINREG** with one or more of the many options available, while others will require the use of different instructions entirely.

As a brief reminder, a simple linear regression can be handled using an instruction like this:

```
linreg rate
# constant ip mldiff ppisum
```

This regresses the series RATE on a constant and three regressors. You can find the full example in the file `ExampleThree.rpf`. The output produced by this **LINREG** is described in detail in the *Introduction*.

Here is another example, also taken from the `ExampleThree.rpf` example program:

```
linreg rate
# constant ip grm2 grppi{1}
```

Here, we use the lag notation “{1}” to include the one-period lag of GRPPI as an explanatory variable. Note *not* `grppi(-1)` as is used in some other software.

The Regressions Wizard

You can use the *Statistics—Linear Regressions* wizard to estimate linear regression models. In addition to the standard least squares model, the wizard supports many of the variations described throughout this chapter.

2.2 Extensions to Linear Regression: A Framework

Limitations of the Standard Normal Linear Model

Unfortunately, the assumptions described on page UG–38 are unlikely to be met with non-experimental data. And they are particularly likely to fail with time series data. For instance, including a lag of y among the explanatory variables violates these assumptions. We need models which are appropriate more broadly than this.

Generalized Method of Moments

Much of the modern theory of linear regression can be understood most easily by writing the model's assumptions in the form:

$$(3) \quad y_t = X_t\beta + u_t$$

$$(4) \quad EZ'_t u_t = 0$$

where Z_t contains the instrumental variables (if any). This is made operational by replacing the expectation by the sample average, thus solving for β the equation

$$(5) \quad \Theta_T \frac{1}{T} \sum Z'_t (y_t - X_t\beta) = 0$$

where Θ_T is a weighting matrix, which comes into play if the dimensions of Z are bigger than those of X . This is known as the Generalized Method of Moments (GMM for short). The unified theory of these estimators (covering non-linear models as well) was developed originally in Hansen (1982). “Generalized,” by the way, refers to the presence of the weighting matrix.

Assumption (4) sometimes comes directly from assumptions about the variables, or it is sometimes derived as the first order conditions of an optimization problem. For instance, least squares minimizes

$$(6) \quad \sum (y_t - X_t\beta)^2, \text{ which has as its first order necessary conditions}$$

$$(7) \quad -2 \sum X'_t (y_t - X_t\beta) = 0$$

Apart from constant multipliers, this is the same condition as would be derived from equation (3) plus the assumption

$$(8) \quad EX'_t u_t = 0$$

This is a relatively weak assumption, as it assumes nothing about the distribution of the residuals, merely requiring that the residuals be uncorrelated with the regressors. Under (3) plus (8), plus some regularity conditions which ensure good behavior of the sample averages, least squares gives consistent estimates of β .

Chapter 2: Linear Regression Models

Consistent, however, doesn't mean that the output from a **LINREG** instruction is correct. The standard errors, t -statistics and covariance matrix of coefficients (and thus any hypothesis tests which follow) are based upon stronger assumptions than merely (8). In particular, those calculations assume that u is homoscedastic (constant variance) and serially uncorrelated.

It's possible to reformulate the model to deal with heteroscedasticity or serial correlation with specific forms, as will be discussed in Sections 2.3 and 2.4. However, if the precise form of this isn't known, applying a generalized least squares (GLS) technique may, in fact, make matters worse. Plus, particularly in the case of serial correlation, the standard GLS method may give inconsistent estimates, as it requires much stronger assumptions regarding the relationship between X and u .

ROBUSTERRORS

An alternative is to apply the simpler least squares technique for estimation, and then *correct* the covariance matrix estimate to allow for more complex behavior of the residuals. In RATS, this is controlled by the **ROBUSTERRORS** option, which is available on most of the regression instructions. The following is a (somewhat) technical description of this process: if you're interested in the actual conditions required for this, consult a graduate level text like Hamilton(1994) or Greene (2012).

If β_0 is the true set of coefficients and β_T is the solution of (5), then (assuming the estimates are consistent and T is big enough), a first order Taylor expansion of (5) gives

$$(9) \quad \Theta_T \frac{1}{T} \sum Z_t' X_t (\beta_T - \beta_0) \approx \Theta_T \frac{1}{T} \sum Z_t' (y_t - X_t \beta_0)$$

which can be rewritten:

$$(10) \quad \sqrt{T}(\beta_T - \beta_0) \approx \left\{ \Theta_T \frac{1}{T} \sum Z_t' X_t \right\}^{-1} \left\{ \Theta_T \frac{1}{\sqrt{T}} \sum Z_t' (y_t - X_t \beta_0) \right\}$$

Assume that Θ_T converges to a constant (full rank) matrix Θ_0 (it's just the identity matrix if we're doing least squares). Assuming that X and Z are fairly well-behaved, it doesn't seem to be too much of a stretch to assume (by a Law of Large Numbers) that the first factor is converging in probability to a constant matrix which can be estimated consistently by (the inverse of)

$$(11) \quad \mathbf{A}_T = \Theta_T \frac{1}{T} \sum Z_t' X_t$$

The second factor is $1/\sqrt{T}$ times a sum of objects with expected value 0. Under the correct assumptions, some Central Limit Theorem will apply, giving this term an asymptotically Normal distribution. The tricky part about the second term is that the summands, while they are assumed to be mean zero, aren't assumed to be independent or identically distributed. Under the proper conditions, this second term is asymptotically Normal with mean vector zero and covariance matrix which can be estimated consistently by:

$$(12) \quad \mathbf{B}_T = \Theta_T \frac{1}{T} \left\{ \sum_{k=-L}^L \sum_t \mathbf{Z}_t' u_t u_{t-k} \mathbf{Z}_{t-k} \right\} \Theta_T'$$

If there is no serial correlation in the u 's (or, more accurately in the Zu 's), L is just zero. The choice of L is governed by the option `LAGS=correlated lags`. We will call the term in braces in (9) `mcov(Z,u)`—short for matrix covariogram. **MCOV** is the name of the RATS instruction which can compute it, although the calculations usually are handled automatically within another instruction. The end result of this is that (in large samples) we have the following approximation for the distribution of the estimator:

$$(13) \quad \sqrt{T}(\beta_T - \beta_0) \sim N(0, \mathbf{A}_T^{-1} \mathbf{B}_T \mathbf{A}_T'^{-1})$$

This general form for a covariance matrix occurs in many contexts in statistics and has been dubbed a “sandwich estimator.”

Newey–West

There is one rather serious complication with this calculation: when L is non-zero, there is no guarantee that the matrix $\mathbf{B}_T$ will be positive definite. As a result, hypothesis tests may fail to execute, and standard errors may show as zeros. (Very) technically, this can happen because the formula estimates the spectral density of $\mathbf{Z}_t' u_t$ at 0 with a *Dirichlet lag window*, which corresponds to a spectral window that is negative for some frequencies (see Koopmans, 1974, p. 306).

RATS provides a broader class of windows as a way of working around this problem. While the default is the window as shown above (which has its uses, particularly in panel data), other windows can be chosen using the `LWINDOW` option. Newey and West (1987) prove several results when the k term in (12) is multiplied by

$$1 - \frac{|k|}{L+1}$$

which is known as the *Bartlett lag window*. As a result, the covariance matrix computed using this is known in econometrics as Newey–West.

The options `ROBUSTERRORS`, `LAGS=lags` and `LWINDOW=NEWKEYWEST` will give you the Newey–West covariance matrix. See [LINDOWoptions.html](#) in the help for descriptions of the other choices for a description of the other choices for `LWINDOW`.

2.3 Heteroscedasticity

We are using the term *heteroscedasticity* to mean that the equation errors are uncorrelated across observations, but have unequal variances in some systematic way. (It is sometimes used to define *any* departure from i.i.d. errors.) There are two common ways to deal with heteroscedasticity: weighted least squares (WLS) estimation, and the White (or Eicker-White) covariance matrix correction. Both are easily implemented in RATS, with the `SPREAD` and `ROBUSTERRORS` options respectively.

Note that neither of these methods is an appropriate remedy for heteroscedasticity that is the result of a poorly specified model. It's possible that a better approach than "correcting" for the problem is to adjust the actual model, usually through a rescaling such as conversion to per capita values, to produce more equal variances.

See Section 3.5 for a discussion of *tests* for heteroscedasticity.

Heteroscedasticity: WLS using the SPREAD Option

Use the `SPREAD` option on an estimation instruction to perform weighted least squares, that is, to correct for heteroscedasticity of a known form:

$$(14) \quad y_t = X_t\beta + u_t, \quad \text{Var}(u_t) = V_t$$

To do weighted least squares, you only need to know a series which is *proportional* to V_t . Weighted least squares improves the precision of the estimates by "downweighting" observations known to have a high residual variance.

spread=*residual variance series*

The residual variances, V_t , are proportional to the entries of the *residual variance series*. Like the `SMPL` option, you can use a series or a formula.

Note that `SPREAD` specifies the *variances*, or at least a series proportional to them. The "WEIGHT" option used in many other statistics packages requests $p_t = 1/\sqrt{V_t}$. RATS has a `WEIGHT` option, but that's for doing probability weights for the observations. The `SPREAD` option is available on the estimation instructions **LINREG**, **STWISE**, **SUR**, and **ESTIMATE**, and the related instructions **MAKE**, **VCV**, **CMOM**, **MCOV**, **RATIO**, **PANEL**, and **PSTATS**.

The `SPREAD` option implements WLS as follows: If $p_t = 1/\sqrt{V_t}$, and *weighted* variables are those multiplied by p_t and *unweighted* variables are not scaled, then:

Coefficients	computed by regressing $p_t y_t$ on $p_t X_t$ (<i>weighted</i> regression)
R² , other goodness of fit	computed from the <i>weighted</i> regression
Durbin-Watson	computed using <i>unweighted</i> residuals
Residuals	are <i>unweighted</i>

RATS leaves the residuals saved in %RESIDS (or using the *residuals* parameter) in unweighted form so they correspond with the actual data. If you use them later in a **VCV** or **RATIO** instruction, you must again use the `SPREAD` option.

Example

The example file `HETERO.RPF` analyzes a linear regression of food expenditures on income for a cross section of 40 individuals. One would expect that not only does the mean of spending go up with income, but also the variance of the residuals. That the variance also increases with income is fairly clear from looking at a scatter plot, where the `LINES` option is used to put the regression line on the x-y scatter:

```
linreg food
# constant income
scatter(style=dots,lines=%beta,vmin=0.0,$
        hlabel="x = weekly income in $100",$
        vlabel="y = weekly food expenditures in $")
# income food
```

The form of the variance is less clear. If we assume it's directly proportional to income, we compute the weighted least squares estimator by

```
linreg(spread=income) food
# constant income
```

Heteroscedasticity: The ROBUSTERRORS Option

With the `ROBUSTERRORS` option, **LINREG** computes the regression using least squares, but then computes a consistent estimate of the covariance matrix allowing for heteroscedasticity, as in Eicker (1967) and White (1980). *Note that the coefficients themselves do not change, only their standard errors.* For the case of a simple linear regression, the covariance matrix is computed using the results of Section 2.2 with $Z = X$, $L = 0$ and $\Theta = \mathbf{I}$. This simplifies the formula quite a bit to

$$(15) \quad (\beta_T - \beta_0) \sim N\left(0, \left(\sum X_t' X_t\right)^{-1} \left(\sum X_t' u_t^2 X_t\right) \left(\sum X_t' X_t\right)^{-1}\right)$$

If you know the form of heteroscedasticity, this will not be as efficient as weighted least squares—the advantage is that you do not *need* to know the form. It's possible to combine both of these ideas: for instance, if you think that the variance is related to population, but are unsure of the form of that relationship, you might estimate using a “best guess” form for the `SPREAD` option, then include `ROBUSTERRORS` to correct for any remaining problems. In `HETERO.RPF`, this does the least squares regression with “White” standard errors:

```
linreg(robust) food
# constant income
```

Chapter 2: Linear Regression Models

Feasible GLS

While sometimes there may be good theoretical reasons for the spread function to take a particular form, more often all that may be known is that it is likely to be some function of some of the variables in the data set. Under those circumstances, you may be able to estimate the function and apply the weights obtained from that. This process is called Feasible or Estimated GLS (FGLS or EGLS). Greene (2012, Chapter 9) has an especially detailed look at the subject.

Because the spread function has to be positive, the most common form used for it is an exponential. The free parameters are estimated by a regression of the log of the squared residuals on some set of variables. If the explanatory variable is the log of X , then the variance is being modelled as a power function on X . For instance, in `HETERO.RPF`, the weighted least squares estimator was done assuming the variance was proportional to income. The following allows it to be an arbitrary power of income. First, a least squares regression is run, then a regression is run of the log of the squared OLS residuals on the log of income:

```
linreg food
# constant income
*
set esq = log(%resids^2)
set z   = log(income)
*
linreg esq
# constant z
```

If the coefficient on z were near one, it would confirm the earlier choice for the form of the variance (it's actually 2.32).

`PRJ` is now used to compute the fitted values from the auxiliary regression. Since that predicts the log variance, the `exp` of it is used for the `SPREAD` option:

```
prj vhat
linreg(spread=exp(vhat)) food
# constant income
```

The slope estimates actually don't change much with the feasible GLS compared with OLS, but the standard error drops from 2.09 for least squares with White standard errors to .97 for GLS.

It's possible to estimate both the regression parameters and the variance parameters jointly, but that requires non-linear methods (Chapter 4).

2.4 Serial Correlation

Correlation between errors at different time periods is a more serious problem than heteroscedasticity. Again, there are two basic ways to treat this: estimate by Generalized Least Squares (GLS) taking into account a specific form of serial correlation, or estimate by simple least squares, then compute a covariance matrix which is robust to the serial correlation. See Section 3.6 for discussion of *tests* for serial correlation.

For GLS, RATS offers only one “packaged” form of serial correlation correction, which is for first order. There is a well-developed (though now rather old) literature on estimation with first-order autoregressive (AR1) errors. More complicated error structures can be handled using filtered least squares (see the **FILTER** instruction), non-linear least squares (Section 4.8), or a RegARIMA model (Section 6.4.4). Note, however, that statistical practice is moving away from tacking the dynamics onto a static model through the error term towards models which are designed to produce serially uncorrelated errors by incorporating the dynamics directly using lagged variables, such as with VAR’s (Chapter 7) and ARDL models (page UG–59).

The AR1 Instruction

AR1 computes regressions with correction for first order autocorrelated errors by estimating a model of the form:

$$(16) \quad y_t = X_t\beta + u_t, \quad u_t = \rho u_{t-1} + \varepsilon_t$$

Its syntax is very similar to **LINREG**, except that it has some options for choosing the estimation method for the serial correlation coefficient ρ . You can also input a specific value for ρ .

Example **AR1.RPF** demonstrates several ways of handling serially correlated errors in a regression of *Y on X*. One very simple possibility is to run a “first difference” regression, which would be correct if $\rho=1$, and will be reasonable if ρ is close to that. That can be done most easily with

```
ar1(rho=1.0) y  
# constant x
```

Note, by the way, that this zeroes out the **CONSTANT**, which will show a zero coefficient and zero standard errors.

If ρ needs to be estimated, you have a choice between two optimization criteria, and within each, two ways of computing ρ (iterative or search):

- Simple least squares, skipping the first observation (**METHOD=CORC** and **METHOD=HILU**, Cochrane–Orcutt and Hildreth–Lu respectively).
- Maximum likelihood (**METHOD=MAXL** and **METHOD=SEARCH**).

For completeness, RATS also offers Prais-Winsten, which is a full-sample GLS estimator, with **METHOD=PW**.

Chapter 2: Linear Regression Models

The iterative methods (CORC and MAXL) are much faster; however, they don't guarantee that you have found the global optimum. CORC, in particular, can converge to the wrong root if you have lagged dependent variables. As, with modern computers, speed is no longer much of an issue in these models, the default estimation procedure is the slower but steadier HILU.

In large samples, there should be only a slight difference between the methods, unless the results show multiple optima. However, in small samples, there can be a substantial difference between maximum likelihood and the least squares estimators. Maximum likelihood includes an extra data point, which can have a major impact when there aren't many data points to start. Maximum likelihood steers the estimates of ρ away from the boundaries at plus and minus one, with the difference becoming more noticeable as ρ approaches those boundary values.

AR1.RPF uses three of these, HILU, CORC and MAXL.

```
ar1(method=hilu) y
# constant x
ar1(method=corc) y
# constant x
ar1(method=maxl) y
# constant x
```

As one would hope, HILU and CORC give (effectively) identical results. MAXL comes in with a somewhat higher value of .95 vs .89, which isn't an unreasonable difference given that there are only 40 data points, and MAXL can use all of them.

Serial Correlation: The ROBUSTERRORS Option

On **LINREG**, **NLLS**, **NLSYSTEM**, **SUR**, **GARCH**, **LDV**, **DDV** and **MAXIMIZE**, you can use the ROBUSTERRORS option, combined with the *LAGS=correlated lags* option, to compute an estimate of the covariance matrix allowing for serial correlation up to a moving average of order *correlated lags*. This is sometimes known as the HAC (Heteroscedasticity and Autocorrelation Consistent) covariance matrix.

ROBUSTERRORS is important in situations where:

- You do not know the form of serial correlation, so a particular generalized least squares (GLS) procedure such as **AR1** may be incorrect, or
- GLS is inconsistent because the regressors (or instruments) are correlated with past residuals. Brown and Maital (1981) and Hayashi and Sims (1983) are two of the earliest examples of the proper analysis of such settings.

In some situations, the proper value of *correlated lags* is known from theory. For instance, errors in six-step-ahead forecasts will be a moving average process of order five. If not known, it has to be set to catch *most* of the serial correlation.

In general, it's a good idea to also include the `LWINDOW=NEWKEYWEST` to get the Newey-West covariance matrix, or one of the other non-truncated windows. Otherwise the covariance matrix produced can be defective in a way that isn't noticed until you try to use it to do hypothesis tests. Note that in the Newey-West window, if `LAGS=1`, the contribution of the lag term is cut in half. In a situation like this, even if the correlations are known to be zero after the first lag, you might be better off adding some extra lags just to make sure the Newey-West window doesn't cut off most of the attempted correction.

An example where **AR1** would be a mistake is seen in Section 6.8 of Hayashi (2000) which analyzes the relationship between spot and forward exchange rates. A market efficiency argument implies that the regression

$$(17) \quad A_t = \alpha + \beta P_t + u_t, \quad A_t = \text{actual depreciation and } P_t = \text{predicted depreciation}$$

should have coefficients satisfying $\alpha = 0, \beta = 1$. However, because these are weekly data of 30-day forward rates, the residuals will almost certainly be serially correlated, up to four moving average lags. However, an AR1 "correction" won't work. AR1, in effect, runs OLS on

$$(18) \quad A_t - \rho A_{t-1} = \alpha(1 - \rho) + \beta(P_t - \rho P_{t-1}) + (u_t - \rho u_{t-1})$$

The transformed residual $(u_t - \rho u_{t-1})$ can't be assumed to be uncorrelated with the transformed regressor $(P_t - \rho P_{t-1})$, since the information which led to the surprise u_{t-1} will get incorporated into P_t . Instead, the equation is estimated by OLS, with standard errors corrected for the serial correlation:

```
linreg(robusterrors, lags=4, lwindow=neweywest) s30_s
# constant f_s
```

AR1.RPF includes a similar treatment:

```
linreg(robust, lwindow=neweywest, lags=4) y
# constant x
```

2.5 Instrumental Variables and Two-Stage Least Squares

Applicability

The RATS instructions **LINREG**, **AR1** (autocorrelation correction), **NLLS** (non-linear least squares), **PREGRESS** (panel regression), **SUR** (linear systems estimation) and **NLSYSTEM** (non-linear systems estimation) all support instrumental variables estimation. For the first four, RATS does (a variant of) two-stage least squares; for the last two, it is done using the Generalized Method of Moments. See Section 4.9 for more on GMM.

In all cases, you do the instrumenting by setting up the list of available instruments using the instruction **INSTRUMENTS**, and then including the **INST** option on the estimation instruction.

The Instruction INSTRUMENTS

RATS uses the **INSTRUMENTS** instruction to maintain the list of instruments. With smaller models, you probably need to set this just once. With larger simultaneous equations models, you may not have enough observations to use all the exogenous and predetermined variables. If so, you will probably need to change the list for each equation. Use the **ADD** or **DROP** options to make small changes to the list. **NLSYSTEM** has a special option (**MASK**), which can do a joint estimation with differing sets of instruments.

Note that you must set the instrument list *before* you do the estimation. A procedure in which the instruments depend upon the parameters being estimated cannot be done (easily) with RATS.

Note that *no* variable, not even the **CONSTANT**, is included in an instrument set automatically. Also note that there is no concept of a specific variable or set of variables “instrumenting out” a particular explanatory variable.

Technical Information

RATS does not, literally, do two sets of regressions, though the effect is the same as if it did. Instrumental variables for a linear regression is based upon the assumptions used in Section 2.2:

$$(19) \quad y_t = X_t\beta + u_t$$

$$(20) \quad EZ_t'u_t = 0$$

where Z_t are the instruments. In solving

$$(21) \quad \Theta_T \frac{1}{T} \sum Z_t' (y_t - X_t\beta) = 0,$$

the weight matrix is chosen to be

$$(22) \quad \Theta_T = \left(\sum X_t' Z_t \right) \left(\sum Z_t' Z_t \right)^{-1}$$

which is the matrix of regression coefficients of the X 's on the Z 's. This gives

$$(23) \quad \hat{\beta} = \left[\left(\sum X_t' Z_t \right) \left(\sum Z_t' Z_t \right)^{-1} \left(\sum Z_t' X_t \right) \right]^{-1} \left[\left(\sum X_t' Z_t \right) \left(\sum Z_t' Z_t \right)^{-1} \left(\sum Z_t' y_t \right) \right]$$

The calculation of the covariance matrix depends upon the options chosen. If you *don't* use ROBUSTERRORS, the assumption is made that

$$(24) \quad E(u_t | Z_t) = \sigma^2$$

which gives us the covariance matrix estimate

$$(25) \quad \hat{\sigma}^2 \left[\left(\sum X_t' Z_t \right) \left(\sum Z_t' Z_t \right)^{-1} \left(\sum Z_t' X_t \right) \right]^{-1}$$

More generally the covariance matrix will be estimated according to the formulas from Section 2.2.

If you use the SPREAD option with instrumental variables, all the formulas above are adjusted by inserting the reciprocal of the “spread” series into each of the sums:

$$(26) \quad \Theta_T = \left(\sum X_t' (1/V_t) Z_t \right) \left(\sum Z_t' (1/V_t) Z_t \right)^{-1}$$

with similar changes to all of the other sums.

Examples

Example KLEIN.RPF estimates Klein's Model I (see, for instance, Greene, 2012, p. 332). The endogenous explanatory variables are (current) PROFIT, PRIVWAGE and PROD (production). The exogenous and pre-determined variables from the model are CONSTANT, TREND, the policy variables GOVTWAGE, GOVTEXP and TAXES and lags of CAPITAL, PROD and PROFIT.

```
instruments  constant trend govtwage taxes govtexp $
              capital{1} prod{1} profit{1}
linreg(inst)  cons
# constant  profit{0 1}  wagebill
linreg(inst)  invst
# constant  profit{0 1}  capital{1}
linreg(inst)  privwage
# constant  prod{0 1}  trend
```

Chapter 2: Linear Regression Models

INSTRUMENT.RPF is based upon example 9.5 from Wooldridge (2010). It estimates labor supply and labor demand functions using 2SLS. The HOURS and LWAGE (log wage) variables are assumed to be endogenous. EDUC, AGE, KIDSLT6, KIDSGE6, NWIFEINC, EXPER and EXPERSQ (experience and its square) are assumed to be exogenous and enter the hours or wage equation or both. EXPER and EXPERSQ are excluded from the hours equation. AGE and the family variables (KIDSLT6, KIDSGE6, NWIFEINC) are assumed not to directly affect the wage equation.

This sets the instrument set to the full list of exogenous variables and estimates the hours equation by 2SLS:

```
instruments constant educ age kidslt6 kidsge6 nwifeinc $
    exper expersq
linreg(instruments) hours
# constant lwage educ age kidslt6 kidsge6 nwifeinc
```

This checks the reduced form for LOG(WAGE) and sees if the instruments excluded from the labor demand function (EXPER and EXPERSQ) work. If the coefficients on these are effectively zero, we have a weak set of instruments. This is true even if the R^2 on the reduced form is high, since, that can be high because of the correlation with the variables already included in the regression.

```
linreg lwage
# constant educ age kidslt6 kidsge6 nwifeinc exper expersq
```

This estimates the wage equation by 2SLS, and the reduced form for HOURS (the endogenous variable included in the wage equation):

```
linreg(instruments) lwage
# constant hours educ exper expersq
linreg hours
# constant educ age kidslt6 kidsge6 nwifeinc exper expersq
```

2.6 Recursive Least Squares

The instruction **RLS** does Recursive Least Squares. This is the equivalent of a sequence of least squares regressions, but done in a very efficient way. The main purpose of recursive estimation is to determine if a model is stable. If a model is correctly specified as

$$(27) \quad y_t = \mathbf{x}_t\beta + u_t, u_t \text{ is i.i.d. } N(0, \sigma^2)$$

the series of *recursive residuals* is

$$(28) \quad (y_t - \mathbf{x}_t\beta_{t-1}) / \sqrt{1 + \mathbf{x}_t(\mathbf{X}'_{t-1}\mathbf{X}_{t-1})^{-1}\mathbf{x}'_t}$$

for $t > K$, where K is the number of regressors, $\mathbf{X}_{t-1}$ is the matrix of explanatory variables through period $t-1$, and β_{t-1} the estimate of β through $t-1$.

These residuals have the property that they are *also* i.i.d., while least squares residuals have a finite sample correlation. A failure of the recursive residuals to behave like this type of process will lead to rejection of the base assumptions: either the coefficients aren't constant through the sample, or the variance isn't constant, or both. See Section 3.8 for more on this.

Note that **RLS** is designed for recursive *estimation* only. If you need to do something more than that (for instance, you want to generate forecasts at each stage), you can use the instruction **KALMAN**, which does the same type of sequential estimation, but only updates one entry for each instruction. If you want to allow the (true) coefficients in (27) to be time-varying, you need to set up a state-space model (Chapter 10).

RLS has much the same format as **LINREG**, and the output will largely match that from a **LINREG**, since the **RLS** over the full sample will be the same as full sample least squares. Note, however, that **RLS** will not do instrumental variables estimation.

The residuals produced by **RLS** using either the built-in %RESIDS series or the *resids* parameter are the recursive residuals, not the finite sample OLS residuals.

The special options that distinguish **RLS** from **LINREG** are:

COHISTORY	saves the sequential estimates of the coefficients in a VEC[SERIES].
SEHISTORY	saves the sequential estimates of the standard errors of the coefficients in a VEC[SERIES].
SIGHISTORY	saves the sequential estimates of the standard error of the regression into a series.
CSUMS	saves the cumulated sum of the recursive residuals in a series.
CSQUARED	saves the cumulated sum of squared recursive residuals in a series.
DFHISTORY	saves the regression degrees of freedom in a series.

Chapter 2: Linear Regression Models

Example

```
rls(csquared=cusumsq) c / rresids
# constant y
```

Post-sample predictive test. The cumulative sum of squares is in the series CUSUMSQ. While the numerator could also be written more simply as $RRESIDS(1993:1)^2$, we write it this way as it will generalize easily to more than one step.

```
compute fstat=(cusumsq(1993:1)-cusumsq(1992:1))/$(
    (cusumsq(1992:1)/(%ndf-1))
cdf(title="Post-Sample Predictive Test") ftest fstat 1 %ndf-1
```

Harvey-Collier functional misspecification test. This can be done by just computing the sample statistics on the recursive residuals and examining the t-stat and significance level provided therein.

```
stats rresids 1952:1 1993:1
```

The ORDER option

By default, **RLS** generates the estimates in entry sequence. It first works through the initial observations until finding a set which generates a full rank regression.

Entries are then added one by one, and the “history” series are updated. If you think that the more likely alternative to stable coefficients and variance is an instability related to some variable other than the entry sequence, you can use the option **ORDER** to base the sequence on a different variable. Note that this *does not* rearrange the data set. Keeping the data set itself in the original order is very important if the model includes lags. If you use **ORDER**, the entries in any of the series created as described earlier are maintained in their original order. You can use the option **INDEX** to find out the order in which entries were added to the model. This is a **SERIES[INTEGER]**, defined from the first entry in the regression to the last. If, for instance, you use **INDEX=IX** for a regression run over 1974:3 through 2017:4, **IX(1974:3)** will be the smallest entry in the regression range for the order series, while **IX(2017:4)** will be the largest.

This is an example of the use of **RLS** with the **ORDER** option. It does an arranged autoregression test for a TAR (threshold autoregressive) model (see Section 11.5). The **RLS** does recursive estimation ordered on the first lag of **DLR**. The recursive residuals are then regressed on the explanatory variables. Under the null of no threshold effect, the coefficients should pass a test for zero values.

```
diff lr / dlr
set thresh = dlr{1}
rls(order=thresh) dlr / rresids
# constant dlr{1 2}
linreg rresids
# constant dlr{1 2}
exclude(title="Arranged Autoregression Test for TAR")
# constant dlr{1 2}
```

2.7 Systems Estimation

Background Information

RATS can estimate systems of *linear* equations with the instruction **SUR**. This uses the technique of Seemingly Unrelated Regressions for standard regressions, and three-stage least squares for instrumental variables.

If you need to estimate a system of *non-linear* equations, or have restrictions across equations other than simple equality constraints, use the instruction **NLSYSTEM** (Section 4.10). If you can (reasonably) use **SUR**, you should do so, as it is much faster and is usually much easier to set up.

SUR is very different from the single-equation estimation instructions. You specify the system using a set of equations, not regressor lists on supplementary cards. You need to construct these equations in advance using **EQUATION**, or possibly by **LINREG** with the option **DEFINE**. The equations can be organized into a **MODEL** using **GROUP**, or you can input the equation information directly into **SUR**.

Technical Information

We use the following basic notation:

$$(29) \quad y_{it} = X_{it}\beta + u_{it}, \quad t = 1, \dots, T; \quad i = 1, \dots, N$$

$$(30) \quad E[u_t u_t'] = \Sigma, \quad u_t' = (u_{t1}, \dots, u_{tN_t})$$

The variables should be defined over the same intervals across the cross-section elements i . RATS drops from the calculation any entry t for which *any* variable involved in the regression is missing.

These estimators are described in Section 10.2 of Greene (2012). The GLS (SUR) estimator is

$$(31) \quad \beta = \left(\mathbf{X}' (\Sigma^{-1} \otimes \mathbf{I}) \mathbf{X} \right)^{-1} \left(\mathbf{X}' (\Sigma^{-1} \otimes \mathbf{I}) \mathbf{y} \right)$$

where β and $\mathbf{y}$ are formed by stacking vectors from the N equations, and $\mathbf{X}$ is formed by stacking augmented X_i matrices: matrices with columns of zeros for explanatory variables in the other equations. The covariance matrix of the estimates is

$$(32) \quad \left(\mathbf{X}' (\Sigma^{-1} \otimes \mathbf{I}) \mathbf{X} \right)^{-1}$$

If you specify a matrix with the **CV** option, RATS will use *that* matrix for Σ . Otherwise, it computes OLS (for **SUR**) or 2SLS (for **SUR (INST)**) estimates of the equations and uses the estimated covariance matrix of the residuals:

$$(33) \quad \hat{\Sigma} = \frac{1}{T} \sum_{t=1}^T \hat{u}_t \hat{u}_t'$$

Chapter 2: Linear Regression Models

Note that some programs use a slightly different estimate of Σ , preferring the formula

$$(34) \quad \hat{\Sigma} = \frac{1}{T-K} \sum_{t=1}^T \hat{u}_t \hat{u}_t'$$

where K is the number of regressors per equation (assuming a system with equations of a similar form). This doesn't affect the coefficient estimates, since a scale factor in Σ will cancel out. However, the standard errors will be slightly different.

Iterating

By default, RATS applies a single application of feasible GLS, as described above. However, if the `ITER` option is non-zero, RATS recomputes Σ and reestimates the system. It repeats this process until either the iteration limit is exceeded, or the change in coefficients is small. The convergence criterion is controlled by the option `CVCRT`. See Section 4.1 on how RATS determines convergence.

The `%SIGMA` matrix and the `CVOUT` option save the Σ at the final estimates, which is *not* the same as the Σ used in computing the estimates.

Inputting a Σ

You can input your own value for the Σ matrix using the `CV` option. As noted above, by default RATS computes Σ from a preliminary estimate of the model. If you have too few observations to estimate Σ freely (you need at least as many observations as equations), you *will* have to input Σ yourself.

Note that the estimates of the standard errors of coefficients will be incorrect if your input Σ doesn't provide estimates of the residual variances.

SUR.RPF example

`SUR.RPF` computes joint GLS estimates for a two company subset for Grunfeld's investment equations. The Grunfeld data is a common "textbook" data set for small panels. The full data set has annual data from 1935 to 1954 on 11 companies with investment, capital stock and the firm's market value. The two countries included in the example are GE and Westinghouse.

Because `SUR` operates off `EQUATIONS`, the following are used to define them:

```
equation geeq ige
# constant fge cge
equation westeq iwest
# constant fwest cwest
```

Since this is also doing OLS estimates, it's also possible to estimate by `LINREG` and define the equation at the same time:

```
linreg(define=geeq) ige
# constant fge cge
```

If the `EQUATIONS` have already been defined, the `LINREGS` can be done with:

```
linreg(equation=geeq)   ige
linreg(equation=westeq) iwest
```

There are two ways to get the description of the overall model into the `SUR` instruction: with supplementary cards or with a `MODEL`. With just two equations, the former is probably easier, but we'll show both. The unrestricted `SUR` can be done with

```
sur(vcv)    2
# geeq
# westeq
```

or with

```
group grunfeld geeq westeq
sur(model=grunfeld)
```

Hypothesis Tests

The test procedures of Chapter 3 work somewhat differently for `SUR` than for `LINREG` and other single-equation estimation procedures. The main differences are:

- You cannot use `EXCLUDE` and `SUMMARIZE` for a test involving a regressor which appears in more than one equation of the system. This is because these two instructions list restrictions according to the regressor names, which won't uniquely identify a coefficient.
- With `RESTRICT`, `MRESTRICT` and `TEST`, keep in mind that the coefficient numbers are based upon the position in the "stacked" system. For instance, if the first equation has 3 variables, the coefficients in the second equation will have numbers 4, 5, etc.
- `RESTRICT(CREATE)` will not work because it is designed for a single equation and cannot properly handle the printing for a multivariate system. Do `RESTRICT(REPLACE)` followed by `SUR(CREATE)` instead.
- RATS uses the second set of formulas from Section 3.3. Thus, it reports the tests as χ^2 not F .

For instance, the following tests for equality between the "F" (market value) coefficients. In the stacked coefficient vector, the `FGE` coefficient is position 2 and the `FWEST` is position 5:

```
restrict(title="Test of equality of F coefficients") 1
# 2 5
# 1 -1 0
```

As described in the final point, the test statistic shows as a χ^2

```
Test of equality of F coefficients
Chi-Squared(1)=      3.203911 with Significance Level 0.07346239
```

Chapter 2: Linear Regression Models

Restrictions Across Equations

SUR can estimate a system subject to equality constraints across all the equations using the **EQUATE** option:

```
sur(model=grunfeld,equate=| 2,3 |)
```

which forces the second (**Fxx**) and third (**Cxx**) coefficients to be equal across equations. Note that this rearranges the coefficient vector, so the coefficients which are forced equal are listed first. In this case, there are actually only four free coefficients: **Fxx**, **Cxx**, **CONSTANT** in **GE** and **CONSTANT** in **WEST**. The output will display the information in the usual equation by equation form, but you will note that the coefficient numbers are out-of-sequence.

For more complicated restrictions, use **NLSYSTEM** instead (Section 4.10) even if the model is linear.

Transforming Equations into FRMLs

If you need to transform the equations into formulas for system simulation (Chapter 8), you have two options.

1. You can use a set of **FRML** instructions with the **EQUATION** option *after* the **SUR** instruction:

```
sur(inst) 3
# conseq
# investeq
# profeq
frml(equation=conseq) consfrml
frml(equation=investeq) invfrml
frml(equation=profeq) proffrml
```

2. You can use the **FRML** option on the **EQUATION** instructions you use to define the model.

Generalized Method of Moments

While the default behavior for **SUR** with the **INSTRUMENTS** option is three stage least squares, it has options for supporting more general weightings of the instruments. See Section 4.10 for more information about this.

2.8 Distributed Lags

Background

We are using the phrase *distributed lag* to refer to a model with a form such as

$$(35) \quad y_t = \sum_{l=L_0}^L \beta_l X_{t-l} + u_t$$

where the explanatory variable X is different from y . This *distributes* the effect of X on y across a number of *lags*, hence the name. We are not using this term when X is the same as y —we refer to such models as autoregressions, which have *very* different statistical properties from the models which we are describing here.

Using RATS

RATS offers several major advantages for distributed lag estimation, among them:

- You can specify lag ranges using the compact and flexible notation:
`series{lag1 TO lag2}`.
- RATS computes the estimates in a memory and time-efficient fashion.
- RATS can easily handle polynomial distributed lags, as well as splines and other restricted forms.
- You can use spectral methods for estimation or analysis of results.

DISTRIBLAG.RPF example

The DISTRIBLAG.RPF estimates a distributed lag of long-term interest rate series (composite yield on long-term U.S. Treasury bonds) on a short-term one (yield on 90 day Treasury bills), monthly from 1947:1 to 2007:4. This same data set is used in example files AKAIKE.RPF (page UG–61) and also in examples PDL.RPF, SHILLER.RPF and GIBBS.RPF to demonstrate various ways to estimate distributed lags. An unrestricted distributed lag from 0 to 24 is done with

```
linreg longrate
# constant shortrate{0 to 24}
```

Summary Measures

Because of multicollinearity, the individual coefficients in a distributed lag regression usually are poorly determined. For instance, in the unconstrained lag estimate, only the 0 and 24 are (individually) statistically significant at conventional levels. Thus we are interested more in summary measures, in particular, the sum and the shape of the lag distribution. There are several instructions or options which can be very helpful in obtaining this information:

- The instruction **SUMMARIZE** is designed specifically for computing the sum of a set of lag coefficients.
- The lag coefficients themselves can be extracted from the %BETA vector defined by **LINREG**.

Chapter 2: Linear Regression Models

This uses **SUMMARIZE** to compute the sum of the full set of lag coefficients. Then lag coefficients are pulled out of *the* %BETA vector into LAGDIST, using SET. The lag coefficients are in slots 2 through 26 of %BETA, corresponding to lags 0 to 24, in order (slot 1 is the coefficient on the CONSTANT). The NUMBER option on **GRAPH** causes it to label entry 1 as 0, 2 as 1, etc, thus getting the lag labels correct.

```
summarize
# shortrate{0 to 24}
set lagdist 1 25 = %beta(t+1)
graph(header="Lag Distribution-Long Unconstrained",number=0)
# lagdist 1 25
```

Estimation Techniques

There are four basic ways to approach distributed lag estimation:

1. Unrestricted long lags, as advocated by Sims (1974), which has just been demonstrated.
2. Data-determined lag length with no shape restrictions.
3. “Hard” shape restrictions, such as polynomial (Almon) distributed lags or splines.
4. “Soft” shape restrictions, such as Shiller’s smoothness prior (Shiller, 1973).

Unrestricted long lags are easy to estimate. The others take a bit more work, though we do have a procedure for doing standard polynomial distributed lags (PDL’s).

Data-determined lag lengths

These use one of the information criteria such as Akaike or Schwarz (Section 2.9), or a general-to-specific testing strategy to find the “optimal” lag length.

“Hard” restrictions

These are done using restricted least squares (Section 2.10) typically with **ENCODE** and **LINREG (UNRAVEL . . .)**. You should note that the *t*-statistics on individual coefficients may end up being extremely high in such a regression. This is because each tests whether its coefficient can be made zero *while maintaining the shape restriction*, which is often nearly impossible. The most common of these is the polynomial or Almon lag. The procedure **@PDL** handles these types of estimates; see **PDL.RPF** for an example.

Geometric distributed lags (or Koyck lags), which are “hard,” infinite (rather than finite) lags, are a special case of transfer functions. See the instruction **BOXJENK** for more on that.

“Soft” restrictions

Compute these by mixed estimation, described in Section 1.2 of the *Additional Topics* PDF. An example is provided on **SHILLER.RPF**.

ARDL (AutoRegressive Distributed Lags)

An increasingly common way to handle this type of analysis is the ARDL model (AutoRegressive Distributed Lag) which combines the distributed lag with lagged dependent variables. The standard distributed lag model (35) usually leaves very high serial correlation in the residuals. Adding lags of the dependent variable to the regression gives a model which can be represented in lag form as:

$$(36) \quad \alpha(L)y_t = \beta(L)x_t + u_t$$

The dynamic responses of y to x can be computed by doing the polynomial division $\beta(L) / \alpha(L)$. The RATS polynomial functions can be used to do this.

The following (also from `DISTRIBLAG.RPF`) estimates an ARDL for the interest rates, with three lags of the dependent variable and current to four lags of the explanatory variable. These were chosen using a preliminary stepwise regression (done with **ST-WISE**).

```
linreg longrate
# constant longrate{1 2 3} shortrate{0 to 4}
```

The following extract the lag polynomials from the regression. (On the `%EQNxxxxx` functions equation “0” is the most recent regression). `%EQNLAGPOLY` recognizes the `LONGRATE` is the dependent variable, so it returns in `ARPOLY` the coefficients for

$$(37) \quad \beta_0 + \beta_1 L + \beta_2 L^2 + \beta_3 L^3 + \beta_4 L^4$$

while `DLPOLY` (for the explanatory variable) will be

$$(38) \quad 1 - \alpha_1 L - \alpha_2 L^2 - \alpha_3 L^3$$

Then `DLPOLY` is divided by `ARPOLY` and expanded out to the 24th degree.

```
compute arpoly=%eqnlagpoly(0,longrate)
compute dlpoly=%eqnlagpoly(0,shortrate)
compute ardlpoly=%polydiv(dlpoly,arpoly,24)
```

This copies the expanded polynomial into the series `LAGDIST` and graphs it (as with the unrestricted lags, using the `NUMBER` option to get the lags labeled correctly).

```
set lagdist 1 25 = ardlpoly(t)
graph(header="Lag Distribution-ARDL(3,4)",number=0)
# lagdist
```

This computes the long-run response by evaluating the polynomials at the value 1.

```
disp "LR Response" %polyvalue(dlpoly,1.0)/%polyvalue(arpoly,1.0)
```

This gives a similar value to the **SUMMARIZE** on the unrestricted lag (.941 vs .926) and the lag distributions are broadly similar except the unrestricted lag turns up sharply at the final lag. That’s a common feature for long unrestricted lags on data with high serial correlation.

2.9 Information Criteria (IC)

A number of criteria have been proposed to allow the data to determine the length of a distributed lag (or more generally to select a model). These combine a measure of fit with a penalty for the number of parameters. We select the lag length by minimizing the function over different choices for the lag length. Of these, the two most commonly used are the Akaike Information Criterion (AIC) (Akaike, 1973) and the Schwarz (1978) Criterion, which is variously known as the SC, SIC, SBC (Schwarz Bayesian Criterion) or BIC (Bayesian Information Criterion). There are a number of equivalent forms of these—the ones that we will use are

$$\begin{aligned}\text{Akaike:} & \quad (-2\log L) / T + 2K/T \\ \text{Schwarz:} & \quad (-2\log L) / T + \log(T)K/T\end{aligned}$$

where K is the number of regressors and T is the number of observations. Because this is likelihood-based, it can be used for many types of models, not just those estimated by least squares. These are “standardized” by division by the number of observations, which makes the scales somewhat more manageable.

The Schwarz criterion puts a heavier penalty on additional parameters and so it will never choose a model larger than Akaike. There are conflicting theoretical results about which of these is “better.” If the correct model is included in the collection of models examined, SBC will, given enough data, choose it, while AIC won’t do so necessarily—even in very large samples, it can pick models which are too big. (SBC is “consistent”, AIC isn’t). However, if the correct model *isn’t* included (for instance, the actual lag length is infinite), then AIC proves to be better at picking an approximate model. (AIC is “efficient”, SBC isn’t). For more information, see the discussion in Brockwell and Davis (1991).

When you use an IC, it’s very important that you *run the regressions over the same interval* so they will be directly comparable. The most common use is to select a lag length (or lengths in the case of an ARIMA model). If you just run the regressions without prior planning, the regressions for the short lags will use more data points than those with long lags. If the early data points turn out to be harder to fit than the later ones, this will tend to favor longer lags, even if they are of little help in an “apples vs apples” comparison.

@REGCRITS Procedure

The procedure **@REGCRITS** computes the information criteria after a regression. In addition to the AIC and SBC, it also computes the Hannan–Quinn and (log) FPE (Final Prediction Error) criteria, which have their own proponents. To use this, after running the regression, just do

@regcrits

Use the **NOPRINT** option if you want to suppress the output—the computed criteria are saved in the variables **%AIC**, **%SBC**, **%HQCRIT** and **%LOGFPE** (log of the FPE). You can use the **TITLE** option to give a more meaningful description than “Information Criteria”.

AKAIKE.RPF Example

Example `AKAIKE.RPF` computes the Akaike and Schwarz criteria for a distributed lag model. The easiest and fastest way to carry out the set of regressions is to use **CMOMENT** combined with **LINREG (CMOMENT)**: the regressions will come out of the same cross-product matrix, ensuring that they all use the same sample and also cutting the calculation time (though the latter will be almost unnoticeable with a linear model like this).

This puts a **NOPRINT** on the **LINREG**, since we only need the summary information, and puts the information out using relatively simple **DISPLAY** instructions. One advantage of using standardized criteria is that the values tend to be representable with the something like the two-digits left being used here.

```
cmom
# constant shortrate{0 to 24} longrate
*
disp "Lag" @10 "Akaike" @20 "Schwarz"
do maxlag=0,24
    linreg(cmom,noprint) longrate
    # constant shortrate{0 to maxlag}
    compute akaike = (-2.0*%logl+%nreg*2.0)/%nobs
    compute schwarz = (-2.0*%logl+%nreg*log(%nobs))/%nobs
    disp maxlag ##.### @10 akaike @20 schwarz
end do
```

A fancier way to show the information (and actually not much harder once you get used to it) is to create a **REPORT** (Section 1.9).

```
report(action=define,hlabels=||"Lags","Akaike","Schwarz"||,title=
"Distributed Lag IC")
do maxlag=0,24
    linreg(cmom,noprint) longrate
    # constant shortrate{0 to maxlag}
    compute akaike = (-2.0*%logl+%nreg*2.0)/%nobs
    compute schwarz = (-2.0*%logl+%nreg*log(%nobs))/%nobs
    report(row=new,atcol=1) maxlag akaike schwarz
end do
```

This tags with a ***** the minimum in the 2nd and in the 3rd columns.

```
report(action=format,tag=minimum,special=onestar,atcol=2,tocol=2)
report(action=format,tag=minimum,special=onestar,atcol=3,tocol=3)
```

This uses a common format of three decimals to the right and forces the numbers to align on the decimal point for easy reading.

```
report(action=format,picture="*.###",align=decimal)
report(action=show)
```

2.10 Restricted Regressions

If you merely want to *test* restrictions, you can use an instruction like **EXCLUDE** or **RESTRICT** as described in Chapter 3. What this section describes are ways to estimate a (linear) model subject to (linear) restrictions. More general constrained optimization (non-linear or inequality constraints) is covered in Section 4.4. If you have a linear model with inequality constraints, you can use those methods or quadratic programming (Section 13.2).

RATS provides two methods for computing restricted linear regressions. The first is to use **ENCODE** to construct new variables as linear combinations of regressors. You estimate the model using these constructed variables and use the **UNRAVEL** option on the estimation instruction to report coefficient values and other statistics in terms of the original variables. The second method is to estimate the unrestricted model first, and then use **RESTRICT** or **MRESTRICT** with the **CREATE** option to compute the restricted regression. Both methods are described below.

With ENCODE and UNRAVEL

The **ENCODE** instruction combined with the **UNRAVEL** option work by reparameterizing the regression to incorporate the restrictions, coding the restrictions into the regressors. When you include the **UNRAVEL** option on the estimation instruction, RATS substitutes back for the original regressors. It does not print the regression in terms of the coded variables, just the final form. For example, a simple equality constraint on two coefficients can be handled using:

Set up 1×2 matrix with a weight of 1.0 on both series:

```
declare rect r(1,2)
compute r=||1.0,1.0||
```

Create the encoded variable $Z = X1 + X2$:

```
encode r / z
# x1 x2
```

Estimate the regression. With UNRAVEL, results will be reported in terms of $X1$ and $X2$ rather than Z :

```
linreg(unravel) rate
# constant z
```

The most common use of these instructions is computing restricted distributed lags such as polynomial distributed lags (see example file `PDL.RPF`):

```
dec rect r(4,13)
ewise r(i,j)=j^(i-1)
encode r / enc1 enc2 enc3 enc4
# x{0 to 12}
linreg(unravel) y
# constant enc1 enc2 enc3 enc4
```

With RESTRICT or MRESTRICT

The similar instructions **RESTRICT** and **MRESTRICT** offer the other way for doing restricted linear regressions. They impose the restriction *after* the unrestricted model has been estimated.

After you do the unrestricted regression, apply either **RESTRICT (CREATE)** or **MRESTRICT (CREATE)** to add the restrictions. With **CREATE**, these do the following:

1. Compute the new coefficient vector and covariance matrix of coefficients subject to the restrictions. See for the formulas used in this procedure.
2. Compute new summary statistics. RATS recomputes all the standard variables, such as %RSS and %NDF.
3. Replace the old regression with the new restricted regression: *any further hypothesis tests will apply to the restricted regression, not the original one.*

For example:

```
linreg logq
# constant logk logl loge
restrict(create) 1
# 2 3 4
# 1. 1. 1. 1.
```

forces the coefficients 2, 3, and 4 (the coefficients on the series LOGK, LOGL, and LOGE) to sum to 1.

Which to Choose

In general, the two methods are almost reciprocals of each other when it comes to the difficulty of setting them up. One or two restrictions (if they apply across a number of regressors) will require a large number of “encoded” variables. And an **ENCODE/UNRAVEL** which produces just a few series from a large number will require many direct restrictions. In general, the **ENCODE** and **UNRAVEL** combination will be useful mainly in the situations such as the tightly restricted distributed lags. Otherwise, use **RESTRICT** or **MRESTRICT**.

However, in many cases, the most straightforward way to handle a restricted estimation is to use non-linear methods (Chapter 4), even if the equation is fully linear.

For instance, the last example can be done with **NLLS**, estimating both the unrestricted and restricted regressions using

```
nonlin(parmset=base) a0 bk b1 be
nonlin(parmset=constraint) bk+b1+be==1.0
frml cobb logq = a0+bk*logk+b1*logl+be*loge
* Unrestricted:
nlls(frml=cobb,parmset=base) logq
* Restricted:
nlls(frml=cobb,parmset=base+constraint) logq
```

2.11 Doing it Your Way: LINREG with CREATE

If you use **LINREG** with the option **CREATE**, **LINREG** *does not* compute the regression. Instead, it displays standard regression output using information which *you* supply. You can use this when you've done a "non-standard" calculation of the coefficients, the covariance matrix or both. By using **LINREG (CREATE)**, you can get the proper *t*-statistics and gain access to the hypothesis testing instructions: **EXCLUDE**, **SUMMARIZE**, **TEST**, **RESTRICT** and **MRESTRICT** (Section 3.2).

You can use the following **LINREG** options with **CREATE**:

lastreg/[nolastreg]

With **LASTREG**, RATS uses the regressors from the preceding regression. You do not need to include a supplementary card.

coeff=*VECTOR of coefficients* [default:%BETA]

covmat=*SYMMETRIC array covariance matrix* [default:%XX]

The **COEFF** option supplies the vector of coefficients and the **COVMAT** option supplies the matrix which will become the %XX matrix. *You may not dimension %BETA or %XX* (the default **COEFF** and **COVMAT** arrays) yourself. However, once you have completed a regression, you can use matrix instructions to alter them.

form=f/chisquared

If you change %XX or use **COVMAT**, and the new matrix is itself an estimate of the covariance matrix, use the option **FORM=CHISQUARED**, as described on the next page.

title=*Title to identify the estimation procedure* ("User")

You can use this to include a phrase identifying the estimation method inside the regression output.

regcorr=*number of restrictions*

Use this option if you have computed the coefficients subject to restrictions. This allows **LINREG** to compute the proper degrees of freedom.

residuals=*input residuals series*

This option allows you to provide your own series of residuals. RATS will use this series in computing the summary statistics.

equation=*equation to use*

You can use the **EQUATION** option as an alternative to a supplementary card to input a regression which is different from the preceding one (meaning you cannot use **LASTREG**). The *equation* should be an equation which you have already defined: it supplies the list of explanatory variables and dependent variable. Use the **COEFF** and **COVMAT** options to input the coefficients and covariance matrix.

The FORM Option

Section 3.3 lists two sets of formulas used by the hypothesis testing instructions:

1. **FORM=F**: These use as the covariance matrix of coefficients $\sigma^2 (\mathbf{X}'\mathbf{X})^{-1}$ (or its equivalent). $(\mathbf{X}'\mathbf{X})^{-1}$ is stored in the array %XX by **LINREG**, **NLLS** or **AR1**. The regression instruction computes an estimate $\hat{\sigma}^2$ of σ^2 and stores it internally.
2. **FORM=CHISQUARED**: These use as the covariance matrix the array %XX alone, that is, %XX does not need to be scaled by the residual variance. This method is used for **DDV**, **LDV**, **GARCH**, **SUR**, **MAXIMIZE**, **NLSYSTEM**, **CVMODEL** or **DLM**, and also either **LINREG** or **NLLS** with the **ROBUSTERRORS** option.

Normally, RATS automatically chooses which to use based upon which estimation technique was used last. However, the procedures on the next page will change the situation to one in which **FORM=CHISQUARED** is correct. Use that option to inform RATS about this. **FORM=F/CHISQUARED** is an option for **LINREG** with the **CREATE** option, and on **EXCLUDE**, **SUMMARIZE**, **TEST**, **RESTRICT** and **MRESTRICT**. Note that if you use **FORM=CHISQUARED** on **LINREG (CREATE)**, you do not need to use it on hypothesis tests which follow it.

Example

This is from Example 9.3 from Greene(2012). It does one of the Davidson-MacKinnon (1993) refinements on White standard errors. We first do the standard White errors using **LINREG** with **ROBUSTERRORS**.

```
open data states.wks
data(org=cols,format=wks) 1 50 expend pcaid pop pcinc
set pcexp = expend/pop
linreg(robusterrors) pcexp / resids
# constant pcaid pcinc
```

Use PRJ to help compute the standardized residuals. Those are used on the MCOV instead of the simple OLS residuals. (If you just put RESIDS on the MCOV instead of RESDM, you'd be computing the White covariance matrix).

```
prj (xvx=xvx)
set resdm = resids/sqrt(1-xvx)
mcov(lastreg) / resdm
compute %xx=%xx*%cmom*%xx
linreg(create,lastreg,form=chisquared,$
      title="OLS with DM(2) Standard Errors")
```

2.11.1 Recomputing a Covariance Matrix

Current statistical practice includes a large set of procedures which produce consistent estimates for coefficients, while giving inconsistent estimates of the covariance matrix of the estimated coefficients. For example:

- Least squares with heteroscedasticity or serial correlation of the residuals.
- Robust estimators computed with iterated weighted least squares.
- Heckman's iterated least squares procedure for truncated samples (Heckman, 1976).

RATS can handle the first automatically if you use the `ROBUSTERRORS` option, as described in Section 2.2. You need to do the computations yourself in other situations. This relies upon an extension of the general results from that section.

Computational Strategy

A basic strategy for computing the covariance matrix which works in a wide variety of situations is the following: If the estimator solves the first order conditions

$$(39) \quad \sum X_t' f(y_t - X_t \beta) = 0$$

for some function $f(u)$, the covariance matrix of $\hat{\beta}$ can be estimated by the “sandwich” $\mathbf{A}^{-1} \mathbf{B} \mathbf{A}^{-1}$ where

$$(40) \quad \mathbf{B} = \sum X_t' X_t f^2(y_t - X_t \beta), \quad \text{and}$$

$$(41) \quad \mathbf{A} = \sum X_t' X_t f'(y_t - X_t \beta)$$

You might have to change the sign on $\mathbf{A}$ to make it positive definite. A sign change will cancel out of the calculation because $\mathbf{A}^{-1}$ appears twice. You can compute $\mathbf{B}$ and $\mathbf{A}$ with the instruction `MCOV` using as *residuals* (in the notation of `MCOV`):

- the series f for computing $\mathbf{B}$
- the series f' with the option `NOSQUARE` for computing $\mathbf{A}$.

Use the following steps:

1. Compute your estimates.
2. Compute the series of values $f(u)$ and $f'(u)$.
3. Compute $\mathbf{A}$ and $\mathbf{B}$ using `MCOV`.
4. Replace `%XX` with $\mathbf{A}^{-1} \mathbf{B} \mathbf{A}^{-1}$ using the instruction

```
compute %xx=%mqform(b,inv(a))
```
5. Use `LINREG` with the options `CREATE` and `FORM=CHISQUARED` to reprint the regression, using the new standard errors and covariance matrix.

2.12 Robust Estimation

We concern ourselves with estimating β in the model

$$(42) \quad y_t = X_t\beta + u_t$$

While least squares gives consistent and asymptotically Normal estimates of β under a fairly broad range of conditions, it's also well known that it is sensitive to outliers or a fat-tailed u distribution—see, for instance, the discussion in Greene (2012), Chapter 7. It's possible to drop “outliers” from the data set. For instance, the following is from example file `ROBUST.RPF`. This estimates a regression by least squares, computes the standardized (more precisely the internally studentized) residuals and reruns the regression dropping from the sample any data point where the standardized residual is greater than 2.5 in absolute value.

```
linreg logy / resids
# constant logk logl
prj (xvx=px)
set stdresids = resids/sqrt(%seesq*(1-px))
*
linreg (smp1=abs(stdresids)<=2.5) logy
# constant logk logl
```

However, if those observations can reasonably be assumed to be valid within the model (42), and not just recording errors or data points where the model simply fails, it might make more sense to choose an alternative estimator which won't be as sensitive as least squares to tail values.

One such estimator is LAD (Least Absolute Deviations), which is

$$(43) \quad \hat{\beta} = \underset{\beta}{\text{minimizer}} \sum_t |y_t - X_t\beta|$$

This is provided by the RATS instruction **RREG** (Robust Regression). LAD is consistent and asymptotically Normal under broader conditions than are required for least squares. It will be less efficient if the u 's are better behaved, with about 60% efficiency for Normal u 's. Its main drawback relative to least squares is that it is much more difficult to compute—the minimand isn't differentiable, and it requires a specialized variant of linear programming to compute the estimator. There is also some question about the best way to estimate the covariance matrix—for details, see the description of **RREG**. **RREG** (for doing LAD) has a similar form to **LINREG**. In `ROBUST.RPF`, the command is simply:

```
rreg logy
# constant logk logl
```

which here gives results quite similar to the outlier-adjusted **LINREG**, though with somewhat higher standard errors.

Chapter 2: Linear Regression Models

Some alternative robust estimators can be generated which have behavior in the tails similar to LAD, but near zero are more like least squares. For instance, the following objective function:

$$(44) \sum_i \frac{u_i^2}{(c^2 + u_i^2)^{1/2}}$$

will look like the absolute value in the tails, when u is much larger than the constant c , but will look like the square when c is larger than u and thus the constant dominates the denominator. By keeping the minimand differentiable, this can be estimated more simply: either by iterated weighted least squares or by a standard nonlinear estimation routine.

If you're using iterated weighted least squares, computing the estimator itself is rarely a problem, since you just need to put **LINREG** with the **SPREAD** option inside a loop which recalculates the **SPREAD** series. The more complicated part is computing the covariance matrix of the coefficients. The result of the iteration process gives an inconsistent estimate of this, as it does not take into account the dependence of the **SPREAD** series on β . The procedure described in Section 2.11.1 needs to be used to correct the covariance matrix. In the example, this does the "M" estimator, starting out from least squares (a constant series for **SPREAD**). This does up to 10 iterations or until convergence determined using the **%TESTDIFF** function which compares the old and new parameter vectors.

```
compute c = sqrt(%seesq)
dec vector beta0
*
set spread = 1.0
do iters=1,10
    compute beta0=%beta
    set spread = sqrt(c^2+resids^2)
    linreg(noprint,spread=spread) logy / resids
    # constant logk logl
    if %testdiff(beta0,%beta)<.0001
        break
end do iters
disp "Iterations Taken" iters
```

The sandwich estimator for the covariance matrix is computed and the regression displayed using

```
set f      = resids/spread
set fprime = c^2/spread^3
mcov(matrix=b,lastreg) / f
mcov(matrix=a,lastreg,nosquare) / fprime
linreg(create,lastreg,form=chisquared,covmat=%mqform(b,inv(a)), $
    title="Iterated Weighted Least Squares")
```

3. Hypothesis Testing

RATS has a large and flexible set of testing instructions. These allow you to test virtually any form of linear hypothesis. With the proper steps, you can implement even fairly complicated procedures such as a Hausman specification test.

This chapter describes and demonstrates tests for heteroscedasticity (Section 3.5), serial correlation (3.6), exogeneity or causality (3.7), structural stability (3.8), functional form (3.9), misspecification (3.10), unit roots (3.11) and cointegration (3.12).

RATS is also capable of doing “computationally intensive” testing procedures, such as those which use Monte Carlo methods, or randomized orderings. See Chapter 16 for more on that subject.

There are also many procedures available for a wide variety of hypothesis tests. Some are included with RATS, while many others—written by the RATS user community—are available for downloading from the Estima web site (www.estima.com)

Please note that RATS, unlike some other software, does not produce standard “batteries” of hypothesis tests after a regression. It may provide some comfort to see your model pass a whole set of “specification” tests, but because these are applied regardless of whether they are actually of interest in a specific situation, they may end up missing areas where your model might be failing.

Technical Details

Heteroscedasticity

Serial Correlation

Causality

Structural Stability

Specification

Unit Roots and Cointegration

3.1 Introduction

Theoretical Tests vs Specification Tests

Hypothesis tests generally fall into one of two categories: tests of a theoretical interest to the model, and specification or diagnostic tests of the assumptions used to estimate a model. The first (examples are constant returns to scale, unbiasedness in prediction) are generally performed as tests on the coefficients of an estimated model—a less-restricted model is estimated first, and the implied restrictions on the coefficients are tested. Many of these are fairly straightforward applications of standard RATS instructions covered in Sections 3.2. The second (such as a test for serial correlation) generally involves some type of secondary calculation—either a second regression or a test calculation, both of which are most often applied to the residuals from the initial estimates. These more often require a calculation specific to the hypothesis and quite often use a RATS procedure to do the work.

Common Errors in Interpreting Results

The single most common error in interpreting tests is mixing up the null and alternative. Classical hypothesis tests have a null which is being tested and an alternative. Formally, we either accept or reject the null. Informally, we may say that we are “accepting” the alternative when we reject the null, but that often isn’t what the test is designed to do. This is particularly true of specification tests. As an example, the test for “ARCH” (page UG–80) has a null of homoscedastic residuals against an alternative suggested by an ARCH model. Rejecting the null does not mean that an ARCH model is a good description of the data (that is, we don’t “accept” the ARCH model), just that homoscedasticity *isn’t*.

Another common error is ignoring “maintained assumptions”: ones that are present under both the null and alternative (unfortunately not always clearly expressed). If a maintained assumption, is, in fact, false, you could end up rejecting a hypothesis for the wrong reason. An example here is the Chow test for structural change (page UG–87) where the maintained assumption is that the residuals are homoscedastic, and thus the difference between variance estimates in the three-sample F test form of that would have to be due to differences in the coefficients. If there is, in fact, a change in the variance (instead), you could end up rejecting, even if the coefficients themselves are the same in the samples.

Effect of Large Sample Sizes

If you have a very long data series (finance data sets are often thousands of observations), it will be quite rare for you to get a time series model which passes diagnostic tests at a conventional .05 level. Why this is the case is explained in a very accessible fashion in Leamer(1978). In short, if you stick with a .05 level for Type I error, all the information from your huge data set goes towards reducing Type II error, which is most easily done by rejecting the null most of the time.

A more detailed explanation is provided in the June 2018 *RATSLetter*.

3.2 RATS Instructions

RESTRICT	This instruction can test any set of linear restrictions and can also be used for doing restricted regressions. This is the most general testing instruction, so it is the hardest to use. TEST , EXCLUDE and SUMMARIZE are more specialized, and easier to use.
MRESTRICT	is similar to RESTRICT , but uses matrices rather than supplementary cards for the specification of restrictions.
EXCLUDE	tests exclusion restrictions: a restriction that every one of a set of coefficients is zero.
TEST	is a generalized version of EXCLUDE which allows testing restrictions of the form $\beta_i = \text{constant}$. You can also use it to test for equality between two coefficient vectors.
SUMMARIZE	computes a linear combination of coefficients and its standard error or a non-linear function of the coefficients with its linearized standard error.
RATIO	tests restrictions on systems of several equations by forming a likelihood ratio statistic from the covariance matrices. Use it when the restricted and unrestricted maximum likelihood estimates of the system can be computed using single equation methods.
CDF	provides a set of “tables” for the Normal, F , t and χ^2 distributions. You provide a computed test statistic and choose the distribution (and degrees of freedom, if necessary) and CDF prints the marginal significance level of the statistic. Use CDF when the test statistic must be computed in a manner not covered by the other instructions.

In addition to these instructions, you can also report your own test statistics in a more flexible fashion by using the **DISPLAY** or **REPORT** instructions. The functions %ZTEST, %TTEST, %FTEST and %CHISQR compute significance levels for the standard testing distributions.

The TITLE option

All of the instructions listed above have a **TITLE** option which allows you to include a descriptive title for your test in the output. For example,

```
exclude(title="Test of Cobb-Douglas as a restriction on translog")
# logl2 logk2 loglk
```

Chapter 3: Hypothesis Testing

Interpreting the Output

The testing instructions print out their result in a form similar to:

```
F(4,133) = 2.350870 with Significance Level 0.05737263
```

This reports the distribution, the degrees of freedom (if applicable), the test statistic, and the marginal significance level (or *P*-value). This is the probability that a random variable with the appropriate distribution will *exceed* the computed value (in absolute value for Normal and *t* distributions). A *P*-value close to *zero* leads you to reject the null hypothesis. For instance, a marginal significance level of .02 would cause you to reject a hypothesis at the .05 level, but not at the .01 level. So, the test statistic above, with a marginal significance level of .057 would not quite be significant at the .05 level.

Suppose you are regressing a variable *Y* (with 100 data points) on a constant term and an explanatory variable *X*, and you want to test the hypothesis that the slope coefficient is one. One way to do this would be:

```
linreg y  
# constant x  
test(title="Test of Unit Slope")  
# 2  
# 1.0
```

The null hypothesis being tested is that the second coefficient is 1.0. Output such as:

```
Test of Unit Slope  
F(1,98)= 0.52842 with Significance Level 0.46900386
```

would be strong evidence in support of the null hypothesis, suggesting that the coefficient on *X* is indeed one. However, a result such as:

```
F(1,98)= 18.10459 with Significance Level 0.00004789
```

would represent a strong rejection of the null, suggesting that the coefficient is not equal to one.

If you want to do a one-tailed test for a Normal or *t*-distributed statistic, just divide the reported marginal significance level by two.

Applicability of the Testing Instructions

The five instructions **RESTRICT**, **MRESTRICT**, **EXCLUDE**, **TEST** and **SUMMARIZE** are all based upon “Wald” test calculations as described in . They can be applied to the results of any of the following “regression” instructions:

```
LINREG, AR1, SUR, ITERATE, DDV, LDV, RREG, PREGRESS, RLS.
```

You can use any but **EXCLUDE** after **BOXJENK**, **CVMODEL**, **DLM**, **FIND**, **MAXIMIZE**, **NLLS**, **NLSYSTEM**, or **GARCH**. You cannot do tests based directly upon an **ESTIMATE** or **KALMAN**.

For **SUR**, **DLM**, **MAXIMIZE**, **NLSYSTEM**, **CVMODEL**, **DDV**, **LDV**, **GARCH**, or for any others where you have used the **ROBUSTERRORS** option, the test statistics are reported as Chi-squared rather than F .

Wizards

There are wizards for **EXCLUDE**, **TEST** and **RESTRICT**. All are reached by choosing the *Statistics—Regression Tests* wizard.

Variables Defined

All testing instructions set values for the following variables:

%CDSTAT	the computed test statistic (real)
%SIGNIF	the marginal significance level (real).

Which distribution is used depends upon the instruction and the situation.

Examples

```
linreg(robusterrors) pcexp
# constant pcaid pcinc large dpcaid dpcinc
*
exclude(title="Sample Split Test-Robust Standard Errors")
# large dpcaid dpcinc
```

Because this is done with a **LINREG** with **ROBUSTERRORS**, the results are shown as a χ^2 . It is also shown (for easier comparison with a similar test done *without* the corrected standard errors) as the equivalent F with infinite denominator degrees of freedom—the value of that F statistic will be the chi-squared divided by the test degrees of freedom:

```
Sample Split Test-Robust Standard Errors
Null Hypothesis : The Following Coefficients Are Zero
LARGE
DPCAID
DPCINC
Chi-Squared(3)=      5.365171 or F(3,*)=      1.78839 with Significance Level 0.14692899
```

The following estimates a linear regression and tests that the second and third coefficients sum to 1. With one degree of freedom it shows this both as a t and as an F .

```
linreg logc
# constant logy logc{1}
restrict(title="Direct Test of Long-Run MPC=1") 1
# 2 3
# 1.0 1.0 1.0
```

```
Direct Test of Long-Run MPC=1
t(46)=      1.930439 or F(1,46)=      3.726596 with Significance Level 0.05973058
```

3.3 Technical Information (Wald Tests)

This section describes the computations used by the five regression-based testing instructions: **EXCLUDE**, **SUMMARIZE**, **TEST**, **RESTRICT** and **MRESTRICT**. All of these compute the “Wald” test, though many “LM” tests are computed by applying these to auxiliary regressions.

A set of Q linear restrictions on the coefficient matrix β can be written

$$\begin{array}{ccc} \mathbf{R} & \beta & = \mathbf{r} \\ Q \times K & K \times 1 & Q \times 1 \end{array}$$

We list below the formulas for the test statistic (used by all five instructions) and the restricted coefficient vector and covariance matrix (for **RESTRICT** and **MRESTRICT** with the **CREATE** option). The first form is for single equation regressions and the second for most non-linear estimation procedures, systems of equations and regressions with corrected covariance matrices.

In these formulas, $(\mathbf{X}'\mathbf{X})^{-1}$ may not precisely be that matrix, but will be its appropriate analogue. $\Sigma_{\mathbf{x}}$ is the estimated covariance matrix of coefficients. Either matrix is saved by RATS under the name %XX.

Test statistic

$$F(Q, t-K) = \frac{(\mathbf{r} - \mathbf{R}\hat{\beta})' [\mathbf{R}(\mathbf{X}'\mathbf{X})^{-1} \mathbf{R}']^{-1} (\mathbf{r} - \mathbf{R}\hat{\beta})}{(Q\hat{\sigma}^2)}$$

$$\chi^2(Q) = (\mathbf{r} - \mathbf{R}\hat{\beta})' [\mathbf{R}\Sigma_{\mathbf{x}}\mathbf{R}']^{-1} (\mathbf{r} - \mathbf{R}\hat{\beta})$$

Restricted coefficient vector

$$\hat{\beta}_{\mathbf{R}} = \hat{\beta} + (\mathbf{X}'\mathbf{X})^{-1} \mathbf{R}' [\mathbf{R}(\mathbf{X}'\mathbf{X})^{-1} \mathbf{R}']^{-1} (\mathbf{r} - \mathbf{R}\hat{\beta})$$

$$\hat{\beta}_{\mathbf{R}} = \hat{\beta} + \Sigma_{\mathbf{x}} \mathbf{R}' [\mathbf{R}\Sigma_{\mathbf{x}}\mathbf{R}']^{-1} (\mathbf{r} - \mathbf{R}\hat{\beta})$$

Restricted covariance matrix

$$\text{Var}(\hat{\beta}_{\mathbf{R}}) = \hat{\sigma}^2 \left\{ (\mathbf{X}'\mathbf{X})^{-1} - (\mathbf{X}'\mathbf{X})^{-1} \mathbf{R}' [\mathbf{R}(\mathbf{X}'\mathbf{X})^{-1} \mathbf{R}']^{-1} \mathbf{R} (\mathbf{X}'\mathbf{X})^{-1} \right\}$$

$$\text{Var}(\hat{\beta}_{\mathbf{R}}) = \Sigma_{\mathbf{x}} - \Sigma_{\mathbf{x}} \mathbf{R}' [\mathbf{R}\Sigma_{\mathbf{x}}\mathbf{R}']^{-1} \mathbf{R} \Sigma_{\mathbf{x}}$$

If the matrix in [...] is not invertible, RATS will, in effect, drop any redundant restrictions and reduce the degrees of freedom appropriately. If RATS makes such an adjustment, you will see a message like

Redundant Restrictions. Using 4 Degrees, not 8

3.4 Technical Information (Lagrange Multiplier Tests)

The Testing “Trinity”

The three main types of tests are the Wald, Likelihood Ratio and Lagrange multiplier, known as the testing trinity. This is covered in just about any modern graduate econometrics text, such as Greene (2012) or Wooldridge (2010). The formulas used in the Wald test were described in the previous section.

Lagrange multiplier tests start by estimating a model under the null hypothesis. The gradient of the (log) likelihood or analogous optimand will be zero in the restricted model itself, and if the null is true, it would be expected to be close to zero in the alternative (expanded) model. The LM tests are often much easier to compute than the other two tests, because they require only that you be able to compute the restricted model, and the gradient (and some approximation to the information matrix) of the expanded model evaluated at the restricted model. Since the extra parameters are typically zero under the null, the gradient often simplifies quite a bit when computed under the null.

While the LM tests have a general structure which unites them, they can take a wide variety of forms once you simplify the gradient calculation in a specific case. However, there are two main forms which appear repeatedly: the auxiliary regression and the outer product of the gradient (OPG). Note, by the way, that these two can be different ways of doing the same calculation.

Auxiliary Regression LM Tests

An auxiliary regression is a secondary regression which includes some variables generated from your primary regression, typically a function of the residuals. The actual test is usually either for the regression having a zero R^2 , or for some regressor or regressors having zero coefficients. The latter type of test can be done by just using an **EXCLUDE** on the coefficients in question. Depending upon the situation, the R^2 tests may be based upon either the centered R^2 or the uncentered one, the latter if a zero constant is implied by the null. The variable %TRSQ returns the number of observations times the uncentered R^2 , while %TRSQUARED will give the number of observations times the *centered* R^2 .

For instance, the following is part of example 6.3 in Wooldridge (2010). It does an overidentification test, determining if there is correlation between the residuals from an instrumental variables regression and the list of instruments.

```
instruments constant exper expersq motheduc fatheduc huseduc
linreg(instruments) lwage / residu
# constant exper expersq educ
linreg residu
# constant exper expersq motheduc fatheduc huseduc
cdf(title="Overidentification Test by Auxiliary Regression") $
chisqr %trsq 2
```

Chapter 3: Hypothesis Testing

OPG Tests

The outer product of the gradient form of the test can be used when the gradient can be written as $\sum g_t$ with sufficient regularity conditions that

$$\sum g_t \left(\sum g_t' g_t \right)^{-1} \sum g_t'$$

is asymptotically distributed as a χ^2 .

You will often see this described as being done by a regression of a vector of 1's on the gradient components. The test statistic is then the sum of squared fitted values from that regression, which is the same (in this case) as the number of observations minus the sum of squared residuals. RATS offers a more convenient way to compute this in most cases. In order to run this (literally) as a regression of 1's on the gradient components, you need to create series for those components. However, in a wide range of cases, the gradient components will just be the product of residuals (or some other series) with a set of already existing variables. If this is the case, you can compute the test statistic using **MCOV** with the **OPGSTAT** option.

```
mcov (opgstat=computed value) / resids  
# gradient variables
```

The “gradient variables” are the variables which are gradient components when they are multiplied by the *resids* series. Typically, these are the original regressors together with one or more generated series which are supposed to have a zero correlation with the residuals under the null. Remember that if the original regressors *are* included in this list, the degrees of freedom of the test will be reduced, as those will typically have gradient components which are forced to sum to zero under the null hypothesis.

This is from the same example as on the last page. It does a “robust” overidentification test, determining if there is a correlation between the residuals from an instrumental variables regression and the residuals from regressions of the two potentially endogenous variable on their instruments.

```
linreg motheduc / rmoth  
# constant exper expersq educat  
linreg fatheduc / rfath  
# constant exper expersq educat
```

```
mcov (opgstat=lmrobust) / resids  
# rmoth rfath  
cdf (title="Robust Overidentification Test") chisqr lmrobust 2
```

Note that you can also use the **@RobustLMTtest** procedure for this.

3.5 Testing for Heteroscedasticity

The Null Hypothesis

The null hypothesis is

$$y = \mathbf{X}\beta + u, \quad E[u] = 0, \quad E[uu'] = \sigma^2 \mathbf{I}$$

There are many alternatives ranging from simple (different variances over different periods) to elaborate (ARCH: autoregressive conditional heteroscedasticity). Several of these are common enough to have standard procedures that are supplied with RATS. In particular, the procedure `@RegWhiteTest`, applied immediately after a regression, can be used to do the Breusch-Pagan or White tests described here.

Most tests have two forms: an F -test and a LM form, which give similar, but not identical significance levels.

HETEROTEST.RPF example

The example file `HETEROTEST.RPF` is based upon a hedonic price regression for housing price, adapted from Wooldridge(2009). The three explanatory variables are the lot size, square footage and number of bedrooms. This works with a linear specification. (Wooldridge also looks at a semi-log model, which is probably more appropriate and doesn't show heteroscedasticity.)

Because we keep re-estimating this (possibly over different samples), we define an `EQUATION` so we can use `LINREG (EQUATION)` to estimate:

```
equation linff price
# constant lotsize sqrft bdrms
*
linreg(equation=linff)
```

Breusch-Pagan Test

Breusch and Pagan (1979) describe a Lagrange Multiplier test against the very general alternative

$$\sigma_i^2 = h(z_i' \alpha)$$

where z_i is some set of variables, such as regressors. Under LM test principles, the function h washes out of the test statistic, so the single test works simultaneously against all alternatives based upon z . We show here a slight modification to Breusch and Pagan, due to Koenker (1981), which does not require an assumption of Normal residuals. This does the two forms described above using the full set of explanatory variables as z_i :

```
set usq = %resids^2
linreg usq
# constant lotsize sqrft bdrms
exclude(title="Breusch-Pagan Test for Heteroscedasticity")
# lotsize sqrft bdrms
cdf(title="Breusch-Pagan Test, LM Form") chisqr %trsquared 3
```

Chapter 3: Hypothesis Testing

The LM version can be done using the procedure `@RegWhiteTest` with the option `type=bp`. You need to do this right after the regression that you want to test, so we re-estimate the model:

```
linreg(equation=linff)
@RegWhiteTest(type=bp)
```

Harvey's Test

This is the same idea as Breusch and Pagan, but with *exp* as the *h* function. The *exp* function is a reasonable functional form in practice, because it produces positive values and, if the “z” variables are in log form, gives a power function for the variance.

This does Harvey's test using only the log of lot size:

```
linreg(equation=linff)
set logusq = log(%resids^2)
set llotsize = log(lotsize)
linreg logusq
# constant llotsize
cdf(title="Harvey Test") chisqr %trsquared 1
```

White's Test

White (1980) describes a test which has power against alternatives that affect the consistency of the least squares covariance matrix. It is a variation of the Breusch-Pagan test, where z_i consists of the regressors, their squares and products. Because there can be duplication among the regressors and products of regressors, you may need to drop some of those. However, if you do end up with collinearity among the regressors, RATS will simply zero out the redundant variables, and reset the degrees of freedom of the regression. The calculation for the degrees of freedom in the `CDF` instruction below will give the correct test value if such an adjustment is made. However, because of the complexity of the setup for this test, we would recommend that you use the procedure `@RegWhiteTest`.

```
linreg(equation=linff)
set usq = %resids^2
set lotsq = lotsize^2
set sqrftsq = sqrft^2
set bdrmsq = bdrms^2
set lotxsqrft = lotsize*sqrft
set lotxbdrms = lotsize*bdrms
set sqftxbdrms = sqrft*bdrms
*
linreg usq
# constant lotsize sqrft bdrms $
lotsq sqrftsq bdrmsq lotxsqrft lotxbdrms sqftxbdrms
cdf(title="White Heteroscedasticity Test") chisqr $
%trsquared %nobs-%ndf-1
linreg(equation=linff)
@RegWhiteTest
```

Goldfeld-Quandt Test

The Goldfeld-Quandt test has as its alternative hypothesis that one (identifiable) segment of the sample has a higher variance than another. The test statistic is computed by running the regression over the two subsamples and testing the ratio of the estimated variances.

In a cross section data set, the segments are usually determined by the values of one of the variables. In some cases, you will have zero/non-zero variables identifying the segments, and can use SMPL options to select the subsamples directly. Sections 8.3.3 and 8.4.2 of Hill, Griffiths, Lim (2008) look at the variance of wages for those living in metropolitan areas versus those in rural areas. The variable METRO is equal to 1 for city dwellers, and 0 for those in rural areas, so we can use SMPL options directly:

```
linreg(smpl=metro) wage
# constant educ exper
compute rssmetro=%rss,ndfmetro=%ndf
linreg(smpl=.not.metro) wage
# constant educ exper
compute rssrural=%rss,ndfrural=%ndf

compute gqstat=(rssmetro/ndfmetro)/(rssrural/ndfrural)
cdf(title="Goldfeld-Quandt Test") ftest gqstat ndfmetro ndfrural
```

In other cases, you will need to sort or rank the values of one of the regressors to identify the subsamples. We recommend using **ORDER** with the **RANKS** option for this. The following does that as part of **HETEROTEST.RPF**.

```
order(ranks=lotranks) lotsize
linreg(equation=linff,smpl=lotranks<=36)
compute rss1=%rss,ndf1=%ndf
linreg(equation=linff,smpl=lotranks>=53)
compute rss2=%rss,ndf2=%ndf
cdf(title="Goldfeld-Quandt Test") ftest $
(rss2/ndf2)/(rss1/ndf1) ndf2 ndf1
```

Note that if the alternative hypothesis is that the variances are simply *different*, not that a specific half is greater than the other, then either a very small or a very large F will lead to rejection of the hypothesis. For a 5% two-tailed F , reject if the significance level is either less than .025 or greater than .975.

Also, if you suspect that the variance is related to a continuous variable, breaking the sample into two parts means that each subsample will have some observations close to the break value—this will reduce the power of the test. In this situation, the usual advice is to have a third subsample in the middle which isn't included in the test. The example above is excluding (roughly) 20% of the observations in the middle. Because **ORDER** gives tied entries an average rank, the exact number of elements in either of the subsets isn't controlled by the number you choose. For instance, if three entries are tied for 35, 36 and 37, they will all be assigned rank=36, so there would actually be 37 data points in the first partition.

ARCH Test

ARCH (for **A**uto**R**egressive **C**onditional **H**eteroscedasticity) was proposed by Engle (1982) as a way to explain why large residuals tend to clump together. The model (for first order ARCH) is

$$u_t \sim N\left(0, \sigma^2 (1 + \alpha u_{t-1}^2)\right)$$

The test is to regress the squared residual series on its lag(s). This should have an R^2 of zero under the null hypothesis. This uses a different data set as testing for ARCH rarely makes sense in a cross section data set like that used before.

One error users sometimes make is applying a test for ARCH to the data rather than residuals. What goes into the test should be serially uncorrelated (as much as possible). In this example, the data just need to have the mean removed first:

```
diff(center) dlogdm / resids
set usq = resids^2
linreg usq
# constant usq{1}
cdf(title="Test for ARCH(1)") chisqr %trsquared 1
```

Testing for higher order ARCH just requires adding extra lags of USQ and increasing the degrees of freedom on the **CDF** instruction.

This can be done more simply using the **@ARCHTEST** procedure:

```
@archtest(lags=1,form=lm) resids
```

You can also do a single **@ARCHTEST** to get a sensitivity table with tests for different lags by using the **SPAN** option. This, for instance, does test for ARCH for lags 1 to 6.

```
@archtest(lags=6,form=lm,span=1) resids
```

It's important to remember that if you reject the null, you are *not* concluding that an ARCH (or more likely GARCH) model will fit the data, just that the type of clustering of large residuals that is consistent with ARCH or GARCH is present. For more on ARCH and GARCH models, see Chapter 9.

3.6 Serial Correlation

The Hypotheses

In all cases considered, the null hypothesis is absence of serial correlation:

$$y = \mathbf{X}\beta + u \quad E[u] = 0, \quad E[uu'] = \mathbf{D} \text{ (diagonal)}$$

$\mathbf{D}$ is usually required to be scalar ($\sigma^2\mathbf{I}$) as well. Alternatives range from first order autoregressive errors to high order moving average errors.

Residual serial correlation is generally a much more serious issue than heteroscedasticity. The whole point of many models is to describe the dynamics of a process, so leaving serial correlation behind is often the sign of a flawed model.

Standard Tests

RATS instructions such as **LINREG** include the Durbin-Watson test for first order serial correlation. While it's routinely reported in regression results, the assumptions underlying it are rarely met in practice. It is biased towards acceptance of the null when there are lagged dependent variables in the regression, as well as for ARMA models. As such it's now used more as a rough guide, where values near 2 are OK, values far from it, not.

BOXJENK, **AR1** and **ITERATE** compute in addition the Ljung-Box Q for higher order serial correlation. The number of correlations used depend upon the sample size and are decided by RATS. If you would like to control this yourself, use the instructions or procedures below.

The Q Test

The Ljung-Box Q (1978) tests against high-order serial correlation. It has low power against specific alternatives, such as first order serial correlation, but can detect the more subtle problems which arise because of a misspecified ARMA model. It also can be applied in a wide range of circumstances, not just linear regressions.

BOXJENK includes Q tests for a fixed number of lags in the summary statistics. You can also use the **QSTATS** option on **CORRELATE** or **CROSS** to do Q tests on autocorrelations or cross-correlations. For instance, the **CORRELATE** here computes Q tests for lags 1 through 8, 1 through 16, and 1 through 24 (the **SPAN** option controls the gaps between tested numbers of lags).

```
boxjenk(ar=2,ma=1,diffs=1) y / resid
correlate(qstats,number=24,span=8,dfc=%NARMA) resid
```

The Q test has a known asymptotic distribution (under the null of no serial correlation) only when applied to the residuals from an ARMA model, where it is asymptotically χ^2 with degrees of freedom equal to the number of correlations minus the number of estimated ARMA parameters (which **BOXJENK** puts into the variable **%NARMA**). RATS does not correct the degrees of freedom in any other situation. You can include your own correction using the **DFC** option as is done above.

Chapter 3: Hypothesis Testing

A simpler approach is to use the procedure **@REGCORRS** to analyze the autocorrelations of residuals. This can produce either a graph or a report (or both) showing the correlations and Q tests. The syntax of this procedure is

```
@regcorrs ( options )      series of residuals
```

where the main options are

```
[graph]/nograph  
report/[noreport]  
method=yule/[burg]  
number=number of autocorrelations to compute  
dfc=degrees of freedom correction for the Q statistic
```

The advantage of the graph and/or detailed report over just a single Q test is that you can more easily determine if the diagnostics are showing any “correctable” serial correlation. Time series models can often be adjusted to largely eliminate residual serial correlation if it’s on short lags (1 or 2 for instance), or at and around seasonals. However, if an apparently significant Q is due mainly to a “spike” at a location which is hard to explain (lag 7 for instance), you may have to stick with the model you have and chalk up the significant Q to the behavior of this particular sample.

SCTEST.RPF example

SCTEST.RPF uses one of the series from West and Cho(1995). The change in the log of the Canadian/US exchange rate would not be expected to show serial correlation, so this tests that. Note that here there is no model to estimate. This does Q Tests using both **CORRELATE** and the **@REGCORRS** procedure.

```
correlate(qstats,number=40,span=10) xcan  
@regcorrs(number=40,graph,report,nocrit,$  
      title="Correlations of Can$ Exchange Rate") xcan
```

West-Cho Test

The Q test relies upon an assumption of conditional homoscedasticity to generate its asymptotics. West and Cho(1995) propose a similar test which uses a robust estimate of the fourth moments to deflate the correlations. (Interestingly, this test is basically a footnote in a paper on a different subject). This “robust” Q test can be done using the **@WestChoTest** procedure. Its principal option is the **NUMBER=***number of autocorrelations to compute*. The following is from SCTEST.RPF:

```
@westchotest(number=40) xcan
```

Cumulated Periodogram Test

This competes with the Q test as a test for general serial correlation. Because it uses frequency domain techniques, the technical details are covered in the description of **CACCUMULATE** in the *Reference Manual*. However, all you need to do is invoke the procedure **@CUMPDGM** to use the test. From **SCTEST.RPF**:

```
@cumpdgm xcan
```

@CUMPDGM reports a Kolmogorov–Smirnov statistic and generates a graph. The K–S test is a general test for whether an empirical distribution comes from a hypothesized distribution. **@CUMPDGM** prints (large-sample) critical values for the test.

General LM Tests (Breusch-Godfrey)

The Q , West-Cho and Cumulated Periodogram tests apply in a wide range of situations, but generally have fairly low power against specific low order alternatives. For diagnostics in a linear regression, Godfrey (1978) and Breusch (1978) have proposed a test against an alternative of serial correlation of order N , in either autoregressive or moving average form. This involves regressing the residuals on the original explanatory variables plus one or more lags of the residuals (here one):

```
linreg %resids
# constant logc{1} logy %resids{1}
cdf(title="Breusch-Godfrey SC Test") chisqr %trsqr 1
```

Durbin's h

The Durbin-Watson test is biased towards a finding of no serial correlation when the model contains a lagged dependent variable. Durbin (1970) proposed an alternative test for such situations, using the “ h -statistic.” While you *can* compute h using RATS (as shown below), we would strongly urge you to use the Breusch-Godfrey test (above), since it is simpler and more robust. (Durbin’s result is, in effect, the B-G test with terms eliminated by assumption).

```
linreg logc
# constant logc{1} logy
compute h=sqrt(%nobs)*%rho/sqrt(1-%nobs*%stderrs(2)^2)
cdf(title="Durbin h test") normal h
```

3.7 Granger-Sims Causality/Exogeneity Tests

Granger (1969) proposed a concept of “causality” based upon prediction error: X is said to *Granger-cause* Y if Y can be forecast better using past Y and past X than just past Y . Sims (1972) proved the following: in the distributed lag regression

$$(1) \quad Y_t = \sum_{j=-\infty}^{\infty} b_j X_{t-j} + u_t,$$

$b_j = 0$ for all $j < 0$ if and only if Y fails to Granger-cause X .

“Cause” is a loaded term, and many articles have been written about whether this concept is a proper definition of causality. Regardless, the test is important for several reasons:

- Certain theories predict absence of Granger causality from one variable to another. Such a theory can be rejected if causality is found.
- It is a specification test (Section 3.10) in distributed lag models such as (1). If the coefficients on future X are non-zero, then a one-sided Y on X regression is a poorly specified dynamic relationship. Also, if the test fails, attempts to correct for serial correlation in estimating a one-sided distributed lag are likely to produce inconsistent estimates.
- Its relationship to prediction is important in building good, small forecasting models.

Testing Methods

The two basic methods for causality testing are:

- The “Granger test” regresses Y on lagged Y and lagged X and tests lags of X .
- The “Sims test” regresses X on past, present and future Y , and tests leads of Y .

The example file `CAUSAL.RPF` looks at the GDP-M1 causality question. With both series in logs, an eight lag Granger test (for M1 causing GDP) can be done with

```
linreg gdph
# constant gdph{1 to 8} fm1{1 to 8}
exclude(title="Granger Causality Test")
# fm1{1 to 8}
```

A Sims test (following the procedure outlined in the paper) and using eight lags and testing four leads is done with

```
set filtm1 = fm1-1.50*fm1{1}+.5625*fm1{2}
set filtgdp = gdph-1.50*gdph{1}+.5625*gdph{2}
*
linreg filtm1
# constant filtgdp{-4 to 8}
exclude(title="Sims Causality Test")
# filtgdp{-4 to -1}
```

Note that the dependent variable and the tested variable are reversed between the two tests.

While the two testing procedures are equivalent theoretically, they are different in practice, because they must be estimated using finite parameterizations of the autoregression (for Granger) and distributed lag (for Sims), which do not directly correspond. Geweke, Meese and Dent (1982) examined several forms of causality tests and found that the Sims test was sensitive to failure to correct for serially correlated residuals. They proposed as an alternative a test using a two-sided distributed lag augmented with lagged dependent variables. Although the lag distribution on X is changed completely by the addition of the lagged dependent variables, the X coefficients are still one-sided under the null. This is also done on the original (logged here) variables rather than pre-filtered ones.

```
linreg fml
# constant gdph{-4 to 8} fml{1 to 8}
exclude(title="Geweke-Meese-Dent Causality Test")
# gdph{-4 to -1}
```

Causality Test and Unit Roots

If the series have unit roots (Section 3.11), the block zero restrictions in the Granger form are a type which has a non-standard distribution according to the results in Sims, Stock and Watson (1990) —you can't rearrange the regression so the block exclusion is strictly on differences. If you are willing to assume that the series have unit roots but *aren't* cointegrated, you can do the test on first differences, but two series can be cointegrated even if one fails to Granger cause the other, so that may not be an innocuous assumption. If it's important to do careful inference, you can bootstrap the test statistic: see example GRANGERBOOTSTRAP.RPF.

In the literature, there's an alternative procedure for testing "Granger causality" in the presence of unit roots by use of an augmented regression which adds an untested lag to the Granger test regression (generally known as the Toda-Yamamoto test). Unfortunately, despite having appeared in quite a few papers, this is statistically incorrect and should not be used. See the story in the June 2018 Newsletter.

Extensions and Non-Extensions

Certain testing procedures have been called "Granger causality" when they really aren't, or at least aren't an interesting null hypothesis. For instance, if you run an ARDL model (UG-59) with y on lags of y and lags *and current* x , a test of just the lags of x isn't a test for Granger non-causality and in practice isn't all that interesting.

Extensions to more than two variables (Section 7.4) are only interesting if they take an entire block of variables out of the system.

3.8 Structural Stability Tests or Constancy Tests

The Hypotheses

The null hypothesis here is one of structural stability: coefficients (and, typically, the variance) are the same over the sample. The alternatives can range from the precise, with a break at a known point in the sample, to the broad, where the coefficients aren't the same, but nothing more specific is assumed. If an estimated model fails to show stable coefficients, inferences using it, or forecasts generated from it, may be suspect. In general, the precise alternatives will generate standard types of tests, while the broad alternatives generate tests with non-standard distributions. And if there is, indeed, a rather sharp break, the more precise alternatives will offer higher power. But if there is a general misspecification error which leads to a more gradual change in the coefficients, a test of a hard break might not fare as well.

Tests with a known change point are known generically as “Chow tests”. There are several forms of this, depending upon the assumptions made. These are usually not very hard to implement (at least in linear models) since they require only estimating the model over two or three samples. See “Chow Tests” on page UG–87.

The tests with *unknown* change points usually require a fair bit of number-crunching, since every possible break in a range of entries must be examined. In most cases, you should rely upon a RATS procedure (such as `@STABTEST`, for the Hansen stability test) to do these. Some “tests” are actually not formal tests, but rather, are graphical displays of the behavior of some statistic, which are examined informally for evidence of a change in the model. For instance, while the individual tests in a series of sequential “*F*-tests” might have (approximately) *F* distributions, the maximum of them won't. These are useful mainly if the resulting statistics leave little room for doubt that the parameters aren't constant. Even if the asymptotic distribution for the most extreme *F* isn't known, if the statistic has a nominal *p*-value of .00001, it's probably a safe bet that the model has a break.

Tests based upon recursive residuals (Section 2.6) offer an even more general alternative than the unknown change point. Many tests can be constructed from the recursive residuals because they have the property that, under the null hypothesis, they are independent with constant variance. The behavior of such a process is easy to analyze, so a deviation from that will lead to a rejection of constancy.

Note well that these tests are specification tests for the original model—testing whether the single specification seems to be valid. They do *not* provide a usable alternative model. A rejection means the original model isn't stable. *Why* it isn't stable (missing variable? policy shift?) is a separate issue. The location of the “breaks” might be useful in determining the source, but not much more.

Structural Breaks Chapter

A more extensive description of tests for structural breaks and estimates of models with breaks is provided in Chapter 11.

Chow Tests

The term “Chow test” is typically applied to the test of structural breaks at known locations. Formally, in the model

$$y_t = X_t\beta_1 + u_t \quad t \in T_1$$

$$y_t = X_t\beta_2 + u_t \quad t \in T_2$$

$$\dots \quad \dots$$

$$y_t = X_t\beta_n + u_t \quad t \in T_n$$

the null is $\beta_1 = \beta_2 = \dots = \beta_n$. The general alternative is that this is not true, though it’s possible for the alternative to allow for some of the coefficients to be fixed across subsamples.

There are two ways to compute the test statistic for this:

1. Run regressions over each of the subsamples and over the full sample. The subsample regressions, in total, are the unrestricted “regression”, and the full sample regression is the restricted regression. Compute a standard F -statistic from the regression summary statistics. With more than two categories, it will probably be simplest to use the **SWEEP** instruction to do the calculations.
2. Run the regression over the full sample, using dummies times regressors for subsamples 2 to n . Test an exclusion restriction on all the dummies.

The first procedure is usually simpler, especially if there are quite a few regressors. However, it is only applicable if the model is estimated appropriately by ordinary least squares. You must use the second method if

- you need to correct the estimates for heteroscedasticity or autocorrelation (by using the **ROBUSTERRORS** option), or,
- you are using some form of instrumental variables, or
- you are allowing some coefficients to be constant across subsamples.

Both procedures require that you have enough data points in each partition of the data set to run a regression. An alternative (“Chow Predictive Test” on page UG–90) can be used when a subsample (usually at the end of the data set) is too short. In effect, this estimates the model holding back part of the sample, then compares that with the fit when the remainder of the sample is added in.

CHOWTEST.RPF example

The **CHOWTEST.RPF** example tests a regression for differences between large and small states. Large states are those with a population above 5,000 (in thousands). With this data set (and most cross section data sets), use the option **SMPL** to do the subsample regressions. With time series data sets, you can handle most sample splits with different *start* and *end* parameters on **LINREG**.

Chapter 3: Hypothesis Testing

The values of the variables %RSS and %NDF are used to calculate the test statistic. RSSLARGE, RSSSMALL and RSSPOOL are the residual sums of squares of the large state, small state and pooled regressions. Similar variables are defined for the degrees of freedom. The value for the unrestricted regression is the sum for the split samples.

```
linreg(smpl=pop<5000) pcexp
# constant pcaid pcinc
compute rsssmall=%rss , ndfsmall=%ndf
*
linreg(smpl=pop>=5000) pcexp
# constant pcaid pcinc
compute rsslarge=%rss , ndflarge=%ndf
*
linreg pcexp
# constant pcaid pcinc
compute rsspool=%rss
```

The test statistic is then computed and displayed with with

```
compute rssunr=rsssmall+rsslarge , ndfunr=ndfsmall+ndflarge
compute fstat = ( (rsspool-rssunr)/%nreg ) / (rssunr/ndfunr)
cdf(title="Chow test for difference in large vs small") $
fctest fstat %nreg ndfunr
```

While it might be overkill in this case, the **SWEEP** instruction is very useful when you need a regression which is split into more than two categories. To use it, you must create a series with different values for each category in the sample split:

```
set sizes = %if(pop<5000,1,2)
```

Then use **SWEEP** with the option `GROUP=category series`. The first supplementary card has the dependent variable (there could be more than one in a systems estimation) and the second has the explanatory variables. This will do a separate regression on each category. Save the covariance matrix of the residuals (in this case a 1×1 matrix, since there is only the one “target” variable):

```
sweep(group=sizes,cvout=cv)
# pcexp
# constant pcaid pcinc
```

The sum of squared residuals will be %NOBS times the 1,1 element of the covariance matrix. The number of explanatory variables is in %NREG, while the total number of regressors across categories is in %NREGSYSTEM so the unrestricted degrees of freedom is the number of observations less that.

```
compute rssunr=cv(1,1)*%nobs
compute ndfunr=%nobs-%nregsystem
compute fstat = ( (rsspool-rssunr)/%nreg ) / (rssunr/ndfunr)
cdf(title="Chow test for difference in large vs small") $
fctest fstat %nreg ndfunr
```


The Chow test with dummy variables is more generally applicable than the subsample regression method above, but the cost is a more complex procedure, especially if the number of regressors or number of subsamples is large. This is because a separate dummy must be constructed for each regressor in each subsample beyond the first. Thus, there are $(n-1) \times k$ dummies. This is not much of a problem here, since $n=2$ and $k=3$, but had we split the sample four ways, we would need nine **SETS**. For the more complicated cases, you would probably want to create a **VECTOR** or **RECTANGULAR** of **SERIES** to handle the various interaction terms.

Once the dummies (or more accurately, subsample dummies times regressors) are set up, the procedure is straightforward: estimate the model over the whole sample, including regressors and dummies, and test the joint significance of the dummies.

Because this example uses the **ROBUSTERRORS** option to correct the covariance matrix for heteroscedasticity, the test statistic will be reported as a χ^2 with three degrees of freedom. Without that option, it will give identical results to the calculations above.

LARGE is a dummy which is one for the large states. We then set up dummies for **PCAID** and **PCINC**:

```
set large = pop>=5000
*
set dpcaid = pcaid*large
set dpcinc = pcinc*large
```

This computes the regression with the original explanatory variables and the dummy and dummied-out regressors and tests those added variables:

```
linreg(robusterrors) pcexp
# constant pcaid pcinc large dpcaid dpcinc
exclude(title="Sample Split Test-Robust Standard Errors")
# large dpcaid dpcinc
```

Chapter 3: Hypothesis Testing

CONSTANT.RPF example

CHOWTEST.RPF uses a cross section data set. There are many fewer tests for stability for those than there are for time series data sets, such as the one used in this example. As a general rule, rejection of stability for a cross section model means that the model was somehow specified wrong—an incorrect functional form, or poor choice of a proxy. With time series data, it's quite possible that the model simply breaks down part way through the sample, due to changes in laws, technology, etc.

CONSTANT.RPF is based upon an example from Johnston and DiNardo (1997). It's a linear model using quarterly data from 1959:1 to 1973:3. The remainder of the section describes the tests used in it.

@STABTEST procedure

The first test in the example is Bruce Hansen's (1991) test for general parameter stability, which is a special case of Nyblom's (1989) stability test. This is based upon the behavior of partial sums of the regression's normal equations for the parameter and variance. For the full sample, those are zero, and (if the model is stable) the sequence of partial sums shouldn't stray too far from zero. The **@STABTEST** procedure generates test statistics for the overall regression (testing the joint constancy of the coefficients and the variance), as well as testing each coefficient and the variance individually. It also supplies approximate *p*-values. **@STABTEST** both estimates the linear regression and does the test, so it includes the full specification just like a **LINREG**:

```
@stabtest y 1959:1 1973:3
# constant x2 x3
```

Chow Predictive Test

The Chow predictive test can be used when a subsample is too short to produce a sensible estimate on its own. It's particularly useful for seeing whether a small "hold-back" sample near the end of the data seems to be consistent with the estimate from the earlier part. In **CONSTANT.RPF**, the regression is run over the sample through 1971:3, then again through 1973:3. The difference between the sums of squares divided by the number of added data points (8) forms (under the null) an estimate of the variance of the regression that's independent of the one formed from the first subsample, and thus it generates an *F*. Note that, in this case, there *is* enough data to do a separate regression on the second subsample. However, it's a very short subsample, and the standard Chow test would likely have relatively little power as a result.

```
linreg(noprint) y 1959:1 1971:3
# constant x2 x3
compute rss1=%rss,ndf1=%ndf
linreg(noprint) y 1959:1 1973:3
# constant x2 x3
compute f=((%rss-rss1)/8)/(rss1/ndf1)
cdf(title="Chow Predictive Test") ftest f 8 ndf1
```

Tests Based Upon Recursive Residuals

As mentioned in Section 2.6, if the model is in fact, correctly specified with i.i.d. $N(0, \sigma^2)$ errors, then the recursive residuals produced by the **RLS** instruction are i.i.d. (Normal), while standard regression residuals have at least some in-sample correlation by construction. There are quite a few tests that can be used to test the null that the recursive residuals are i.i.d., and the failure of those tests can be seen as a rejection of the underlying assumptions. The following instruction does the recursive estimation and saves quite a few of the statistics generated:

```
rls (sehist=sehist, cohists=cohists, sighist=sighist, $
    csum=cusum, csquared=cusumsq) y 1959:1 1973:3 rresids
# constant x2 x3
```

The recursive residuals are (by construction) zero at the start of the estimation range to the point where there are just enough data points to estimate the model, which (barring a problem in the explanatory variables) will be the number of regressors. For convenience, **RLS** actually makes them missing values. Thus, the first usable residual will be at the start of the estimation range plus the number of regressors. That's computed by this into the variable **RSTART**:

```
compute rstart=%regstart()+%nreg
```

Next is a graph of the recursive residuals with the (recursively estimated) standard error bands. This doesn't form a formal test; however, if there is a break, it's likely that the residuals will, for a time, lie outside the bands until the coefficients or variance estimates adjust.

```
set lower = -2*sighist
set upper = 2*sighist
graph(header="Recursive Residuals and Standard Error Bands") 3
# rresids
# lower / 2
# upper / 2
```

Next is the more formal CUSUM test (Brown, Durbin and Evans, 1975). Under the null, the cumulated sums of the recursive residuals should act like a random walk. If there is a structural break, they will tend to drift above the bounding lines, which here are set for the .05 level.

```
set cusum = cusum/sqrt(%seesq)
set upper5 rstart 1973:3 = .948*sqrt(%ndf)*$
    (1+2.0*(t-rstart+1)/%ndf)
set lower5 rstart 1973:3 = -upper5
graph(header="CUSUM test") 3
# cusum
# lower5 / 2
# upper5 / 2
```

Chapter 3: Hypothesis Testing

This can be done more simply with the **@CUSUMTESTS** procedure, which does both the CUSUM test and the CUSUMQ test (for the square). The CUSUMQ test is mainly aimed at testing stability of the variance. **@CUSUMTESTS** takes the recursive residuals as the input series:

```
@cusumtests rresids
```

The final graph is a set of one-step Chow predictive F -tests. This is basically the same information as the recursive residuals graph with a different presentation. Again, this is an informal test. This graph is designed to present not the sequential F 's themselves, but the F 's scaled by the .05 critical value. At the start of the sample, the F 's are based upon very few denominator degrees of freedom, so F 's that are quite large may very well be insignificant. Anything above the "1" line (shown on the graph with the help of the **VGRID** option) is, individually, statistically significant at the .05 level.

```
set seqf = (t-rstart)*(cusumsq-cusumsq{1})/cusumsq{1}  
set seqfcval rstart+1 * = seqf/%invftest(.05,1,t-rstart)  
graph(vgrid=||1.0||,header=$  
    "Sequential F-Tests as Ratio to .05 Critical Value")  
# seqfcval
```

3.9 Functional Form Tests

Introduction

This section describes tests which are designed to determine if the functional form for a regression equation is adequate. For instance, should an equation be estimated in logs or levels? Should it be linear in the variables, or should a higher power, like a quadratic, be used? Note that some of these procedures are “non-nested”, and so can result in rejecting both of the possibilities, or accepting both.

RESET Tests

RESET stands for Regression Equation Specification Error Test. These were proposed by Ramsey (1969). If the regression equation

$$y_t = \mathbf{X}_t\beta + u_t$$

is specified correctly, then adding non-linear functions (and, in particular, powers) of the fitted values $\mathbf{X}_t\hat{\beta}$ should not improve the fit significantly. This is designed to test whether in

$$y_t = f(\mathbf{X}_t\beta) + u_t$$

the “ f ” function is just $f(x) = x$. However, it can pick up other types of failures.

You can get the fitted values using the instruction **PRJ**. The test is carried out by generating the needed powers of the fitted values, and running an auxiliary regression with the original specification plus the powers. Test an exclusion on the higher powers. The following, from an example in Verbeek (2008), does a cubic RESET.

```
linreg lprice
# constant llot bedrooms bathrms airco
prj fitted
set fit2 = fitted^2
set fit3 = fitted^3
linreg lprice
# constant llot bedrooms bathrms airco fit2 fit3
exclude(title="RESET test with quadratic and cubic")
# fit2 fit3
```

In practice, you’ll find it easier to use the **@RegRESET (h=power)** procedure. Use this immediately after the regression that you want to test:

```
linreg lprice
# constant llot bedrooms bathrms airco
@RegRESET (h=3)
```

Note that RESET tests can run into numerical problems if you get above a 3rd degree test because of high collinearity among the powers. **@RegRESET** offers a **PC** option to reduce the test to a selected number of principal components of the powers.

MacKinnon-White-Davidson Test

Davidson and MacKinnon (1981) and MacKinnon, White and Davidson (1983) proposed a method for testing the functional form for the *dependent* variable. The most common use of this would be for determining whether levels or log is more appropriate. This is a non-nested test, so it's possible to reject each in favor of the other, or to fail to reject either. The test procedure is to add the difference between the levels fit and the exponentiated log fit to the log equation, and to add the difference between the log fit and the log of the levels fit to the levels equation. The added coefficient should be insignificant if the original equation was properly specified. You can just look at the *t*-statistic on it.

```
linreg m1
# constant tbilrate realgdp
prj linearfit
set logm1 = log(m1)
set logy  = log(realgdp)
set logr  = log(tbilrate)

linreg logm1
# constant logr logy
prj logfit
set loggap = logfit-log(linearfit)
set lingap = linearfit-exp(logfit)

linreg logm1
# constant logr logy lingap
linreg m1
# constant tbilrate realgdp loggap
```

3.10 Specification Tests

Introduction

Specification test is a very general term which really can cover many of the tests described in previous sections. The one absolute requirement for performing a specification test is this:

The model must incorporate more assumptions than are required to estimate its free coefficients.

For instance, if our model, in total, consists of

$$y_t = X_t\beta + u_t \quad E(X_t u_t) = 0, \beta \text{ unknown}$$

we have nothing to test, because the least squares residuals will be, by construction, uncorrelated with the X 's. We get specification tests when we go beyond these assumptions:

- If we assume, in addition, that the u 's are homoscedastic or serially uncorrelated, we can test those additional assumptions (Section 3.5 and Section 3.6), because we don't use them in estimating β by least squares (though we may rely on them for computing the covariance matrix).
- If we assume, in addition, that $E(u_t | X_t) = 0$, we can test whether various other functions of X are uncorrelated with the residuals. The RESET test (page UG–93) is an example of this.
- If we assume, in addition, that $E(X_s u_t) = 0$ for all t, s (strict econometric exogeneity), we get, in the distributed lag case, Sims' exogeneity test (page UG–84), which tests whether or not u_t is uncorrelated with X_{t+j} .

Two basic strategies are available:

- Compare estimates of β computed with and without the additional assumptions. If the additional assumptions are true, the estimates should be similar. This gives rise to "Hausman tests" (Hausman, 1978).
- If the assumptions can be written as orthogonality conditions (u_t uncorrelated with a set of variables), seeing whether or not these hold in sample is a test of the overidentifying restrictions. The general result for this style of test is given by Hansen (1982).

Hausman Tests

Hausman tests operate by comparing two estimates of β , one computed making use of all the assumptions, another using more limited information. If the model is correctly specified, $\hat{\beta}_1 - \hat{\beta}_2$ should be close to zero. The difficult part of this, in general, is that the covariance matrix of $\hat{\beta}_1 - \hat{\beta}_2$ is not the covariance matrix of either one alone, it is

$$(2) \quad \text{Var}(\hat{\beta}_1) + \text{Var}(\hat{\beta}_2) - 2 \text{Cov}(\hat{\beta}_1, \hat{\beta}_2)$$

Chapter 3: Hypothesis Testing

and computing the covariance term is quite unappealing. However, in Hausman's (1978) settings, the covariance matrix of the difference simplifies to

$$(3) \quad \text{Var}(\hat{\beta}_1) - \text{Var}(\hat{\beta}_2)$$

where $\hat{\beta}_1$ is the less efficient estimator. This result is valid only when $\hat{\beta}_2$ is efficient and $\hat{\beta}_1$ is based upon the same or a smaller information set.

Note that it is *quite* difficult to do a Hausman test properly using the direct comparison of estimators. To get the correct behavior any “nuisance parameters” such as residual variances need to match up in the covariance estimators in (3), which is usually *not* what you will get by using standard estimators, since each will come up with its own estimator for those. In many cases the Hausman test is identical to a test that is done using an auxiliary regression, and when that's the case, it's usually better to use the alternative method.

Procedure

If you would like to perform a Hausman test by the direct comparison of estimators, follow these steps:

1. Estimate the model using one of the two estimators. (The order only matters if the nuisance parameters from one are needed to do the *estimates* on the other.) Save its coefficient vector (%BETA) and covariance matrix (%XX) into other matrices and save the nuisance parameters if needed later.
2. Estimate the model using the second estimator. Figure out what you have to do to get the comparable estimators for the covariance matrix. Compute the difference between the two properly estimated covariance matrices (less efficient less more efficient).
3. Use **TEST** with the options `ALL` and `VECTOR=saved coefficients` and `COVMAT=difference in covariance`. This tests for equality between the two estimated coefficient vectors. The covariance matrix of the difference will probably not be full rank. However, **TEST** will determine the proper degrees of freedom.

To look at a simple example, suppose

$$(4) \quad c_t = \alpha_0 + \alpha_1 y_t + u_t$$

$$(5) \quad E(u_t) = E(c_{t-1} u_t) = E(y_{t-1} u_t) = 0$$

$$(6) \quad E(y_t u_t) = 0$$

If we take (4) and (5) as maintained hypotheses, we can estimate the equation consistently by instrumental variables. If (6) is also valid, OLS will be efficient. This test is done in the example file `HAUSMAN.RPF`. The first thing to note about this is that the natural estimation range for least squares will have one additional data point compared with instrumental variables, as the latter needs lags of c and y . So the range

has to be controlled to make the estimators match. This estimates the model by least squares and saves a copy of %XX, %BETA and (maximum likelihood) estimate of the residual variance.

```
linreg realcons 1950:2 *
# constant realgdp
compute xxols=%xx,betaols=%beta,sigsqols=%sigmasq
```

Next up is the instrumental variables estimator:

```
instruments constant realgdp{1} realcons{1}
linreg(inst) realcons
# constant realgdp
compute xxiv=%xx,betaiv=%beta
```

With these options, both **LINREG** instructions produce an %XX which needs to be multiplied by an estimate of the residual variance to get the covariance matrix. We need to pick one of those to be the common scaling for the covariance matrix of the difference. Here, we will use the SIGSQOLS from least squares. Different choices for this will yield (slightly) different test results. The **FORM=CHISQUARED** option is needed on the **TEST** because multiplying by SIGSQOLS converts it from an “F” form to a chi-squared form (Section 3.3). We show the output below the instruction:

```
test(covmat=sigsqols*(xxiv-xxols),vector=betaols,all,$
    form=chisquared,title="Hausman Test")
```

```
Null Hypothesis : The Following Coefficients Are Zero
Constant
REALGDP
## X13. Redundant Restrictions. Using 1 Degrees, not 2
Chi-Squared(1)=      22.111856 with Significance Level 0.00000257
```

Note that the degrees of freedom are adjusted to 1. Because the **CONSTANT** is in both the regression and the instrument set, if the slope coefficients are equal the intercepts have to be as well, so the test is really only on the slope. In practice, it's rare for a Hausman test to *have* full rank since some of the assumptions tend to be common between the two models.

In many cases (such as this one), the Hausman test can be computed more easily with an auxiliary regression. In this case, it's the Wu(1973) test, which adds the fitted value from projecting the endogenous explanatory variable onto the instruments to the linear regression and testing the fitted value:

```
linreg realgdp
# constant realgdp{1} realcons{1}
prj yfit
linreg realcons
# constant realgdp yfit
exclude(title="Wu test")
# yfit
```

Chapter 3: Hypothesis Testing

And this latter test can be done even more simply using the `@RegWuTest` procedure immediately after running the instrumental variables regression:

```
instruments constant realgdp{1} realcons{1}
linreg realcons 1950:2 *
# constant realgdp
@RegWuTest
```

“Hansen” Tests

By limiting ourselves to a notation appropriate for those models which can be estimated using single-equation methods, we can demonstrate more easily how these tests work. Assume

$$(7) \quad y_t = f(X_t, \beta) + u_t$$

$$(8) \quad E(Z'_t u_t) = 0$$

with some required regularity conditions on differentiability and moments of the processes. f will just be $X_t\beta$ for a linear regression. The Z 's are the instruments.

If Z is the same dimension as β , the model is just identified, and we can test nothing. Hansen's key testing result is that

$$(9) \quad \mathbf{u}' \mathbf{Z} \mathbf{W} \mathbf{Z}' \mathbf{u} \sim \chi^2$$

when the weighting matrix $\mathbf{W}$ is chosen “optimally.” The degrees of freedom is the difference between the number of orthogonality conditions in (8) and the number of parameters in the β vector.

This test is automatically included in the output from **LINREG**, **AR1**, **NLLS**, **NLSYSTEM** and **SUR** as the *J*-Specification. This generates output as shown below, with the first line showing the degrees of freedom (the difference between the number of conditions in (8) and the number of estimated coefficients) and the test statistic, and the second line showing the marginal significance level:

J-Specification (4)	7.100744
Significance Level of J	0.13065919

These results would indicate that the null hypothesis (that the overidentifying restrictions are valid) can be accepted.

The computed test statistic can be obtained as `%JSTAT`, its significance level as `%JSIGNIF` and the degrees of freedom as `%JDF`. $\mathbf{u}' \mathbf{Z} \mathbf{W} \mathbf{Z}' \mathbf{u}$ is available as `%UZWZU`.

If the weighting matrix used in estimation isn't the optimal choice, there are two ways to “robustify” the test. The first is described in Hansen (1982)—it computes an adjusted covariance matrix $\mathbf{A}$ for $\mathbf{Z}'\mathbf{u}$ so that $\mathbf{u}' \mathbf{Z} \mathbf{A}^{-1} \mathbf{Z}' \mathbf{u} \sim \chi^2$. The other is described in Jagannathan and Wang (1996). It uses $\mathbf{u}' \mathbf{Z} \mathbf{W} \mathbf{Z}' \mathbf{u}$ as the test statistic, but they show that it has an asymptotic distribution which is that of a more general qua-

dratic form in independent Normals. You can choose between these with the option `JROBUST=STATISTIC` (for the first) or `JROBUST=DISTRIBUTION` (for the second).

See page UG–138 for more information.

HANSEN.RPF example

`HANSEN.RPF` does two Hansen J -tests for the consumption equation in Klein's model I. The first is the simplified version for two-stage least squares. The J -statistic is included in the regression output:

```
instruments constant trend govtwage govtexp taxes $
  profit{1} capital{1} prod{1}
linreg(inst) cons / resids
# constant profit{0 1} wagebill
```

You can also do a form of the test by regressing the residuals from the IV estimator on the full instrument set:

```
linreg resids
# constant trend govtwage govtexp taxes $
  profit{1} capital{1} prod{1}
cdf chisqr %trsqr 4
```

The two test statistics are slightly different because the J -statistic produced by the first `LINREG` uses an estimate of the residual variance corrected for degrees of freedom

The second uses the GMM optimal weights (page UG–138) allowing for one lag of autocorrelation.

```
linreg(inst,optimal,lags=1,lwindow=newey) cons
# constant profit{0 1} wagebill
```

This will produce the identical results to the included J -test

3.11 Unit Root Tests

Much of applied macroeconometrics in the past thirty years has been devoted to an analysis of the implications of “unit root” and related behaviors in time series data. The econometrics governing the tests is technically difficult, with a large body of published papers discussing the proper methods. Because of this, we have chosen not to code specific tests directly into RATS. Instead, we rely upon procedures, which can more easily be adapted to changes in current practice. We will discuss here procedures to do the *basic* forms of the tests.

The *Time Series—Unit Root Tests* wizard offers a simple way to access the most important of these. Among the tests listed there are those that allow for structural breaks—those are covered in Section 11.6. There are also tests using panel data (page UG–423) rather than just a single time series.

For additional technical details on unit root testing, see the *RATS Programming Manual* by Enders and Doan, which is a PDF document included with RATS.

Before you run unit root tests on your data, you might want to ask yourself why you are doing it. There have been many papers published that have included unit root tests even though there was nothing in the paper that actually depended upon the result of the tests. Worse than pointless unit root tests, we have had more than a few users who are almost paralyzed by the (incorrect) idea that you can’t run regressions when some of the variables are “*I(1)*”. It’s true that *static* regressions (ones without lags) on series involving unit roots have a good chance of being “spurious” (Granger and Newbold, 1974), but regressions that have proper handling of dynamics (lags) such as vector autoregressions (Chapter 7) and ARDL’s (page UG–59) are generally fine. See “Spurious Regressions” in the help for more.

Dickey-Fuller Test

The most common choice in published empirical work nowadays is the Dickey-Fuller test (from Fuller, 1976 and Dickey and Fuller, 1979) with empirically chosen augmenting lag length. This can be done using the procedure **@DFUNIT**:

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

series Series to be tested for unit roots.

start end Range to use in testing. By default, the full range of the series.

The main options are:

```
det=none/[constant]/trend
```

This chooses the deterministic parts to include in the model.

lags=number of additional lags [0]
maxlags=maximum number of additional lags to consider
method=[input]/aic/bic/hq/ttest/gtos
signif=cutoff significance level for METHOD=TTEST or GTOS[.10]

These select the method for deciding the number of additional lags for an “augmented Dickey-Fuller test”, which is the number of lags on the difference to include in the regression to handle the shorter-term dynamics. If METHOD=INPUT, the number of lags given by the MAXLAGS (or LAGS) option is used. If AIC, the AIC-minimizing value between 0 and MAXLAGS is used; if BIC, it’s the BIC-minimizing value, and if TTEST or GTOS, the number of lags for which the last included lag has a marginal significance level less than the cutoff given by the SIGNIF option. Which you should use is largely a matter of taste, though AIC and TTEST are probably the most common choices. See Section 2.9 for more on the information criteria.

The augmented test has the same asymptotic distribution under the null as the standard Dickey–Fuller test does in the AR(1) case. Note that some authors describe the number of lags in a Dickey-Fuller test as the number of lags in the AR, not (as is done here) by the number of *additional* lags on the difference. If that’s the case, their “3” lag test would be done using LAGS=2.

UNITROOT.RPF example

For examples, UNITROOT.RPF looks at the two series examined in Chapter 17 of Hamilton (1994). One is the U.S. 3-month Treasury bill rate, which is assumed not to be trending, and the other is log GNP, where a trend is assumed. While, for illustration, this demonstrates all the unit root tests that we cover in this section, in practice you would do either one or at most two, unless you get conflicting results.

The Dickey-Fuller tests for T-bills with several choices for handling the augmenting lags are:

```
@dfunit(det=constant) tbill  
@dfunit(det=constant,maxlags=6,method=gtos) tbill  
@dfunit(det=constant,maxlags=6,method=aic) tbill
```

In this case, both GTOS and AIC pick the full set of six lags, which might indicate that a somewhat longer lag length might be in order. The similar treatment for log GNP uses the DET=TREND option:

```
@dfunit(det=trend) lgnp  
@dfunit(det=trend,maxlags=6,method=gtos) lgnp  
@dfunit(det=trend,maxlags=6,method=aic) lgnp
```

Here both end up picking 2 augmenting lags.

Other Tests

There are two obvious problems with the standard Dickey-Fuller tests:

1. The test depends upon the “nuisance” parameter of the extra lags to remove serial correlation.
2. The deterministics change their meanings as the model moves between the null (unit root) and the alternative (stationary). For instance, under the unit root, the constant is a drift rate, while it determines the mean for a stationary process.

The *Phillips-Perron* test (from Phillips, 1987 and Phillips and Perron, 1988) is similar to the Dickey-Fuller, but uses a non-parametric correction for the short-term serial correlation. The Dickey-Fuller and Phillips-Perron test each tend to exhibit rather poor behavior in the presence of certain types of serial correlation—see the Monte Carlo analysis in Schwert (1989). However, the types of serial correlation for which the PP test does poorly are much more common than the ones for which it does well, so it is much less commonly used now than DF. In RATS, the Phillips-Perron test can be done using the **@PPUNIT** procedure. The examples on `UNITROOT.RPF` are:

```
@ppunit(det=constant,lags=12,table) tbill  
@ppunit(det=trend,lags=12,table) lgnp
```

The `TABLE` option displays a sensitivity table which shows how the test statistic depends upon the lag length in the window for the non-parametric correction. There really isn’t the same collection of methods for objectively choosing the non-parametric lag length, so this allows you to see if the decision is sensitive to the choice. The statistics tend to stabilize once the lag window is long enough to handle the bulk of the serial correlation.

There have been several different approaches to dealing with the second issue. One is to replace the Dickey-Fuller “Wald” test with a Lagrange multiplier test. This is done with the *Schmidt-Phillips test* (Schmidt and Phillips 1992) which is executed using the **@SPUNIT** procedure. As with the Phillips-Perron test, this deals with the short-term serial correlation using non-parametric methods. **@SPUNIT** uses the `P` option (for the power of time) rather than the `DET` option used by the other tests as they allow for higher powers of time than just a linear trend (though there is little call for quadratic and above). The examples are:

```
@spunit(p=0,lags=12) tbill  
@spunit(p=1,lags=12) lgnp
```

A similar idea is to improve the estimate of the trend using GLS. Probably the most popular form is the test developed in Elliott, Rothenberg and Stock(1996). Since first differencing the data is inappropriate if the data are trend-stationary, they quasi-difference the data using a filter which is “local to unity” (close to a unit root, but not quite), and use that to estimate the trend. The detrended filtered data are then subjected to a Dickey-Fuller test (without any deterministics). The procedure for do-

ing this is **@ERSTEST**. There is little to be gained from using this for a non-trending series, so the example just applies it to GNP:

```
@erstest(det=trend,lags=12) lgnp
```

All of the tests described so far have had the unit root as the null. This makes sense since it's the “simple” hypothesis while the alternative of stationarity is composite. However, it is possible to construct a test with a null of stationarity—this is shown in Kwiatkowski, et. al(1992). The variance of the deviations from trend for a stationary process is bounded, while it's unbounded for a non-stationary process so if the process wanders too far to be compatible with stationarity, we conclude that it's non-stationary. This can be done using the **@KPSS** procedure. As with the Phillips-Perron and Schmidt-Phillips tests, this requires a lag window estimator. Note that the hypothesis is reversed, so if you do both KPSS and one of the other tests, you would hope to get opposite results regarding acceptance of the null.

```
@kpss(det=constant,lmax=12) tbill  
@kpss(det=trend,lmax=12) lgnp
```

Bayesian Tests

The tests above are all “classical” tests. A very different procedure implements the Bayesian odds ratio test proposed by Sims (1988). This is the procedure **@BAYESTST**. This is more of an intellectual curiosity because it doesn't allow for deterministic components and so really has little practical use.

3.12 Cointegration Tests

If we take two series, each of which has a unit root, then, in general, any linear combination of them will also have a unit root. However, it is *possible* for there to exist a linear combination which is stationary instead. Such series are said to be *cointegrated* (Engle and Granger, 1987). The textbooks by Hamilton (1994) and Enders (2014) are good references on the subject. Of these, Hamilton is the more theoretical; Enders the more applied. For a more complete discussion, Juselius(2006) is a monograph on the subject which ties directly into the CATS add-on.

Cointegration is a property of two or more series. Some examples where cointegration might possibly be seen are

$$(10) \quad \log Y_t - \log C_t \quad \text{consumption proportional to income}$$

$$(11) \quad \log P_t - \log S_t - \log P_t^* \quad \text{purchasing power parity (P's are prices, S exchange rate)}$$

In neither case would we expect these linear combinations to be *exactly* constant. However, if the variables are cointegrated, we would expect these residuals to stay close to a fixed value if observed over a long span of time. If there were still a unit root in the residuals, we would expect to see them wandering farther away as the sample size increased.

Note that before a set of series can be *COintegrated*, they must be (individually) *integrated* ($I(1)$). This rather important point has been missed by people trying to get results quickly, but not carefully. If one of the input series is stationary, rather than $I(1)$, you will almost certainly find “cointegration”, because $1 \times$ the stationary series $+ 0 \times$ the others is a stationary linear combination. There is nothing wrong with including both integrated and stationary variables in an analysis—you just have to understand how they interact when you look at cointegrating rank statistics.

As with unit root tests, there are quite a few cointegration testing procedures available on the Estima web site (www.estima.com). As with unit root tests, we are somewhat skeptical of the value of much of the empirical work done on cointegration. For analyzing joint behavior of a set of time series in greater detail, we would recommend the program CATS by Dennis, Hansen and Juselius (available separately from Estima). CATS adopts a structured approach to cointegration, emphasizing the testing of specific restrictions. See page UG–107 for more information.

Most of the testing procedures are based upon a two-step procedure of estimating a cointegrating vector and then performing a stationarity test on the residuals. The result underlying this is that if two variables X_{1t} and X_{2t} are cointegrated, then if the regression

$$(12) \quad X_{1t} = \alpha X_{2t} + u_t$$

is run, the estimate of α is not only consistent (even though it disregards possible simultaneous equations bias), but is “superconsistent,” converging to the true value at a faster rate than for a typical regression coefficient (Stock, 1987). But what happens if they *aren't* cointegrated?

Then the estimated coefficient in (12) is likely to be garbage; this is the “spurious regression” of Granger and Newbold (1974). It’s one thing to test a restriction like (10) and (11) that is rooted in economic theory. It’s quite another to blindly estimate a “cointegrating vector” and to rely upon asymptotic distribution theory to save us from an incorrect inference. If, for instance, we were to estimate the coefficients in (11) and came up with

$$(13) \log P_t - .05 \log S_t - .50 \log P_t^*$$

and we were able to reject non-cointegration, would we seriously believe that (13) represents some type of long-term equilibrium condition? We would be on much safer ground reaching the conclusion that this was due to sampling error. After all, we are attempting to infer very long-run behavior of a set of series from a finite sample.

Testing a Known Cointegrating Vector

The simplest test for a *known* cointegrating vector is a standard unit root test (Section 3.11) applied to the residual from the hypothesized linear combination of the variables. The test has the same asymptotic distribution as when applied to any other observed series.

Example file COINTTST.RPF is adapted from Hamilton. It examines whether the Purchasing Power Parity (PPP) restriction (11) is a cointegrating vector for Italian and U.S. data. The first step is testing whether the series are $I(1)$ with Dickey-Fuller tests (all three pass):

```
@dfunit(lags=12,trend) uscpi
@dfunit(lags=12,trend) italcpi
@dfunit(lags=12,trend) exrat
```

The test for PPP being a cointegrating relation is done with:

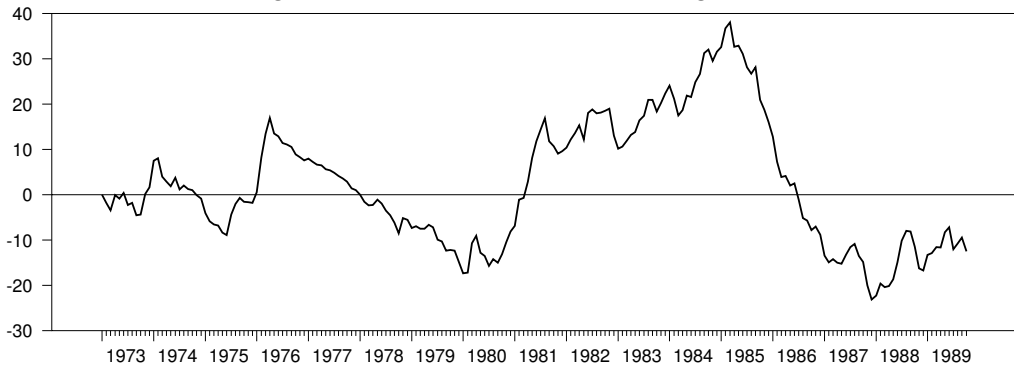
```
set ppp = uscpi-exrat-italcpi
@dfunit(lags=12) ppp
```

If the variables are cointegrated and we have used the correct cointegrating vector, then this series should *fail* a unit root test. As this ends up accepting the unit root, we reject the null hypothesis of cointegration—the deviations from PPP are *not* stationary (which seems clear from the graph at the top of the next page).

Testing an Unknown Cointegrating Vector

The simplest testing procedure for cointegration with an *unknown* cointegrating vector is to apply a unit root test to the residuals from a regression involving the variables. This is the *Engle-Granger test*. Since the sum of squares of a non-stationary linear combination should be quite a bit higher than those for a stationary linear combination, we would expect that least squares would zero in on a stationary linear combination if it exists. Thus it’s even more important in this case to make sure the input variables are themselves $I(1)$. Because the coefficients are now estimated, the

Figure 19.3 The real dollar-lira exchange rate



critical values for the unit root test are different and get more negative the more variables we include in the cointegrating regression. There are two procedures for doing this: `@EGTestResids` takes as input the residuals from the cointegrating regression. `@EGTest` takes the set of variables and does the preliminary regression itself.

`@EGTEST` has most of the same options as `@DFUNIT` but because the number of endogenous variables isn't fixed, they are input to the procedure using a supplementary card. Although it's unlikely to be interesting in this case (for the reasons described above), the Engle-Granger test can be applied to the three series in the example with something like:

```
@egtest(lags=12)
# uscpi italcpi exrat
```

An alternative to the regression-based tests is the likelihood ratio approach, which is the basis for the CATS software. The likelihood approach allows for testing sequentially for the rank of cointegration from 0 (no cointegration, N separate stochastic trends) up to N , which would mean no unit roots. See the discussion in Hamilton or Juselius.

The procedure `@JOHMLE` does the basic Johansen likelihood ratio test. In our example:

```
@johmle(lags=6,det=constant)
# uscpi italcpi exrat
```

DET=CONSTANT is appropriate for this procedure because that means a constant in each equation outside the cointegrating vector, which allows for trends in the variables.

The *Time Series—Cointegration Test* operation provides easy access to several of these testing procedures.

3.12.1 CATS Cointegration Analysis Procedures

As noted earlier, if you need to do any kind of comprehensive cointegration testing and analysis, we recommend that you use the CATS (cointegration analysis of time series) package for RATS (available separately from Estima).

CATS is a large collection of RATS procedures, which are invoked by a single call to the main **@CATS** procedure. Once loaded into memory, CATS provides an extensive suite of hypothesis testing and analysis tools via a highly interactive menu- and dialog-driven interface (with the option of batch-mode analysis as well). These include operations for selecting lag lengths, selecting and testing the choice of cointegration rank, testing a wide variety of restriction hypotheses, generating various types of graphical analysis, exploring $I(2)$ models, and much more. You can also export the estimated model from CATS for further analysis in RATS.

A Look at CATS

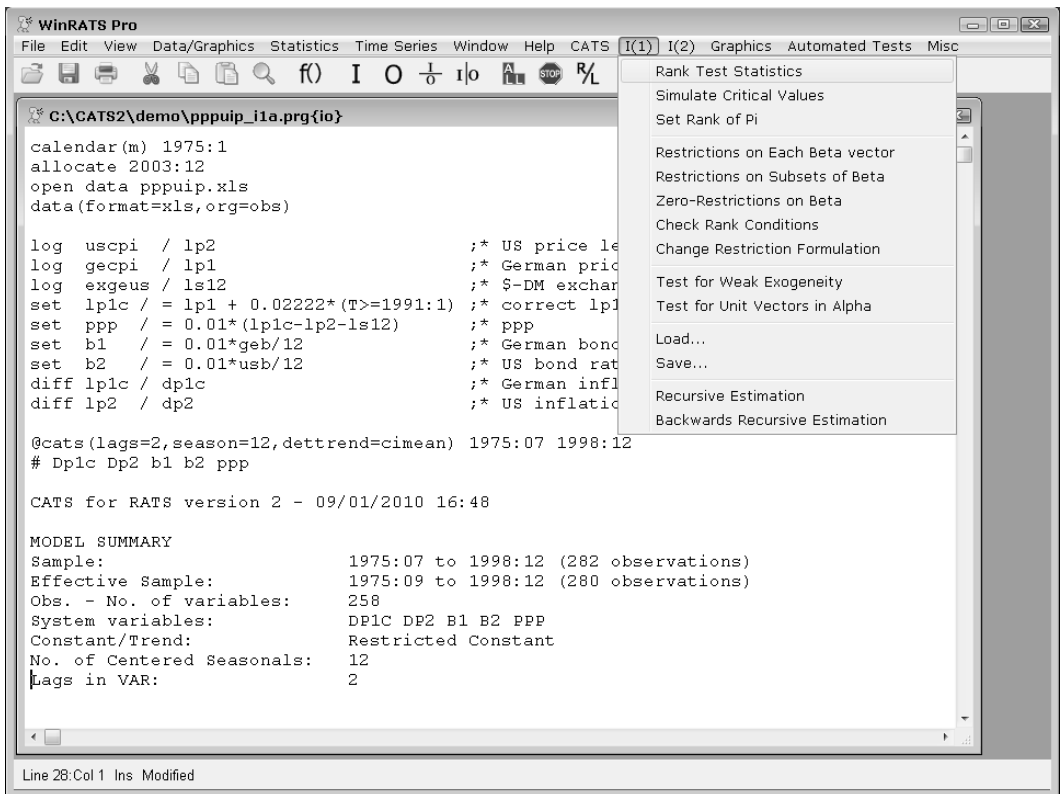
Here is one of the sample programs included with CATS. As you can see, this reads in some data series, does some data transformations, and executes the **CATS** procedure, specifying the VAR model to be analyzed (CATS automatically transforms the model into error correction form).

```
calendar(m) 1975:1
allocate 2003:12
open data pppuip.xls
data(format=xls,org=obs)

log uscpi / lp2          ;* US price level
log gecpi / lp1          ;* German price level
log exgeus / ls12        ;* $-DM exchange rate
set lp1c / = lp1 + 0.02222*(T>=1991:1) ;* post-unification fix
set ppp / = 0.01*(lp1c-lp2-ls12)        ;* purchasing power parity
set b1 / = 0.01*geb/12      ;* German bond rate
set b2 / = 0.01*usb/12     ;* US bond rate
diff lp1c / dp1c          ;* German inflation
diff lp2 / dp2            ;* US inflation

@cats(lags=2,season=12,dettrend=cimean)
# Dp1c Dp2 b1 b2 ppp
```

Here's what the screen looks like after CATS completes its initial set of computations and adds six new menus to the menu bar. We've opened the I(1) menu, which contains the primary operations for setting the cointegration rank and testing various restrictions on the model.



CATS Cointegration Wizard

There is a wizard on the Time Series menu which will help you set up the call to the CATS procedure.

Cointegration Textbook

For a comprehensive treatment of cointegration analysis, we recommend *The Cointegrated VAR Model: Methodology and Applications*, by Katarina Juselius (Oxford University Press, 2006).

CATS version 2 was developed in conjunction with the writing of the textbook, so the book will be of particular interest to anyone using CATS, but anyone interested in the topic of cointegration (or in sound principles for econometric analysis) should find it valuable.

A PDF “workbook” and example code showing how to implement many of the examples from the textbook are available on our website.

4. Non-Linear Estimation

RATS offers a range of instructions for estimating non-linear models. Which should be used depends upon the underlying assumptions. The first half of the chapter describes the elements that the estimation instructions have in common: the optimization techniques and the methods of describing the model. The optimization algorithms are also used for special purpose instructions for estimating ARCH and GARCH models (Chapter 9), state-space models (Chapter 10) and probit and related models (Chapter 12). If you need more information about the algorithms, see Chapter 4 in the free e-book, *RATS Programming Manual, 2nd ed.*

The second half of this chapter describes and demonstrates the general purpose non-linear estimation instructions (**NLLS**, **NLSYSTEM**, **MAXIMIZE** and **FIND**). These are heavily used later in the remainder of the *User's Guide*, but particularly in Chapter 11 where they are needed for estimating transition and Markov switching models.

General Principles

Newton-Raphson and Hill-Climbing

Simplex and Genetic Algorithms

Constrained Optimization

Covariance Matrices

NONLIN and FRML

Non-linear Least Squares

Method of Moments

Systems Estimation

General Optimization

Troubleshooting

4.1 General Principles and Problems

Machine Precision

There are certain problems with which all non-linear estimation techniques in RATS (and other software) must deal. One difficulty is that, on a computer, floating point numbers are represented only approximately. Most systems running RATS show fourteen to fifteen significant digits. This does not mean, however, that you can get answers from estimation problems correct to fourteen digits. Intermediate calculations can often push the significance limit long before that. For instance, $f(x) = 50 + (x - 1)^2$ will give identical values for $x=1$ and $x=1.00000001$. Because the computer can't distinguish between the function value for these two (or anything in between), we can't get greater accuracy than eight digits in minimizing f .

A related problem is that floating point numbers have a huge, but limited, range, typically with around 10^{300} as the largest number and 10^{-300} as the smallest positive number. This can cause problems if a calculation is done in such a way that intermediate calculations overflow the range. If, for instance, you attempt to calculate the binomial coefficient ${}_{200}C_{100}$ using `%FACTORIAL(200) / %FACTORIAL(100) ^ 2`, RATS will produce a missing value, as `%FACTORIAL(200)` overflows. As these difficulties typically arise where a logarithm of the big number is needed, the way to work around this is to make sure you take logs early on and add or subtract them as needed, rather than risking overflow by multiplying and dividing huge numbers. Here the log of the binomial coefficient can be computed by `%LNGAMMA(201) - 2.0 * %LNGAMMA(101)`. For normal densities, use `%LOGDENSITY` rather than `%DENSITY` if you'll eventually be taking logs anyway.

Convergence and Scale of Parameters

A non-linear optimization is considered “converged” when the change from one iteration to the next is “small.” RATS defines the change in a vector of parameters by taking the maximum across its elements of

$$\min(|\beta - \beta_0| / |\beta_0|, |\beta - \beta_0|)$$

where θ_0 is the parameter's value before this iteration. This uses the relative change for parameters which, on the previous iteration, had absolute values of 1 or larger, and absolute changes for smaller values. This is compared with the convergence criterion that you set (typically by the `CVCRIT` option). The default value of this is 10^{-5} for most methods. A value much smaller than that (say 10^{-8}) probably is unattainable given the precision of floating point calculations on most machines, and even if the estimation succeeds in meeting the tighter value, you are unlikely to see much effect on the final coefficients.

However, you need to be careful how you parameterize your function. In particular, if you have a parameter which has a natural scale which is quite small (say .0001), then a change of .000009 might have a substantial effect on the function value, but would be small enough to pass the convergence test. Rescaling the parameter (replac-

ing it wherever it appears with .001 times itself, or scaling a variable it multiplies by .001, in some cases multiplying the dependent variable by constant like 100) can help here. Other options are to parameterize it in log or square root form if appropriate. These also help with the accuracy of numerical derivatives if RATS needs to take them.

RATS *does* allow you to set the optimizers so that they consider convergence achieved if the change in the function value is small. This is done using the instruction **NLPAR** with the **CRITERION=VALUE** option. Convergence on function value is usually quite a bit easier to achieve than convergence on coefficients. You should probably choose this only when just the optimized function value matters, and the coefficients themselves are of minor importance.

A Guide to the Optimization Algorithms

The following table lists the general purpose optimization algorithms used in RATS. Note that there are also some specialized algorithms used for certain problems, such as the linear and quadratic programming algorithms used in **LQPROG** and the specialized optimizers used in LAD and quantile regressions in **RREG**. These are for more general optimization and most are found in many different instructions. There are also optimization techniques (such as EM (expectation-maximization)) which require specific calculations for specific problems. The algorithms below are more generally applicable. The derivative-based methods are described in detail in Section 4.2 and the derivative-free methods are covered in Section 4.3.

Method Name (in options)	Type of algorithm	Applications
BFGS	Derivative-based, uphill only	General optimization
BHHH	Derivative-based, uphill only	Likelihood-based only
GAUSS(-Newton)	Derivative-based, uphill only	Non-linear least squares only
NEWTON(-Raphson)	Derivative-based, uphill only	Specific models
SIMPLEX	Derivative-free, uphill only, narrow search	General optimization
GENETIC	Derivative-free, uphill only, broad-search	General optimization
ANNEALING	Derivative-free, up-and-down, broad search	General optimization
GA (genetic annealing)	Derivative-free, up-and-down, broad search	General optimization

Throughout this, we're assuming that we're talking about maximizing a function. Minimizing can be represented by maximizing minus the function.

Chapter 4: Non-Linear Estimation

The ideal optimization problem has a function which is continuously analytically differentiable (multiple times) and globally concave. (The log likelihood of the probit is a good example of that). If that's all we ever needed to do, we wouldn't need a range of optimization algorithms, with different settings within each. In practice, we have optimization problems with derivatives that have to be approximated numerically, or sometimes no derivatives at all. They can have multiple modes, or, even if there is only a single mode, they can fail to be globally concave. There can also be boundary issues, where the function either isn't computable outside a certain range (for instance, a variance can't be negative), or where the optimization problem itself requires constraints on certain parameters.

In general, derivative-based methods are faster, but are much more sensitive to choice of initial guesses and are limited in use only to models which are (at least) twice continuously differentiable.

The “uphill only” algorithms assume that you are starting on the right “hill” and move up from there, one hopes to the top. While it's *possible* for an early iteration to jump a considerable distance and actually switch “hills”, you shouldn't expect it. If you have any doubts about whether your function might have multiple local optima, it's a good idea to try re-estimating from several extra sets of initial guesses, to see if you converge just to the one set of estimates.

“Up-and-down” algorithms make no such assumptions. As a result, they are better able to handle surfaces with multiple modes. However, there is a considerable cost in time, since they may be working multiple actual modes, or may be looking just at different parts of the same hill—there's no way to tell based just upon the function value. The (simulated) ANNEALING and GA (genetic annealing) algorithms have the property that they can make “downhill” moves. This makes them particularly useful in dealing with multi-modal surfaces since they don't have to find a spot on another hill that has a higher function value than the current location (the uphill-only algorithms require finding a better function value to switch “hills” which generally requires a lucky, wild step). Needless to say, making both uphill and downhill moves takes considerably longer than making only uphill moves, so you should employ them only if you have a real concern that you might have a very ill-behaved surface.

4.2 Derivative-Based Methods

If the function being optimized is quadratic:

$$(1) \quad F(\mathbf{x}) = \mathbf{a} + \mathbf{c}'\mathbf{x} + \frac{1}{2}\mathbf{x}'\mathbf{Q}\mathbf{x}$$

and $\mathbf{c}$ and $\mathbf{Q}$ are known, then the maximum of F (if $\mathbf{Q}$ is negative definite) can be found in one step by $\mathbf{x} = -\mathbf{Q}^{-1}\mathbf{c}$. If, for instance, you feed **NLLS** (non-linear least squares) a function which is linear in the parameters (and thus the sum of squares is quadratic), the algorithm will usually report convergence in two iterations: the first finds the minimum and the second makes no further progress and thus passes the convergence test.

While we usually don't have quadratic functions when we do non-linear optimization, there is a large set of tools which can be applied to functions which are twice continuously differentiable, and thus have a second order Taylor expansion. If θ_k is the current estimate, then, for θ in a neighborhood of θ_k ,

$$(2) \quad F(\mathbf{x}) \sim F(\mathbf{x}_k) + \mathbf{g}'(\mathbf{x} - \mathbf{x}_k) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_k)' \mathbf{H}(\mathbf{x} - \mathbf{x}_k)$$

where $\mathbf{g}$ is the gradient and $\mathbf{H}$ the Hessian (matrix of second derivatives) evaluated at θ_k .

Newton-Raphson takes the updated θ to be the optimizer of (2), which is

$$(3) \quad \mathbf{x} = \mathbf{x}_k - \mathbf{H}^{-1}\mathbf{g}$$

There are two main drawbacks to the use of Newton-Raphson as an optimizing algorithm:

1. It requires calculation of the second derivatives of $\mathbf{H}$. If there are n parameters, this means $n(n+1)/2$ second derivatives. Computing these can be very time-consuming, and, if it has to be done numerically, can be subject to considerable approximation error.
2. Until the estimates are near the optimum, it is possible that $\mathbf{H}$ won't show the correct curvature. Because Newton-Raphson is really searching only for a zero in the gradient, it could head for a minimum when you want a maximum, or vice versa.

RATS uses Newton-Raphson only for logit, probit and related models (estimated via the **DDV** and **LDV** instructions) which have fairly simple second derivatives. And even then, it uses the Newton-Raphson prediction only to give the direction of movement, not the distance.

Hill-Climbing Methods

These are a very general collection of techniques for maximizing a twice-continuously differentiable function. A single iteration takes the following steps:

1. Compute the gradient ($\mathbf{g}$)
2. Select a direction by premultiplying the gradient by a matrix, that is $\mathbf{d}=\mathbf{G}\mathbf{g}$. Make sure the directional derivative is positive, so that (at least locally) we are going “uphill” in that direction. If not, modify the direction.
3. Select a distance to travel in that direction: $\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda \mathbf{d}$

When RATS needs to make an adjustment in the direction in step 2, it does it by adding in a multiple of the gradient. RATS is looking for a direction in which

$$(4) \quad \frac{\mathbf{d} \bullet \mathbf{g}}{\|\mathbf{d}\| \|\mathbf{g}\|} \geq \alpha$$

where α is a control parameter. On most iterations, α is zero. However, if RATS detects that the estimation isn’t proceeding smoothly, it will switch to .00001 (by default), which you can reset using the ALPHA option on **NLPAR**. If the initial direction fails the test, RATS replaces it with $(\mathbf{d} + \gamma \mathbf{g})$ where this is solved for the value of γ which just meets (4). While you might expect that making α higher would speed up the process by picking a direction in which the function is “steeper,” this is not the case. In fact, taking $\mathbf{d} = \mathbf{g}$, which maximizes the left side of (4), gives us the infamous “steepest descent” (really “steepest ascent” for maximization problems) which is well known for producing very *slow* progress towards the maximum because of constant severe direction changes from iteration to iteration.

Subiteration (Line Search)

The process of selecting λ is called *line search*. RATS refers to the attempts to find a suitable λ as *subiteration*. There are two distinct philosophies which can be used in the line search stage. One is called exact (or perfect) line search, which maximizes the function over λ . In inexact line search, we look for a value of λ at which a condition is satisfied which guarantees progress towards the maximum. The choice between these two methods is controlled by the EXACTLINESEARCH option on **NLPAR**. The default is NOEXACT. For this, RATS tests

$$(5) \quad \delta \leq \frac{F(\mathbf{x}_{k+1}) - F(\mathbf{x}_k)}{\lambda \mathbf{d} \bullet \mathbf{g}} \leq 1 - \delta$$

where δ (a number between 0 and .5) is controlled by the DELTA option on the instruction **NLPAR**. The middle part is the ratio between the directional arc derivative with respect to λ and the directional derivative itself. This condition is described in, for instance, Berndt, Hall, Hall and Hausman (1974). The choice of subiteration method rarely affects whether you are able to achieve convergence. It *can* affect how quickly you achieve it. The default (NOEXACT) is almost always faster if you have fewer than thirty parameters, because it reduces the number of function evaluations.

RATS applies the hill climbing technique with four different methods of computing the multiplying matrix $\mathbf{G}$. Some of these are specialized to particular functions and are thus not available in all situations. The four methods (Newton–Raphson, Gauss–Newton, BFGS, and BHHH) are described below.

Newton–Raphson

Used by **DDV** and **LDV**, Newton–Raphson takes $\mathbf{G} = -\mathbf{H}^{-1}$. It converges *very* quickly once it is close to the optimum.

Gauss–Newton

Gauss–Newton is the primary method for the instructions **NLLS** and **BOXJENK** and is an option on **DLM**. An extension to multiple equations is used by **NLSYSTEM**. This method is specific to non-linear least squares estimation. It is an algorithm for solving problems like:

$$(6) \quad \min_{\mathbf{x}} \sum_{t=1}^T u_t^2 \quad \text{where } u_t = f(y_t, \mathbf{x})$$

Gauss–Newton is based upon a linearization of the residuals around θ_k .

$$(7) \quad u_t(\mathbf{x}_k) = - \left[\frac{\partial u_t}{\partial \mathbf{x}} \right] (\mathbf{x} - \mathbf{x}_k) + u_t(\mathbf{x})$$

This takes the form of a least squares regression of $u_t(\mathbf{x}_k)$ on the partial derivatives and is the same as a hill-climbing step (after converting to a maximization) with

$$(8) \quad \mathbf{G} = - \left(\sum_{t=1}^T (\partial u_t / \partial \mathbf{x})' (\partial u_t / \partial \mathbf{x}) \right)^{-1}$$

The objective function for instrumental variables (with Z as the instruments) is

$$(9) \quad \left(\sum_{t=1}^T u_t Z_t \right) \mathbf{W} \left(\sum_{t=1}^T Z_t' u_t \right)$$

where $\mathbf{W}$ is a weighting matrix, typically $\mathbf{W} = \left(\sum Z_t' Z_t \right)^{-1}$. The same type of linearization gives

$$(10) \quad \mathbf{G} = - \left(\left(\sum_{t=1}^T (\partial u_t / \partial \mathbf{x})' Z_t \right) \mathbf{W} \left(\sum_{t=1}^T Z_t' (\partial u_t / \partial \mathbf{x}) \right) \right)^{-1}$$

BFGS (Broyden, Fletcher, Goldfarb, Shanno)

The BFGS method (described in Press, et. al., 2007) is used by the instructions **BOXJENK**, **CVMODEL**, **DLM**, **FIND**, **GARCH**, **MAXIMIZE**, and (with certain options) **NLSYSTEM**.

G generally starts with a diagonal matrix, the form of which varies from instruction to instruction. At each iteration, it is updated based upon the change in parameters and in the gradient in an attempt to match the curvature of the function. The basic theoretical result governing this is that if the function is truly quadratic, and if exact line searches are used, then in n iterations, **G** will be equal to $-\mathbf{H}^{-1}$. If the function isn't quadratic, **G** will be an approximation to $-\mathbf{H}^{-1}$.

Because the estimate of **G** produced by BFGS is used in estimating the covariance matrix and standard errors of coefficients, you need to be careful not to apply BFGS to a model which has already been estimated to a fairly high level of precision. If BFGS uses fewer iterations than the number of parameters being estimated, the **G** will be poorly estimated and, hence, the standard errors derived from them will be incorrect. All of the instructions which allow BFGS will let you use the **HESSIAN** option to set an initial guess for **G**. If you have a model which is nearly converged, and use an initial **G** which is close to the correct matrix as well (for instance, using %XX from a previous estimation), you can have a bit more confidence in the resulting covariance matrix. If you don't have a better initial **G**, move your guess values for the parameters away from the optimum to force a longer estimation sequence.

BHHH (Berndt, Hall, Hall and Hausman)

BHHH implements the proposal for choosing **G** recommended in Berndt, Hall, Hall and Hausman (1974). Because it can only be applied to specific types of optimization problems, the only RATS instructions which can use it are **MAXIMIZE** and **GARCH**. The function being maximized has the form:

$$(11) \quad F(\mathbf{x}) = \sum_{t=1}^T f(y_t, \mathbf{x})$$

where F must be the actual log likelihood function (possibly omitting additive constants. It chooses **G** as $\mathbf{J}^{-1}$ where

$$(12) \quad \mathbf{J} = \sum_{t=1}^T \left[\frac{\partial f}{\partial \mathbf{x}}(y_t, \mathbf{x})' \frac{\partial f}{\partial \mathbf{x}}(y_t, \mathbf{x}) \right]$$

Under fairly general conditions, $-\mathbf{J}$ will have the same asymptotic limit (when divided by T) as the Hessian **H**. The information equality is used in deriving this, which is why F is required to be the log likelihood.

If the function is not the log likelihood, the algorithm will still usually end up at the correct maximum, as a good choice of **G** generally affects only how quickly the hill is climbed. Convergence, however, is likely to be slow, and the standard errors and covariance matrix will be incorrect.

4.3 Derivative-Free Methods

RATS also offers several methods which do not require a differentiable function. In general, continuity is the only requirement for these.

Simplex Method

The simplex method is a search procedure which requires only function evaluations, not derivatives. It starts by selecting $K+1$ points in K -space, where K is the number of parameters. The geometrical object formed by connecting these points is called a *simplex*. (Note that the only relationship between this and the simplex method of linear programming is that each method utilizes a simplex). One of these vertices represents the initial values. RATS makes each of the other points equal to the initial point plus a perturbation to one and only one of the parameters. Notice that, unlike a general grid search, this initial set of points does not need to enclose the optimum.

The basic procedure at each step is to take the *worst* of the $K+1$ vertices, and replace it with its reflection through the face opposite. Where hill-climbing methods look for a direction where the function increases, the simplex method moves uphill by eliminating the directions where the function decreases. As a result, the simplex method is more robust to initial conditions and the behavior of the function, but is much slower to converge than hill-climbing techniques when the latter are appropriate.

There is a tendency for the simplex method to get stuck in situations where the points are too much in a line to provide information about the shape. To prevent this, RATS perturbs the vertices every 30 function evaluations. (How often this is done is controlled by the JIGGLE option on **NLPAR**). For more information, see, for instance, Press, et. al. (2007).

RATS uses the simplex method (exclusively) to estimate the parameters in **ESMOOTH**. While **ESMOOTH** generates models which can be estimated by non-linear least squares models, the Gauss–Newton algorithm that applies to such problems does not deal well with the unstable regions in the parameter space.

You can choose to use the simplex method for estimation for **FIND** (where it's the default method), **BOXJENK**, **CVMODEL**, **DLM**, **GARCH**, **MAXIMIZE**, **NLLS** and **NLSYSTEM** (where it's an option). **FIND** is the “catch-all” optimization instruction. If you can't do it any other way, you can probably do it with **FIND**. See more in Section 4.13.

For the other seven instructions, the most important use of **SIMPLEX** is to refine initial estimates before using one of the derivative-based methods, which are more sensitive to the choice of initial estimates. This is usually done by using the **PMETHOD** (and **PITERS**) options to choose a “preliminary” method. However, *don't overdo this*. If the plan is to switch to a derivative-based algorithm, somewhere between 5 and 20 simplex iterations is generally enough. For instance, you may find that

```
maximize (pmethod=simplex,piters=5,method=bfgs) formula
```

works better than the **BFGS** (or **BHHH**) alone. Note, by the way, that RATS counts iterations for the simplex minimization in a non-standard way: each “iteration” in-

Chapter 4: Non-Linear Estimation

cludes $2K$ changes to the parameter set. This roughly equalizes the number of function evaluations per iteration with the other methods.

Genetic algorithm

Genetic algorithms are designed to handle difficult optimization problems by applying lots of compute power and a stylized model of “evolution,” with random “mutations.” RATS applies a variation of this called *differential evolution*. Given a “population” of parameter vectors, at each iteration each vector is compared with a possible successor. Whichever is the “fitter” of the two (the one with the better function value) is retained, the other discarded. There are a number of schemes for generating successors, all of which require that at least one parameter be “mutated,” that is, have some random number added to it. In differential evolution, the size of a mutation in a parameter is determined from the difference between that parameter in two randomly selected elements of the population. If a particular parameter takes values which are still fairly widely dispersed throughout the population, then mutations will be large. If, on the other hand, it becomes clear that an individual parameter is fairly well determined, so that its values are tightly bunched, future changes in it will be small.

METHOD=GENETIC is available as an option on the instructions **BOXJENK**, **CVMODEL**, **DLM**, **FIND**, **GARCH**, **MAXIMIZE**, **NLLS** and **NLSYSTEM**.

There are four options on **NLPAR** which govern the genetic algorithm:

```
populate=scale for population size [6]
mutate=[simple]/shrink
crossover=probability of taking mutated parameter [.5]
scalefactor=scale factor for mutations [.7]
```

The **POPULATE** option is the simplest of these to explain: it just indicates the factor by which the number of parameters is multiplied to get the population size. Note, however, that the population size is never allowed to be below 20. As mentioned above, at each iteration, each of the parameter vectors in the population is compared against a trial vector, to be replaced if the trial vector is better. The other three options govern how this trial vector is generated. One existing parameter vector is chosen as the base for this. Two other parameter vectors are chosen at random and their difference is taken. In the **SIMPLE** form of mutation, this difference is multiplied by **SCALEFACTOR** and added to the base. A binomial trial is performed for each component. With probability equal to the **CROSSOVER**, the trial value is chosen; otherwise the original value is retained. A large value of **SCALEFACTOR** (bigger than one) is useful if you need to search a wide area. The **SHRINK** form for mutation is similar but, instead of using a randomly selected base vector, it uses a linear combination of the original vector and the best vector from the previous iteration.

It's likely that the ideal set of choices for this will vary depending upon the function being optimized. If you don't seem to be making good progress, try different values.

Simulated Annealing

Simulated annealing (option **METHOD/PMETHOD=ANNEALING**) is an algorithm originally proposed to solve (or at least approximately solve) certain combinatorial problems such as the traveling salesman problem (minimizing the total distance required to hit each of a set of stops exactly once). As the number of stops grows, it becomes intractable to simply look at all possibly routes and pick the shortest one. But it's a difficult problem to optimize by trying change up the route to (always) shorten it because it's very easy to get trapped in a locally shortest route—shortening the path may require taking a “right” rather than a “left” 15 or 20 moves farther back, and backtracking through all that is basically no different from the (impossible) testing of all routes.

The key to simulated annealing is that it can make changes that can either shorten or lengthen the route. The name is based upon the annealing process in metallurgy, where the energy state generally goes down, but can occasionally go up, as metal cools. At a high “temperature” the moves in the “wrong” direction are more likely, and become less likely as the “temperature” declines. For the traveling salesman problem, there may be a key switch that requires quite a bit of later reshuffling, but which overall reduces the length. The annealing algorithm makes it possible to try that switch at a high temperature (when, at least initially, it seems to make things worse) and then steadily fix the rest of the path as the process “cools”.

The general structure of the algorithm requires an “energy function” (the function to be optimized or some transformation of it), and an “annealing schedule”, which tells what the initial “temperature” is and when and by how much it “cools”. With RATS, the probability of moving from the current energy level e to a proposed energy level e' when at temperature T is

$$P(\text{move} | e, e', T) = \exp(-(e' - e) / T)$$

which is known as the Metropolis condition. It's possible to make a “bad” move (to a higher energy level) and it's also possible to not make a “good” move. However, the probability of either of those goes down as T decreases. (Some descriptions of simulated annealing have this set to always take a good move).

That's common to all uses of simulated annealing. What's specific to an application is the generation of possible moves (proposals). For the traveling salesman problem, what's been found to work is a random choice of path reversals and path breaks. However, we're interested in applying this to functions defined in continuous parameter spaces, not integers. Corana et. al (1987) describe an adaptive method for generating proposals in such situations. Each parameter maintains its own range (which they call $v(i)$). Batches of single-move proposals are generated, where each proposal randomly perturbs only one of the parameters using that parameter's current range (uniform draw from + or - the range from the current value). Moves or non-moves are determined by the Metropolis test. At the end of a batch, you look at the probabilities of moves for each parameter. If it's too high (the target is 50% moves), the range is

Chapter 4: Non-Linear Estimation

widened and if it's too low, the range is narrowed. Multiple batches are run at each temperature level to form an "iteration". At the end of an iteration, the best set of parameters seen so far are re-loaded, and the temperature is reduced. At a high temperature, quite a few "bad" moves (even some really bad ones) will be accepted, and the parameter space will be widely searched. As the temperature is reduced, you will generally only make good moves and the test range for the parameters will squeeze down so you are only making (small) local changes.

METHOD=ANNEALING (and PMETHOD=ANNEALING, which is what is more typically used) are available on **BOXJENK**, **CVMODEL**, **DLM**, **FIND**, **GARCH**, **MAXIMIZE** and **NLLS**. Each of those instructions has a T0 option to control the initial temperature. For a maximum likelihood function (all of the above but **BOXJENK** without the MAXL option and **NLLS**), the default is the maximum of 5 and the square root of the number of parameters, which seems to work fine in practice. For non-linear least squares functions, it's 20% of the function value at the initial guesses. The ITER (for METHOD) and PITER (for PMETHOD) options control the number of "iterations".

The other controls are in the **NLPAR** instruction, and generally don't need adjustment. ASCHEDULE controls the rate at which the temperature is lowered per iteration. The default is .85. AINNER (default of 20) is the size of the inner batches described above. After each inner batch, the ranges are adjusted to try to push the acceptance ranges towards .5. The AVADJUST option (default of 2.0) controls the range adjustment (Corana, et. al. call this Cu). Higher values lead to more aggressive changes. AOUTER controls the number of times the inner batches are run per iteration. This is a multiplier on the number of parameters. (The inner batch size is fixed). The default is 3.0—however, a minimum of 25 are done, regardless of the number of parameters. For example, if you have 15 parameters, there will be 3.0×15 outer loops over 20 (AINNER default) inner loops, each of which will do one function evaluation per parameter, thus, a single "iteration" would involve $3 \times 15 \times 20 \times 15 = 13500$ function evaluations. By contrast, a BFGS iteration typically would require 15 plus a few extras for the directional search, so you can get an idea of how much longer it will take. In practice, use it (at most) only for preliminary iterations and keep the number of PITERS small (5 or less).

Genetic Annealing

Genetic Annealing (option METHOD/PMETHOD=GA) is a cross between the genetic algorithm and simulated annealing. It's mainly the genetic algorithm (thus tracks a population of parameter sets) which are mutated at each point, but, instead of only picking the "fittest" (that is, replacing one parameter set only if the mutated one produces a better function value) uses an annealing schedule so it might replace it with one that's not as good.

An "iteration" is a pass through the algorithm at a given temperature. Within each, it does several batches of mutations. Within a single batch of mutations, each member of the population is mutated and tested to see if it should be replaced. After a batch of mutations, the control parameters for the mutations are adjusted to try to keep

the percentage of acceptances at around 50%. (The crossover scale is increased if the acceptance rate is below the target and decreased if it's above it).

METHOD=GA (and PMETHOD=GA, which is what is more typically used) is available on the same instructions as ANNEALING and has the same set T0 option on those instructions and annealing controls on **NLPAR**. The **NLPAR** controls that are common to the genetic algorithm are POPULATE (multiplier on the number of parameters for the size of the population) and MUTATE (method of mutation). The other two genetic method controls (CROSSOVER and F) are used, but have values which adapt to the situation. (F is changed to try to keep the acceptance rate at 50%).

If you have 15 parameters, a single iteration would (with the default settings) take 3.0×15 loops over the mutation batch, each of which would have 6.0 (POPULATION default multiplier) $\times 15$ function evaluation. It will take (roughly) the same number of function evaluations per iteration as simulated annealing if the AOUTER times the POPULATION value for genetic annealing is the same as AOUTER times AINNER for simulated annealing. (Simulated annealing has the extra “inner” loop, but that's over the number of parameters, while genetic annealing loops over the population size, which is considerably larger). In comparison with the regular genetic algorithm, it takes that entire outer loop within an iteration ($AOUTER \times$ the number of parameters times through that), so requires much *more* computation, but then it does a broader scan of the parameter space.

4.4 Constrained Optimization

The previous sections have described optimization problems where the free parameters are allowed to take any values. Now some functions have implicit bounds on the parameters. Variance parameters, for instance, often are forced to be positive by the presence of a $\log(v)$ term in the function. There is no need to do anything special to see that this is enforced. The function value will be NA if v goes non-positive, and so RATS will be forced to take a smaller step into positive territory.

However, there are situations in which the parameters are constrained, not by the nature of the function, but by the nature of the problem. If a parameter represents a physical amount of something, a negative value may be nonsensical but an unconstrained optimization might well give us a negative estimate.

If you have a single parameter subject to this type of imposed bound, constrained optimization is simple: estimate the model unconstrained. If the parameter has a permitted value, you're done. Otherwise, re-estimate with the bound imposed.

However, if you have several constraints, it becomes much more complicated to work through the possible combinations. Instead, you can let RATS do the work for you.

Types of Constraints

There are three basic types of constraints that you can impose on your non-linear parameters. These are all imposed using the instruction **NONLIN**. Note that you must define the parameters before they can appear in a constraint expression—see the examples later in this section.

1. Substitution constraints. These give one parameter as a function of others, for instance, $B3=B1*B2$. This type of constraint is handled directly during the estimation process, so RATS doesn't need any special techniques.
2. Equality constraints. These are similar to substitution constraints, except that the equation is not explicitly solved for one of the variables. An example is $B1*B2==B3*B4$. (Note that you *must* use the `==` operator). However, if you can solve out for one variable, do it. For instance, here, unless you have a strong possibility that $B2$ will be forced to zero, using $B1=B3*B4/B2$ will be more efficient.
3. Inequality constraints. Examples are $B1 \geq 0$ and $B1+B2+B3 \leq 1.0$. *These cannot be strict inequalities*. For instance, you can't restrict a parameter to be positive: it must be constrained to be non-negative.

For some examples, see pages UG–130 and UG–145 and the description of **NONLIN** in the *Reference Manual*.

Algorithm

On each iteration, RATS first determines which of the inequality constraints are to be considered “active.” This includes all constraints which are violated, and any others which have non-zero Lagrange multipliers from the previous iteration. Let the active constraints (including any equality constraints) be represented as $c(\beta) = 0$.

The Lagrangean for maximizing $f(\beta)$ subject to these is

$$(13) \quad \mathcal{L} = f(\beta) - \mu' c(\beta)$$

where μ is the vector of Lagrange multipliers. RATS maintains an estimate of

$$(14) \quad \mathbf{G} = -\left(\frac{\partial^2 \mathcal{L}}{\partial \beta \partial \beta'}\right)^{-1}$$

It computes $\mathbf{g} = \partial f / \partial \beta$ and $\mathbf{N} = \partial c / \partial \beta$. Using these, a new set of Lagrange multipliers is computed using the formula:

$$(15) \quad \alpha = (\mathbf{N}'\mathbf{G}\mathbf{N})^{-1}(\mathbf{N}'\mathbf{G}\mathbf{N} - c)$$

These are then used to compute the direction vector

$$(16) \quad \mathbf{d} = \mathbf{G}(\mathbf{N}\alpha + \mathbf{g})$$

This direction vector would, in a perfect world, take us to a zero gradient point for the Lagrangean.

With the direction chosen, the question then is how far to move in that direction. RATS searches over the step size λ using the following penalty function

$$(17) \quad f(\beta + \lambda \mathbf{d}) - (1/r) \left\| c(\beta + \lambda \mathbf{d}) \right\|^2$$

The scale factor r changes from iteration to iteration, becoming smaller as the estimates come closer and closer to meeting the constraints. With a new set of parameters in hand, RATS then updates the estimate of $\mathbf{G}$ using the BFGS formula and moves on to the next iteration. *Any* constrained estimation done with a hill-climbing method will use this variant of BFGS, even if you selected the BHHH or Gauss–Newton methods.

Constrained optimization is usually quite a bit slower than a similar problem with no constraints. This is not due to the extra calculations done in each iteration. Instead, it simply takes more iterations, often many more. Even if the set of active constraints stays the same throughout the estimation process, as the Lagrange multipliers change in the early going, the Lagrangean itself changes. Thus, we're trying to hit a moving target. Once the Lagrange multiplier estimates settle down, the estimates will start converging more smoothly.

When setting up your constraints, be aware of the sensitivity to scale. If, for instance, you were to put in a constraint such as `1000000*B1>=0`, a fairly modest negative value of `B1` would cause a large value in this constraint's component of c . This would likely so overwhelm the other constraints that all the early iterations would be devoted to forcing this single constraint to hold.

The REJECT Option

Most RATS instructions that do non-linear optimization include a REJECT option. With this, you provide a function or formula that evaluates to “true” (non-zero) for any situation where you want the overall function to return an NA. An example is the option

```
reject=(abs(g1+g2)>=1.0)
```

which (in the situation to which this applies) eliminates from consideration any unstable combinations of the G1 and G2 parameters.

In the early iterations of a non-linear estimation, it’s quite possible for rather extreme values to come up for evaluation. In most cases, you’ll just get a really low likelihood (or a natural NA, if, for instance, they would require taking the log of a negative number), and they’ll be rejected on that basis. REJECT can (and should) be used if something worse happens than a really low likelihood.

You *cannot* use REJECT as a cheap way to impose a constraint on parameters when the boundary is feasible. If, for instance, your function is computable at $x \geq 1$ and you want to impose $x \leq 1$, the option REJECT=(X>1.0) will cause the optimization to fail to converge if the function is increasing at $x = 1$. Given the shape of the function, the optimization will naturally be testing values of x larger than 1. When it’s told that the value there is NA (as a result of the REJECT), it will cut the step size, but will probably have to cut it quite a few times before getting down below 1.0. The next iteration will likely need even more subiterations in order to get under 1, etc. Eventually, it will hit a subiterations limit.

Instead, use the constrained optimization technique described earlier, by including

```
x x<=1.0
```

in the parameter set. This uses a penalty function for values above 1.0, but that penalty function starts relatively small, so at least initially (given the shape of the function), it will have a higher total (likelihood less penalty) for values above 1. The penalty function increases as you iterate more, eventually dominating the function value and forcing x towards 1. You’ll probably end up with the final estimated x being something like 1.000001.

4.5 Covariance Matrices

The main (and in some cases the only) point of the optimization methods is to answer the question: for what parameters is this function maximized (minimized)? However, most of the interesting optimizations depend upon sample data. Assuming that there is some randomness in the observed data, the point estimates will vary depending upon the sample. This means that providing the point estimates alone will not fully describe the available information—it is also desirable to report some measure of the sensitivity of the estimates to the sample.

The only optimization techniques available in RATS that can provide estimates of the standard errors and covariance matrix are the hill-climbing methods described in 4.2. The derivative-free methods from 4.3 can produce point estimates only.

The Basic Framework

With most of the non-linear estimation instructions in RATS, the function being optimized takes the form

$$(18) \quad F(\beta) = \sum_{t=1}^T f(y_t, \mathbf{X}_t, \beta)$$

The exceptions to this are **FIND**, where you provide $F(\beta)$ directly, and **CVMODEL**, where you provide sufficient statistics for such a sum.

If θ_T is the estimate using data through T and θ_0 is the true parameter, then a first-order Taylor expansion of the gradient is

$$(19) \quad \frac{\partial F}{\partial \beta}(\beta_T) \approx \frac{\partial F}{\partial \beta}(\beta_0) + \frac{\partial^2 F}{\partial \beta \partial \beta'}(\beta_T - \beta_0)$$

The left-hand side of this is zero, so we can solve to get

$$(20) \quad \sqrt{T}(\beta_T - \beta_0) \approx - \left(\frac{1}{T} \frac{\partial^2 F}{\partial \beta \partial \beta'} \right)^{-1} \frac{1}{\sqrt{T}} \frac{\partial F}{\partial \beta}(\beta_0)$$

Using arguments similar to those in Section 2.2, the second factor on the right can be written

$$(21) \quad \frac{1}{\sqrt{T}} \sum_{t=1}^T \frac{\partial f}{\partial \beta}(y_t, \mathbf{X}_t, \beta)$$

where the summands have expected value zero. Again, under the proper conditions (see, for instance, Hayashi, 2000, section 7.3) this will have an asymptotic Normal distribution with mean zero and a covariance matrix that can be estimated consistently by

$$(22) \quad \mathbf{B}_T = \frac{1}{T} \left\{ \sum_{k=-L}^L \sum_t \frac{\partial f}{\partial \beta}(y_t, \mathbf{X}_t, \beta)' \frac{\partial f}{\partial \beta}(y_{t-k}, \mathbf{X}_{t-k}, \beta) \right\}$$

Chapter 4: Non-Linear Estimation

Combined with

$$(23) \mathbf{A}_T = \frac{1}{T} \sum_t \frac{\partial^2 f}{\partial \beta \partial \beta'}(y_t, \mathbf{X}_t, \beta)$$

we get the familiar “sandwich” expression

$$(24) \sqrt{T}(\beta_T - \beta_0) \sim N(0, \mathbf{A}_T^{-1} \mathbf{B}_T \mathbf{A}_T'^{-1})$$

$\mathbf{B}_T$ can be calculated from the derivatives computed during each iteration. It’s $\mathbf{A}_T$ that we don’t necessarily have. The only hill-climbing method that computes it is Newton–Raphson, which is used only by **DDV** and **LDV**. However, BFGS produces an approximation to the (inverse) Hessian. As stated in Section 4.2, BFGS exactly produces the Hessian in K iterations (K =number of parameters) if the function $F(\beta)$ is quadratic, and exact line searches are used. If a function is only locally quadratic (which is a required assumption to get the covariance matrix using this type of analysis), then BFGS produces an approximation. But be careful: if you start BFGS very close to the final estimates, it won’t have a chance to “map out” the curvature, as the gradient will already be quite close to zero. To give BFGS a decent chance to develop the curvature estimates, you may need to start with initial conditions pulled away from the optimum.

Simplifying Assumptions

In many cases, the calculation of the covariance matrix can be simplified. In some (for instance, with **CVMODEL** or **FIND**) it *has* to be, as the information to compute $\mathbf{B}_T$ isn’t available. All of these start by assuming the L in (22) is zero. A non-zero L means that the partial derivatives are correlated across time. If the observations are independent, or the model is assumed to exhaust any correlation, this should be a safe assumption.

If $F(\beta)$ is the log-likelihood function, then the information equality applies (see, for instance, Hamilton, 1994, section 14.4). $\mathbf{A}_T$ and $\mathbf{B}_T$ have the same expected values; they aren’t, however, equal. Whichever one we can compute most easily can substitute for the other. In the case of **METHOD=BHHH**, we are computing $\mathbf{B}_T$ (with $L=0$) at every iteration to get the **G** matrix in the hill-climbing procedure. For **METHOD=BFGS**, it’s $\mathbf{A}_T$ (more accurately, its inverse) that we’re computing (approximately).

Cautions

Different methods of estimation will generally produce point estimates which agree to the precision that you request. The covariance matrices, however, can differ substantially. It's not uncommon for standard errors to disagree at the second or third significant digit. This is because they are all based upon the asymptotic equivalence of matrices that are not identical in sample.

By default, **MAXIMIZE** assumes that the information equality applies. If $F(\beta)$ is *not* the log likelihood function (other than missing some additive constants), the simplifying assumptions above are wrong, and the covariance matrix is likely to be quite wrong as well. For instance, if you use **MAXIMIZE** to fit a least squares model by maximizing $\sum(-u_t^2)$, the point estimates will be correct, but the standard errors will be off by a factor of $\sqrt{2\sigma^2}$, since the actual log likelihood element is $-.5(u_t^2/\sigma^2)$, not just $-u_t^2$.

Because **FIND** can be used for optimization problems which are not based upon sample data, it doesn't automatically compute and display standard errors. You need to include the **STDERRS** option if you want it to do so and you must be using **METHOD=BFGS**. The calculation of these assumes the information equality holds.

NLLS and NLSYSTEM

The covariance matrix calculations for **NLLS** and **NLSYSTEM** are similar to those above but exploit the special structure of the problem. For instance, for **NLLS**,

$$(25) \quad F(\beta) = \sum_{t=1}^T (y_t - f(\mathbf{X}_t, \beta))^2$$

$$(26) \quad \frac{\partial F}{\partial \beta} = -2 \sum_{t=1}^T (y_t - f(\mathbf{X}_t, \beta)) \frac{\partial f}{\partial \beta}(\mathbf{X}_t, \beta)$$

$$(27) \quad \frac{\partial^2 F}{\partial \beta \partial \beta'} = 2 \sum_{t=1}^T \frac{\partial f}{\partial \beta}(\mathbf{X}_t, \beta)' \frac{\partial f}{\partial \beta}(\mathbf{X}_t, \beta) - 2 \sum_{t=1}^T (y_t - f(\mathbf{X}_t, \beta)) \frac{\partial^2 f}{\partial \beta \partial \beta'}(\mathbf{X}_t, \beta)$$

The second sum in (27) is the sum of the product of residuals times a function of the exogenous variables only. When divided by T , it isn't unreasonable to assume that this will become negligible in large samples. Thus, this term is ignored in computing $\mathbf{A}_T$, which eliminates the need to compute second derivatives. The convenient simplification here is to assume that

$$(28) \quad E(u_t^2 | \mathbf{X}_t) = \sigma^2$$

which, when applied to (26) gives us

$$(29) \quad E(\mathbf{B}_T) = \sigma^2 E(\mathbf{A}_T)$$

ROBUSTERRORS, LWINDOW, LAGS and CLUSTER

The ROBUSTERRORS option is available on the instructions **MAXIMIZE**, **NLSYSTEM**, **NLLS**, **LDV**, **DDV** and **GARCH** to do the full covariance matrix calculation described by (22), (23) and (24). This can only be used with METHOD=BFGS on **MAXIMIZE** and METHOD=GAUSS (the default) for **NLLS** and **NLSYSTEM**.

The LAGS option, when used together with ROBUSTERRORS, gives a non-zero value to L in (22). This is subject to the same caution as for linear models (Section 2.1): if you don't use a choice for LWINDOW like NEWKEYWEST, the covariance matrix computed could fail to be positive definite.

ROBUSTERRORS with CLUSTER=*SERIES with category values* allows for arbitrary patterns of correlation in the gradient within the categories given by the values of the series you provide. You must have at least as many categories as parameters to estimate, and generally should have *many* more.

QMLE (Quasi-Maximum Likelihood Estimation)

QMLE is not a separate option in any RATS instruction. Instead, to do QMLE, you maximize using standard methods a likelihood that you either know or suspect is misspecified. However, there are quite a few situations where the misspecified likelihood is adequate for obtaining consistent estimates of the parameters—an obvious example would be least squares, which is the (true) maximum likelihood estimator for i.i.d. Gaussian disturbances, but is consistent under much broader circumstances.

While the parameter estimates might be consistent, a covariance matrix that comes from the inverse of the information matrix won't be correct. In general, the ROBUSTERRORS option corrects for the failure of the information equality when you're not using the correct likelihood.

4.6 Setting Up Your Model: NONLIN and FRML

While some of the instructions covered in this chapter are special-purpose instructions which handle only one type of model, many of them allow you to, and in fact require you to, provide the function to be optimized. In addition, you have to indicate the set of parameters which are to be estimated. This second task is done with the instruction **NONLIN**. The first is usually done with the help of the instruction **FRML**.

PARMSETS

A PARMSET is a specialized data type which consists of a list of parameters (a “parameter set”) for non-linear estimation and constraints on them (if any). PARMSETS are constructed and maintained using the instruction **NONLIN**, and are used by the instructions **CVMODEL**, **DLM**, **FIND**, **GARCH**, **MAXIMIZE**, **NLLS**, and **NLSYSTEM**. PARMSETS can be combined using the standard “+” operator. You can set up vectors or matrices of them or pass them as procedure parameters.

The Instruction NONLIN

NONLIN sets up or edits the list of free parameters in the model. In its simplest form, it simply defines a list of free parameters to be estimated by a later instruction, such as **NLLS**:

```
nonlin alpha gamma delta
nonlin g b0 b1 b2
```

NONLIN can accept either single REAL variables as parameters or a complete VECTOR or other real array, or an array of real-valued arrays, such as a VECTOR[VECTOR]. For instance,

```
declare vector b(5)
nonlin b rho
```

will create a parameter set which includes the five elements of B plus the single value RHO. A vector used in this way doesn’t have to be dimensioned at the time of the **NONLIN** instruction, but must be dimensioned before being used for estimation.

The above instructions create an internal PARMSET, which is the one used by the estimation instructions unless you provide a different one. To create a different PARMSET, use the PARMSET option on the **NONLIN**.

```
nonlin(parmset=boxcoxparms) sigma lambda
nonlin(parmset=regparms) b
```

The first of these creates a PARMSET named BOXCOXPARMS which includes the two variables SIGMA and LAMBDA. The second creates REGPARMS. You can combine one PARMSET with another using the “+” operation, either in a **COMPUTE** instruction or directly on the estimation instruction. For instance,

```
compute fullparm=boxcoxparms+regparms or
maximize(parmset=boxcoxparms+regparms) ...
```

Chapter 4: Non-Linear Estimation

The functions %PARMSPEEK and %PARMSPOKE can be used (with a named PARMSET) to move values into and out of a parameter set. For instance, in the last example, %PARMSPEEK (BOXCOXPparms) will return a vector with elements SIGMA and LAMBDA in order, and %PARMSPOKE (BOXCOXPparms,V) would make SIGMA=V(1) and LAMBDA=V(2). This can be handy when you wish to parameterize a function using single variable names (which makes it easier to change the function, and read the estimation output), but you also need to be able to move information into and out of the parameter set easily. For instance,

```
nonlin(parmset=simszha) a12 a21 a23 a24 a31 a36 a41 $
      a43 a46 a51 a53 a54 a56
...
compute axbase=%parmspeek(simszha)
...
compute %parmspoke(simszha,axbase+saxx*au)
```

In the application from which this is taken, it's necessary to reset the parameters using a matrix operation. However, parameterizing the function as A(13) would have been clumsy, since adding or deleting a parameter would have required a complete rewrite of six lines defining a matrix. By using %PARMSPEEK and %PARMSPOKE, you can set up the PARMSET in the most convenient fashion and still be able to set the elements using matrix operations.

If you need to refer to the default parameter set, use the %PARMSET() function.

Using NONLIN to Impose Constraints

NONLIN is also used to add constraints on the parameters. See Section 4.4 for more on constrained optimization.

```
nonlin(parmset=base) g b0 b1 b2
nonlin(parmset=constraint) b0>=0.0 b1>=0.0 b2>=0.0
nlls(parmset=base) ...
nlls(parmset=base+constraint) ...
```

This does **NLLS** on a model, the first time without constraints imposed, the second with non-negativity constraints on the three “b” parameters. Note that constraints are only obeyed for certain types of non-linear optimization methods. In particular, the derivative-free algorithms cannot handle an implicit equality constraint.

Constraints can also apply to all elements of a vector or matrix. For instance, the following creates a PARMSET with the single real variable A and a vector with NLAGS elements B, and constrains the B's to be positive.

```
dec vect b(nlags)
nonlin a b b>=0
```

FRMLS

A FRML is a specialized data type which describes a function of the entry number T. Usually this function includes references to current and lagged data series. For non-linear estimation, it will also use the parameters that are to be estimated.

You can define vectors and arrays of FRMLs, although you must take some special care in defining the elements of these in a loop. FRMLs can also be passed as procedure parameters.

FRMLs used in non-linear estimation need to be created using the instruction **FRML**. FRML's can also be created by **LINREG** and some other instructions from the results of regressions, but these have the estimated coefficients coded into them and thus are of no use for further estimation.

FRML's usually produce a real value, but you can also create formulas which produce matrices—these are used, for instance, by the instructions **CVMODEL** and **DLM**. You need to do a **DECLARE** instruction as the first step in creating such a formula. For instance, to make A a formula which returns a RECTANGULAR, do

```
declare frml[rect] a
```

The Instruction FRML

The **FRML** instruction is used to define the formula or formulas that you want to estimate. **FRML** will usually take the form:

```
frml formula depvar = function(T)
```

The *depvar* (dependent variable) is often unnecessary. Some simple examples:

```
nonlin k a b
frml logistic = 1.0/(k+a*b^t)
```

is a formula for a logistic trend. The parameters K, A and B will be estimated later.

```
nonlin b0 b1 b2 gamma sigmasq
frml olsresid = pcexp-b0-b1*pcaid-b2*pcinc
frml varfunc = sigmasq*(pop^gamma)
```

translates the following into FRMLs: $PCEXP_t - \beta_0 - \beta_1 PCAID_t - \beta_2 PCINC_t$ and $\sigma^2 POP_t^\gamma$ with $\beta_0, \beta_1, \beta_2, \gamma$ and σ^2 representing the unknown parameters.

```
nonlin i0 i1 i2 i3 i4
frml investnl invest = $
    i0+i1*invest{1}+i2*ydiff{1}+i3*gnp+i4*rate{4}
```

translates an investment equation which depends upon current GNP and lags of INVEST, YDIFF and RATE. You could also do this (more flexibly) with:

```
frml(regressors,vector=invparms) investnl invest
# constant invest{1} ydiff{1} gnp rate{4}
nonlin invparms
```

Chapter 4: Non-Linear Estimation

Recursive FRMLs

You can create a FRML to compute a function which depends upon its own value at the previous data point. Such *recursively defined* functions are not uncommon in time series work. For instance, in GARCH models, this period's variance depends upon last period's. In models with moving average terms, this period's residual depends upon last period's. We will demonstrate how to handle this for a geometric distributed lag. (This is for purpose of illustration. You can estimate this more easily with **BOXJENK**).

$$(30) \quad y_t = \beta_0 + \beta_1 \left\{ \sum_{s=0}^{\infty} \lambda^s X_{t-s} \right\}$$

We can generate the part in braces (call it Z_t) recursively by

$$(31) \quad Z_t = X_t + \lambda Z_{t-1}$$

The parameter THETA represents the unobservable

$$(32) \quad Z_0 = \sum_{s=0}^{\infty} \lambda^s X_{-s}$$

At each T, the formula below sets XLAGS equal to the new value of Z , and then uses this value to compute the next entry. %IF is used to distinguish the first observation from the others, as it needs to use THETA for the pre-sample value of Z . (The START option from Section 4.7 provides a more flexible way to handle the different treatment of the first observation).

```
declare real xlags  
nonlin beta0 beta1 theta lambda  
frml geomdlag = (xlags = x + lambda*%if(t==1949:1,theta,xlags)), $  
beta0 + beta1 * xlags
```

Notice that the formula includes two separate calculations: the first computes XLAGS, and the second uses this to create the actual value for the formula. This ability to split the calculation into manageable parts is a great help in writing formulas which are easy to read and write. Just follow each preliminary calculation with a comma. The final value produced by the formula (the expression after the last comma) is the one that is used.

The handling of the initial conditions is often the trickiest part of both setting up and estimating a recursive function. The example above shows one way to do this: estimate it as a free parameter. Another technique is to set it as a “typical” value. For instance, if X were a series with zero mean, zero would not be an unreasonable choice. The simplest way to set this up is to make XLAGS a data series, rather than a single real value.

```
set xlags = 0.0  
nonlin beta0 beta1 lambda  
frml geomdlag = (xlags=x+lambda*xlags{1}),beta0+beta1*xlags
```

When the formula needs an initial lagged value of XLAGS, it pulls in the zero.

Using Sub-FRMLs

Many of the models which you will be estimating will have two or more distinct parts. For instance, in a maximum likelihood estimation of a single equation, there is usually a model for the mean and a model for the variance. While both parts are needed for the complete model, there is no direct interaction between the two. You might very well want to alter one without changing the other.

This can be handled within RATS by defining separate FRMLs for each part, and then combining them into a final FRML. Redefining one of the components has no effect upon the other. For instance,

```
nonlin b0 b1 b2 gamma sigmasq
frml olsresid = pcexp-b0-b1*pcaid-b2*pcinc
frml varfunc = sigmasq*(pop^gamma)
frml likely = %logdensity(varfunc(t),olsresid(t))
```

defines the log likelihood function for a model with Normally distributed errors whose variance is proportional to a power of the series POP. The variance model and regression residuals model are represented by separate formulas. The **LIKELY** formula doesn't need to know anything about the two formulas which it references.

There is one minor difficulty with the way the model above was coded: the single **NONLIN** instruction declares the parameters for both parts. This is where **PARMSETS** can come in handy. If we rewrite this as

```
nonlin(parmset=olsparms) b0 b1 b2
frml olsresid = pcexp-b0-b1*pcaid-b2*pcinc
nonlin(parmset=varparms) gamma sigmasq
frml varfunc = sigmasq*(pop^gamma)
frml likely = %logdensity(varfunc(t),olsresid(t))
```

then the **PARMSET** for the complete model is **OLSPARMS+VARPARMS**.

Creating a FRML from a Regression

In the last example, the **OLSRESID** formula was fairly typical of the “mean” model in many cases: it's linear in the parameters, with no constraints. This could be estimated by **LINREG** if it weren't for the non-standard model of the variance.

This is a relatively small model, but it is possible to have a model for the mean which has many more explanatory variables than this one. Coding these up as formulas can be a bit tedious. Fortunately, **FRML** provides several ways to simplify this process. We demonstrated the **REGRESSORS** option earlier—it takes the right side of the formula from a list of regressors. For example, we can set up this model with the following:

```
frml(regressors,parmset=olsparms,vector=b) olsmodel
# constant pcaid pcinc
nonlin(parmset=varparms) gamma sigmasq
frml varfunc = sigmasq*(pop^gamma)
frml likely = %logdensity(varfunc(t),pcexp-olsmodel(t))
```

Chapter 4: Non-Linear Estimation

The first FRML instruction does the following:

1. Creates OLSMODEL as the formula $B(1) + B(2) * PCAID + B(3) * PCINC$.
2. Puts the 3-element vector B into the PARMSET named OLSPARMS.

The final function has to be altered slightly because OLSMODEL gives the explained part of the model, not the residual. Notice that, with this way of setting up the model, you can change the mean model by just changing the list of explanatory variables on the supplementary card.

You can also create formulas following a LINREG instruction by using FRML with the LASTREG option, or by using the EQUATION option to convert an estimated equation. For instance, the following sets up the same model, but uses the least squares estimates for the guess values for the parameters in the mean model, and the (constant) least squares variance as the guess for the variance model.

```
linreg pcexp
# constant pcaid pcinc
frml (lastreg,parmset=olsparms,vector=b) olsmodel
nonlin(parmset=varparms) gamma sigmasq
compute sigmasq=%sigmasq,gamma=0.0
frml varfunc = sigmasq*(pop^gamma)
frml likely = %logdensity(varfunc(t),pcexp-olsmodel(t))
```

Defining FRMLs in a Loop

You can create VECTORS or other arrays of FRMLs. This can be very handy when you have a large number of FRMLs with a similar form. You have to be careful, however, if the FRML's are defined in a loop. Wherever you use the loop index in the formula definition, you must prefix it with the & symbol.

```
dec vector b(n)
dec vect[frml] blackf(n)
nonlin gamma b
do i=1,n
  frml blackf(i) s(i) = (1-b(&i))*gamma+b(&i)*market
end do i
```

The &i's are needed in the formula because $(1-B(i)) * GAMMA + B(i) * MARKET$ is a perfectly good formula, which would be calculated using the value of i at the time the formula is *used*, not at the time it was defined. The &i returns the value of i as the formula is defined, so, for example, the first formula is defined as:

```
S(1) = (1-B(1))*GAMMA+B(1)*MARKET
```

rather than

```
S(i) = (1-B(i))*GAMMA+B(i)*MARKET
```

4.7 The START and ONLYIF Options

The START option

The **START** option is available on the non-linear estimation instructions **DLM**, **GARCH**, **MAXIMIZE**, **NLLS**, and **NLSYSTEM**, as well as the data transformation instructions **SET**, **CSET**, **GSET** and **SSTATS**. This evaluates an expression once per function evaluation, before the main function being estimated is evaluated. The **START** expression is *not* re-evaluated at each time period, so it is appropriate only for portions of an estimation that are time-invariant. This can be used to handle pre-sample values or time-consuming calculations that are time-invariant.

For instance, our original coding for the recursive formula on page UG–132 (with the pre-sample value included as a parameter) can be rewritten as:

```
declare real xlags
nonlin beta0 beta1 theta lambda
frm1 geomdlag = (xlags=x+lambda*xlags),beta0 + beta1 * xlags
```

with the option **START=(XLAGS=THETA)** on the instruction which estimates the model. This will cause **XLAGS** to be initialized to the current value of **THETA** at the start of every function evaluation. Note that this will no longer require coding the start period into the function, since the **START** is executed before the estimation period whenever that might be.

START is particularly useful for the **DLM** instruction, which estimates state-space models. Most inputs to **DLM** aren't time-varying, but some may depend upon parameters that we need to estimate, and some can be quite large and complicated.

If you need to do several calculations in your **START** option, you need to enclose them in the “empty function” **% (. . .)**, for instance

```
START=% (XLAGS=THETA, BETA1=EXP (LBETA) )
```

The **% (. . .)** is just to group the calculations together into a single option, since the comma alone is used to separate options.

The ONLYIF option

The **START** option is evaluated before anything else on a function evaluation. In some cases, most of the work in the estimation is handled by this. **ONLYIF** is very similar to **REJECT** (page UG–124), except the test works the opposite way, and **ONLYIF** applies before **START**, while **REJECT** is applied after it. **ONLYIF** can be used to avoid doing the **START** calculation for values for which that would be illegal (or just poorly-behaved). For instance, if the **START** calculation would fail for values of **RHO** bigger than one, you would want the combination **ONLYIF=(RHO<=1.0)**, **START=start calculation**. If **START** computes a value **P** which might be outside a legal range of [0,1], you would want to use **START=start calculation, REJECT=(P<0.OR.P>1)**.

4.8 Non-Linear Least Squares

Background

RATS can estimate, by non-linear least squares, models of the form

$$(33) \quad y_t = f(X_t, \beta) + u_t$$

with objective function

$$(34) \quad \sum_{t=1}^T u_t^2$$

This is done using the instruction **NLLS**. **NLLS** can also be used for GMM/non-linear two-stage least squares (Section 4.9), while **NLSYSTEM** (Section 4.10) performs non-linear *systems* estimation and multivariate GMM, and **MAXIMIZE** (4.11) is designed to handle more general problems.

The Instruction NLLS

The syntax for the **NLLS** instruction, which does the actual estimation is:

```
nlls (frml=formula, other options)   depvar start end resids
```

Before you can run **NLLS** you need to:

- Set the parameter list using **NONLIN**.
- Set up the function using **FRML**.
- Set initial values for the parameters, using, for instance, **COMPUTE**.

For instance, the following estimates the parameters for a CES production function

```
log Qt = log γ -  $\frac{\nu}{\rho}$  log ( δ Kt-ρ + (1 - δ) Lt-ρ ) + ut

nonlin lgamma delta nu rho
frml ces = lgamma-nu/rho* $
      log( delta * k^(-rho) + (1.-delta) * l^(-rho) )

compute lgamma=1.0
compute delta=0.4
compute      nu=0.8
compute      rho=0.6

nlls (frml=ces, trace) logq
```

Technically, **NLLS** is set up for models strictly of the form (33). However, you can use * in place of *depvar*—**NLLS** interprets this as a series of zeros. Since R^2 and related statistics are meaningless in such cases, they are omitted.

By default, **NLLS** estimates using the Gauss–Newton algorithm, which is described in Section 4.2. The covariance matrix is calculated as detailed in 4.5. However, it also

supports the derivative-free methods such as `SIMPLEX` (Section 4.3). These can be applied if the function is continuous but not differentiable. They can also be used to refine initial guesses before using the standard method, or to make a broader search of the parameter space.

Example

The example `NLLS.RPF` analyzes a consumption function of the form

$$C = \beta_0 + \beta_1 Y D^{\beta_2}.$$

This is adapted from Examples 6.7 and 6.8 from Martin, Hurn and Harris(2013). Some non-linear models have obvious guess values, and this is one of them, as, with $\beta_2=1$, it is linear. The basic steps for estimating the model are:

```
nonlin beta0 beta1 beta2
frml nlconst cons = beta0+beta1*inc^beta2
linreg cons
# constant inc
compute beta0=%beta(1),beta1=%beta(2),beta2=1.0
nlls(frml=nlconst)
```

Hypothesis tests

You can test hypotheses on the coefficients using **TEST**, **RESTRICT** and **MRESTRICT**. Coefficients are numbered by their positions in the parameter vector. You cannot use **EXCLUDE** since it requires a variable list rather than coefficients. In this case, we can test for $\beta_2=1$ with

```
test(title="Test of linearity")
# 3
# 1.0
```

The hypothesis tests are based upon a quadratic approximation to the sum of squares surface at the final estimates. This is the “Wald” test. If you estimated using the option `ROBUSTERRORS`, they will report a chi-squared test. If not, they will report an *F*. The *F* is not strictly valid, as the numerator in the *F* is only asymptotically chi-squared; however, it most likely has better small sample properties than the asymptotic distribution, since it allows for the fact that the variance of the residuals is estimated and not a known constant.

A likelihood ratio test can be constructed easily since **NLLS** computes the log likelihood and puts it into `%LOGL`. Save the `%LOGL` from the constrained model (which here is just the **LINREG**) and use **CDF**:

```
cdf(title="LR Test of linearity") chisqr 2.0*(loglunr-%logl) 1
```

Constrained Estimation

NLLS can handle parameter sets with constraints imposed. However, it doesn’t use the Gauss–Newton algorithm in this case. Instead, it uses constrained BFGS as documented in Section 4.4.

4.9 Method of Moments Estimators (Univariate)

RATS can produce generalized method of moments estimators (Hansen, 1982) for models whose orthogonality conditions can be expressed as

$$(35) \quad E[Z_t' f(y_t, X_t, \beta)] = 0$$

for a closed form function f and a list of instruments Z which does not depend upon θ . This is a very broad class, which includes two-stage least squares and other instrumental variables estimators. If f is linear, this can be done with the simpler **LINREG** (Section 2.5), so here we will be dealing with models where f is non-linear. The extension to multiple equations is treated in Section 4.10. As with **LINREG**, you do instrumental variables by setting the instruments list using **INSTRUMENT** and including the **INST** option when estimating. The following estimates the same basic model as in the previous section (taken from a different textbook, so it uses different variable names), but by non-linear two-stage least squares, using lags of consumption and income as the instruments.

```
nonlin a b g
linreg realcons
# constant realdpi
compute a=%beta(1),b=%beta(2),g=1
frml cfrml realcons = a+b*realdpi^g
*
instruments constant realcons{1} realdpi{1 2}
nlls(frml=cfrml,inst) realcons 1950:3 *
```

For model (33) and for instrument set Z , nonlinear two-stage least squares minimizes

$$(36) \quad \left(\sum_{t=1}^T u_t Z_t \right) \mathbf{W} \left(\sum_{t=1}^T Z_t' u_t \right), \text{ where } \mathbf{W} = \left(\sum_{t=1}^T Z_t' Z_t \right)^{-1}$$

This is what **NLLS** does by default. Most of the remainder of the section describes other ways to compute $\mathbf{W}$.

The OPTIMALWEIGHTS and WMATRIX Options

NLLS with the **INST** option minimizes the quadratic form in (36) for some choice of $\mathbf{W}$, which weights the orthogonality conditions. Hansen shows that a more efficient estimator can be obtained by replacing the two-stage least squares weight matrix $(\mathbf{Z}'\mathbf{Z})^{-1}$ by the inverse of

$$(37) \quad \text{mcov}(\mathbf{zu}) = \sum_{k=-L}^L \sum_t Z_t' u_t u_{t-k} Z_{t-k}$$

This matrix plays a key role in the covariance matrix calculations described in Section 2.2, and some of its numerical properties are discussed there.

RATS provides the option **OPTIMALWEIGHTS** on **LINREG** and **NLLS** for doing this estimation directly. For example:

```
instruments constant z1{1 to 6}
linreg(inst,optimalweights,lags=2,lwindow=newey) y1
# constant x1 x2 x3
```

For **LINREG**, **OPTIMALWEIGHTS** tells RATS to first compute the two-stage least squares estimator, use the residuals to compute the matrix shown in (37) and take its inverse as the weighting matrix. This is a “two-step” estimator. **NLLS** is similar, but isn’t as simple because, unlike a linear model, the first step can’t be done as a single matrix calculation since it requires minimization across the parameters. Instead of two-step, **NLLS** resets the weight matrix with each iteration using the current residuals in computing (37). With either instruction, you can retrieve the last weight matrix used as the SYMMETRIC array %WMATRIX.

The example file **GIV.RPF** estimates the free parameters (the discount rate β and coefficient of risk aversion σ) where the first-order condition for allocation of consumption across time yields the condition:

$$E_{t-1} \left[\beta R_t (C_t / C_{t-1})^\sigma - 1 \right] = 0$$

Theoretically, anything that’s part of the information set at time $t-1$ can be used as an instrument. The following uses the **CONSTANT** (which will almost always be an instrument) and six lags of the two data series and uses the **OPTIMALWEIGHTS** calculation with no lags in (37). (You wouldn’t expect serial correlation in the moment conditions for this model.)

```
nonlin discount riskaver
frml h = discount*realret(t)*consgrow(t)^riskaver-1
compute discount = .99,riskaver = -.95
*
instruments constant consgrow{1 to 6} realret{1 to 6}
nlls(inst,frml=h,optimal) *
```

If you wish to provide your own weighting matrix, you can use the **WMATRIX** option on **LINREG** or **NLLS**. These steps would be used to compute your own weight matrix:

1. Estimate the model in the standard way, saving the residuals.
2. Use **MCOV** to compute the $\text{mcov}(\mathbf{Z}, \mathbf{u})$ matrix. To get the proper scale for the covariance matrix, the weight matrix needs to be the inverse of a matrix which is $O(T)$.
3. Re-estimate the model with the option **WMATRIX=new weighting matrix** (if you’ve computed the inverse) or **IWMATRIX=mcov matrix** (matrix before inversion). If this is an (intentionally) sub-optimal weight matrix, you can use **ROBUSTERRORS**, **LAGS** and **LWINDOW** to correct the covariance matrix.

J-Tests and the %UZWZU Variable

Another result from Hansen is that if optimal weights are used, the value of $\mathbf{u}'\mathbf{Z}\mathbf{W}\mathbf{Z}'\mathbf{u}$ is asymptotically distributed χ^2 with degrees of freedom equal to the number of overidentifying restrictions. (If the number of instruments equals the number of parameters, the model is just identified and $\mathbf{u}'\mathbf{Z}\mathbf{W}\mathbf{Z}'\mathbf{u}$ will be zero, at least to machine-precision.) He proposes this as a test of these assumptions—a test sometimes known as the *J*-test (page UG–98). **LINREG** and **NLLS** compute and print the *J*-statistic and its significance level using the weight matrix you supply, or the optimal weight matrix if you use the **OPTIMALWEIGHTS** option. This is included in the regression output, and is available afterwards in the variables %JSTAT, %JSIGNIF and %JDF (test statistic, significance level, and degrees of freedom respectively).

In **GIV.RPF**, the following is used to do specification tests using several different lag lengths on the instruments. Note that this uses a common estimation range (determined by the six lag set which was estimated first).

```
compute start=%regstart()
dofor nlag = 1 2 4 6
    instruments constant consgrow{1 to nlag} realret{1 to nlag}
    nlls(inst,noprint,frml=h,optimal) * start *
    cdf(title="Specification Test for "+nlag+" lags") $
        chisqr %jstat 2*nlag-1
end dofor
```

If the weight matrix used in estimation isn't the "optimal" one, there are two ways to adjust the specification test. One uses the alternative form for the test statistic (Hansen's Lemma 4.1). The other, proposed by Jagannathan and Wang (1996), uses the standard *J*-statistic but uses its (non-standard) distribution. You choose between these with the option **JROBUST=STATISTIC** or **JROBUST=DISTRIBUTION**.

In addition, the **LINREG** and **NLLS** instructions (and the systems estimators **SUR** and **NLSYSTEM**) define the variable %UZWZU. %UZWZU is the value of $\mathbf{u}'\mathbf{Z}\mathbf{W}\mathbf{Z}'\mathbf{u}$ for whatever weight matrix is used. In the case of simple 2SLS, it will be the *J*-statistic times the estimated variance. Note that if the model is just identified (that is, if the number of free coefficients is equal to the number of instruments), %UZWZU is zero, theoretically. It may be very slightly different due to a combination of computer roundoff error and (for non-linear models) the differences between the estimated and true parameters.

The ZUMEAN option

The model (35) can be a bit too restrictive in some applications. For instance, in financial econometrics, it's not uncommon for a model to generate something similar to (35), but with a fixed but non-zero value for the expectation. The calculations done for GMM can easily be adjusted to allow for such a fixed mean for the moment conditions. To do this with RATS, use the `ZUMEAN=vector of expected values`. The vector of expected values should have the same dimensions as the set of instruments.

```
nonlin delta gamma
frml fx = delta*cons^(-gamma)
instruments r1 to r10
compute [vect] zumean=%fill(10,1,1.0)
nlls (frml=fx,inst,zumean=zumean,optimalweights)
```

The CENTER option

If the model is just identified

$$(38) \sum_t Z'_t u_t$$

will be equal to zero. If the model is overidentified, at least some elements would be expected to be non-zero. Hall(2000) recommends subtracting off the sample mean from each Zu term in (37), that is, to compute

$$(39) \sum_{k=-L}^L \sum_t (Z'_t u_t - \alpha_{zu}) (Z'_{t-k} u_{t-k} - \alpha_{zu})'$$

This will have no effect if the model is just-identified (the weight matrix has no effect on the estimates in that case anyway), but he shows that it improves the performance of specification tests for the overidentifying restrictions.

4.10 Non-Linear Systems Estimation

NLSYSTEM vs. SUR

The instruction **NLSYSTEM** estimates a system of equations by non-linear least squares or (for instrumental variables) by the generalized method of moments. The analogous instruction for linear systems is **SUR**, which is described in Section 2.7. **NLSYSTEM** uses formulas (FRMLS) rather than the equations used by **SUR**. For models which can be estimated with both instructions, it is slower than **SUR** for two reasons:

- **SUR** does not have to *compute* derivatives with each iteration (the derivatives of the linear functions are just the explanatory variables).
- **SUR** uses every possible means to eliminate duplicate calculations. The cross-products of regressors, dependent variables, and instruments can be computed just once.

For a linear model without complicated restrictions, use **SUR**. For a linear model with restrictions, **NLSYSTEM** may be simpler to set up and the speed difference is likely to matter only with a large model or very large data set.

NLSYSTEM vs. NLLS

Like **NLLS**, **NLSYSTEM** can do either instrumental variables or (multivariate) least squares. In most ways, **NLSYSTEM** is just a multivariate extension of **NLLS**. There are three important differences, though:

- When you use **NLLS**, you specify the dependent variable on the **NLLS** instruction itself. With **NLSYSTEM**, you *must* include the dependent variable when you define the FRML.
- **NLSYSTEM** with **INST** will automatically compute the optimal weighting scheme described in the last section.
- Primarily because of the preceding point, you will rarely use **ROBUSTERRORES** with **NLSYSTEM**. **ROBUSTERRORES**, in fact, is a request to compute a sub-optimal estimator and correct its covariance matrix.

Technical Details

(40) $\mathbf{u}_t = (u_{1t}, \dots, u_{nt})'$ is the vector of residuals at time t ($\mathbf{u}$ depends upon θ), and

$$(41) \quad \Sigma = E\mathbf{u}_t\mathbf{u}_t'$$

Multivariate non-linear least squares solves

$$(42) \quad \min_{\beta} \sum_t \mathbf{u}_t' \Sigma^{-1} \mathbf{u}_t$$

For instrumental variables, further define

$$(43) \quad \mathbf{Z}_t = (z_{1t}, \dots, z_{rt})'$$
 as the vector of instruments at t , and

$$(44) \quad \mathbf{G}(\beta) = \sum_t \mathbf{u}_t \otimes \mathbf{Z}_t$$

then Generalized Method of Moments solves

$$(45) \quad \min_{\beta} \mathbf{G}(\beta)' [\mathbf{SW}] \mathbf{G}(\beta)$$

where $\mathbf{SW}$ is the system weighting matrix for the orthogonality conditions. By default, it is just $\Sigma^{-1} \otimes (\mathbf{Z}'\mathbf{Z})^{-1}$, where $\mathbf{Z}$ is the $T \times r$ matrix of instruments for the entire sample.

In either case, the estimation process depends upon certain “nuisance parameters”—the estimates will change with Σ and $\mathbf{SW}$. Unless you use options to feed in values for these, **NLSYSTEM** recomputes them after each iteration. The objective function is thus changing from one iteration to the next. As a result, the function values that are printed if you use the **TRACE** option will often seem to be moving in the wrong direction, or at least not changing. However, the old versus new values on any single iteration will always be improving.

The continuous recalculation of weight matrices for instrumental variables can sometimes create problems if you use the **ZUDEP** option. This allows for general dependence between the instruments (“ $\mathbf{Z}$ ”) and residuals (“ $\mathbf{u}$ ”). You may not be able to freely estimate the covariance matrix of $n \times r$ moment conditions with the available data. Even if $n \times r$ is less than the number of data points (so the matrix is at least invertible), the weight matrices may change so much from iteration to iteration that the estimates never settle down. If this happens, you will need to switch to some type of “sub-optimal” weight matrix, such as that obtained with **NOZUDEP**, and use the **ROBUSTERRORS** option to correct the covariance matrix.

Multivariate Least Squares

The first order necessary conditions for minimizing (42) are

$$(46) \quad \sum_t \frac{\partial \mathbf{u}_t'}{\partial \beta} \Sigma^{-1} \mathbf{u}_t = 0$$

Ignoring the second derivatives of $\mathbf{u}$, a first order expansion of the left side in (46) (which is the gradient $\mathbf{g}$) at θ_k is

$$(47) \quad \sum_t \frac{\partial \mathbf{u}_t'}{\partial \beta} \Sigma^{-1} \mathbf{u}_t + \left(\sum_t \frac{\partial \mathbf{u}_t'}{\partial \beta} \Sigma^{-1} \frac{\partial \mathbf{u}_t}{\partial \beta} \right) (\beta - \beta_k)$$

Setting this to zero and solving for θ puts this into the general “hill-climbing” framework (if minimization is converted to maximization) from Section 4.3 with

$$(48) \quad \mathbf{G} = \left(\sum_t \frac{\partial \mathbf{u}_t'}{\partial \beta} \Sigma^{-1} \frac{\partial \mathbf{u}_t}{\partial \beta} \right)^{-1}$$

$\mathbf{G}$ ends up being the estimate of the covariance matrix of the estimates. If you use **ROBUSTERRORS**, the recomputed covariance matrix is

$$(49) \quad \mathbf{G} \text{ mcov}(\mathbf{v}, 1) \mathbf{G}, \text{ where}$$

Chapter 4: Non-Linear Estimation

$$(50) \quad \mathbf{v}_t = \frac{\partial \mathbf{u}'_t}{\partial \beta} \Sigma^{-1} \mathbf{u}_t \quad (\text{the summands from (46)})$$

Example of Multivariate Least Squares

The example `CONSUMER.RPF` uses **NLSYSTEM** to estimate equations from an expenditure system given in Theil (1971), Section 7.6. It works with four groups of commodities: Food, Vice, Durable Goods and Remainder. Data for each consists of its value share $\times$ (percent) change in quantity (`WQ . . .`) and the (percent) change in price (`DP . . .`). For commodity group i , we have the equation

$$(51) \quad WQ_{it} = u_i DQ_t + \sum_j \pi_{ij} DP_{jt} + \varepsilon_{it}$$

This system of equations is redundant, as the sum of the ε over i must be zero. We thus estimate only three of the four equations. We are interested in two constraints:

$$(52) \quad \sum_j \pi_{ij} = 0 \quad (\text{homogeneity})$$

$$(53) \quad \pi_{ij} = \pi_{ji} \quad (\text{symmetry})$$

To simplify the former, we parameterize the equations using an “excess” parameter for π_{i4} . This parameter has a value of 0 if constraint (52) holds.

We use different `PARMSETS` to handle the different problems. For instance, the `PARMSET` “RELAX”, when added in with the base parameter set, relaxes the homogeneity assumptions by allowing the π_{i4} to be estimated. The `PARMSET` “SYMMETRY”, when added to the others, imposes the symmetry constraints:

```
nonlin(parms=base) mu1 mu2 mu3 p11 p12 p13 p21 p22 p23 p31 p32 p33
nonlin(parms=relax) ep14 ep24 ep34
nonlin(parms=symmetry) p12=p21 p13=p31 p23=p32
```

The three `FRML`’s for the three equations we will estimate are:

```
frml ffood wqfood = mu1*dq+p11*(dpfood-dprem)+$
      p12*(dpvice-dprem)+p13*(dpdura-dprem)+ep14*dprem
frml fvice wqvice = mu2*dq+p21*(dpfood-dprem)+$
      p22*(dpvice-dprem)+p23*(dpdura-dprem)+ep24*dprem
frml fdura wqdura = mu3*dq+p31*(dpfood-dprem)+$
      p32*(dpvice-dprem)+p33*(dpdura-dprem)+ep34*dprem
```

Because the equations are linear, the guess values aren’t particularly important, and we start everything at 0. The ones that are really important to be zero are the `EP14`, `EP24` and `EP34` since zeroing those *imposes* homogeneity. That’s why those get reset before the third **NLSYSTEM**:

```
compute mu1=mu2=mu3=0.0
```



```
compute p11=p12=p13=p21=p22=p23=p31=p32=p33=0.0
compute ep14=ep24=ep34=0.0
nlsystem(parmset=base) / ffood fvice fdura
nlsystem(parmset=base+relax) / ffood fvice fdura
compute ep14=ep24=ep34=0.0
nlsystem(parmset=base+symmetry) / ffood fvice fdura
```

GMM Estimation

GMM estimation uses similar methods—the first order conditions of the optimization problem (45) are expanded, ignoring the second derivative of the residuals. The estimated covariance matrix of coefficients is

$$(54) \quad \mathbf{A} = \left(\frac{\partial \mathbf{G}'}{\partial \beta} [\mathbf{SW}] \frac{\partial \mathbf{G}}{\partial \beta} \right)^{-1}$$

If you use the **ROBUSTERRORS** option (with a sub-optimal weight matrix), the covariance matrix becomes **ABA**, with **B** dependent on the options chosen:

with **NOZUDEP**, **ROBUSTERRORS**, and **LAGS=0**:

$$(55) \quad \mathbf{B} = \left(\frac{\partial \mathbf{G}}{\partial \beta} \right)' [\mathbf{SW}] (\Sigma \otimes \mathbf{Z}'\mathbf{Z}) [\mathbf{SW}] \left(\frac{\partial \mathbf{G}}{\partial \beta} \right)$$

With **ZUDEP**, **ROBUSTERRORS**, and **LAGS=0**, or **NOZUDEP**, **ROBUSTERRORS**, and **LAGS>0**

$$(56) \quad \mathbf{B} = \left(\frac{\partial \mathbf{G}}{\partial \beta} \right)' [\mathbf{SW}] \text{mcov}(\mathbf{Z} \otimes \mathbf{u}, 1) [\mathbf{SW}] \left(\frac{\partial \mathbf{G}}{\partial \beta} \right)$$

Example of Multivariate GMM

The example file **CHANKAROLYI.RPF** estimates the model for interest rates from Chan, et al (1992). The model generates a conditional mean and a conditional variance. Note how the **VARIANCE** formula uses the square of the **EPS** formula:

```
nonlin alpha beta gamma sigmasq
frml eps = y1{-1}-(1+beta)*y1-alpha
frml variance = eps(t)^2-sigmasq*y1^(2*gamma)
```

Different information sets (instruments) and different ways of computing the weight matrix will give different estimates. The example does four **NLSYSTEMS**, two of which show here are the just-identified and overidentified (2 conditions x 3 instruments vs 4 free parameters) estimated with the default **NOZUDEP** handling:

```
instruments constant y1
nlsystem(instruments) / eps variance
instruments constant y1{0 1}
nlsystem(instruments) / eps variance
```

4.11 More General Maximization

The Instruction **MAXIMIZE**

MAXIMIZE is designed for many of the estimation problems that specialized instructions like **LINREG** and **NLLS** cannot handle. Its primary purpose is to estimate models by maximum likelihood, but it is more general than that.

MAXIMIZE is perhaps most often used for estimating the less-standard variations of ARCH and GARCH models that the built-in **GARCH** instruction can't handle. These models are covered in detail in Chapter 9.

The problems that **MAXIMIZE** can solve are those of the form

$$(57) \max_{\beta} \sum_{t=1}^T f(y_t, X_t, \beta)$$

where f is a RATS formula (FRML). Things to note about this:

- **MAXIMIZE** only does maximizations (directly), but can, of course, do minimizations if you put a negative sign in front of the formula when you define it.
- **MAXIMIZE** does not check differentiability and will behave unpredictably if you use methods **BFGS** or **BHHH** with a function that is not twice-differentiable. Differentiability is not an issue with the derivative-free methods (Section 4.3).

If even this is too narrow for your application, you will need to try the instruction **FIND**—see Section 4.12.

Setting Up for **MAXIMIZE**

The steps in preparing to use **MAXIMIZE** are the same as for **NLLS** (Section 4.8):

- Set the parameter list using **NONLIN**.
- Set up the function using **FRML**.
- Set initial values for the parameters, using **COMPUTE** or **INPUT**. In some cases, such as recursive ARCH/GARCH models, you will also need to initialize one or more series used to hold values such as residuals or variances.

Most of the ideas described in Section 4.6 for simplifying the specification of a model through the use of “sub-FRMLs” and separate **PARMSETS** are most likely to be used when the final FRML is to be estimated using **MAXIMIZE**.

Example

Example 16.9 in Greene (2008) estimates a stochastic frontier model. This is basically a log-linear production model, except that the residuals have two components, one of which is not permitted to be positive—you can fall short of, but can't exceed, the (unobservable) production function. The model is written

$$y_t = X_t\beta + v_t - u_t, u_t \geq 0, v_t \sim N(0, \sigma_v^2)$$

Without the u_t , this is a conventional model which can be estimated by least squares. Under the assumption that u_t is “half-Normal”, that is, $u_t \sim N(0, \sigma_u^2) \times I_{[0, \infty)}$, the log likelihood for entry t can be written most conveniently as

$$\log f_N(\varepsilon | \sigma^2) + \log 2F_N(-\lambda\varepsilon | \sigma^2), \text{ where } \varepsilon = y_t - X_t\beta, \sigma^2 = \sigma_v^2 + \sigma_u^2, \lambda = \frac{\sigma_u^2}{\sigma_v^2 + \sigma_u^2}$$

and f_N and F_N are the density and distribution functions for the Normal with the given variance. (The formula in Greene looks slightly different, but is equivalent). The 2 multiplier attached to the F_N adjusts for the u_t being supported on just half the real line, thus requiring doubling of the integrating constant. If $\sigma_u^2 = 0$, then $\sigma^2 = \sigma_v^2$ and $\lambda = 0$, so this collapses (as it should) to the least squares log likelihood $\log f_N(\varepsilon | \sigma^2)$.

The Normal density and distribution functions are provided as %DENSITY and %CDF, though the more convenient function for the density for likelihood calculations is %LOGDENSITY which takes the form %LOGDENSITY(VARIANCE, X). Note that all density functions in RATS include their integrating constants.

We need to estimate the parameters in the regression function, plus λ and σ . If we want, we can always solve out for the component variances from those. An obvious source for guess values is the linear regression. However, $\lambda = 0$ is on the boundary, so we need to start that away from 0. The full example is MAXIMIZE.RPF. The most important part of that is:

```
linreg logq
# constant logk logl
nonlin b0 bk bl sigma lambda
compute b0=%beta(1),bk=%beta(2),bl=%beta(3)
compute sigma=sqrt(%seesq),lambda=.1

frml resid = logq-b0-bk*logk-bl*logl
frml frontier = eps=resid,$
    %logdensity(sigma^2,eps)+log(2*%cdf(-eps*lambda/sigma))

maximize(method=bfgs,title=$
    "Frontier Model with 1/2 Normal Errors") frontier
```

This estimates the linear regression, sets up the parameter set, uses **COMPUTE** to set the guess values, defines the formula **RESID** for the residual ε , uses that to define the log likelihood formula **FRONTIER**, and maximizes the sum of **FRONTIER** across the data set. Note the use of the **TITLE** option on **MAXIMIZE**.

Chapter 4: Non-Linear Estimation

Multivariate Likelihoods

You can use **MAXIMIZE** to estimate multivariate Normal likelihood functions using the `%LOGDENSITY` function. (`%LOGDENSITY` can be used for both univariate and multivariate Normals). The log likelihood element for an n -vector at time t is, in general,

$$(58) \quad -\frac{n}{2} \log 2\pi - \frac{1}{2} \log |\Sigma_t| - \frac{1}{2} \mathbf{u}'_t \Sigma_t^{-1} \mathbf{u}_t$$

This is computed by the function `%LOGDENSITY(SIGMA,U)` where `SIGMA` is the covariance matrix and `U` the vector of deviations from the means of the components.

Although `%LOGDENSITY` is by far the most commonly used, RATS offers more than a dozen other density functions for other distributions, including `%LOGTDENSITY` and `%LOGGEDDENSITY` for t and GED distributions, respectively. All of these density functions include the integrating constants, so models estimated using different distributions will give comparable likelihood values.

The biggest problem is creating the FRML which will allow you to compute this as the final step. It is often a good idea to create a `FRML[VECTOR]` to compute `U` and a `FRML[SYMMETRIC]` for `SIGMA`. Of course, if Σ doesn't depend upon time, you don't need to create a FRML just to compute it. If all parameters of Σ are freely estimated, you can even **DECLARE** and **DIMENSION** it and add it to the nonlinear parameter set as a `SYMMETRIC` array.

The following adds on to `CONSUMER.RPF` to estimate by maximum likelihood. Because Σ is unconstrained, this should give identical answers to the first **NLSYSTEM** instruction. As just mentioned, the covariance matrix is put into the parameter set `SIGMAPARMS` as an unrestricted 3×3 matrix.

```
dec frml[vect] ufrml
frml ufrml = ||wqfood-ffood,wqvice-fvice,wqdura-fdura||

dec symm sigma(3,3)
nonlin(parmset=sigmaparms) sigma
frml mvlikely = %logdensity(sigma,ufrml)

compute sigma=%identity(3)

maximize(parmset=base+sigmaparms,itors=400) mvlikely
```

4.12 Using FUNCTION

While FRMLs are general enough to handle a very wide range of non-linear optimizations through instructions like **NLLS** and **MAXIMIZE**, there are others that fit into one of those general frameworks, but have a function that is too complicated to be computed using a single formula. You may have a numerical integral, or a less standard transcendental or probability density function that can be computed using a sequence of RATS instructions, but can't be done with just one. In those situations, you can use a **FUNCTION** to do the required calculation. A **FUNCTION** is a sequence of RATS instructions which takes a set of parameters (perhaps empty) and produces a return value.

The following, for instance, returns the “digamma” function (or a very good numerical approximation of it) of its argument z . The digamma is the derivative of the $\log \Gamma$ function. Since this example was originally written, we have added a built-in `%DIGAMMA` function to RATS for doing this computation. However, we think the techniques demonstrated in this example program may be helpful. We use `%%` as a prefix for user-defined functions, to distinguish them from built-in functions and variables:

```
function %%DiGamma z

if z<=0 {
    compute %%DiGamma=%NA
    return
}
compute zp=z
compute %%DiGamma=0.0
while (zp<30) {
    compute zpr=1/zp
    compute %%DiGamma=%%DiGamma-zpr
    compute zp=zp+1
}
compute zpr=1/zp
compute %%DiGamma=%%DiGamma+log(zp)+$
    zpr*(-.5+zpr*(-1.0/12.0+zpr^2*(1.0/120.0-zpr^2/252.0)))

end
```

Once you've executed these lines, you can use `%%digamma(argument)` to get the digamma function of the argument.

The **COMPUTE** `%%DiGamma` instructions are the ones that set the return value. Whatever is the last value assigned to the function name before a **RETURN** or the **END** of the function is the value the function returns. For instance, here we return `%NA` if the argument is out of range (non-positive) right at the start; otherwise, a recursion is used to push the argument to a level where a simple series expansion can be employed.

Chapter 4: Non-Linear Estimation

If this were intended as a “production” version of the function, there would be a minor change which would be handy. This uses the two variables `ZP` and `ZPR` in addition to function name `%%DIGAMMA` and argument `Z`. As this function is written, those would go into the regular symbol table, where they might conflict with names used in a main program. A slight improvement would be to add the line

```
local real zp zpr
```

right after the **FUNCTION** line. This makes those two variables “local” to the function, so you don’t have to worry about hitting a name in the caller. **LOCAL** is like **DECLARE**, but is used only within a **FUNCTION** or **PROCEDURE**. Local variables are described in Section 15.2.

When you write a function, RATS assumes that the function value and the parameters are all real-valued. They can, however, take other types. You indicate this with the **TYPE** instruction, which needs to be included early in the function code, before any instructions which actually do anything. The following returns the sum of the logs of the diagonal elements of a SYMMETRIC matrix.

```
function %%sumlogdiag a  
type symmetric a  
local integer i  
compute %%sumlogdiag=0.0  
do i=1,%rows(a)  
    compute %%sumlogdiag=%%sumlogdiag+log(a(i,i))  
end do i  
end
```

Note, by the way, that this could be done without **FUNCTION** using built in expressions: `%SUM(%LOG(%XDIAG(A))`. In general, it’s better to use the built-in functions where possible.

An example of the use of a **FUNCTION** for doing non-linear least squares with a complicated objective function is provided on the example `BONDS.RPF`. Suppose that we want to estimate a yield-curve using data from many bonds. For each bond, we have the following information:

- Periods to maturity (series `MATURITY`).
- The coupon (series `COUPON`).
- The current market value (series `VALUE`).

We will model the function $\delta(T)$, which gives the present value of \$1 at T , by

$$(59) \quad \delta(T) = \exp\left(-T\left(a_0 + a_1 T + a_2 \max(0, T - C)\right)\right)$$

The term multiplying $-T$ is a piecewise linear function with a join point at C (which will be called `CUSP` in the program).

We need to minimize over the parameters of (59) the gaps between the present values predicted by the model and the actual market values. However, the present value for a bond (which is a single entry in the data set) requires a sum over all the coupons due until maturity. We can't use annuity formulas for this because the interest rate isn't fixed. So we create a **FUNCTION** that takes as its argument the number of the bond and uses the data in the three series described above to compute the present value of the bond given the parameter settings being evaluated. The start of is:

```
function BondPV bond
type real BondPV
type integer bond
local real mdate cdate
```

In practice, you would start with those first three lines and add the **LOCAL** after you've written the rest, once you know what other local variables you will need. The **FUNCTION** code needs to give a value to **BondPV** before it returns. In this case, we will keep setting and resetting the value of this as we add terms to the present value. To start, we take the discounted value of the redemption at maturity:

```
compute mdate=maturity(bond)
compute BondPV=100.0 * $
      exp(-mdate*(a0+mdate*a1+%max(mdate-cusp,0.0)*a2))
```

The next part of this walks backwards through the coupon payments as long as there are full coupon periods, adding the discounted value to what's in **BondPV**.

```
compute cdate=mdate
while (cdate > 0.0) {
      compute BondPV=BondPV + coupon(bond) * $
            exp(-cdate*(a0+cdate*a1+%max(cdate-cusp,0.0)*a2))
      compute cdate=cdate-1
}
```

And finally we adjust for the simple interest payable by the purchaser for the initial coupon. **CDATE** will be $-(\text{fraction of period})$. Note that you don't need a **RETURN**, just **END** to indicate that the function is finished.

```
compute BondPV=BondPV+coupon(bond)*cdate
end
```

With that done, we still need a **FRML** which gives the value for each entry. Here, all this **FRML** needs to do is return the value of the function for entry T . The first three lines are best placed before the **FUNCTION** since it needs **A0**, **A1**, **A2** and **CUSP**:

```
nonlin a0 a1 a2
compute a0=.030,a1=a2=0.0
compute cusp=2.0
frml BondPrice value = BondPV(t)
*
nlls(robusterrors,frml=BondPrice) value
```

4.13 Using FIND

FIND is the “catch-all” optimization instruction. If you can’t do it any other way, you can probably do it with **FIND**. Its great advantage is that it can optimize an expression that takes several RATS instructions to calculate and doesn’t fall into one of the forms treated by the existing optimization instructions. For instance, we have used it to estimate “hyperparameters” in Kalman filter problems.

FIND works quite differently from the other estimation instructions. The **FIND** instruction itself controls a set of other instructions, which are the ones which actually compute the function value. The syntax is

```
find(options)  maximum/minimum/root  expression
statement block: instruction or block of instructions used in computing expression
end find
```

Like the other non-linear estimation instructions, the options include **METHOD**, **ITERS** and **CVCRT**. The default method, however, is **SIMPLEX** (Section 4.3).

Just as with the other estimation instructions, you need to put your free parameters into a **PARMSET** using **NONLIN** and give them initial guess values. The main difference, though, is that the function being optimized is supplied by you, through the *expression* on **FIND** and an instruction or set of instructions which are used to compute it. Typically, *expression* will just be a real variable which will be set by those instructions. If so, you also need to **DECLARE** that variable, since RATS won’t see the instruction that sets it until after the **FIND**.

The program **NONLINEAR.RPF** shows three ways to estimate the parameters in the production function:

$$(60) \log y_i + \theta y_i = \beta_1 + \beta_2 \log K_i + \beta_3 \log L_i + \varepsilon_i, \varepsilon_i \sim N(0, \sigma^2) i.i.d.$$

Because of the transformation of the dependent variable, there’s a Jacobian term that must be included in the likelihood. The model can be estimated by **MAXIMIZE**, or by **NLLS** with the **JACOBIAN** option. An alternative is to observe that, given θ , all the other coefficients can be estimated by a least squares regression. The following uses **FIND** to search for the maximum likelihood θ . The regression part of the likelihood is concentrated into the single value **%LOGL** which is then adjusted by the sum of the log Jacobians. **FIND** controls a loop much as **DO** does. The instructions between **FIND** and **END** are executed each time a new function evaluation is needed.

```
nonlin theta
find maximum %logl
    set yadjust = log(y)+theta*y
    linreg(noprint) yadjust
    # constant logk logl
    sstats / log(1+theta*y)>>yterms
    compute %logl=yterms+%logl
end find
```


4.14 EM Algorithm

The EM algorithm (Dempster, Laird and Rubin, 1977) is a general method for handling situations where you have a combination of observed data and unobserved or latent data, and where the likelihood would have a convenient form if the unobserved data were known. It's an iterative algorithm, which repeats the pair of an E (or Expectations) step and an M (or Maximization) step.

Let y represent the full record of the observed data and x the full record of the unobserved data. Let Θ represent the model parameters being estimated. We're assuming that the log likelihood as a function of both x and y is $\log f(y, x | \Theta)$ which is itself fairly easy to handle. Unfortunately, since we only see y , maximum likelihood needs to maximize over Θ the log marginal density:

$$(61) \quad \log f(y | \Theta) = \log \int f(y, x | \Theta) dx$$

which can be quite difficult in the (non-trivial) case where x and y aren't independent. The result in the DLR paper is that repeating the following process will (under regularity conditions) eventually produce the maximizer of (61):

E Step: Let Θ_0 be the parameter values at the end of the previous iteration. Determine the functional form for

$$(62) \quad Q(\Theta | \Theta_0) = E_{x|y, \Theta_0} \log f(y, x | \Theta)$$

M Step: Maximize $Q(\Theta | \Theta_0)$ with respect to Θ .

The E step is, in many cases, simpler than the calculation in (61) because it integrates the log density, which often is a much simpler functional form than the density itself. Where the augmented data set (y, x) is jointly Normal, a small simplifying assumption in the E step allows you to use the even simpler

$$\tilde{Q}(\Theta | \Theta_0) = \log f(y, E(x | y, \Theta_0) | \Theta)$$

that is, you just do a standard log likelihood maximization treating x as observed at its expected value given the observable y and the previous parameter values Θ_0 .

While EM can be very useful, it does have some drawbacks. The most important is that it tends to be very slow reaching final convergence. While (62) can often be maximized with a single matrix calculation (if, for instance, it simplifies to least squares on a particular set of data), that is only one iteration in the process. The next time through, the function changes since Θ_0 has changed. It also (for much the same reason) gives you point estimates but not standard errors.

The best situation for EM is where the model is basically a large, linear model controlled (in a non-linear way) by the unobserved x . Even if (61) is tractable, estimating it by variational methods like BFGS can be very slow. Because EM takes care of the bulk of the parameters by some type of least squares calculation, the iterations are much faster, so taking more iterations isn't as costly. And in such situations,

Chapter 4: Non-Linear Estimation

EM tends to make much quicker progress towards the optimum, again, because it handles most of the parameters constructively. Where possible, the ideal is to use EM to get close to the optimum, then switch to full maximum likelihood, which has better convergence properties, and gives estimates for the covariance matrix.

However, don't overuse it. Some papers recommend using EM because the author didn't have access to software that could directly maximize (61). If the original problem can be handled by maximum likelihood, try that first. Only try EM if that proves to be too slow.

Because of the nature of algorithm, which has the maximization *inside* of an iterative process, EM usually involves a **DO** loop over the calculations, although you can sometimes write (62) in a form that you can input into **MAXIMIZE**. However, you need to insert the E step calculation between each iteration. You can't use the **START** option (Section 4.7), as it is associated with each function evaluation, not each iteration.

Instead, there's a separate (similar) option called **ESTEP**. This is available on **DLM**, **MAXIMIZE**, **NLLS** and **NLSYSTEM**, though it is most often used with **MAXIMIZE**. **MAXIMIZE** with **ESTEP** is often simpler to set up than the "classical" use of EM with specialized calculations for optimizing (62). It will always be slower to execute than if you work out the simple maximizers in the M step (it's basically doing the same optimization without knowing the structure of the problem), but you should only worry about execution time if execution is too slow. If you use **MAXIMIZE** with **ESTEP**, use **METHOD=BHHH**, not (the default) **METHOD=BFGS**. Because the function being optimized changes from iteration to iteration, the BFGS update steps aren't valid.

The example file `EMEXAMPLE.RPF` estimates a "mixture model" (Section 11.7.1) using maximum likelihood (which is in this case fairly simple), the classical optimized EM and **MAXIMIZE** with **ESTEP**. This has a mixture of two Normals with different variances. Which branch applies to each observation is unknown—the x are the unobserved branches. The parameters to be estimated are the two variances and the unconditional probability p of being on the first branch. This is the classical EM calculation, which here requires a probability-weighted estimate for the variances, and an average probability across observations to estimate p . One drawback is that you have to do your own convergence checks, done here using the `%TESTDIFF` function.

```
compute oldparms=||p,sigsq1,sigsq2||
do iters=1,500
  set pstar = f1=exp(%logdensity(sigsq1,xjpn)), $
              f2=exp(%logdensity(sigsq2,xjpn)), $
              p*f1/(p*f1+(1-p)*f2)
  sstats(mean) rstart rend pstar>>p $
              pstar*xjpn^2>>sumsq1 (1-pstar)*xjpn^2>>sumsq2
  compute sigsq1=sumsq1/p,sigsq2=sumsq2/(1-p)
  compute newparms=||p,sigsq1,sigsq2||
  if %testdiff(newparms,oldparms)<1.e-6
    break
  compute oldparms=newparms
end
```

4.15 Troubleshooting

If all the functions you were trying to maximize were globally concave and computable over the entire parameter space, there would be little need for this section. Unfortunately, that's not the real world. If you do any significant amount of work with non-linear optimization, you will run into occasional problems. Make sure that you understand the effects of numerical precision and initial guess values described in Section 4.1. Make sure that you check for warnings that the estimation hasn't converged. **THESE ARE IN CAPITAL LETTERS FOR A REASON.** If your parameters don't make sense, often it's because the model hasn't properly estimated. Until you have a converged model, it really doesn't matter whether the results are sensible.

Of course, almost all problems are due to “bad” initial guess values, in the sense that if you could start out at the optimum, everything would be simple. And many problems can be solved by coming up with better settings for these. However, note the caution on using the BFGS standard errors (page UG–116) if the guess values are “too good”, so that BFGS doesn't have a chance to determine the curvature.

If you haven't tried using preliminary simplex iterations (Section 4.3) before using BFGS or BHHH, we would suggest that you start with that. If you have, and you still are having problems, read further.

Tracing Progress

Most non-linear estimation instructions include an option `TRACE`, which produces intermediate output showing the progress of the iterations. The form of this varies depending upon the estimation technique being used, but this is fairly common:

```
Non-Linear Optimization, Iteration 6. Function Calls 39.
Cosine of Angle between Direction and Gradient  0.0116839. Alpha used was 0.00
Adjusted squared norm of gradient 0.534343
Diagnostic measure (0=perfect) 1.4860
Subiterations 1. Distance scale 1.000000000
Old Function = -250.152278      New Function = -250.105608
New Coefficients:
      1.876483      14.467947      -0.175269      0.604792
```

`TRACE` can be helpful in seeing what is going on with your estimation. If the function value seems to stabilize, but the squared norm of the gradient doesn't go to zero, and the diagnostic measure stays high (above 2), the model may be only barely identified, with a “ridge” rather than a peak in the surface. Very slow progress at the start is often a sign of poor initial guess values, possibly very near a zone where the function is either explosive or simply uncomputable (due to things like logs of negative numbers). This is a good sign that you need to try several different sets of guess values, as there wasn't an obvious “best” direction from your original ones.

“Missing Values And/Or SMPL Options Leave No Usable Data Points”

RATS will produce this message if it can't find any data points at which it can compute an objective function. In some cases, there is a simple fix. For instance, if you are maximizing a function which includes a term of `LOG (SIGSQ)` and you haven't given `SIGSQ` an initial guess value, the standard initialization value of zero will leave the function uncomputable. If you did this, you should see a warning like

```
## NL6. NONLIN Parameter SIGSQ Has Not Been Initialized. Trying 0
```

These warnings usually are of little consequence, since zero is, for many parameters, a quite reasonable initial guess value. If, however, you're having missing value problems, be careful about this.

The other common source for this problem is a failure to set up a recursively defined function properly. If you have an ARCH/GARCH model or one with moving average errors, you need to give a hard value for the *start* parameter since the recursion has to have a known initial point. See the “Recursive FRML's” part of Section 4.6 if you need more help.

Zero Standard Errors

If the model seems to converge reasonably, but your output shows that some coefficients have (true) zero standard errors, this will almost always be because that coefficient doesn't actually affect the function value. Most commonly, this is simply because the parameter in question doesn't appear in the formula(s) being estimated—either you forgot to include it in the formula, or inadvertently specified a parameter you didn't need on the **NONLIN** list. More subtly, it can be a parameter which multiplies a series or subcalculation which takes a zero value.

“Subiterations Limit Exceeded”

This message is issued when the subiteration process described in Section 4.2 fails to produce a step size meeting the criteria described there. This is a very serious problem—the predicted rate of change of the function in the chosen direction is quite wrong. If this occurs on a very early iteration, it's *possible* that increasing the subiterations limit from its default value of 30 will fix the problem. However, you're more likely to have success by doing some preliminary simplex iterations (`PMETHOD=SIMPLEX`, `PITERS=5` for instance) to improve the initial guess values.

The most common cause for the estimation stalling out like this is a poorly scaled parameter (page UG–110). If you have a parameter showing with `.0000xxx` or smaller, it's possible that its scale is making calculations difficult. A common example of this is a parameter estimating a variance, since it goes with the square of the scale of the data. As mentioned on page UG–110, sometimes simply multiplying the dependent variable by (say) 100 will fix this, since it multiplies variances by 10000.

If that doesn't apply, there are two other potential sources:

1. Estimates at or near the boundary of the computable zone
2. Collapse to near-singularity of the Hessian

If you have a function value which can't be computed over some region (a variance goes negative, a covariance matrix becomes singular, a probability exceeds 1) and the estimates have moved close to that region, it may be hard for general hill-climbing methods to work because the function value may be going from computable to uncomputable over a short distance. If this is what has happened (check the coefficient values on the TRACE output), you might be able to push the estimates away by restarting the estimation (by executing another estimation instruction) with simplex iterations. If it *still* doesn't move, you've probably found at least a local maximum at the boundary—you'll need to try the entire process from a different set of initial guess values to see if there is a higher global optimum away from the boundary.

The collapse to near-singularity of the Hessian is the more likely culprit if you're using METHOD=BFGS. The instruction

```
display %cvtocorr(%xx)
```

which displays the last inverse Hessian as a correlation matrix will likely show you some off-diagonal values very near 1.0 (such as .9999). In some cases, you may be able to fix this by just re-executing the instruction, which will reinitialize the BFGS process. If, however, you have some very highly correlated parameters, this will probably quite quickly reproduce the problem. If that's the case, you may need to find a different (equivalent) parameterization. Concentrating out a parameter or parameters, can, where possible, help out quite a bit. And substitutions like $bx^p \Rightarrow b^*(x/x_0)^p$ and $\alpha + \varphi x_{t-1} \Rightarrow \alpha^*(1 - \varphi) + \varphi x_{t-1}$, generally produce better behavior in the parameter pairs involved. The first of these is designed to control the scale of b , while the latter is to fix the sensitivity of α to φ as φ gets near one.

Occasionally, a problem like this may be fixed by tweaking the algorithm a bit by using **NLPAR (EXACTLINE)** (page UG-114) to force exact line search or by using **NLPAR (DERIVES=SECOND)** or **NLPAR (DERIVES=FOURTH)** to compute more accurate (but slower) numerical derivatives. These might work, but there's no guarantee — EXACTLINE sometimes makes things worse.

Slow Convergence (Many Iterations)

The more parameters that you try to estimate, the more iterations you're likely to need. If your function seems to be making very slow progress, check the TRACE output, paying particular attention to the distance scale on the subiterations:

```
Subiterations 2. Distance scale 0.500000000
```

If you're consistently seeing distance scales of 1.0 or 0.5, you may just need to be patient. Numbers like those mean that the predictions for the behavior of the function value seem to be accurate. If, on the other hand, you're seeing short steps like

```
Subiterations 5. Distance scale 0.062500000
```

you may need to rethink the parameterization of the model as described above.

Chapter 4: Non-Linear Estimation

You can sometimes improve the behavior of the estimates by doing small sets of iterations on subsets of the parameters. Something like

```
maximize (iters=20,parmset=pset1,noprint) maxfrml
maximize (iters=20,parmset=pset2,noprint) maxfrml
maximize (parmset=pset1+pset2) maxfrml
```

The first **MAXIMIZE** will do 20 iterations of the parameters in PSET1 given the initial guess values in PSET2. The second will do 20 iterations on PSET2 given the refined values of PSET1. Then the final **MAXIMIZE** will estimate the full set. Because the subiterations apply the same step size multiple to the entire parameter set, the overall process may be slowed down by a few poorly estimated parameters.

Slow Convergence (Time)

If each iteration takes a long time, there are two possible reasons: the function evaluations themselves are slow (computationally intensive), or there are a large number of parameters. You could, of course, be dealing with both. If you need to speed up your function evaluations, look for redundant calculations:

- If any FRML or FUNCTION includes a calculation that doesn't depend upon the parameters, do it once (prior to the non-linear estimation) and save the results.
- Look carefully at your FRML(s) to see if there's anything in them that isn't time-varying. Something that depends upon the parameters, but not on time, can usually be calculated once using the **START** option and stored into an array, which is then used in the other calculations. See Section 4.7.
- Within a FRML, avoid making multiple references to the same sub-FRML. For instance, if **RESIDA** is a FRML, the following are equivalent, but the second computes **RESIDA** just once rather than twice:

```
frml frontier = log(theta) - .5*resida^2 + $
               log(%cdf(-lambda*resida))

frml frontier = alpha=resida, $
               log(theta) - .5*alpha^2 + log(%cdf(-lambda*alpha))
```

If the size of the parameter set is what is slowing things down, you might want to think about the suggestion above for using subsets of the parameters. This will give you the biggest improvement in speed if you can optimize over a less complicated formula for different subsets.

5. Introduction to Forecasting

This chapter provides an introduction to forecasting with RATS, including the various types of forecasting models and the instructions used to generate forecasts. It also discusses ways to store and graph forecasted values, produce forecasts using rolling regressions, compute forecast errors, generate Theil U statistics, and troubleshoot forecasting problems.

Chapter 6 provides additional information on univariate forecasting methods, including Box–Jenkins ARIMA models and exponential smoothing techniques. See Chapter Chapter 7 for more information on forecasting VAR models. Chapter 8 discusses issues related to simultaneous equations and model solving and Chapter 10 includes forecasting with state-space models.

Forecasting Concepts

Dynamic vs. Static Forecasting

PRJ, UFORECAST, and FORECAST Instructions

Saving and Graphing Forecasts

Forecast Errors

Computing Theil U Statistics

Rolling Regressions and Forecasts

Comparing Forecasts

Troubleshooting

5.1 Introduction

Forecasting With RATS

This chapter provides an overview of the techniques and models available in RATS, introduces the instructions you can use to produce forecasts, and discusses some important issues related to producing good forecasts. Most of the information in this chapter applies to all the methods available in RATS, and is recommended reading for anyone doing forecasts with any type of model. See Chapter 6 for additional details on ARIMA, exponential smoothing, and spectral techniques, Chapter 7 for Vector Autoregressions (VAR's), Chapter 8 for Simultaneous Equations, and Chapter 10 for state-space models.

The main forecasting instructions are **FORECAST** and **UFORECAST**. **UFORECAST** (Univariate FORECAST) is the simplest of these—as the name implies, it is used only for forecasting (linear) univariate models. **FORECAST** can handle models with many equations, and with nonlinearities. You can also use **PRJ** for forecasting certain simple types of models, though its main use is analyzing the in-sample properties of a fitted model.

Note that this chapter deals only with forecasts of the mean of a process, and only for models that are linear in the endogenous variables. Linearity permits simple calculations of the predicted mean of the process. Forecasts of the variance are covered in Chapter 9 on ARCH and GARCH models. Forecasts of models which are non-linear in the endogenous variables (such as threshold models) require simulation methods (Chapter 16).

Regression Models

Consider the basic regression model:

$$(1) \quad y_t = \mathbf{X}_t\beta + u_t$$

Such a model is generally not a self-contained forecasting model unless it is a simple regression on deterministic trend variables and/or lags of the dependent variable. Otherwise, to forecast the future values of y , we either need to know the future values of $\mathbf{X}_t$, or the forecasts must be made conditional upon some assumed values for them. It's also possible to “close” the model by constructing additional equations which will forecast the future $\mathbf{X}$'s.

As a simple example, suppose that we are going to forecast a series of orders using a regression on a linear trend. We have data through 2018:6, but want forecasts out to 2019:12. In order to do this, we need to define the trend series out to 2019:12.

```
data(format=xls,org=columns) 1986:1 2018:6 orders
set trend 1986:1 2019:12 = t
linreg orders
# constant trend
prj forecast 2018:7 2019:12
```


Models With Lagged Dependent Variables

If the right hand side of your model includes lags of the dependent variable (that is, if $\mathbf{X}$ in equation (1) contains lagged values of y), the model is referred to as a *dynamic* model. One example would be the simple autoregressive model:

$$(2) \quad y_t = \alpha + \beta y_{t-1} + u_t$$

For such models, you have the choice between computing *static* forecasts or *dynamic* forecasts.

Static forecasts are computed as a series of one-step-ahead forecasts, using only actual values for lagged dependent variable terms. You can only compute static forecasts in-sample (or up to one period beyond the end of the sample), because you must have actual data available for the lagged dependent variable terms. Static forecasting will always produce the same forecasted values for a given time period T , regardless of the point in time at which you start computing forecasts.

Dynamic forecasts are multi-step forecasts, where forecasts computed at earlier horizons are used for the lagged dependent variable terms at later horizons. For example, the forecasted value computed for time T will be used as the first-period lag value for computing the forecast at time $T+1$, and so on.

For dynamic forecasts, use **UFORECAST** or **FORECAST**, which do dynamic forecasting by default. If you want static forecasts instead, you can use the **STATIC** option on **UFORECAST** or **FORECAST** or, for simple linear models, you can use the **PRJ** instruction.

Note that if you are only forecasting one step ahead, the static and dynamic forecasts will produce the same result. And if your model is itself static (has no lagged dependent variables), you will get identical results with any of the methods just described.

Time Series Models

In “pure” time series models, future values of a series (or group of series) depend *only* on its (their) past values. This category includes Box–Jenkins (ARIMA), exponential smoothing, and spectral models, as well as VAR systems that do not include other exogenous variables. For all except exponential smoothing and spectral methods, you can use **FORECAST** to produce dynamic forecasts or, with the **STATIC** option, one-step-ahead static forecasts (in-sample only). (**UFORECAST** can be used for single equations).

An ARIMA model with moving average terms also requires lags of the residuals for forecasting. For instance, if

$$(3) \quad y_t = \alpha + \varphi y_{t-1} + u_t + \theta u_{t-1}$$

in order to forecast period $T+1$, a value is needed for the u_T (u_{T+1} is a post-sample shock and is set to zero). When you use **BOXJENK** to estimate an ARIMA model, it will automatically save these.

5.2 Producing Forecasts

Static Forecasting with PRJ

The instruction **PRJ** can be used to compute fitted values or forecasts for a linear model after you estimate the model with **LINREG**, **STWISE**, **BOXJENK** or **AR1**. They are calculated using the most recently estimated regression. **PRJ** does static forecasts only, and so can only be used out-of-sample (for more than one period) if your model itself is static. An example of such a use of **PRJ** is provided on page UG–160.

Using UFORECAST and FORECAST

For more general forecasting of linear and non-linear models, use **UFORECAST** or, for multi-equation models or additional flexibility, **FORECAST**. As noted earlier, these produce dynamic forecasts by default—use the **STATIC** option to get static forecasts. Remember, for models that do *not* have a dynamic structure (that is, they have no lagged dependent variables), either method will produce the same results. The next paragraphs discuss the use of the instructions in various situations.

Single-Equation Linear Models

If you are estimating your model with **LINREG**, use the **DEFINE** option to save the estimated equation, which can then be forecast. This, and many of the other samples are from **BASICFORECAST.RPF**, which uses a data set with monthly data through 1994:12. The final year of this is “held back” for evaluation, so the estimates run only through 1993:12. For actual forecasting, you would use the full data set in the estimates.

```
cal(m) 1954:1
open data rsales.dat
data(format=prn,org=columns) 1954:1 1994:12
*
set logrsales = log(rsales)
set time = t
*
linreg(define=saleseq) logrsales * 1993:12
# constant time
```

You can use the *Time Series—Single-Equation Forecasts* wizard to generate the forecasting instruction, or you can type the command directly. For example:

```
uforecast(equation=saleseq) logfore 1994:1 1994:12
or
forecast(steps=12) 1
# saleseq logfore
```

The parameters for the **UFORECAST** are the series for the forecasts (**LOGFORE**), and the range. The option **EQUATION=SALESEQ** gives the equation to use.

The basic syntax of **FORECAST** is shared by several other instructions (**IMPULSE**, **ERRORS**, **SIMULATE** and **HISTORY**). The **STEPS** option indicates that we want to forecast for 18 steps. We could also use the options **FROM=1994:1**, **TO=1994:12** to set the forecast range. The supplementary card supplies the equation to forecast (**SALESEQ**), and the series into which the forecasts are saved (**LOGFORE**).

You can also use the **SMPL** instruction to specify the forecast range. Note that this changes the default range for any subsequent instructions, so be careful using it.

```
smp1 1994:1 1994:12
forecast 1
# ordereq logfore
```

AR1 Models

The **DEFINE** option on the **AR1** instruction produces a linearized equation from the original non-linear model:

```
open data housing.dat
calendar(m) 1983:1
data(format=free,org=col) 1983:1 1989:10 housing construct rates
*
ar1(define=firsteq) housing * 1988:10
# constant construct{0 to 6} rates{0 to 9}
*
uforecast(equation=firsteq) firstfore 1988:11 1989:10
```

This reads in data for **HOUSING**, **CONSTRUCT** and **RATES**, fits an AR1 model, and forecasts the model for 12 steps starting in 1988:11.

Multiple Equation Systems

FORECAST can also be used to produce forecasts for systems of equations. The equations are usually defined using either **EQUATION** instructions or the **SYSTEM** instruction and then estimated using **ESTIMATE** or **SUR**. However, your system might consist of equations estimated using several **LINREG** instructions, or it might include identity equations (see the next page for details) or other equations whose coefficients you have set yourself using **EQUATION** or **ASSOCIATE**.

The forecasting instructions don't "care" how the equations in the model were defined or estimated. As long as the necessary data are available and the equation coefficients have been estimated or set, you can produce forecasts.

To forecast a system of equations, you can use the *Time Series—VAR (Forecast/Analyze)* wizard to generate the appropriate instruction, or type in the **FORECAST** instruction directly. If using **FORECAST** directly, you can either:

- group the equations into a **MODEL** and use the **MODEL=name** option to forecast the model, or

Chapter 5: Forecasting

- list each of the equations in the system on a separate supplementary card

If you create the equations using a **SYSTEM** instruction, you can use the **MODEL** option on **SYSTEM** to automatically group those equations into a **MODEL**. Otherwise, use the **GROUP** instruction to construct the **MODEL** from a list of existing equations

For example:

```
group ourmodel houseeq>>fhousing conseq>>fconstruct rateeq>>frates
forecast(model=ourmodel,from=1989:11,to=1990:12)
or
```

```
forecast(from=1989:11,to=1990:12) 3
# houseeq  fhousing
# conseq   fconstruct
# rateeq   frates
```

Note that a **MODEL** can include non-linear formulas as well as equations.

Non-Linear Models

Non-linear models are constructed using formulas (**FRMLs**). These are typically defined using the instruction **FRML**, although they can also be created using **LINREG** or **EQUATION**. You can estimate them using **NLLS**, **NLSYSTEM**, or **MAXIMIZE**.

To generate forecasts for a non-linear model, you *must* group them into a model and use the **MODEL** option on **FORECAST**—you cannot list **FRMLs** on supplementary cards. This is true even when forecasting a single formula.

Using Identities

In some cases, you may need or want to include identities in your forecasting model. You might have accounting identities in a simultaneous equations model (see also Chapter 8), or, if you are forecasting a model estimated in logs, you might want to include an identity **FRML** to generate levels from the logs.

Identity equations can be created using the **EQUATION** instructions, while identity formulas are created using **FRML**. Identities should be listed after all estimated equations on the supplementary cards, or on the list in a **GROUP** instruction.

These two instructions define the same identity, one (**RSUMID**) as a **FRML** and one (**RSUMEQ**) as an **EQUATION**.

```
frml(identity) rsumid    rsum  = rate+rate{1}

equation(identity,coeffs=||1.0,1.0||) rsumeq rsum
# rate{0 1}
```

5.3 Generating and Using Results

Saving Output from UFORECAST and FORECAST

UFORECAST and **FORECAST** provide tremendous flexibility for handling output. You can:

- save your forecasts directly into series, which you can then print, graph, write to a data file, or use for further analysis.
- use the **PRINT** option to display the forecasts in the output window.
- use the **WINDOW** option to display the forecasts in a spreadsheet-style window. This provides convenient viewing of the forecasts, and also allows you to *Copy and Paste* the forecasts into spreadsheet programs or other applications, or export the data directly to a file with the *File—Export* operation.

Saving Forecasts in Series

This is straightforward for **UFORECAST**. For **FORECAST**, you can save your forecasts into series in three ways:

- If you are using supplementary cards to list the equations (rather than using the **MODEL** option), you can simply provide series names for the forecasts as the second parameter on each supplementary card. This example forecasts the equation **SALESEQ** and stores the forecasts in a series called **LOGFORE**:

```
forecast (from=1994:1, to=1994:12) 1
# saleseq logfore
```

If there are several equations, you don't have to save forecasts for each of the equations. Just leave off the *forecasts* parameter on a supplementary card if you don't need that equation's forecasts.

- When using the **GROUP** instruction to create a model, use the ">>series" notation on **GROUP** to provide the names for the forecast series. When you forecast the model (using the **MODEL** option), the forecasts will be stored in those series (see the example on page UG-163).
- You can use the **RESULTS** option on **FORECAST** to save the forecasts in a **VECTOR** of **SERIES**.

You can save your forecasts into existing series (including the actual dependent variables), or you can have **FORECAST** create new series (see "Saving Series in a Loop" for an exception). Note that saving your forecasts back into the dependent variable(s) has no effect on the actual forecasts produced, because the forecasts are computed and stored internally before being saved to the target series.

Chapter 5: Forecasting

This example computes forecasts for an interest rate series and stores them in a new series called OLSFORE. The forecasts intentionally overlap the estimation range.

```
linreg(define=irateeq) rate 1960:1 1995:8
# constant ip grm2 grppi{1}
uforecast(equation=irateeq) olsfore 1995:1 1996:2
```

And this example forecasts a MODEL, and uses the RESULTS option to save the forecasts in a VECTOR of SERIES called FORECASTS:

```
forecast(model=gdpmod,results=forecasts,steps=24)
```

Saving Series in a Loop

If you would like to use **FORECAST** inside a loop (or other compiled section, such as a procedure) and want to save the forecasts into *new* series, you must either use the RESULTS option to store the forecasts, or else create the new series prior to the beginning of the compiled section (using, for instance, **DECLARE**, **SET**, or **CLEAR**).

The WINDOW and PRINT Options

The WINDOW option allows you to display the forecasts in a report window. This provides a convenient way to view the results, and it also allows you to *Copy* and *Paste* the data directly into a spreadsheet program or other application, or export the data using the *File-Export* menu operation. The argument of the WINDOW option provides the title for the output window. For example, the following forecasts twenty-four steps of a six variable model, putting the forecasts into a window named “Forecasts.”

```
forecast(model=sixvar,window="Forecasts",steps=24)
```

The PRINT option on **FORECAST** causes the forecasted values to be displayed in the output window. PRINT also inserts the table into the *Report Windows* list on the *Windows* menu so you can reload it. Both PRINT and WINDOW can be used with any of the other methods of saving or displaying output.

```
forecast(model=gdpmod,results=forecsts,print,steps=24)
```

Graphing Forecasts

If you save your forecasts into series, you can graph them using the **GRAPH** or **SCATTER** instructions. This example forecasts log sales, saves the forecasts in a series called LOGFORE and then graphs both the forecasts and (part of) the original series (LOGRSALES).

```
uforecast(equation=saleseq) logfore 1994:1 1994:12
graph(key=below,header="Log Trend Model") 2
# logrsales 1993:1 1994:12
# logfore
```

Saving Forecasts to Files

To save your forecasts to a file, you can either:

- save the forecasts into a series and output the series using **COPY**, or
- use the **WINDOW** option to display your forecasts in a window. You can export the data by doing *File-Export*, or you can use *Edit-Copy* to copy the data to the clipboard for pasting into another application.

For example, this creates a file named `LOGMODELFORE.XLS` for the forecasts.

```
uforecast(equation=saleseq) logfore 1994:1 1994:12
open copy logmodelfore.xls
copy(format=xls,org=cols,dates) / logfore
close copy
```

Forecasting with Rolling Regressions

Computing forecasts based on a series of rolling regressions (that is, a series of regressions performed over changing sample periods) is a common practice, and an easy one to implement in RATS.

Here, we first run the regression through 1991:12, and compute a forecast for 1992:1. Then we estimate through 1992:1 and forecast 1992:2, and so on. Note that we have used the **NOPRINT** option on **LINREG**. We really aren't interested in seeing 36 regressions—we want the forecasts.

```
clear rhat
do regend=1991:12,1994:11
    linreg(noprint) logrsales * regend
    # constant time logrsales{1}
    uforecast rhat regend+1 regend+1
end do
```

At the end of this, the series **RHAT** will have a series of one-step out-of-sample forecasts over the period 1992:1 to 1994:12.

The last example used the same starting date for all regressions. Suppose, instead, you want to estimate over a “moving window” of, say, five years. There are various ways to handle the loop indexes to accomplish this—here's one of them (this program also saves the estimated coefficients from each regression):

```
clear rhat coef1 coef2 coef3
do regend=1991:12,1994:11
    linreg(noprint) logrsales regend-59 regend
    # constant time logrsales{1}
    compute coef1(regend) = %beta(1)
    compute coef2(regend) = %beta(2)
    compute coef3(regend) = %beta(3)
    uforecast rhat regend+1 regend+1
end do
```

Chapter 5: Forecasting

Both these examples compute only one-step-ahead forecasts, so saving the forecasts from each step in the rolling regression is easy. Suppose you instead want to compute twelve-step-ahead forecasts, and save *only* the twelfth forecast step from each rolling regression. If you simply did:

```
uforecast rhat regend+1 regend+12
```

then forecast step 12 from the first trip through the loop would be overwritten by forecast step 11 from the second trip through the loop, and so on. Thus an additional instruction is needed to copy the forecasts at later horizons to another series., as in

```
clear rhat rhat12  
do regend=1991:12,1993:12  
  linreg(noprint) logrsales regend-59 regend  
  # constant time logrsales{1}  
  uforecast rhat regend+1 regend+12  
  compute rhat12(regend+12)=rhat(regend+12)  
end do
```

Another way is to use **FORECAST** with the **SKIPSAVE** option, which is designed for just this sort of situation. For example:

```
clear rhat12  
do regend=1991:1,1993:12  
  linreg(define=foreeq,noprint) logrsales regend-59 regend  
  # constant time logrsales{1}  
  forecast(skip=11,steps=12)  
  # foreeq rhat12  
end do
```

See “Using the SKIPSAVE Option” on page RM–194 in the description of **FORECAST** for more information.

5.4 Forecast Performance

Standard Errors of Forecast

If you use **PRJ** to compute forecasts, you can use the **STDERRS** option to produce a series of the standard errors of projection. This estimates the model through 1993:12, does forecasts through 1994:12, and creates series **FORE**, **UPPER** and **LOWER** with forecasts and upper and lower bounds (using 1.96 standard errors) transformed back into levels from logs.

```
linreg(define=saleseq) logrsales * 1993:12
# constant time
prj(stderrs=stderrs) logfore 1994:1 1994:12
set upper 1994:1 1994:12 = exp(logfore+1.96*stderrs)
set lower 1994:1 1994:12 = exp(logfore-1.96*stderrs)
set fore 1994:1 1994:12 = exp(logfore)
```

The standard errors are given by the formula

$$s\sqrt{1 + x_t(\mathbf{X}'\mathbf{X})^{-1}x_t'}$$

where s is the standard error of the regression and $\mathbf{X}'\mathbf{X}$ is its cross-product matrix. This is a special formula which only works in static forecasting situations. The variance of projection that it computes consists of two parts: one due to the error in the regression equation itself (which would be present even if the coefficients were known with certainty) and one due to the sampling error of the coefficients. In a dynamic forecast, neither of these has such a simple form. Because forecasts are fed forward, the forecast errors from early steps accumulate. And the forecasts at higher steps depend upon the estimated coefficients in a highly non-linear way. How these are computed for a dynamic model is discussed more fully in the *Reference Manual* in the description of the instruction **ERRORS**.

UFORECAST and **FORECAST** also have **STDERRS** options, but these only include the part of the standard error which is produced by the regression equation, so they are computed as if the coefficients were known exactly.

Because **FORECAST** is designed to handle more than one equation, its **STDERRS** option returns a **VECTOR[SERIES]**. This pulls out the forecasts and standard errors for the second equation in the model:

```
forecast(model=tmodel, steps=5, results=forecasts, stderrs=stderrs)
set lower 37 41 = forecasts(2)+stderrs(2)*%invnormal(.025)
set upper 37 41 = forecasts(2)+stderrs(2)*%invnormal(.975)
```

One-Step Forecast Errors

By using the **STATIC** and **ERRORS** options on **FORECAST** or **UFORECAST**, you can easily compute and save a series of forecast errors (the differences between the forecasted value and the actual value) for each period in the forecast range. For a dynamic model, remember that these are one-step forecasts. As with **RESULTS** and **STDERRS**, the **ERRORS** option for **FORECAST** returns a **VECT[SERIES]**.

Chapter 5: Forecasting

@UFOREERRORS Procedure

This is a procedure for analyzing forecast errors for a single variable. You provide it with the series of actual values and forecasts and it computes a variety of statistics on the errors.

```
@UForeErrors ( options )   actual   forecast   start   end
```

Parameters

<i>actual</i>	series of actual values
<i>forecast</i>	series of forecasts
<i>start end</i>	range of forecasts to analyze (by default, the range of the series forecast)

Options:

```
[print]/noprint  
title="...descriptive title for forecasts..."  
window="...title of window..." (if you want a separate report window)
```

This computes and defines the following statistics: the mean forecast error (%%FERRMEAN), mean absolute error (%%FERRMAE), root mean square error (%%FERRMSE), mean percentage error (%%FERRMPE), mean absolute percentage error (%%FERRMAPE) and root mean square percentage error (%%FERRMSPE).

The last three are defined only if the actual values are positive throughout the test range. They are also all defined as decimals, not true percentages, that is, they'll report a value of 0.11 for an 11% error.

THEIL and the Theil U and other Performance Statistics

For larger models, or for multiple step forecasts, you can use the instruction **THEIL** to produce forecast performance statistics. You use it to compute a series of forecasts within the data range. The forecasts are compared with the actual values and a variety of statistics are compiled from the forecast errors: the mean error, the mean absolute error, root mean square error and a Theil's U statistic. The last is a ratio of the root mean square error for the model to the root mean square error for a "no change" forecast. This is a convenient measure because it is independent of the scale of the variable. The statistics are compiled separately for each forecast horizon, that is, if you do 12 forecast steps, you'll get separate information for each variable and for each forecast step from 1 to 12.

THEIL is usually applied to ARIMA models (Chapter 6) or VAR's (Chapter 7). Its use is a standard part of the methodology for forecasting with VAR's.

5.5 Comparing Forecasts

Forecasts can be compared informally by using the performance statistics described in the previous section. However, these are subject to sampling error, and so, while one forecast procedure may produce a better RMSE than another in a particular sample, it may not be clear whether that difference is really significant.

Among the procedures which have been proposed for an actual test is that of Diebold and Mariano (1995). This is based upon the use of a loss function, which could be the squared errors, but also could be a loss function tailored to the situation, like an error in the sign, if calling a direction right is more important than the actual forecast value.

The calculation is simple to perform in RATS once you've generated the sets of forecasts. Compute the series of the differences between the loss functions for the two forecasts. Regress this on a `CONSTANT`, requesting robust standard errors with the appropriate number of lags. (If you're comparing K step forecasts, this would generally be $K-1$). The null is that the intercept is zero. Now, typically the alternative is that one set of forecasts is better than the other (not just that they're different), so a one-tailed test is appropriate. Note that the series of forecasts should all be for the same forecasting horizon. Don't mix 1 to N step forecasts in a single test.

We have provided the procedure `@DMARIANO` which directly handles this test for the two most important loss functions: squared errors and absolute errors. Note that the default on the `LWINDOW` option for the procedure is `TRUNCATED`, which follows the recommendation of the authors. However this *could* produce a non-positive standard error; unlikely in practice, but possible.

For instance, consider the rolling regressions example on page UG-167. That generated one set of one-step forecasts (into the series `RHAT`). Suppose the alternative is the "naive" forecast of the previous period's return value, and that our loss function is the squared error. The Diebold-Mariano test can be implemented by

```
set naive 1992:1 1994:12 = logrsales{1}
@dmariano logrsales naive rhat
```

Limitations of the Diebold–Mariano Test

The Diebold-Mariano test should *not* be applied to situations where the competing models are nested. Examples of nested models are $AR(1)$ vs $AR(2)$, no change vs any $ARIMA(p,1,q)$. In the example above, the models (the no-change forecast and the regression model) would have been nested if the lagged dependent variable were included. When the models nest, under the null the population forecast errors are asymptotically perfectly correlated, and the Diebold-Mariano statistic fails to have an asymptotic Normal distribution. An alternative testing procedure for those situations is the Clark–McCracken test (Clark and McCracken, 2001), which is implemented in the `@ClarkForeTest` procedure.

5.6 Troubleshooting

Why Don't I Get Forecasted Values?

At some point, you may discover that you are getting NA's (missing values) for one or more of the periods for which you expected to get forecasts. Almost without exception, this will be because required right-hand-side values were not available, either due to missing values in the data set, a mistake in specifying the forecast range, or (for simultaneous equation systems) a mistake in the model specification itself.

Consider a simple forecast model where series y at time t depends only on series x at time t plus a constant term, a model you could estimate using:

```
linreg(define=yeq) y
# constant x
```

Remember, if you are forecasting this as a single-equation model (rather than as part of a system), you cannot possibly compute a forecasted value for $Y(t)$ unless the series X already contains a valid number at time t . Suppose these are quarterly data, and you have data for X through 2018:2. If you do:

```
uforecast(equation=yeq,print) yhat 2018:3 2018:4
```

the forecasts will be reported as "NA" because X (upon which Y depends) is itself undefined (NA) for these periods. To get out-of-sample forecasts for Y , you will need to supply values for X yourself over the forecast range (using **SET** or **COMPUTE** or by reading them in from a data file), or you will need to define and estimate a model for X and compute forecasts, either independently, or as part of a system which also includes the equation for Y .

Finding the Problem

First, turn on the **PRINT** option on your forecasting instruction so you can see the forecasts as RATS is calculating them.

If these look correct, but the series in which you save the forecasts have NA's, then you have a problem with the way you are saving or using the forecasts. Make sure that you're using variable names in the correct positions.

If the displayed forecasts coming out of the forecasting instruction show problems, there's a problem with either your data or the model (or both). Check the following:

1. If you are forecasting a multiple-equation system, make sure your equations have been defined and estimated properly, and that you've remembered to include all the required equations on supplementary cards or in the **MODEL**. In particular, make sure that you've included any required identity equations.
2. Check your data! Many users have wasted a great deal of time searching for programming mistakes when the real problem was simply missing data in one or more right-hand-side variables. If you are having trouble getting forecast values, do yourself a favor and check the right-hand-side data series carefully, by using

the **PRINT** instruction or double-clicking the series in the Series Window list to display the actual series used in the model. A common mistake is verifying that original series are OK, but forgetting that transformations (such as log of a negative number) can produce NA's. Check the data that are actually being used in the model!

3. Check the range for which you are trying to compute forecasts—make sure you aren't starting your forecasts too far beyond the end of your available data. Suppose you have a simple ARIMA model for a monthly series:

```
boxjenk(ar=1,define=areq) y
```

and you have data for y through 2017:12. You could produce forecasts out to any horizon if you start forecasting at entry 2018:1 (or earlier). However, if you tried to start at 2018:2 (or later), you would get NA's at every forecast horizon. That's because y at 2018:2 depends on y at 2018:1 and you don't have data for that observation.

6. Univariate Forecasting

This chapter focuses on forecasting a single variable using time series methods, such as ARIMA modeling, exponential smoothing, and spectral methods. For a general introduction to basic forecasting concepts and the instructions used to produce forecasts, please see Chapter 5. That chapter also provides information on regression-based forecasting models. For forecasting with more than one variable, see Chapter 7 for Vector Autoregressions and Chapter 8 for simultaneous equations. You can also use State Space Models (Chapter 10) for forecasting either one or several variables.

Transformations and Differencing

Exponential Smoothing

ARIMA Models

ARMAX Models

ARIMA Procedures (BJIDENT, BJEST, BJFORE and More)

Spectral Forecasting

Intervention Modeling

6.1 Time Series Methods

Background

Time series modeling techniques forecast the future by

1. modeling the correlation between a series (or group of series) and its (their) past,
2. assuming that the relationship between current and past will continue into the future, and
3. computing forecasts on the basis of those assumptions.

They are extrapolative techniques which incorporate no information other than that available in the past data.

Methods

RATS supports three methods for univariate time series forecasting:

Exponential smoothing (Section 6.3)

is a small collection of models which are often adequate for forecasting. Unlike the other methods, it does not really attempt to model the autocorrelation. You choose a model based upon the two most important aspects of time series behavior: trend and seasonality.

Box-Jenkins (Section 6.4)

offers a broad collection of models for fitting the serial correlation pattern. Because of the number of choices permitted, it can handle many types of time series. Choosing an appropriate model, however, is something of an art.

Spectral forecasting (Section 6.5)

uses frequency domain techniques to fit a “generic” Box-Jenkins model. It produces very acceptable forecasts relative to Box-Jenkins models, but is less robust to errors in preparing the data for forecasts. It also requires more data than Box-Jenkins.

You can also forecast using state-space models (done in RATS using the instruction **DLM**—see Chapter 10), and do multivariate time series forecasting with vector autoregressive models, described in Chapter 7. And you can produce forecasts using a variety of standard regression models. See Chapter 5 for details on that.

In the remainder of this chapter, we’ll examine some issues that are common to all three of these modeling techniques, and then discuss each of them in detail. We will also look at some related topics, including ways to evaluate forecast performance, intervention models, and combined ARIMA/regression models (ARMAX or RegARIMA).

6.2 Common Issues

There are several issues which all three of these time series methods must address, and which are examined in this section:

- Should you transform the data prior to the analysis? Usually this boils down to a question of logs vs levels.
- How should you represent the series trend? Should you difference the series?
- How should you treat seasonality?
- How many parameters can you safely estimate given the available data?

Preliminary Transformations

Preliminary transformations are simple transformations with two purposes: to straighten out trends; and to produce approximately uniform variability in the series over the sample range.

What we are trying to eliminate in the second point is a systematic tendency for the magnitude of the variance to depend upon the magnitude of the data. Suppose that the variance of a series is significantly higher when its values are large. If we estimate a model for this series by least squares, the least squares algorithm will concentrate upon fitting the high-variance part, largely ignoring the rest of the data.

Usually, you will either do nothing to the data, take square roots, or take logarithms. How do you decide? The following guidelines should help:

- Do you usually think of the series in terms of its growth rates (GNP, money, prices, etc.)? If so, take logs.
- Is the percent growth rate of the series an almost meaningless concept? Usually, do nothing. Possibly take the square root if the series clearly is more variable at higher values.

Typically, *if your decision is not obvious from looking at the data, your choice will make little difference to the forecasts.*

Differencing

We will look at this together with the decision of whether or not to include an intercept. This is a crucial decision for two reasons. First, some forecasting methods (especially spectral procedures) are computationally sensitive to mistakes here, particularly to *underdifferencing*. Second, this determines the trend behavior of the forecasts, so an incorrect choice will have major effects on longer-term forecasts. Modeling methodologies which do not include the correct trend model are likely to produce poor forecasts.

The table on the following page suggests some differencing and intercept combinations for various types of series behavior. Often, you can decide how to difference solely on the basis of your knowledge of the data.

Chapter 6: Univariate Forecasting

Diff	Intercept	Behavior of Series
0	Yes	Clusters around a mean level (unemployment rate?)
1	No	Doesn't trend, but doesn't seek a level (interest rate?)
1	Yes	Trends at a fairly constant rate (real GDP?)
2	No	Trends at a variable rate (price index?)

Seasonality

Seasonality poses several serious problems for forecasters:

- It can take many forms: additive or multiplicative, stationary or non-stationary, stochastic or deterministic.
- Seasonality usually accounts for *most* of the variance of the series, making it difficult to determine the non-seasonal behavior.
- Every method of dealing with seasonality induces degrees of freedom problems. This will either cause a loss of usable data points, or require estimation of seasonal parameters with very few data points.

If seasonality is essential to the task at hand, you have two basic courses of action:

- You can seasonally adjust the data, forecast the adjusted data and the seasonal part separately, and then combine them. The main problem is that the seasonal “model” is done independently and may be inappropriate for the data series.
- You can use a methodology which incorporates seasonality directly. All three of the forecasting methods permit this. However, you will only be able to handle adequately certain forms of seasonality.

Parsimony

While the “Principle of Parsimony” is usually associated with Box and Jenkins, the idea applies to other forecasting models as well. Keep in mind two general principles:

- Don't ask the data to do too much work. The less data you have, the fewer parameters you can estimate, and the more important your judgment becomes.
- Don't take your model too seriously. Time series models are designed to *fit* the serial correlation properties of the data, not *explain* them. The goal is to find a model which fits the data reasonably well with as few parameters as possible.

The most carefully thought-out model is worthless if you can't estimate it with the data available. You might think that estimating four parameters with thirty data points is not asking too much of the data. However, experience has shown that a three parameter model which fits almost as well will forecast better most of the time. *This is true even if the difference is statistically significant.*

6.3 Exponential Smoothing

Background

The exponential smoothing (ES) methodology chooses one from a small group of models which focus upon the trend and seasonal behavior of the data. Because those two aspects dominate the variance of the series, properly chosen ES models perform well relative to more complicated methods on a wide range of data series.

These techniques were designed for dealing with data where there is a smooth “signal” obscured by substantial measurement noise. While they can adequately capture many economic data series, they do it primarily by not smoothing at all. In fact, it isn’t uncommon for the α parameter, which governs the amount of smoothing, to have an optimal value bigger than 1 which means that past data are overweighted. This typically arises because the data have been time-averaged. If, for instance, a monthly series is constructed as an average of daily values, a sudden change late in the month won’t fully be reflected in the observed data until the next period.

Advantages

ES techniques are computationally simple, so you can quickly apply them to a large number of series. Plus, the small set of choices simplify the process of choosing the “best” ES model and make them ideal for very small data sets.

Disadvantages

The class of models is too narrow for some data series. For instance, a “no trend” model actually is a random walk model, so its forecasts remain at the final smoothed level. For series which return to a mean level, this behavior is very bad, especially at longer horizons. Also, when you have enough data to entertain a richer class of models, other methods are likely to do better.

Using the ESMOOTH Instruction

The **ESMOOTH** instruction implements exponential smoothing. You can type in the instruction directly, or you can use the *Time Series—Exponential Smoothing Wizard* to generate it.

You will need to choose both the trend model and the seasonal model, and decide between setting the smoothing parameters yourself or estimating them. You choose the models with the **TREND** and **SEASONAL** options on **ESMOOTH**. The choices available for these are:

```
trend=[none]/linear/exponential/select  
seasonal=[none]/additive/multiplicative/select
```

If you choose **SELECT**, RATS will test all three choices for that option and select the best fitting model.

You can set the smoothing parameters with the **ALPHA**, **GAMMA**, and **DELTA** options (all default to 0.3), or you can use the **ESTIMATE** option to have RATS estimate the parameters. With **ESTIMATE**, RATS chooses the parameters by minimizing the in-

Chapter 6: Univariate Forecasting

sample squared one-step forecast errors. If you have a small amount of data (twenty or fewer data points), you should *not* try to estimate. Setting the parameters to reasonable values will usually be superior to relying upon imprecise estimates.

Examples

The first example uses **ESMOOTH** to forecast a pair of U.S. interest rates. This example is supplied on the file `EXPSMOOTH1.RPF`:

```
open data haversample.rat
calendar(m) 1978
data(format=rats) 1978:1 2007:4 fcm30 ftbs3
```

FCM30 = Yield on 30 Year Treasury Bonds

FTBS3 = Yield on 3 Month Treasury Bills

Forecast twelve periods out-of-sample using simple smoothing model with fixed parameter of .8. Append the forecasts to the series for ease in graphing.

```
esmooth(alpha=.8,forecast=ftbs3,steps=12) ftbs3
esmooth(alpha=.8,forecast=fcm30,steps=12) fcm30
```

Graph with more informative key labels, showing the last six months of actual data, followed by the forecasts. Include a vertical line at 2007:4.

```
graph(grid=(t==2007:4),key=below,$
      klabels=||"30 Year Bonds","3 Month Bills"||) 2
# fcm30 2006:11 2008:4
# ftbs3 2006:11 2008:4
```

This next example “seasonally adjusts” Canadian retail sales, storing the adjusted data in `CANRETSAX`, and graphs a comparison of this series with the “official” seasonally adjusted version over a six year period in the middle of the sample. This is supplied on the file `EXPSMOOTH2.RPF`:

```
open data oecdsample.rat
calendar(m) 1960
data(format=rats) 1960:1 2007:3 canrett canretts
esmooth(trend=select,seasonal=select,smooth=canretsax,$
      initial=start) canrett
```

```
graph 3
# canrett 1994:1 1999:12
# canretsax 1994:1 1999:12
# canretts 1994:1 1999:12
```

6.4 Box-Jenkins Models

Background

The Box-Jenkins methodology provides a broad range of models for fitting the serial correlation patterns of both seasonal and non-seasonal data. The class of models is rich enough that there is often a choice which fits the data adequately with very few parameters.

The fitting process as described in textbooks (Box, Jenkins, Reinsel (2008), Brockwell and Davis (2002), DeLurgio (1998), Enders (2014), and Granger and Newbold (1986) are among the many worthy examples) is an interactive process between the forecaster and the data. We describe the process here briefly, but you will need to consult an appropriate text if you are new to ARIMA modeling.

Advantages

Box-Jenkins models cover a much larger range of series than exponential smoothing models. They require less data than spectral methods, which don't use the data in quite so "sharp" a fashion. There is a well-developed literature on the methodology, and they have proven to be quite successful in practice.

Disadvantages

The selection of a model depends strongly upon individual judgment. The range of possible models (especially with seasonal data) requires you to make quite a few choices. As a result, two people facing the same data may come up with very different models. Fortunately, if two models provide a nearly identical fit, they will usually produce nearly identical forecasts.

The selection procedure can be difficult to apply, especially with seasonal data. If you have a hard time coming up with an adequate model and you have 100 or more data points, you might try spectral methods instead. On the other end, Box-Jenkins requires more data than exponential smoothing: Granger and Newbold, for instance, recommend at least 40-50 observations.

The Model Selection Process

First, you need to make sure that you are working with a stationary series. An examination of the series can help determine if some preliminary transformation is needed to give a stationary variance. Then, the sample autocorrelations and partial autocorrelations will help you determine whether you need to apply differencing or other transformations (such as taking logs) to produce a stationary series. Only then can you begin fitting an ARMA model to your stationary series.

In order to develop ARIMA models, you need to be familiar with the behavior of the theoretical autocorrelation and partial autocorrelation functions (ACF and PACF) for various "pure" AR, MA, and ARMA processes. See Enders (2014) or similar texts for descriptions and examples of theoretical ACF and PACF behavior.

Chapter 6: Univariate Forecasting

Potential model candidates are determined by examining the sample autocorrelations and partial correlations of your (stationary) series, and looking for similarities between these functions and known theoretical ACF and PACF. For example, your sample correlations may behave like a pure MA(2) process, or perhaps an ARMA(1,1).

In most cases, you will have several possible models to try. Selecting a final model will usually require fitting different models and using information such as the summary statistics from the estimation (t -statistics, Durbin–Watson statistics, etc.), the correlation behavior of the residuals, and the predictive success of the model to determine the best choice (keeping in mind the principle of parsimony discussed earlier).

Some textbooks recommend an “automatic” model selection procedure which examines all the ARIMA models with a small number of AR and MA parameters (say, all combinations of between 0 and 4 with each), and chooses the model based upon one of the information criteria, like the Akaike or Schwarz criterion. (The procedure **@BJAUTOFIT**, described later in this section, can be used to do this). For many series, this works fine. However, it’s hard to entertain models with lag skips using this type of search. For instance, the search just described would require checking 25 models, but if we allow for *any* set of lags for each polynomial, such as lags 1 on AR and 1 and 4 on the MA, we’re up to 1024 possible candidates, 32 combinations for each polynomial. For a series with seasonality, it’s very likely that the “best” model will have this type of lag structure. So understanding how to choose and adjust a model is still likely to be important in practice.

Tools

The primary RATS instructions for selecting, fitting, analyzing, and forecasting Box–Jenkins models are **BOXJENK**, **CORRELATE**, **DIFFERENCE** and **UFORECAST** (or the more general **FORECAST**). You will probably find it helpful to know how to use each of these instructions. However, we have also provided several procedures that tie these instructions together to allow you to do standard tasks with one or two simple procedure calls. We’ll introduce the instructions by walking through an example first, and then discuss the procedures.

6.4.1 A Worked Example

To demonstrate the tools used to estimate and forecast ARIMA models, we will work with an example from Enders 3rd edition, section 2.10. (A similar example with updated data is in the same section of Enders(2014)). The series being modeled is a quarterly interest rate spread, computed as the difference between the interest rate on ten-year U.S. government bonds and the rate on three-month treasury bills, from 1960 to 2008 quarter 1. This program is available in the file `ARIMA.RPF`. The first step is to read in the data and take a look at a graph of the original series:

```
open data quarterly.xls
calendar(q) 1960:1
data(format=xls,org=columns) / tbill r10
graph(header="T-Bill and 10-year Bond Rates",key=upleft) 2
# tbill
# r10
```

Next, we need to compute the interest rate spread. We will also want to examine the first differences of the series, so we will go ahead and compute those as well, and then graph both series. We will use the **SPGRAPH** instruction to display two graphs on a single page:

```
* Compute spread and first difference of spread:
set spread = r10 - tbill
diff spread / dspread

spgraph(vfields=2,$
  footer="FIGURE 2.5 Time Path of Interest Rate Spread")
graph(header="Panel (a): The interest rate spread")
# spread
graph(header="Panel (b): First difference of the spread")
# dspread
spgraph(done)
```

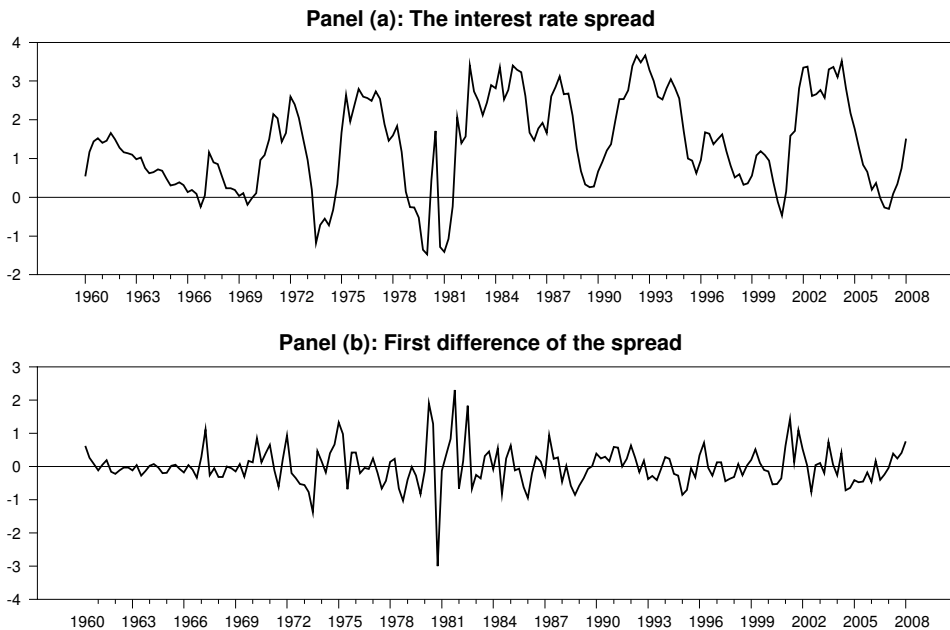


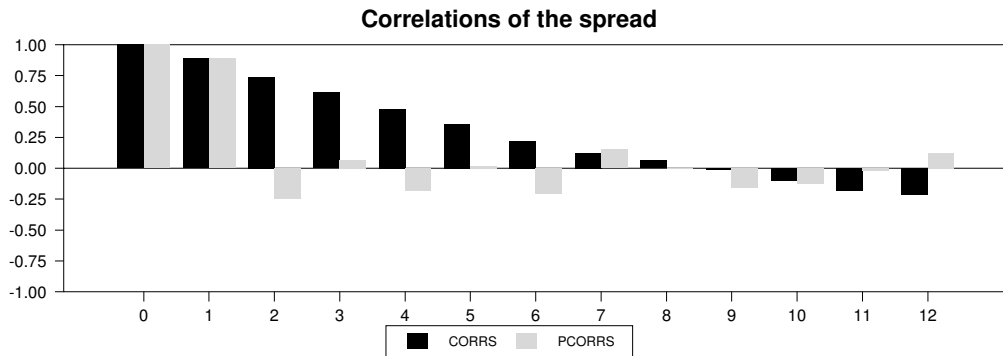
FIGURE 2.5 Time Path of Interest Rate Spread

Looking at the first panel, we don't see strong evidence of a trend or any structural breaks, and the overall behavior suggests that the series is covariance-stationary. The first differences on the other hand (in panel b), behave much more erratically. Based just on looking at the data, it appears that we will want to model the series in levels, rather than in differences.

Chapter 6: Univariate Forecasting

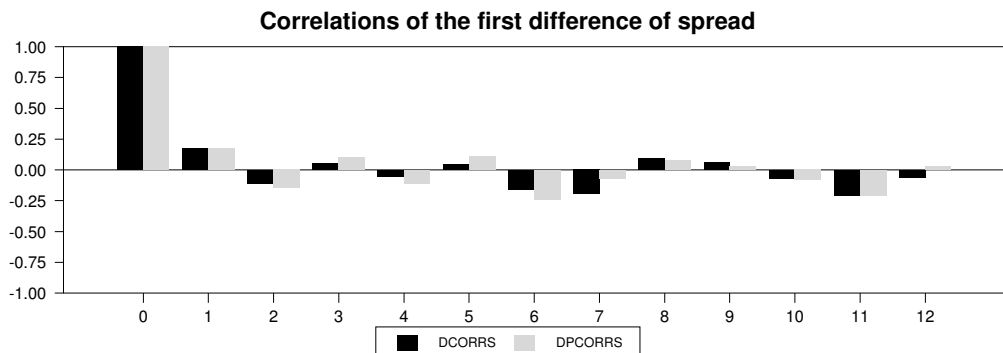
We can now examine the autocorrelation behavior of the series to identify some candidate models. The easiest way to do this is with the procedure **@BJIDENT**, described on page UG–193. First, we'll show here how to generate the required graphs directly. We use the instruction **CORRELATE** to compute the statistics (autocorrelations and partial autocorrelations). The example below shows the options that we like to use for graphing these. Note that this graph includes the 0 lag statistics, which are always 1.0. At this stage of the modeling process (called the *identification phase* by Box and Jenkins), we find it helpful to leave this in as a reference. We'll take it out in a later stage, when we're expecting the autocorrelations at positive lags to be near zero.

```
corr(results=corrs,partial=pcorrs,number=12) spread
graph(header="Correlations of the spread",key=below,$
      style=bar,nodates,min=-1.0,max=1.0,number=0) 2
# corrs
# pcorrs
```



As a further check on whether differencing the series is appropriate, we can examine the correlations of the first difference:

```
corr(results=dcorrs,partial=dpcorrs,number=12) dsread
graph(key=below,style=bar,nodates,min=-1.0,max=1.0,number=0, $
      header="Correlations of the first difference of spread",) 2
# dcorrs
# dpcorrs
```



This series could go either way—the first difference series shows only mild signs of overdifferencing (correlations switching signs). Since you should take the minimal level of differencing to produce stationarity, we'll stick with the undifferenced data.

A Note on Differencing

For models that *do* require differencing, you can either estimate the model using the differenced dependent variable, or you can use the original dependent variable and add a `DIFFS` option to include the differencing as part of the instruction. This has no effect on the estimates, and the only change in the output will be in the R^2 and related measures. There are two advantages to having **BOXJENK** do the differencing:

1. It allows comparable R^2 for models with different choices for the number of differences. Note, however, that the R^2 is *not* considered to be an important summary statistic in ARIMA modeling.
2. If you define a forecasting equation using **BOXJENK**, it will be for the variable of interest, and not for its difference.

Selecting and Estimating Models

Our next step is to examine the autocorrelations and partial autocorrelations of the levels to identify some possible models. We can see that the autocorrelation functions decay fairly slowly, rather than either cutting off immediately (as they would for a pure MA process) or decaying geometrically (as they would for an AR(1) process). This suggests that we will need a higher-order AR model, or a model including both AR and MA terms.

As Enders explains in his text, the fact that there are significant partial autocorrelation values as far out as seven lags suggests that we may need AR terms out to six or seven lags. Also, the pattern of alternating positive and negative partial autocorrelations suggest the presence of a (positive) MA term, as well as one or more AR terms.

Enders starts by estimating six possible models. We will include the code for three of them here—see the `ARIMA.RPF` for the others. First, though, some general points on estimating and comparing ARIMA models using the **BOXJENK** instruction.

RATS offers two criterion functions for fitting ARMA models: conditional least squares and maximum likelihood. The default is conditional least squares—use the `MAXL` option on **BOXJENK** to get maximum likelihood instead. Each method has its advantages. Conditional least squares has a better behaved objective function, and thus is less likely to require “fine tuning” of the initial guess values. However, there are several ways to handle the pre-sample values for any moving average terms, so different software packages will generally produce different results. Maximum likelihood is able to use the same sample range for all candidate models sharing the same set of differencings, and so makes it easier to compare those models. The RATS procedure **@BJAUTOFIT** uses maximum likelihood for precisely this reason. In addition, maximum likelihood estimates should give the same results across software packages.

Chapter 6: Univariate Forecasting

The most interesting summary statistics in the **BOXJENK** output are the standard error of estimate (for conditional least squares) or the likelihood function (for maximum likelihood) as your measure of fit, and also the Q statistic. Ideally, you would like the Q to be statistically insignificant. In addition, it's standard practice to examine the autocorrelations of the residuals, looking for large individual correlations. The Q can pick up general problems with remaining serial correlation, but isn't as powerful at detecting a few larger-than-expected lags. For residuals, it generally makes more sense to do the autocorrelations only (not partials).

Our model is going to include an intercept, as we expect a non-zero mean. There are three main ways that software packages deal with the intercept:

- Remove the mean from the differenced data, then add it back after estimating the model on the mean zero data.
- Include the intercept as a coefficient in a “reduced form” model.
- Separate the model into a “mean model” and a “noise model”.

RATS normally uses the third of these. (You can implement the first using the option **DEMEAN**). For an AR(1) model, the difference between (b) and (c) is that (b) estimates the model in the form $y_t = c + \phi y_{t-1}$, while (c) does $y_t - \alpha = \phi(y_{t-1} - \alpha)$. The two models should produce identical answers (with $c = \alpha(1 - \phi)$) given the same method of estimation. The advantages of the form chosen by RATS is that the estimated coefficient on the constant has a simple interpretation as the mean of the differenced process. It also generalizes better to intervention models (Section 6.4.3) and the like.

In this case, we estimate the candidate models by unconditional least squares. To ensure comparability, they are all run over the same sample range, from 1961:4 to the end of data. 1961:4 is the earliest period for which the model with the longest lag length—the AR(7) model—can be run using conditional least squares.

The estimates in the text use method (b) to handle the mean. As you can see, all other results are unaffected by this choice.

An AR(7) Model

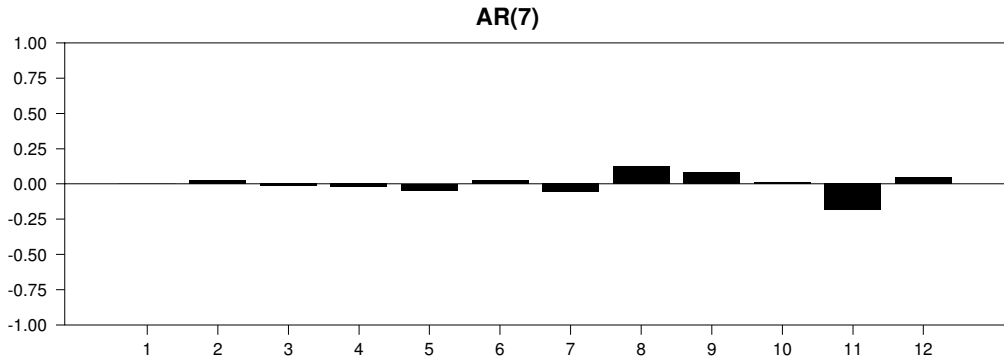
The instruction below estimates our first model, an AR(7) with a constant term:

```
boxjenk(constant,ar=7) spread 1961:4 *
```

This reports the Q for 36 lags as 28.71 with 29 degrees of freedom (the output shows 36–7: the number of autocorrelations minus the number of estimated ARMA parameters), which is not significant, showing no evidence of residual serial correlation.

Next we compute and graph the residual autocorrelations and some additional Q statistics. We use the **DFC** option on **CORR** correct for the degrees of freedom used in the estimation, which will be the number of AR and MA parameters. **BOXJENK** stores that value for us in the **%NARMA** variable. The **NUMBER** and **SPAN** options tell RATS to compute correlations out to 12 lags and to report Q statistics for every 4 lags. (We'll show you an easier way to do this in a moment).

```
correlate (results=rescorrs , number=12 , span=4 , qstats , $
          dfc=%narma) %resids
graph (header="AR (7) " , style=bar , nodates , min=-1.0 , max=1.0 , number=1)
# rescorrs 2 *
```



The autocorrelation plot offers further evidence that there is little remaining correlation in the residuals. These are the *Q* statistics for lags 4, 8, and 12 produced by **CORRELATE**:

Ljung-Box <i>Q</i> -Statistics			
Lags	Statistic	Signif	Lvl
4	0.230		
8	4.501	0.033868	
12	13.072	0.022714	

While these are apparently significant, that is mainly because we have very few degrees of freedom after correcting for the seven estimated ARMA parameters. You can see from the graph why you need to correct for the number of estimated parameters, since the AR(7) model almost completely eliminates autocorrelation for 1 to 7.

While the model seems adequate, we don't yet know if it does a good job of *forecasting* the data, or whether other models might do a better job. The rather large number of parameters suggests there might be a smaller model which will do better.

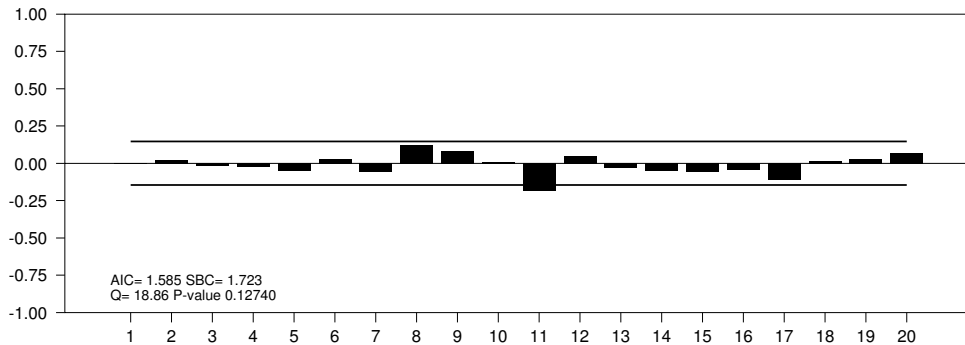
The @REGCORRS Procedure

Although it is useful to know how to compute and graph the correlations using the **CORR** and **GRAPH** instructions, in practice you will probably prefer to use the **@REGCORRS** procedure. **@REGCORRS** displays standard error bands on the correlation plots, Akaike (AIC) and Schwarz Bayesian (SBC) information criteria statistics, and (with the **REPORT** option) produces a table of *Q* statistics for the residuals from the most recent estimation (which are automatically stored in **%RESIDS**). We'll describe the procedure in more detail later, but for now, here's a simple example:

```
boxjenk (constant , ar=7) spread 1961:4 *
```

Chapter 6: Univariate Forecasting

```
@regcorrs (dfc=%narma,number=20,qstats,report,$
method=burg,title="AR(7) model diagnostics")
```



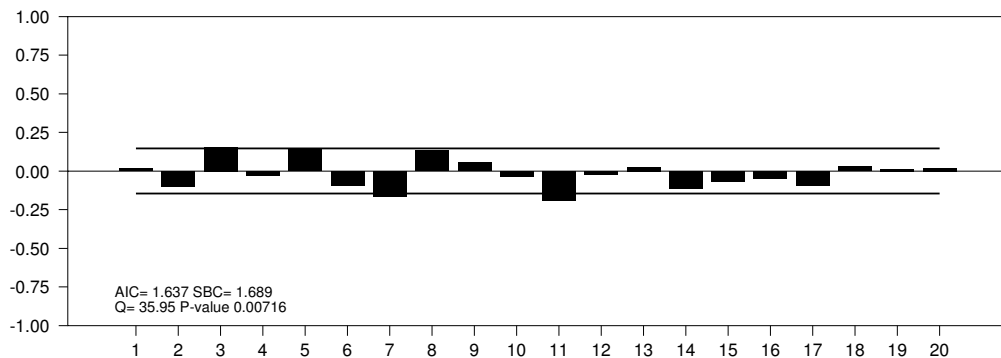
@RegCorrs graph of AR(7) model diagnostics

Other Candidate Models

Given that models with fewer parameters often forecast as well as, or better than, larger models, we will look at two of the smaller models Enders considers—an AR(2) model and an ARMA(1,1):

```
boxjenk(constant,ar=2) spread 1961:4 2008:1
@regcorrs (dfc=%narma,number=20,qstats,report,$
method=burg,title="AR(2) model diagnostics")
boxjenk(constant,ar=1,ma=1) spread 1961:4 2008:1
@regcorrs (dfc=%narma,number=20,qstats,report,$
method=burg,title="ARMA(1,1) model diagnostics")
```

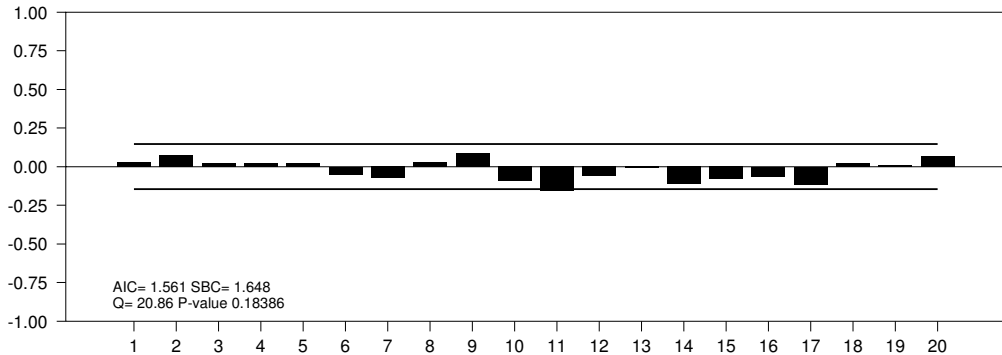
In both cases, the Q statistics and correlation plots suggest the presence of significant correlations in the residuals. For example, here are the results for the AR(2) model:



So, these particular models are not accounting for all of the correlation behavior, and are unlikely to provide reliable forecasts.

Enders finds that an ARMA(2,1) model performs better than the ARMA(1,1) model, but leaves some significant correlations around lag 7, and so he next considers a model with AR terms at lags 1 and 2, and MA terms on lags 1 and 7. This kind of model is handled easily in RATS, by using the in-line matrix notation (`| |value,value,...,value| |`) to provide a list of lags:

```
boxjenk(constant,ar=2,ma=||1,7||) spread 1961:4 2008:1
@regcorrs(dfc=%narma,number=20,qstats,report,$
method=burg,title="ARMA(2,(1,7)) model diagnostics")
```



Both the AR(7) and ARMA(2,(1,7)) models look promising. The AIC criteria (computed by **REGCORRS**) favors the AR(7) model, while the SBC criteria favors the ARMA(2,(1,7)).

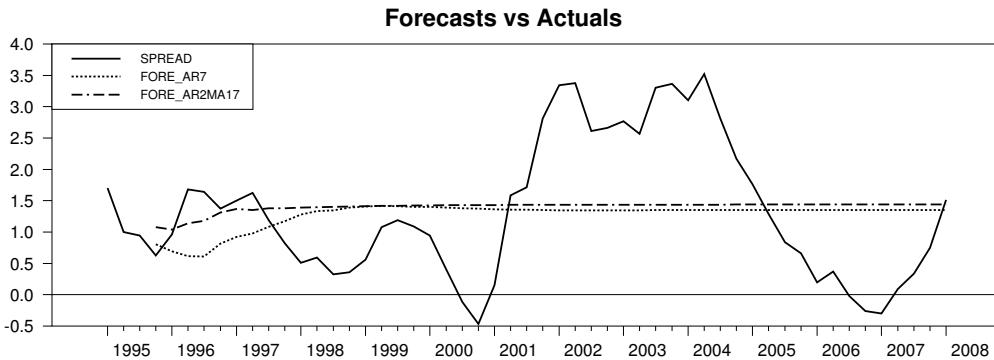
The next step is to compare the forecasting performance of these two models. To do this, we need to include the **DEFINE** option on the **BOXJENK** instructions that estimate the model. We can then use the instruction **UFORECAST** to compute the forecasts.

First, we'll estimate both models through 1995:3, holding back the last fifty observations of data. We then compute forecasts for those fifty periods, and generate a graph comparing the forecasts with the actual values. Note, by the way, that these are multiple step dynamic forecasts.

```
boxjenk(constant,define=ar7eq,ar=7) spread * 1995:3
boxjenk(constant,define=ar2ma17eq,ar=2,ma=||1,7||) spread * 1995:3

uforecast(equation=ar7eq,print) fore_ar7 1995:4 2008:1
uforecast(equation=ar2ma17eq,print) fore_ar2ma17 1995:4 2008:1

graph(header="Forecasts vs Actuals",key=upleft) 3
# spread 1995:1 *
# fore_ar7
# fore_ar2ma17 / 5
```



The results are rather typical of time series forecasts for models that do not include a trend: after the two years or so, the forecast is almost a straight line. As mentioned on page UG-177, the long-term forecasts are dominated by the choice of differencing (if any) plus the intercept. Here, with no differencing applied, both forecasts eventually level out at a constant value.

Now, we'll look at how well the models fare when forecasting just one step beyond the end of the estimation range. Naturally, the one-step forecasts track much better than the multi-step forecasts. This code loops over our 50 periods of holdback data, estimating through entry T-1 and forecasting entry T, for T=1995:4 through 2008:1.

```
do time=1995:4,2008:1
  * Estimate through period TIME-1
  boxjenk(noprint,constant,define=ar7eq,ar=7) spread * time-1
  boxjenk(noprint,constant,define=ar2ma17eq,ar=2,ma=|1,7|) $
    spread * time-1
  * Forecast one step ahead (entry TIME)
  uforecast(equation=ar7eq,static) forecast_ar7 time time
  uforecast(equation=ar2ma17eq,static) forecast_ar2ma17 time time
end do

* Graph the forecast with the actuals.
graph(header="Forecasts vs Actuals",key=upleft) 3
# spread 1995:1 *
# forecast_ar7
# forecast_ar2ma17eq / 5
```

Forecast Errors and Theil U Statistics

For one-step ahead forecasts, we can use the @UFOREERRORS procedure to quickly compute forecast performance statistics:

```
@uforeerrors spread forecast_ar7
@uforeerrors spread forecast_ar2ma17
```

To look at forecast performance over longer horizons, we can use the instruction **THEIL**. This does a series of multiple step forecasts and does parallel analysis on each horizon. One of the statistics that it computes is Theil's U statistic. This is the ratio between the RMS errors of the model's forecasts to the RMS errors of a "naïve" forecast of no change in the dependent variable from the previous value. See Chapter 5 and the description of **THEIL** for details. This has the advantage over the basic error statistics like the RMSE as it is scale independent, and also offers a quick comparison with another forecasting procedure. Here's how to use this for our ARMA model. We compute forecasts for (up to) 8 steps ahead for each starting date in the range 1995:4 through 2008:1. The model is re-estimated after each set of forecasts to use the next available data point.

```
boxjenk(constant,define=ar2ma17eq,ar=2,ma=||1,7||) spread * 1995:3
theil(setup,steps=8,to=2008:1)
# ar7eq
do time=1995:4,2008:1
    theil time
    boxjenk(noprint,constant,define=ar2ma17eq,ar=2,ma=||1,7||) $
        spread * time
end do time
theil(dump,title="Theil U results for ARMA(2,(1,7)) Model")
```

The results are as follows:

Theil U results for ARMA(2,(1,7)) Model:

Step	Mean Error	Mean Abs Err	RMS Error	Theil U	N Obs
Forecast Statistics for Series SPREAD					
1	0.0131677	0.3449180	0.4389769	0.9477	50
2	0.0221656	0.5791501	0.7038170	0.9279	49
3	0.0155860	0.7343455	0.8825325	0.8821	48
4	-0.0035745	0.8302931	1.0263826	0.8485	47
5	-0.0270086	0.9029884	1.1094199	0.8158	46
6	-0.0347677	0.9691531	1.1482820	0.7892	45
7	-0.0416368	0.9942000	1.1574798	0.7613	44
8	-0.0474994	1.0144249	1.1664960	0.7320	43

These results are slightly superior (smaller errors and Theil U's at each horizon) than the results for the AR(7) model, but the differences between the two sets of results are small. Overall, they suggest that both models should provide adequate (and comparable) forecasts.

In Section 6.4.2, we show how to do more of these steps using procedures.

6.4.2 The ARIMA Procedures

RATS includes several procedures designed to simplify the process of selecting, estimating and forecasting ARIMA models. We introduce these procedures below, and apply them to our interest rate example.

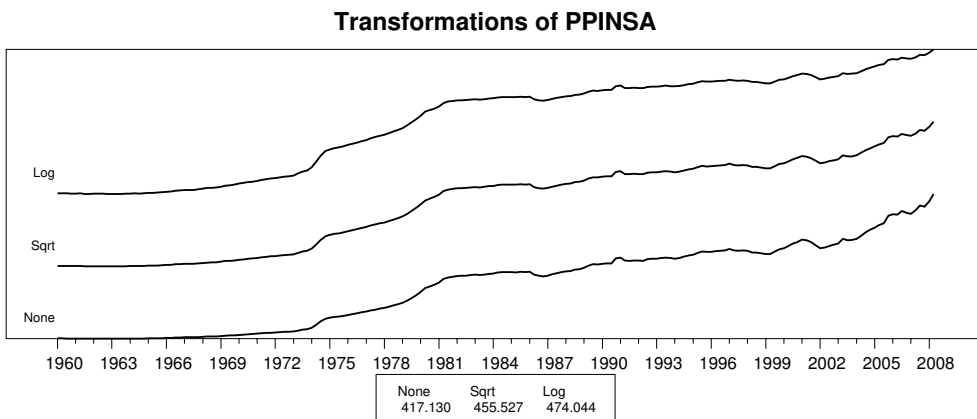
@BJTRANS Procedure

The procedure **@BJTRANS** can help you determine the appropriate preliminary transformation if you are unsure which to use. It graphs the series, its square root and its log on a single graph so you can compare their behavior. As mentioned on page UG-177, you are looking for a transformation which “straightens” out the trend (if there is one), and produces approximately uniform variability. The graph also shows a (very crude) log likelihood of a model under the three transformations. You shouldn’t necessarily choose the transformation based upon which of these is highest, as a close decision might be reversed with a careful model for the transformed processes. However, if there is a very clear difference among the likelihoods, they are likely to be pointing you to the best choice. The syntax is:

```
@bjtrans series start end
```

We can’t apply **@BJTRANS** to the **SPREAD** series, as it contains non-positive values (so the square root and log transforms won’t work). To demonstrate how this works, we can apply the procedure to a producer price index series that is also included on the **QUARTERLY.XLS** file:

```
data(format=xls,org=columns) / ppinsa  
@bjtrans ppinsa
```



Now, this is a series which we expect to be best modeled in logs, and the log likelihood values would support that. The levels (“None”) are clearly inadequate. Note, from the graph, that there is effectively no visible short-term movements in the first half of the sample, while it is considerably less smooth in the second half, but the log transform gives a more uniform variability across the full sample.

@BJDIFF Procedure

@BJDIFF can aid in choosing the proper set of differences for a series. It generates a table of BIC criteria for various combinations of differencings and mean extractions. With non-seasonal data the choice of differencing is usually either quite clear or doesn't matter much. There are more options with seasonal data and the choice isn't quite as obvious. In addition, the usual Box-Jenkins methodology uses a graphical procedure (helped by the **@BJIDENT** procedure) which won't help if you need an automated process.

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

Parameters

<i>series</i>	series to analyze
<i>start, end</i>	range of series to use. By default, the defined range of series. Do not adjust these for the number of differencings.

Options

```
diff=maximum regular differencings[0]  
sdiffs=maximum seasonal differencings[0]  
span=seasonal span [CALENDAR seasonal]
```

```
trans=[none]/log/root  
Transformation to apply to the data
```

```
width=window width for spectral smoothing  
The variance of the "stationary" part is estimated using non-parametric spectral methods. This controls how that's computed. The width depends upon the number of data points.
```

@BJDIFF produces a table with a list of the possible combinations with the minimum BIC choice starred. It also defines %%AUTOD as the recommended number of regular differencings, %%AUTODS as the recommended number of seasonal differencings and %%AUTOCONST as the recommended choice for CONST option.

@BJIDENT Procedure

The procedure **@BJIDENT** assists in identifying a Box-Jenkins model by computing and graphing the correlations of the series in various differencing combinations. Here is the syntax for the **@BJIDENT** procedure, with the most common set of options:

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

Chapter 6: Univariate Forecasting

Parameters

series series for which you want to identify a model

start end range over which you want to compute correlations. Defaults to the maximum usable range given the selected `DIFFS`/`SDIFFS`.

Options

`diffs=maximum regular differencings [0]`

`sdiffs=maximum seasonal differencings [0]`

`@BJIDENT` examines every combination of regular and seasonal differencing between 0 and the maximums indicated. For instance, with each set at 1, it will do 0×0, 0×1, 1×0 and 1×1.

`number=number of autocorrelations to compute [25 if possible]`

`method=[yule]/burg`

Sets the number of autocorrelations to compute and graph. The `METHOD` option selects the method to be used in computing the correlations (see the description of the `CORRELATE` instruction).

`trans=[none]/log/root`

Selects the preliminary transformation.

`[graph]/nograph`

If `GRAPH`, `BJIDENT` creates high-resolution graphs of the correlations.

`report/[noreport]`

`qstats/[noqstats]`

If you use the `REPORT` option, `BJIDENT` will create a table of the computed statistics for the various differencing combinations. If you include `QSTATS` as well, it will compute Ljung-Box *Q* statistics and their significance levels. While some software does the *Q* statistics as part of their “ident” functions, we recommend against it. The *Q*’s are tests for white noise; they have no real implications for stationarity decisions, and at this point, we don’t expect to see the correlations pass a white noise test even with the correct number of differencings.

Examples

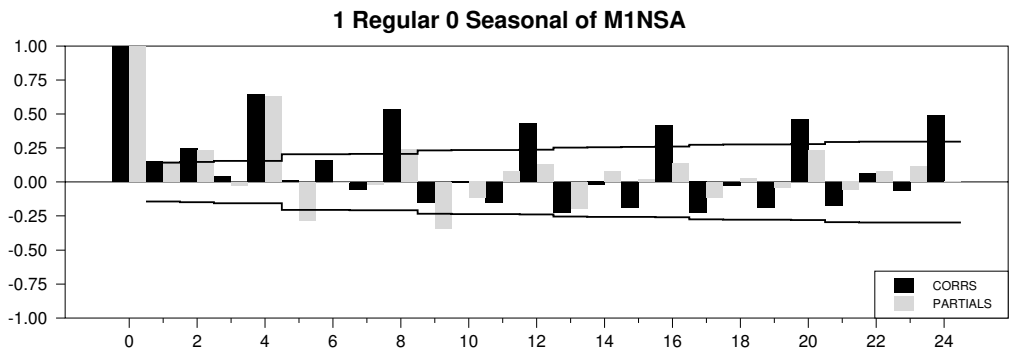
Recall the fairly complicated `DIFF`, `CORR` and `GRAPH` instructions that we used in our earlier example to produce graphs of the correlations for both `SPREAD` and the first difference of `SPREAD`. Using `@BJIDENT`, the following is all you need to produce even fancier versions of these graphs:

```
open data quarterly.xls
calendar(q) 1960:1
data(format=xls,org=columns) / tbill r10 ppinsa
set spread = r10 - tbill
@bjident(diffs=1) spread
```

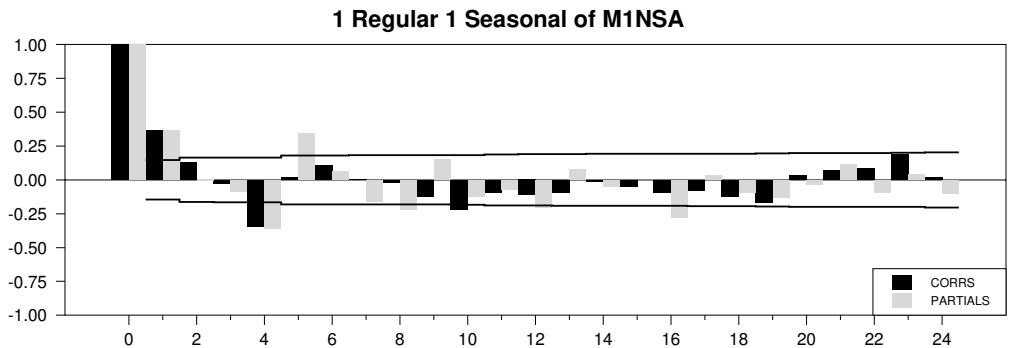
The next example produces graphs for all combinations of 0 or 1 regular difference and 0 or 1 seasonal differences for the log of M1NSA (a non-seasonal adjusted series of the U.S. M1 money supply (also available on `QUARTERLY.XLS` and discussed in Section 2.11 of the Enders text). It shows 24 correlations. We also include the `REPORT` and `QSTATS` options, to generate a report window containing the correlations and Q statistics at each horizon:

```
data(format=xls,org=columns) / tbill r10 ppinsa m1nsa
@bjident(diffs=1,sdifs=1,trans=log,number=24,report,qstats) m1nsa
```

The graphs with no differencing make it clear that we need at least one regular difference. The graph with one regular difference but no seasonal difference suggests that we do in fact need a seasonal difference to deal with the persistent seasonality:



As you can see from the next graph, adding a seasonal difference accounts for most of the remaining correlation, but still does not account for all of the seasonality:



Enders settles on a model with AR(1) and SMA(1) (seasonal moving average) terms for these data.

Chapter 6: Univariate Forecasting

@BJFORE Procedure

The **@BJFORE** procedure provides an easy way to estimate and forecast an ARIMA model with a single instruction. The syntax is as follows:

```
@bjfore(options) series forestart foreend forecasts
```

Parameters

<i>series</i>	Series to be forecast
<i>forestart foreend</i>	Range of entries to forecast
<i>forecasts</i>	Series for computed forecasts

Options

```
diffs=number of regular differences [0]  
sdiffs=number of seasonal differences [0]  
ars=number of autoregressives [0]  
mas=number of moving averages [0]  
sar=number of seasonal autoregressives [0]  
sma=number of seasonal moving averages [0]  
span=seasonal span. Defaults to CALENDAR seasonal
```

```
[constant]/noconstant
```

```
trans=[none]/log/root
```

CONSTANT includes a constant term in the model. TRANS allows you to apply a log or square root transformation to the series. The default is no transformation.

```
[print]/noprint
```

NOPRINT turns off display of **BOXJENK** estimation output.

```
rstart=start of estimation range [earliest possible]
```

```
rend=end of estimation range [forestart-1]
```

These set the range of entries used to estimate the mode. RSTART defaults to the earliest possible entry. REND defaults to (*forestart*-1).

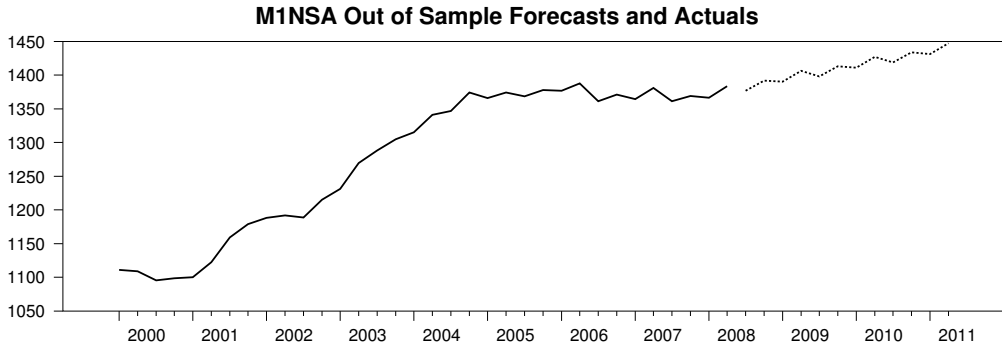
Examples

Returning to our SPREAD example, we can estimate the model through 1995:3 and generate dynamic forecasts for 1995:4 through 2008:1 using just this line:

```
@bjfore(ars=7,constant) spread 1995:4 2008:1 fore resids
```

For our $(1,1,0) \times (0,1,1)$ model for M1NSA, we can estimate the model and graph the results using the following:

```
@bjfore(ars=1,smas=1,trans=log,diffs=1,sdifs=1,constant) m1nsa $  
2008:2+1 2008:2+12 m1fore m1resids  
graph(header="M1NSA: Actuals and Out of Sample Forecasts") 2  
# m1nsa 2000:1 *  
# m1fore
```



@REGCORRS Procedure

We introduced the **@REGCORRS** procedure earlier in this section. Recall that it computes and optionally graphs the residual autocorrelations, computing the Q -statistic and the AIC and SBC. It can be used after any type of single-equation estimation, not just after **BOXJENK**. Here is the formal syntax:

```
@regcorrs ( options )   resids
```

Parameters

resids Series of residuals to analyze. By default, this uses the residuals from the most recent estimation.

Options

```
[graph]/nograph
print/[noprint]
[qstats]/noqstats
report/[noreport]
[criteria]/nocriteria
title="title for report window"
header="graph header string"
footer="graph footer string"
```

If **GRAPH**, **REGCORRS** creates high-resolution graphs of the correlations. If **PRINT**, the output from **CORRELATE** is included in the output window, and, if, in addition, you use **QSTATS**, that output will include the Q -statistics. If **REPORT**, a separate table is displayed with the correlations, partial autocorrelations and Q -statistics for different lag lengths.

number=*number of autocorrelations to compute* [25 if possible]

dfc=*degrees of freedom correction for Q statistics*

method=**[yule]/burg**

NUMBER sets the number of autocorrelations to compute and graph. **DFC** should

Chapter 6: Univariate Forecasting

be set to the number of ARMA parameters estimated; **BOXJENK** sets the variable %NARMA equal to this. The **METHOD** option selects the method to be used in computing the correlations (see the description of the **CORRELATE** instruction). **METHOD=YULE** is used by most other time-series software.

Variables Defined

In addition to the *Q*-statistic variables (%QSTAT, %QSIGNIF and %NDFQ), **@REGCORRS** also computes %AIC and %SBC, which are the Akaike Information Criterion and the Schwarz Bayesian Criterion. The values of these will also be displayed on the graph.

@BJAUTOFIT Procedure

The **@BJAUTOFIT** procedure estimates (by maximum likelihood) all combinations of ARMA terms in a given range, displaying a table with the values of a chosen information criterion, with the minimizing model having its value highlighted with a *. Note that this can take quite a while to complete if the **PMAX** and **QMAX** are large. Also, some of the more complicated models may not be well-estimated with just a simple use of **BOXJENK (MAXL)**. However, these estimation problems generally arise because of poorly identified parameters, which means that a simpler model would fit almost as well and would thus be preferred anyway. The syntax is as follows:

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

Parameters

<i>series</i>	Series to be fit
<i>start end</i>	Range of entries to use for estimation. Default is largest range possible given the data and any differences.

Options

```
pmax=maximum number of AR lags to consider [0]  
qmax=maximum number of MA lags to consider [0]  
crit=[aic]/bic/sbc (criterion to use—BIC and SBC are synonyms)  
const/[noconst] (estimate mean as part of model)  
demean/[nodemean] (extract mean before estimating ARMA models)  
diffs=number of preliminary differences [0]  
sdiffs=number of preliminary seasonal differences [0]
```

Example

We use **@BJAUTOFIT** on the log **SPREAD** data, looking at models up to an ARMA(7,7).

```
@bjautofit(constant,pmax=7,qmax=7,crit=sbc) spread
```

Here, the procedure selects the ARMA(2,1) model (as indicated by the *), which was one of our leading candidates earlier:

BIC analysis of models for series SPREAD

MA									
AR	0	1	2	3	4	5	6	7	
0	618.96	436.75	383.69	343.23	340.28	334.86	314.74	319.97	
1	323.03	313.41	316.03	316.71	321.36	325.10	319.98	324.66	
2	316.51	313.37*	322.32	320.41	324.12	329.98	363.25	335.19	
3	320.84	318.49	321.27	324.01	329.26	333.43	335.75	336.20	
4	319.63	324.70	321.66	329.10	326.68	329.63	330.91	333.26	
5	324.83	326.19	327.67	326.34	325.85	336.49	334.75	335.70	
6	321.86	324.50	322.75	333.07	339.78	333.38	347.77	466.51	
7	322.89	327.09	329.74	338.85	338.12	346.98	359.14	489.79	

The procedure does not allow for models with skipped lags, so it's not surprising that it selects a slightly simpler model than the ARMA(2,(1,7)) we chose after noting some remaining correlation around lag 7 in the ARMA(2,1) residuals. This demonstrates that criterion-based selection tools can be helpful for narrowing down choices or verifying conclusions, but are no substitute for careful analysis and judgement.

The procedure saves the selected AR and MA values in the variables %%AUTOP and %%AUTOQ, so you can estimate the model and analyze the residuals by doing:

```
boxjenk (constant,define=ar2maleq,ar=%%autop,ma=%%autoq) spread
@regcorrs (dfc=%narma)
```

@GMAUTOFIT Procedure

@GMAUTOFIT is similar to @BJAUTOFIT but is designed to choose the more complicated multiplicative seasonal ARMA model to a series. This follows the procedure described in Gomez and Maravall(2001). Because there are four polynomial lengths to choose and seasonal models take much longer to estimate than non-seasonal models, Gomez-Maravall searches for an approximate minimum BIC model. Rather than attempt an exhaustive search across all four simultaneously (though you can do that with the FULL option), it first fixes the “regular” ARMA parameters, then does an exhaustive search over the seasonal parameters only. Once those are chosen, they are fixed and an exhaustive search is done for the regular ARMA parameters. An example of its use is provided by AUTOBOX.RPF.

6.4.3 Intervention Modeling

Background

Intervention modeling allows you to deal with simple types of structural breaks while using an ARIMA model. It does this by adding a deterministic component, such as a dummy variable, to the basic ARIMA model. Its scope is somewhat limited because it requires that the date and general form of the change be known. It also requires the rather strong assumption that the time series properties of the series stay the same after the break. For instance, if a country chooses to devalue its currency and then attempts to keep its value in line with (say) the US dollar, we would know the date, and the form of the break, but the time series structure would likely be quite different before and after, making intervention modeling impractical.

Tools

Intervention models are estimated using **BOXJENK**. However, before running **BOXJENK**, you must create the deterministic components which will create the effect that you want. These are added to the model using the **INPUTS** option. Usually there is just one input, so we'll demonstrate that:

```
boxjenk (inputs=1, other options) series start end  
# input p d q
```

For intervention modeling, it is usually simplest to keep the time series in its undifferenced form. Then, by using the **APPLYDIFFERENCES** option you will be able to figure out the form of the input based upon what change it makes to the *level* of the series. If you work with the differenced series, you will have to work out the input as a shift in the differenced series, which is usually much harder.

Here are some examples of intervention inputs:

For a permanent change in level beginning at T_0

```
set change = (t>=t0)
```

For a short term change in level from T_0 to T_1

```
set change = (t>=t0.and.t<=t1)
```

For a permanent change in growth rate beginning at T_0

```
set change = %max(0.0,t-(t0+1))
```

The instruction **DUMMY** can also be used to create these dummies (with the **LS** (level shift) option for the first two and the **RAMP** option for the third), and others. However, **DUMMY** uses equivalent definitions which are zero at the end of the sample; for instance, **DUMMY (LS=T0)** will be -1 through T_0-1 and 0 afterwards. This makes it easier to extrapolate the effects beyond the end of the sample.

Notes

Even if you know the date of the intervention *and* are satisfied that you understand its form, *and* you are willing to assume that the time series properties are the same except for the structural break, you still have one problem: figuring out the proper form for the base ARIMA model. You can't apply the standard identification techniques to the original series, as the autocorrelations will be thrown off by the structural break. There are a couple of ways to approach this:

1. You can apply the identification procedures to whichever stretch of “stable” data is longest; that is, restrict your attention only to the part of the sample before the break, or the part after it.
2. You can apply the identification procedure to the time series after accounting for the type of change that you are planning to permit. This works best for a level change, as you can just take the residuals from the regression on `CONSTANT` and your dummy variable. If you have a growth rate change, you're almost certainly looking at data which, at minimum, will be differenced once. The equivalent procedure in that case would be to difference the data, and then take the residuals from the regression on `CONSTANT` and a level change dummy starting at `T0`.

The “denominator lags” parameter on the input line can be used if there is a “phase-in” before the permanent change takes full effect. When applied to a temporary effect, the denominator lags create a “phase-out.” A single denominator lag allows for a geometric convergence to the new level. If you use this, you have to be careful interpreting the coefficients. In all the examples above, if you use a simple input without lags, the coefficients on the `CHANGE` variable will be easy to interpret: in the first two, they will show the size of the level shift, and in the last the difference in the growth rate. However, in the two permanent effects, if the coefficient on the input is 3.0 with a denominator lag coefficient of .6, the long-run effect will be $3.0/(1-.6)$ or 7.5.

Example

The U.S. stock market took a very large tumble in October, 1987. We will apply intervention techniques to the S&P 500. Because the series we are using consists of monthly averages (not the ideal choice for a serious analysis of this event—daily closing prices would be better), the October, 1987 value does not fully show the decline, since over half the values averaged into the October value were from before the large drop on October 19. This means that we're going to need a lag in the numerator as well, to give the effect two months to take full effect. We'll also look at two possible ways to model the change: a permanent drop, or a temporary drop with a phase out (see DeLurgio (1998) for another treatment of this same basic example).

We won't go through the process of choosing the base ARIMA model for this series. It proves to be a relatively easy choice of an `ARIMA(0,1,1)` with a `CONSTANT` on the logs. (The `ARIMA(0,1,1)` is the expected model for the time-averages of a series which is likely to be fairly close to a random walk in continuous time). This is example `INTERVENTION.RPF`.

Chapter 6: Univariate Forecasting

```
cal(m) 1947:1
open data haversample.rat
data(format=rats) 1947:1 1993:12 sp500
set logsp_500 = log(sp500)
```

The permanent effect is done with a level shift starting in 1987:10. The temporary effect is a spike at 1987:10.

```
dummy(ls=1987:10) perm
dummy(ao=1987:10) temp
```

The permanent effect is done with lags 0 and 1 on perm

```
boxjenk(diff=1,applydiff,constant,inputs=1,ma=1) logsp_500 $
1980:1 1993:12 presids
# perm 1
```

The temporary effect is done with numerator lags 0 and 1 and a single denominator lag.

```
boxjenk(diff=1,applydiff,constant,inputs=1,ma=1) logsp_500 $
1980:1 1993:12 tresids
# temp 1 1
```

The following are the regression tables from the two estimated models. Don't be deceived by the extremely high t -stat on the $D\_TEMP\{1\}$ coefficient in the second model. It doesn't mean the second model is better. In fact, the first model is a special case of the second with the coefficient on $D\_TEMP\{1\}$ fixed at 1.0, so the two are almost identical. The results show a roughly 25% drop in the index through the two month period in October and November.

1.	CONSTANT	0.010310788	0.002907481	3.54630	0.00050947
2.	MA{1}	0.200127855	0.077450051	2.58396	0.01063889
3.	N_PERM{0}	-0.128493661	0.031743366	-4.04789	0.00007949
4.	N_PERM{1}	-0.137958367	0.031372235	-4.39747	0.00001963
1.	CONSTANT	0.008715909	0.003419433	2.54893	0.01172839
2.	MA{1}	0.317061758	0.074340267	4.26501	0.00003377
3.	N_TEMP{0}	0.001909992	0.020301010	0.09408	0.92515830
4.	N_TEMP{1}	-0.002947065	0.021264954	-0.13859	0.88994688
5.	D_TEMP{1}	-0.952734786	0.072001530	-13.23215	0.00000000

6.4.4 ARMAX and RegARIMA Models

RegARIMA and ARMAX models both combine features of simple regression models with ARIMA modelling. **BOXJENK** supports both approaches. In RATS, both types of models are parameterized in exactly the same way—the differences lie in the estimation technique.

With ARMAX, the focus is on the ARMA process itself, with additional explanatory variables. For RegARIMA, the focus of the estimation is on the mean equation represented by the explanatory variables rather than the ARIMA model “noise” term. The output is switched around so the explanatory variables are listed first.

The RegARIMA approach is used in the X12-ARIMA methodology, supported in Professional versions of RATS through a combination of the **BOXJENK** and **X11** instructions.

APPLYDIFFERENCES

The **APPLYDIFFERENCES** option is used to choose between the following two forms for a mixed model: the first for **APPLYDIFFERENCES**, the second for **NOAPPLY**.

$$(1) \quad (1 - L)^d (1 - L^s)^e (y_t - X_t \beta) = \text{ARMA noise}$$

$$(2) \quad (1 - L)^d (1 - L^s)^e y_t = X_t \beta + \text{ARMA noise}$$

In most applications, (1) is more natural, though with deterministic regressors such as trends and dummies, the model can generally be written either way. For instance, if we need to first difference, and the regression model in the first form has 1, t and the level shift dummy $D_{t \geq T_0}$, then the corresponding regressors in the second form are (zero), 1 and the additive outlier dummy $D_{t=T_0}$, which you get by differencing the X_t . The original intercept giving the overall level of the process is lost in moving between the two, and can be recovered only by referring back to the original data.

There’s a third possible way to add regressors to an ARMA model, which is

$$(3) \quad \varphi(L) \left\{ (1 - L)^d (1 - L^s)^e y_t \right\} = X_t \beta + \theta(L) u_t$$

However, it’s harder to interpret the regression part in this form than in the similar form (2). The two are equivalent if there are no AR terms—if there are AR terms and you need to estimate a model like this, you can use **ITERATE** rather than **BOXJENK**.

REGRESSORS option

To estimate an ARMAX model in RATS, use the **REGRESSORS** option on **BOXJENK**, and follow the instruction with a supplementary card listing (in regression format) the variables in the regression model. The ARMA terms are handled using the standard AR and MA options. You can also use the **INPUTS** option, though that is designed primarily for implementing transfer function and intervention models (Section 6.4.3).

Chapter 6: Univariate Forecasting

The following is part of an example from Diebold (2004). The series of interest is liquor sales. It has both a strong trend and seasonal. The basic regression model is a full set of seasonals plus trend and trend squared:

```
seasonal seasons
linreg lsales 1968:1 1993:12 resid
# time time2 seasons{0 to -11}
```

From the residuals, an AR(3) model is identified for the noise term. The full model is estimated using the **BOXJENK** instruction shown.

```
boxjenk(regressors,ar=3) lsales 1968:1 1993:12 resid
# time time2 seasons{0 to -11}
```

RegARIMA Models

Use the GLS option, rather than REGRESSORS, to do RegARIMA estimation. As with REGRESSORS, you supply a list of explanatory variables on a supplementary card. The output is switched around so the explanatory variables are listed first. GLS forces the use of maximum likelihood estimation and also includes the behavior of the APPLYDIFFERENCES option.

You can employ the automatic outlier detection as part of your estimation. This is done using the OUTLIER option which selects the types of outliers for which to scan. OUTLIER can take a while since it needs to estimate a model for each data point for each type of outlier.

The example file REGARIMA.RPF analyzes a series of (raw) household appliance sales. Because sales are higher on certain days of the week (Friday, Saturday and Sunday), there is a “trading day” effect in the data. We also need to account for a predictable “leap year” effect, since a February with a leap year has one extra day to sell appliances than Februaries which don’t. The following is used to generate the shift regressors, trading day counts (differences between the other days of the week and Sunday) and the leap year dummy:

```
dec vect[series] td(6)
do i=1,6
  set td(i) = %tradeday(t,i)-%tradeday(t,7)
end do i
set lpyr = %if(%period(t)==2,-.25+(%clock(%year(t),4)==4),0.0)
```

The log of series is analyzed, so, for instance, the LPYR dummy estimates the percentage difference due to February 29. After some preliminary analysis, we estimate a final RegARIMA model and extract the adjustment factor with:

```
boxjenk(gls,diffs=1,sdifs=1,sma=1,outlier=ao,$
  adjust=tdadjust) logsales
# constant td lpyr
```

6.5 Spectral Forecasting

Background

Spectral forecasting uses frequency domain methods to fit a “generic” Box-Jenkins model to the data. The preparations (transformation and differencing) are the same as for a BJ model.

It is a non-parametric method, so parsimony is not a well-defined concept. Technically, the estimates of cycles are smoothed to eliminate possibly spurious effects due to sampling error.

The procedure used is technically very complex. For those of you who are familiar with z -transforms, we describe the algorithm in Section 14.9.

Advantages

The advantages of spectral forecasting tend also to be disadvantages. It is an automatic procedure once you have chosen the preliminary transformation and trend model. Given adequate data, it can produce good forecasts quickly and painlessly. It can do better than Box-Jenkins, particularly when no model with just a few parameters seems to work well.

Disadvantages

As an automatic procedure, there are no diagnostics along the way to steer us away from problems. It is less tolerant of mistakes in preliminary transformation and requires more data (100+ observations) than parametric techniques.

Procedures

You can do spectral forecasting using the procedure **@SPECFORE**. The syntax for the procedure is:

```
@specfore ( options )   series   start  end    forecasts
```

Parameters

<i>series</i>	Series to forecast
<i>start end</i>	Range of entries to forecast
<i>forecasts</i>	Series for computed forecasts

Chapter 6: Univariate Forecasting

Options

The options select the transformation and trend model.

diffs=Number of regular differences [0]

sdiffs=Number of seasonal differences [0]

[constant]/noconstant

Select NOCONSTANT to exclude the intercept from the differenced model.

trans=[none]/log/root

Select the appropriate preliminary transformation.

Example

This computes a spectral method forecast of the spread series used in Section 6.4 It's example file SPECFORE.RPF.

```
open data quarterly.xls
calendar(q) 1960:1
data(format=xls,org=columns) * 2008:1 tbill r10
*
* Compute spread
*
set spread = r10 - tbill
*
* Forecasts with ARMA(2,(1,7)) model, holding back the last
* two years of data.
*
boxjenk(constant,ar=2,ma=||1,7||,define=armaeq) $
    spread 1961:4 2006:1
uforecast(equation=armaeq,from=2006:2,to=2008:1) armafore
*
* Forecasts with @SPECFORE procedure
*
@specfore spread 2006:2 2008:1 spfore
*
graph(header="Forecasts of Spread",key=below,klabels=$
    ||"Actual","ARMA Forecasts","Spectral Forecasts"||) 3
# spread 2004:1 2008:2
# armafore
# spfore
*
* Compute and graph out-of-sample forecasts
*
@specfore spread 2008:2 2009:12 outofsample
graph(key=below,header="Out of Sample Forecasts") 2
# spread 2006:1 2008:1
# outofsample
```

7. Vector Autoregressions

Vector autoregressions (VAR's) are dynamic models of a group of time series. In “Macroeconomics and Reality” and later papers, Sims has proposed using VAR's as an alternative to large simultaneous equations models for studying the relationship among the important aggregates. The two main uses of this methodology are:

- to test formally theories which imply particular behavior for the vector autoregression.
- to learn more about the historical dynamics of the economy.

With the use of Bayesian techniques, VAR's have also been employed very successfully for small multivariate forecasting models.

VAR's have become a key tool in modern macroeconometrics. While there is some coverage of the topic in econometrics books such as Greene (2012) and introductory time series books like Diebold (2004), we would recommend more specialized books such as Enders (2014), Hamilton (1994) and Lütkepohl (2006). Of these, Enders is the most applied; the others are more theoretical.

In addition to those, the notes from our *Vector Autoregression* e-course can be very helpful. For more information on that, see

https://estima.com/courses_completed.shtml

Estimation
Testing Lag Length
Exogeneity Tests
Orthogonalization and Covariance Models
Impulse Responses
Decomposition of Variance
Historical Decomposition
Cointegration and Error-Correction
Forecasting with a Prior
Sequential Estimation and the Kalman Filter
Conditional Forecasting

Chapter 7: Vector Autoregressions

7.1 Setting Up a VAR

Formally, a vector autoregression may be written

$$(1) \quad \mathbf{y}_t = \mathbf{X}_t \beta + \sum_{s=1}^p \Phi_s \mathbf{y}_{t-s} + \mathbf{u}_t \quad E(\mathbf{u}_t \mathbf{u}_t') = \Sigma$$

where $\mathbf{y}$ is an N -vector of variables and Φ_s is an $N \times N$ matrix. There are a total of $N^2 L$ free coefficients in this model.

You can set up a standard VAR using the *Time Series—VAR (Setup/Estimate)* wizard, or by using the following instructions directly:

```
system(model=modelname)  
variables          list of endogenous variables  
lags              list of lags  
deterministic    list of deterministic/additional variables in regression format  
end(system)
```

The lags listed on **LAGS** are usually consecutive, for instance, 1 TO 12, but you can skip lags (for instance, 1 2 3 6 12). The list of deterministic variables is usually just **CONSTANT** and possibly seasonal or other dummies, but they can be any variables other than the lagged endogenous variables. For example:

```
system(model=canusa)  
variables usam1 usatbill canm1 cantbill canusxr  
lags 1 to 13  
det constant  
end(system)
```

defines a five-equation, 13-lag VAR model. Note that these instructions simply define the VAR system. The model then needs to be estimated, which is usually done using the **ESTIMATE** instruction (Section 10.2).

Preliminary Transformations

You should choose the transformation for each series (log, level or other) that you would pick if you were looking at the series individually. Thus, you transform exponentially growing series, such as the price level, money stock, GNP, etc. to logs. You will usually leave in levels non-trending series, such as interest or unemployment rates. This is especially important for interest rates in a VAR including prices or exchange rates (which should be in logs) because real interest rate and parity conditions can be expressed as a linear relationship among the variables.

While preserving interesting linear relationships is desirable, you should not shy away from obvious transformation choices to achieve them. For instance, in Doan, Litterman and Sims (1984), two of the variables in the system were government receipts and expenditures. These are obvious candidates to be run in logs, which makes the deficit a non-linear function of the transformed variables. If we had used levels instead of logs, it would have made studying the budget deficit easier as we would

not have needed to linearize. However, the predictions would have been unrealistic since the growth relationship between receipts and expenditures would have been distorted.

Should I Difference?

Our advice is no, in general. In Box–Jenkins modeling for single series (Chapter 6), appropriate differencing is important for several reasons:

- It is impossible to identify the stationary structure of the process using the sample autocorrelations of an integrated series.
- Most algorithms used for fitting ARIMA models will fail when confronted with integrated data.

Neither of these applies to VAR's. In fact, the result in Fuller (1976, Theorem 8.5.1) shows that differencing produces no gain in asymptotic efficiency in an autoregression, *even if it is appropriate*. In a VAR, differencing throws information away (for instance, a simple VAR on differences cannot capture a co-integrating relationship), while it produces almost no gain.

Trend or No Trend?

In most economic time series, the best representation of a trend is a random walk with drift (Nelson and Plosser (1982)). Because of this, we would recommend against including a deterministic trend term in your VAR. In the regression

$$\mathbf{y}_t = \alpha + \gamma t + \beta_1 \mathbf{y}_{t-1} + \dots + \beta_p \mathbf{y}_{t-p} + \mathbf{u}_t$$

if we expect to see a unit root in the autoregressive part, γ becomes a coefficient on a *quadratic* trend, while α picks up the linear trend. As you add variables or lags to the model, there is a tendency for OLS estimates of the VAR coefficients to “explain” too much of the long-term movement of the data with a combination of the deterministic components and initial conditions (Sims, 2000). While this may seem to “improve” the fit in-sample, the resulting model tends to show implausible out-of-sample behavior.

How Many Lags?

If you want to, you can rely on information criteria, such as the Akaike Information Criterion (AIC) or the Schwarz or Bayesian Information Criterion (variously BIC, SBC, SIC), to select a lag length (see “VARLAG.RPF Example”). However, these tend to be too conservative for most purposes. Where we are including an identical number of lags on all variables in all equations, the number of parameters goes up very quickly—we add N^2 parameters with each new lag. Beyond the first lag or two, most of these new additions are likely to be unimportant, causing the information criteria to reject the longer lags in favor of shorter ones. The use of priors (see Section 7.10) is an alternative to relying on short lags or data-driven selection methods.

If the data are adequate, it is recommended that you include at least a year's worth of lags. In fact, a year's worth plus one extra can deal with seasonal effects, which can be present even with seasonally adjusted data (which sometimes “overadjust”).

7.2 Estimation

Once you have defined a VAR model, you need to estimate it. If you use the *Time Series—VAR (Setup/Estimate)* wizard, this is done automatically. Otherwise, you will normally use the **ESTIMATE** instruction, although you can also use **LINREG** or **SUR** for this. All three methods are described below.

Using ESTIMATE

Once you have set up a system, use **ESTIMATE** to do the regression. This estimates all equations in the system independently using least-squares. Usually, the syntax is quite simple. The instruction

```
ESTIMATE   start   end
```

- estimates each of the equations over the range *start* to *end*.
- prints the output for each, following each with a table of *F*-tests for exclusion of lags of each of the variables.
- computes and saves the covariance matrix of residuals (required for variance decompositions) in the array %SIGMA.

You can omit *start* and *end* if you want to use the maximum range. **ESTIMATE** is a quick, efficient instruction which makes the fullest use of the structure of the VAR. However, because it does an equation by equation estimate of the full system, you cannot do any hypothesis tests using the regression-based instructions like **EXCLUDE** and **RESTRICT**. This is not a serious problem, because there are very few interesting hypotheses which can be expressed as restrictions on an individual equation.

Using LINREG

You can use a set of **LINREG** instructions if you really want to be able to focus on an individual equation. If you want to compute or print the variance-covariance matrix of the system, you will need to save the residuals and use the instruction **VCV** to compute and print the *VCV matrix*.

Using SUR

SUR is more often used for a “near-VAR” (where the equations don’t all have the same explanatory variables, Section 7.9). However, even in a full VAR, it can be used if you need to get a full sample covariance matrix and the standard $\Sigma \otimes (\mathbf{X}'\mathbf{X})^{-1}$ isn’t correct. For instance, the following skips lags 1 to $h-1$ in the VAR, which will create serially correlated residuals. **SUR** is used to estimate the model with “Newey-West” correction to the covariance matrix.

```
system(model=linvar)
variables ygap infl rate
lags h to h+nlags-1
det constant
end(system)
*
sur(model=linvar,robust,lwindow=newey,lags=h-1,noprint)
```

7.3 Data Analysis

Goals and Tools

As mentioned in the introduction to the chapter, we have two basic goals in using VAR's to analyze a set of series:

- To test formally theories which imply particular behavior for the vector autoregression.
- To learn more about the historical dynamics of the economy.

We are somewhat limited in what we can test formally using simple VAR's: most of the interesting hypotheses boil down to multivariate generalizations of the Granger-Sims causality tests (Section 3.7). Because the hypotheses will usually be joint restrictions across equations, a simple **EXCLUDE** instruction will not suffice. We use the special instruction **RATIO** which tests such cross-equation restrictions. Section 7.4 discusses testing procedures. The analysis of cointegrating relationships is covered in Section 7.8.

Mostly, VAR's are used to study the dynamics of the data. The coefficients in the estimated VAR are of little use themselves. Instead, we use a set of instructions which provide equivalent information in a more palatable form. The three key instructions are:

IMPULSE	computes impulse responses: if the system is shocked by x , y is the expected response.
ERRORS	computes the decomposition of variance. This decomposes the variance of the forecast errors in a series into the parts attributable to each of a set of innovation (shock) processes.
HISTORY	computes a historical decomposition. The historical data is decomposed into a trend (forecast) and the accumulated effects of the residuals.

All of these can be handled using the *Time Series—VAR (Forecast/Analyze)* wizard.

In addition, the instruction **CVMODEL** can be used to identify, estimate and test models for the contemporaneous relationship among the innovations in the VAR.

7.4 Hypothesis Tests

There are relatively few interesting hypotheses which you can test using just the estimates of a single equation from a VAR. Even the block F -tests produced by **ESTIMATE**, which indicate whether variable z helps to forecast variable x one-step ahead, are not, individually, especially important. z can, after all, still affect x through the other equations in the system.

Thus, most hypotheses will include more than one equation. The testing procedure to use is the Likelihood Ratio. The test statistic we recommend is

$$(2) \quad (T - c) \left(\log |\Sigma_r| - \log |\Sigma_u| \right)$$

where Σ_r and Σ_u are the restricted and unrestricted covariance matrices and T is the number of observations. Under certain conditions, this is asymptotically distributed as a χ^2 with degrees of freedom equal to the number of restrictions. c is a correction to improve small sample properties: Sims (1980, p.17) suggests using a correction equal to the number of variables in each unrestricted equation in the system. To help make this correction, **ESTIMATE** sets the variable %NREG equal to the number of regressors per equation, and %NREGSYSTEM to the number in the whole VAR. Note, by the way, that some hypotheses might have a non-standard distribution in the presence of unit roots (Sims, Stock and Watson, 1990). Their result won't affect the lag length test (VARLAG.RPF example), but *could* affect the exogeneity test (VARCAUSE.RPF).

You can compute (2) yourself using the variable %LOGDET defined by **ESTIMATE**, or you can use the special instruction **RATIO**. To use **RATIO**, you need to compute the two sets of series of residuals. To compute the statistic directly, you need to save the %LOGDET values into variables with different names after each of the **ESTIMATE** instructions. The examples use both techniques, but in practice, you only need to do one or the other.

The Examples

The examples in this section use the same data set. This is a five variable system comprising real GDP (GDPH), unemployment rate (LR), gross private fixed investment (IH), implicit price deflator (DGNP) and M2 (FM2). All source variables are seasonally adjusted. We transform all but the unemployment rate to logs. Both examples use the same set of initial instructions.

VARLAG.RPF Example

VARLAG.RPF formally tests one overall lag length vs another. To do this, we need to run two VAR's *over the same range*. Here this is done by fixing the start period of estimation to 1961:1 on the **ESTIMATE** instructions. The multiplier correction is the %NREG from the longer model ($5 \times 8 + 1 = 41$) and the degrees of freedom is $5 \times 5 \times 4 = 100$ (4 lags of each of 5 variables in 5 equations), which is here calculated using the difference between the values of %NREGSYSTEM. It also demonstrates use of the procedure

@VARLagSelect, which can be used to select the lag length selection automatically. **@VARLagSelect** uses a version of the AIC which corrects for degrees of freedom—when applied to a VAR, the standard AIC will often “select” a model which nearly exhausts the degrees of freedom.

This estimates the longer lag length VAR, saving the log determinant of the covariance matrix into **LOGDETU**, the number of regressors (per equation) into **MCORR**, the total number of regressors across the system into **NTOTAL** and the residuals into the **VECT[SERIES]** called **RESIDS8**.

```
system(model=usamodel)
variables loggdp unemp logi logp logm2
lags 1 to 8
det constant
end(system)
estimate(noprint,resids=resids8) 1961:1 *
compute logdetu=%logdet
compute mcorr=%nreg,ntotal=%nregsystem
```

And this estimates the *shorter* lag length. Note that this uses the same **MODEL** name, which is fine since we’ve already saved everything we needed out of the first set of estimates. The residuals need to go into a separate **VECT[SERIES]** (here called **RESIDS4**) and we also save the log determinant into **LOGDETR**.

```
system(model=usamodel)
variables loggdp unemp logi logp logm2
lags 1 to 4
det constant
end(system)
estimate(noprint,resids=resids4) 1961:1 *
compute logdetr=%logdet
```

The following does the likelihood ratio test for lag length using **RATIO** and the two set of residuals. The number of restrictions is the difference between the values of **%NREGSYSTEM** for the two models.

```
compute nrestr=ntotal-%nregsystem
ratio(degrees=nrestr,mcorr=mcorr,title="Test of 8 vs 4 Lags")
# resids8
# resids4
```

This does the numerically identical calculation using the summary statistics from two **ESTIMATE** instructions. Note that if you do the test this way, you don’t need to save residuals on the **ESTIMATES**.

```
cdf(title="Test of 8 vs 4 Lags") chisqr $
    (%nobs-mcorr)*(logdetr-logdetu) nrestr
```

Chapter 7: Vector Autoregressions

The following uses `@VARLagSelect` procedure, using (corrected) AIC with a maximum of eight lags.

```
@varlagselect(lags=8,crit=aic)
# loggdp unemp logi logp logm2
```

Note that the lag length test (top below) and the (corrected) AIC come to the completely opposite conclusions. The AIC chooses two and rather strongly prefers four lags over eight. As mentioned earlier, this is because of the large number of parameters covered by a single lag in a model this size. The information criteria will usually choose a relatively small model, while “general-to-specific”, while a common choice for univariate models, often chooses a rather large model (often the longest lag allowed).

```
Test of 8 vs 4 Lags
Chi-Squared(100)=      140.198047 with Significance Level 0.00497668

VAR Lag Selection
Lags AICC
 0  -3.569639
 1 -26.274506
 2 -28.568582*
 3 -28.539238
 4 -28.492796
 5 -28.392054
 6 -28.270686
 7 -28.085545
 8 -27.891979
```

VARCAUSE.RPF example

VARCAUSE.RPF does a block exogeneity test. This has as its null hypothesis that the lags of one set of variables do not enter the equations for the remaining variables. This is the multivariate generalization of Granger–Sims causality tests (Section 3.7), and is subject to the same problem of a non-standard distribution if there are unit roots. We test here that the real variables (LOGGDP, UNEMP and LOGI) as a block are exogenous, by comparing two systems, with and without the lags of LOGP and LOGM2.

This does the unrestricted model. The test can be done by looking only at the equations for the (assumed) exogenous block, so the lags of the tested variables are included as “deterministic” variables:

```
system(model=unrestricted)
variables loggdp unemp logi
lags 1 to 4
det constant logp{1 to 4} logm2{1 to 4}
end(system)
estimate(resids=unresids)
compute ntotal=%nregsystem,mcorr=%nreg,loglunr=%logl
compute logdetunr=%logdet
```

This does the restricted model, which is just a straight VAR on the assumed exogenous variables:

```
system(model=restricted)
variables loggdp unemp logi
lags 1 to 4
det constant
end(system)
estimate(resids=resresids)
compute loglres=%logl
compute logdetres=%logdet
```

This does the test with **RATIO** using the two sets of residuals:

```
ratio(degrees=ntotal-%nregsystem,mcorr=mcorr,$
      title="Exogeneity of Real Variables")
# unresids
# resresids
```

This is the numerically identical test using the log determinants of the covariance matrices:

```
compute teststat=(%nobs-mcorr)*(logdetres-logdetunr)
cdf(title="Exogeneity of Real Variables") chisqr $
      teststat ntotal-%nregsystem
```

and this does the same using the saved %LOGL statistics, with the standard test statistic rescaled by $(T-c)/T$ to implement the multiplier correction:

```
compute teststat=-2.0*(%nobs-mcorr)/%nobs*(loglres-loglunr)
cdf(title="Exogeneity of Real Variables") chisqr $
      teststat ntotal-%nregsystem
```

Again, you don't have to do all three of these, as they give identical answers. Just pick one.

7.5 Orthogonalization

Moving Average Representation

IMPULSE, **ERRORS** and **HISTORY** are all based upon the moving average representation of a vector time series:

$$(3) \quad \mathbf{y}_t = \hat{\mathbf{y}}_t + \sum_{s=0}^{\infty} \Psi_s \mathbf{u}_{t-s}$$

where

- $\mathbf{y}$ is an M -variate stochastic process
- $\hat{\mathbf{y}}_t$ is the deterministic part of $\mathbf{y}_t$
- $\{\mathbf{u}_t\}$ is an N -variate white noise process: if $t \neq s$, $\mathbf{u}_t$ and $\mathbf{u}_s$ are uncorrelated. Usually $N=M$, but if there are some linear identities connecting the $\mathbf{y}$ values, M can be greater than N . $\mathbf{u}$ is called an *innovation process* for $\mathbf{y}$.

What is Orthogonalization?

There are many equivalent representations for this model: for any non-singular matrix $\mathbf{G}$, Ψ_s can be replaced by $\Psi_s \mathbf{G}^{-1}$ and $\mathbf{u}$ by $\mathbf{G}\mathbf{u}$. A particular version is obtained by choosing some normalization.

If Ψ_0 is normalized to be the identity matrix, each component of $\mathbf{u}_t$ is the error that results from the one step forecast of the corresponding component of $\mathbf{y}_t$. These are the *non-orthogonal innovations* in the components of $\mathbf{y}$; non-orthogonal because, in general, the covariance matrix $\Sigma = E(\mathbf{u}_t \mathbf{u}_t')$ is not diagonal.

It is often more useful to look at the moving average representation with orthogonalized innovations. If we choose any matrix $\mathbf{G}$ so that

$$(4) \quad \mathbf{G}\Sigma\mathbf{G}' = \mathbf{I}$$

then the new innovations $\mathbf{v}_t = \mathbf{G}\mathbf{u}_t$ satisfy $E(\mathbf{v}_t \mathbf{v}_t') = \mathbf{I}$. These *orthogonalized innovations* have the convenient property that they are uncorrelated both across time and across equations. Such a matrix $\mathbf{G}$ can be gotten from inverting any solution $\mathbf{F}$ of the factorization problem $\mathbf{F}\mathbf{F}' = \Sigma$. There are many such factorizations of a positive definite Σ , among them:

- those based on the Cholesky factorization, where $\mathbf{G}$ is chosen to be lower triangular (next page),
- structural decompositions of the form suggested by Bernanke (1986) and Sims (1986).

Impact Responses

If we write $\mathbf{u}_t = \mathbf{F}\mathbf{v}_t$, then, when $\mathbf{v}_t$ is the i th unit vector, $\mathbf{u}_t$ is the i th column of $\mathbf{F}$. These are known as the *impact responses* of the components of $\mathbf{v}$, since they show the immediate impact to the variables $\mathbf{y}$ due to the shocks $\mathbf{v}$.

Why Orthogonalize?

Orthogonalized innovations have two principal advantages over non-orthogonal ones:

- Because they are uncorrelated, it is very simple to compute the variances of linear combinations of them.
- It can be rather misleading to examine a shock to a single variable in isolation when historically it has always moved together with several other variables. Orthogonalization takes this co-movement into account.

The greatest difficulty with orthogonalization is that there are many ways to accomplish it, so the choice of one particular method is not innocuous. The Bernanke-Sims style decompositions (Section 7.5.2) are designed to overcome some of the objections to the methodology by modeling the decomposition more carefully.

7.5.1 Cholesky Factorization

The standard orthogonalization method used by RATS is the Cholesky. Given a positive-definite symmetric matrix Σ , there is one and only one factorization into $\mathbf{F}\mathbf{F}'$ such that $\mathbf{F}$ is lower triangular with positive elements on the diagonal. This is the Cholesky factorization.

We can obtain several related decompositions by reordering rows and columns of Σ . For instance, if the Σ matrix is

$$\begin{bmatrix} 1.0 & 4.0 \\ 4.0 & 25.0 \end{bmatrix} \text{ its Cholesky factor is } \begin{bmatrix} 1.0 & 0.0 \\ 4.0 & 3.0 \end{bmatrix}$$

If we interchange variables 1 and 2 in the covariance matrix,

$$\begin{bmatrix} 25.0 & 4.0 \\ 4.0 & 1.0 \end{bmatrix} \text{ has a factor of } \begin{bmatrix} 5.0 & 0.0 \\ 0.8 & 0.6 \end{bmatrix}$$

If we switch the rows on the latter array, a new factor of the original Σ is obtained:

$$\begin{bmatrix} 0.8 & 0.6 \\ 5.0 & 0.0 \end{bmatrix}$$

We describe the first factorization of Σ as the decomposition in the order 1-2 and the second as the order 2-1.

The Cholesky factorization is closely related to least squares regression. If we are decomposing the covariance matrix of a set of variables, the i th diagonal element of the factor is the standard error of the residual from a regression of the i th variable on variables 1 to $i-1$.

There is a different factorization for every ordering of the variables, so it will be next to impossible to examine all of them for systems with more than three variables. Usually, you will decide upon the ordering based mostly upon a “semi-structural” inter-

pretation of the model: you might feel, for instance, that within a single time period, movements in y_1 precede movements in y_2 , so y_1 should precede y_2 in the ordering.

Note that when the residuals are close to being uncorrelated, the order of factorization makes little difference. With low correlation, very little of the variance in a variable can be explained by the other variables.

7.5.2 Structural Decompositions

The Cholesky factorizations described above suffer from the problem of imposing a “semi-structural” interpretation on a mechanical procedure. For instance, the “money” innovation is different (sometimes radically so) if you put money first in the ordering than if you put it last. There will rarely be a nice, neat, publicly acceptable way to order the VAR variables.

Bernanke (1986) and Sims (1986) independently proposed alternative ways of looking at the factorization problem which impose more of an economic structure. These have been dubbed “structural VARs” (SVAR for short) or “identified VARs.” The theory of these is developed more fully in Hamilton (1994) and Lütkepohl (2006), while Enders (2014) covers them from a practical standpoint.

First, note that in the VAR model

$$(5) \quad \mathbf{y}_t = \sum_{s=1}^p \Phi_s \mathbf{y}_{t-s} + \mathbf{u}_t \quad ; \quad E(\mathbf{u}_t \mathbf{u}_t') = \Sigma$$

the lag coefficients Φ_s can be estimated by single equation OLS regardless of any restrictions on the Σ matrix. Suppose now that we write down a model for the (non-orthogonal) innovation process $\mathbf{u}$, such as

$$\begin{aligned} u_{1t} &= v_{1t} \\ u_{2t} &= \gamma u_{1t} + v_{2t} \\ u_{3t} &= \delta u_{1t} + v_{3t} \end{aligned}$$

where we assume the $\mathbf{v}$'s are orthogonal. This puts a restriction on Σ : there are six free elements in Σ (in general, there are $N(N+1)/2$) and only five in this setup: γ , δ and the variances of the $\mathbf{v}$'s. We can obtain a related Cholesky factorization by adding either u_{2t} to the u_{3t} equation (order 1–2–3) or vice versa (order 1–3–2). The above model causes u_2 and u_3 to be related only through u_1 .

In general, if we write the innovation model as

$$(6) \quad \mathbf{A} \mathbf{u}_t = \mathbf{v}_t \quad ; \quad E(\mathbf{v}_t \mathbf{v}_t') = \mathbf{D} \quad ; \quad \mathbf{D} \text{ diagonal}$$

and assume Normal residuals, we need to maximize over the free parameters in $\mathbf{A}$ and $\mathbf{D}$ the likelihood-based function:

$$(7) \quad \frac{T}{2} \left\{ \log |\mathbf{A}|^2 - \log |\mathbf{D}| \right\} - \frac{T}{2} \text{trace}(\mathbf{D}^{-1} \mathbf{A} \mathbf{S} \mathbf{A}')$$

where $\mathbf{S}$ is the sample covariance matrix of residuals.

In RATS, structural VARs are estimated using the instruction **CVMODEL**. This actually accepts a broader class of models. The general form is

$$(8) \quad \mathbf{A}\mathbf{u}_t = \mathbf{B}\mathbf{v}_t ; E(\mathbf{v}_t\mathbf{v}_t') = \mathbf{D} ; \mathbf{D} \text{ diagonal}$$

Of course, $\mathbf{B}=\mathbf{I}$ gives the model from before. Typically, a model will use just one of $\mathbf{A}$ and $\mathbf{B}$. A “B” model would come from a view that you know what the orthogonal shocks are and are using the $\mathbf{B}$ matrix to tell which variables they hit. A model using both $\mathbf{A}$ and $\mathbf{B}$ would likely have just a few well-placed free coefficients in $\mathbf{B}$ to allow for correlations among residuals in structural equations. For instance, if you have two structural equations for $\mathbf{u}$, but are unwilling to restrict them to having uncorrelated residuals, a $\mathbf{B}$ matrix with a non-zero coefficient at the off-diagonal location linking the two will allow them to be correlated.

Identification

When you write them down, innovation models *look* a lot like standard simultaneous equations models. Unfortunately, there is no simple counting rule like the order condition to verify identification. If you have more than $N(N-1)/2$ free parameters in $\mathbf{A}$, your model is definitely *not* identified, but it may be unidentified even with fewer. For example, if we start with $\mathbf{D}$ being the identity and $\mathbf{A}$ matrix

$$\begin{bmatrix} 1.0 & 0.5 & 0.0 \\ 0.0 & 1.0 & 1.2 \\ -0.3 & 0.0 & 1.0 \end{bmatrix}, \text{ there is an alternate factorization with } \begin{bmatrix} 1.0 & 1.253 & 0.0 \\ 0.0 & 1.0 & 1.925 \\ -2.303 & 0.0 & 1.0 \end{bmatrix}.$$

The likelihood function will have a peak at each of these.

In most cases, you can rely on the methods in Rubio-Ramirez, Waggoner, and Zha (2010) to check identification. While this is a very technical paper designed to apply to a wide range of situations, it provides as special cases a couple of easy-to-verify rules:

- In an “A” model identified with zero restrictions, the SVAR is identified if and only if exactly one row has j zeros for each j from 0 to $m-1$. Equivalently, if and only if exactly one row has j non-zero elements for each j from 1 to m .
- In a “B” model identified with zero restrictions, the SVAR is identified if and only if exactly one column has j zeros for each j from 0 to $m-1$ with the analogous condition for non-zeros.

For example, the following pattern of zeros and non-zeros:

$$\begin{bmatrix} \bullet & 0 & 0 & \bullet \\ \bullet & \bullet & \bullet & 0 \\ \bullet & \bullet & 0 & \bullet \\ \bullet & \bullet & 0 & 0 \end{bmatrix}$$

Chapter 7: Vector Autoregressions

will not properly identify an “A” model (rows have 2-1-1-2 zeros, not 0-1-2-3 in some order), but will identify a “B” model (columns have 0-1-3-2). Note that identification for the A model fails even though there is nothing “obviously” wrong with it (like two rows being identical).

Referring to the earlier example, we can tell immediately using this method that the model isn’t globally identified, because it has two non-zero elements in each row, rather than 1, 2, and 3 non-zero elements in rows 1 through 3 respectively.

Estimation

Before running **CVMODEL**, you need to create a FRML which describes your **A** or **B** matrix. This must be declared as a `FRML[RECT]`, which is a formula which produces a rectangular matrix. Whatever free parameters you will have in this also need to be put into a parameter set using **NONLIN**. For instance, for the small model:

$$\begin{aligned}u_{1t} &= v_{1t} \\ u_{2t} &= \gamma u_{1t} + v_{2t} \\ u_{3t} &= \delta u_{1t} + v_{3t}\end{aligned}$$

the set up would be

```
nonlin gamma delta
dec frml[rect] afrml
frml afrml = ||1.0,0.0,0.0|-gamma,1.0,0.0|-delta,0.0,1.0||
```

We also need the covariance matrix of residuals, which will usually come from an **ESTIMATE** instruction on a VAR. Now the covariance matrix itself is a sufficient statistic for estimating the free coefficients of the model. However, in order to obtain standard errors for the coefficients, or to test overidentifying restrictions, we need to know the number of observations. RATS keeps track of the number of observations on the most recent estimation, so if you have just estimated the VAR, you won’t need to do anything about that. However, if the covariance matrix was estimated separately, you should use the option `OBS=number of observations` on **CVMODEL**.

Note, by the way, that it is good practice to analyze (that is, calculate and graph impulse responses, Section 7.6) with the simpler Cholesky factor model before you try to estimate your structural model. Structural VAR’s take the underlying lag model as given and model only the contemporaneous relationship. If there’s a problem with the lag model (for instance, explosive roots due to some type of data error), that will be much easier to see with the Cholesky factor model, since it’s fully constructive, and doesn’t depend upon non-linear estimation.

CVMODEL provides three choices for the (log) maximand that it uses in estimating the free coefficients in (8). All are based upon Normally distributed $\mathbf{v}$ ’s, but differ in their handling of the **D** matrix. They fall into two general forms:

$$(9) \quad \frac{T-c}{2} \left\{ \log |\mathbf{A}|^2 - \log |\mathbf{B}|^2 \right\} - \left(\frac{T-c}{2} + \delta + 1 \right) \sum_i \log \left(\mathbf{B}^{-1} \mathbf{A} \mathbf{S} \mathbf{A}' \mathbf{B}'^{-1} \right)_{ii}$$

$$(10) \quad \frac{T-c}{2} \left\{ \log |\mathbf{A}|^2 - \log |\mathbf{B}|^2 \right\} - \frac{T}{2} \sum_i \left(\mathbf{B}^{-1} \mathbf{A} \mathbf{S} \mathbf{A}' \mathbf{B}'^{-1} \right)_{ii}$$

With $c=0$ and $\delta=-1$ in (9), you have the concentrated likelihood. This form is selected by using `DMATRIX=CONCENTRATE`, which is the default. Other values of δ have $\mathbf{D}$ integrated out. This is `DMATRIX=MARGINALIZED`, combined with `PDF=value of  $\delta$` . This uses a prior of the form $|\mathbf{D}|^{-\delta}$ (`PDF` stands for Prior Degrees of Freedom). With $c=0$ in (10), you have the likelihood with $\mathbf{D}=\mathbf{I}$. This is selected with `DMATRIX=IDENTITY`. This requires a different parameterization of the basic factoring model than the other two. The concentrated and marginalized forms both assume that the parameterization of the $\mathbf{A}$ and $\mathbf{B}$ matrices includes a normalization, generally by putting 1's on the diagonal. With `DMATRIX=IDENTITY`, the normalization is chosen by making the $\mathbf{v}$'s have unit variance, so the diagonal in $\mathbf{A}$ or $\mathbf{B}$ has to be freely estimated. This was the choice in Sims and Zha (1999), as they had some concern that the normalization of one of their equations was not innocuous. The maximum likelihood estimates aren't affected by the choice of normalization, but the Monte Carlo integration process (example `MONTESVAR.RPF`) is.

In all of these, c is for correcting the degrees of freedom if you're examining the posterior density. You provide the value of c using the option `DFC=value of  $c$` .

For estimation methods, `CVMODEL` offers `BFGS` (Section 4.2) and the derivative-free methods (Section 4.3). `BFGS` is the only one of these that can estimate standard errors. However, `BFGS`, as a hill-climbing method, depends crucially on whether you are starting on the right "hill." If you want to play safe, start out with the `GENETIC` method, which explores more broadly (but, unfortunately, also more slowly). You can use the `PMETHOD` and `PITERS` options to choose a preliminary estimation method before switching to `BFGS`.

You can also use `CVMODEL (METHOD=EVAL)` to evaluate the log likelihood at the initial guess values. Note that while the integrating constants aren't included in (9) and (10), they *are* included in the log likelihood that `CVMODEL` produces.

The following code estimates the original SVAR. `SIGMA` is the (previously estimated) covariance matrix of residuals:

```
compute gamma=0.0,delta=0.0
cvmodel(method=bfgs,a=afml) sigma
```

Chapter 7: Vector Autoregressions

This is the output produced:

```
Covariance Model-Concentrated Likelihood - Estimation by BFGS
Convergence in      9 Iterations. Final criterion was  0.0000010 <=  0.0000100
Observations                        250
Log Likelihood                    -1594.3951
Log Likelihood Unrestricted        -1589.7001
Chi-Squared(1)                     9.3901
Significance Level                  0.0021816

Variable                        Coeff      Std Error    T-Stat      Signif
*****
1.  GAMMA                        0.4336094435  0.0647412945    6.69757    0.00000000
2.  DELTA                        0.6020698430  0.0605333608    9.94608    0.00000000
```

In the header, it displays the log likelihood of the estimated model, and the log likelihood of an unrestricted model. For a just-identified model, those should be equal; if they aren't you got stuck on a ridge and need to do some extra work with the non-linear estimation options to find the global optimum. If the model is overidentified (which this one is), it produces a likelihood ratio test for the overidentifying restrictions. In this case, the restriction is rejected fairly soundly.

The log likelihood is accessible for further analysis as %FUNCVAL or %LOGL, while the test statistic and significance level are %CDSTAT and %SIGNIF. To get the factor of the Σ that you will need for impulse responses, use the FACTOR option on **CVMODEL**. You can use the DVECTOR option to get the variances of the orthogonal components. Note that if the model is over-identified, the factor matrix generated by **CVMODEL** won't exactly reproduce the covariance matrix if you take its outer product — if you try it, you'll typically see some values matching exactly, while some don't, as the typical overidentified model generally includes some blocks of variables for which the model *is* just-identified.

Directly Modeling the Covariance Matrix

The A and B forms are designed (in effect) to model *a factor of* the covariance matrix. Sometimes, however, we have a model which directly gives the covariance matrix itself. If you have this, you can provide a V formula (a FRML[SYMMETRIC]) which models Σ . The likelihood used is (omitting the constants):

$$(11) \quad \frac{T-c}{2} \left\{ \log |\mathbf{V}^{-1}| \right\} - \frac{T}{2} \text{trace}(\mathbf{S}\mathbf{V}^{-1})$$

For instance, a simple “factor” model would model the covariance matrix as $\mathbf{\Lambda}\mathbf{\Lambda}' + \mathbf{D}$ where $\mathbf{\Lambda}$ are the loadings on the orthogonal factors and $\mathbf{D}$ is a diagonal matrix. With $\mathbf{\Lambda}$ constructed by a vertical concatenation of LLEAD and LREM, and $\mathbf{D}$ being a VECTOR, the following would estimate the free parameters in a fit to the covariance matrix R.

```
cvmodel (v=%outerxx(llead~lrem)+%diag(d.*d) ,obs=%nobs) r
```

7.5.3 Blanchard-Quah/Long-Run Restrictions

Technically, the decomposition proposed by Blanchard and Quah (1989) is a special case of a structural VAR model. However, it reflects a completely different approach to the task of interpreting a VAR.

Blanchard and Quah (BQ) start with (assumed to be) orthogonal shocks. In a two variable model, one is assumed to represent a “supply” shock and one a “demand” shock. It is expected that both shocks will affect both variables, meaning that we must find a restriction somewhere else in order to identify the shocks. The BQ assumption is that the “demand” shock will have no permanent effect on one of the two variables.

Now the question arises: what does it mean for a shock to have only a temporary effect on a variable? This has to be defined in a way that produces a restriction that can be imposed upon a factorization. BQ accomplish this by assuming that the target variable is an integrated variable which is put into the VAR in differenced form. Translating the moving average representation on the difference to the corresponding levels requires merely taking running sums of the MAR coefficients. That is, if

$$(12) \quad (1 - L)\mathbf{y}_t = \sum_{s=0}^{\infty} \Psi_s \mathbf{v}_{t-s}, \text{ then}$$

$$(13) \quad \mathbf{y}_t = \sum_{s=0}^{\infty} \Psi_s^* \mathbf{v}_{t-s}, \text{ where } \Psi_s^* = \sum_{l=0}^s \Psi_l$$

The restriction that a component of $\mathbf{v}$ has no long-run effect on a component of $\mathbf{y}$ can now be written

$$(14) \quad \sum_{l=0}^{\infty} \Psi_l(i, j) = 0$$

This is still not in a form where it can be imposed as a restriction on the factorization. However, if the VAR has been transformed so as to have an invertible auto-regressive representation (in effect, all the variables in it are now stationary), then the infinite sums of the moving average representation can be obtained by inverting the autoregressive representation. If we write the desired model in the inverted form

$$(15) \quad \mathbf{y}_t = \Phi(L)^{-1} \mathbf{B} \mathbf{v}_t, \text{ with } \mathbf{B} \mathbf{B}' = \Sigma, E \mathbf{v}_t \mathbf{v}_t' = \mathbf{I}, \text{ and } \Phi(L) = 1 - \sum_{s=1}^l \Phi_s L^s$$

then the matrix of sums of MA coefficients (the long-run response matrix) is

$$(16) \quad \Phi(1)^{-1} \mathbf{B}$$

$\Phi(1)$ can be obtained after an **ESTIMATE** instruction as %VARLAGSUMS, and more generally can be computed using %MODELLAGSUMS(model).

The BQ restriction on $\mathbf{B}$ (in a two-variable system) is that one of the four elements of this be zero. For simplicity, assume that this is the (1,2) element. (To make it the 1 row, you need to order the variables correctly when setting up the VAR). Now

$\Phi(1)^{-1} \mathbf{B}$ is a factor of $\Phi(1)^{-1} \Sigma \Phi(1)^{-1'}$ and the restriction can be imposed by making it the Cholesky factor of that matrix. The specialized function %BQFACTOR does the calculation of the required factor $\mathbf{B}$.

```
compute    bqfactor=%bqfactor(%sigma,%varlagsums)
```

This gives a factor which has the proper behavior, but it may be necessary to correct the signs of the impact responses. If a shock has a zero long-run impact, then so does the same shock with the opposite sign. We need one sign convention for each column. @ImpactSignFlip is designed to do this—the following will adjust BQFACTOR to make the impacts of each shock positive for GDP (variable 1 in this model):

```
@ImpactSignFlip bqfactor ||+1,+1||
```

Generalizing to Larger Systems

This generalizes to N variables by choosing a restriction that the long-run response matrix is lower triangular; that's what you'll get if you apply %BQFACTOR to a larger system. That, however, is unlikely to be a very interesting structure. More commonly, you'll have a combination of restrictions on the long-run matrix and restrictions on $\mathbf{B}$ (the impact responses) itself. This gives you a combination of short (actually contemporaneous) and long run restrictions.

Unlike the models which are entirely contemporaneous, models with long-run restrictions cannot easily be handled if they're over-identified. Because the maximum likelihood estimates of the lag coefficients are just the OLS estimates regardless of the value of Σ , they can be concentrated out, allowing us to focus only on modelling the contemporaneous relationship. For a just-identified model of Σ , long-run restrictions don't restrict the lag coefficients (even though they're a function of them), since we have enough freedom in the $\mathbf{B}$ matrix to make the restrictions work while achieving the overall maximum for the likelihood. (All just-identified structural VAR's have the same likelihood). If the model is *over-identified*, however, it's almost certain that you can achieve a higher likelihood by adjusting the lag coefficients. While this *could* be done using NLSYSTEM, it involves estimating all the coefficients subject to rather nasty non-linear constraints.

If (as is typical) the restrictions in the model are done entirely by zero restrictions on elements of $\mathbf{B}$ and elements of $\Phi(1)^{-1} \mathbf{B}$, then these can be combined into the form

$$(17) \quad \mathbf{R} \text{vec}(\mathbf{B}) = 0$$

where $\mathbf{R}$ is a $N(N-1)/2 \times N^2$ matrix. This can be inverted to create the unrestricted parameters Θ by

$$(18) \quad \text{vec}(\mathbf{B}) = \mathbf{R}^\perp \Theta$$

where $\mathbf{R}^\perp$ is an orthogonal complement of $\mathbf{R}$ (computed with the %PERP function). The procedure @ShortAndLong can be used to get the factor matrix $\mathbf{B}$. Input to this are the $\Phi(1)^{-1}$, Σ , and pattern matrices for the long and short run restrictions. The

pattern matrices have non-zero elements in the slots which aren't restricted, and zeros in the slots which are restricted to be zero. Each row represents a variable and each column represents a shock. So a 0 in the 1,2 slot of the long-run pattern means that shock 2 is restricted to have no long-run effect on variable 1.

The following is an example of the use of **@ShortAndLong**. While you could use any non-zero value to represent the non-zero slots, we've found that using the . (missing value) makes this easier to read and understand.

```
dec rect lr(4,4)
dec rect sr(4,4)
input sr
. 0 0 .
. 0 . .
. . . .
. . . .
input lr
. 0 0 0
. . . .
. . . .
. . . .
@shortandlong(sr=sr,lr=lr,masums=inv(%varlagsums),factor=b) %sigma
```

The (column) counting rule described on page UG–219 applies here; you just have to count 0's across the two matrices. This is just-identified because the columns have 0, 3, 2 and 1 zeros.

While **@ShortAndLong** can't *estimate* a model unless it's just identified, you can input into it an *underidentifying* set of restrictions so it can compute the representation (18) in terms of a reduced number of parameters. To do this, add the options **NOESTIMATE** and **RPERP=R** *Perp Matrix*. You can then use this with **CVMODEL** to estimate with additional types of restrictions. This, for instance (which is included in the Gali QJE 1992 replication programs) has two short-run, three long-run and one additional restriction that variable 3 not enter shock 3.

```
input sr
. 0 0 .
. . . .
. . . .
. . . .
input lr
. 0 0 0
. . . .
. . . .
. . . .
@shortandlong(sr=sr,lr=lr,masums=inv(%varlagsums),factor=b) %sigma
compute bv=%vec(%bqfactor(%sigma,%varlagsums))
compute [vect] theta=%ginv(rperp)*bv
dec frml[rect] bf af
frml bf = %vectorect(rperp*theta,4)
```

```
frml af = inv(bf(0))
nonlin(parmset=base) theta
nonlin(parmset=r8) af(0) (3,3)==0.0
cvmodel (parmset=base+r8,b=bf,itors=400) %sigma
```

As with the simpler BQ model, models with short- and long-run get the “shape” right, but don’t necessarily give the expected sign, so you might need to change the signs on some of the columns of the factor to give the impacts the correct interpretation.

7.5.4 Isolating a Single Shock

All of the examples seen so far have produced a complete factorization of the covariance matrix. An alternative approach is to isolate one shock with certain characteristics. This might be the requirement that a shock produce certain responses, or that a shock is a particular linear combination of the non-orthogonal shocks. The Blanchard-Quah factorization is actually a form of this: one shock (the demand shock) has the property that it has a zero long-run response, the other is just whatever shock is required to complete a factorization.

You can construct a factorization around any single (non-zero) shock—some rescaling of it will always be part of a factorization. Why do we need a factorization if we’re just interested in the one shock? This is mainly because the decomposition of variance (Section 7.6) can only be computed if you have a full factorization. In general, there will be many ways to complete the factorization, but the fraction of the variance explained by the shock of interest will be the same for all of them.

Using @ForcedFactor

The procedure **@ForcedFactor** computes a factorization which includes a (scale of a) specified column in the factorization (which forces an orthogonalized component to hit the variables in a specific pattern), or which includes a (scale of a) specified row in the inverse of the factorization (which forces an orthogonalized component to be formed from a particular linear combination of innovations). For example, in a four variable system where the first two variables are interest rates:

```
@ForcedFactor sigma ||1.0,-1.0,0.0,0.0|| f1
@ForcedFactor(force=column) sigma ||1.0,1.0,0.0,0.0|| f2
```

F1 will be a factorization where the first orthogonal component is the innovation in the difference between the rates. F2 will be a factorization where the first orthogonal component loads equally onto the two interest rates, and hits none of the other variables contemporaneously.

Another example is from King, Plosser, Stock and Watson (1991). They need a factorization which hits the three variables in the system equally in the long run. By using %VARLAGSUMS (described in Section 7.5.3) this can be done very simply with:

```
compute [rect] a=||1.0|1.0|1.0||
compute x=%varlagsums*a
```

```
@forcedfactor(force=column) %sigma x factor
```

@ForcedFactor will also allow you to control more than one column in the factorization, but the columns other than the first will be linear combinations of itself and the columns to its left. (That is, you can control the space spanned by some set of columns, but not the columns themselves).

7.5.5 Sign Restrictions

Parametric SVAR's have, too frequently, been unable to produce models where shocks have the desired properties, and the types of zero restrictions which allow isolated shocks to be identified as described in Section 7.5.4 aren't always reasonable. Uhlig (2005) and Canova and De Nicolo (2002) proposed an even less structured approach, in which the shock is identified by sign restrictions, satisfying the prior understanding of how a particular shock should behave.

Because it is likely that there are many shocks which satisfy a set of sign restrictions, Uhlig's approach uses a randomization procedure to explore the space of the possible shocks, which requires techniques described in Chapter 16. The basic idea behind it is if you take any factorization $\mathbf{FF}' = \Sigma$, then *any* column vector that is part of any factorization of Σ (Uhlig calls these *impulse vectors*) can be written in the form $\mathbf{F}\alpha$ where $\|\alpha\| = 1$. Thus, the space of single shocks in an SVAR model can be explored by looking the unit sphere in the proper space.

7.5.6 Generalized Impulse Responses

Generalized impulse responses (GIR) were proposed by Pesaran and Shin (1998) as an attempt to avoid the difficulties of identifying orthogonal shocks in VAR models. This is not a particularly complicated idea—in fact, all these do is compute the shocks with each variable in turn being first in a Cholesky order. These can be done quite easily using RATS: `%GIRFMATRIX(%SIGMA)` will compute the “factor” that will give you GIR when you use it to compute impulse responses.

While these can, quite easily, be done in RATS, we do not recommend them unless you understand exactly what you're doing. While coming up with an orthogonalizing model can, in practice, be somewhat difficult, it is a necessary step. A set of responses to highly correlated shocks are almost impossible to interpret sensibly. For instance, you can't run an **ERRORS** instruction to assess the economic significance of the responses, since that requires an orthogonal decomposition, and this *isn't* one.

7.5.7 Examples

The interpretations of the results from the instructions **ERRORS**, **IMPULSE** and **HISTORY** all depend crucially on the factorization chosen for the covariance matrix of residuals.

The CV option

You can provide the instruction with the **SYMMETRIC** array Σ using the **CV** option that is available on each of these instructions. That causes RATS to do a Cholesky decomposition in the order that the equations were listed. To get an alternate Cholesky ordering, use the function **%PSDFACTOR** as shown below.

The FACTOR option

In general, we need a matrix **F** satisfying $\mathbf{FF}' = \Sigma$. With this computed, you use the option **FACTOR=F** on the instruction.

For a Structural VAR

You get **F** using the **FACTOR** option on **CVMODEL**.

```
nonlin gamma delta
dec frml[rect] afrml
frml afrml = ||1.0,0.0,0.0|-gamma,1.0,0.0|-delta,0.0,1.0||
compute gamma=0.0,delta=0.0
cvmmodel(method=bfgs,factor=sfactor,a=afrml) %sigma
errors(model=model3,factor=sfactor,steps=24)
```

For a Blanchard-Quah Decomposition

You get **F** by direct calculation as described in Section 7.5.3. This relabels the shocks to match their interpretations.

```
estimate
compute bqfactor=%bqfactor(%sigma,%varlagsums)
compute bqfactor=%dmult(bqfactor,$
    ||%sign(bqfactor(1,1)),%sign(bqfactor(1,2))||)
impulse(model=rcmodel,factor=bqfactor,steps=80,$
    results=responses,labels=||"Permanent","Transitory"||)
```

For an Alternative Cholesky Ordering

The **%PSDFACTOR(sigma,order)** function computes **F**. This takes as its arguments the covariance matrix and a vector of integers describing the ordering.

```
errors(model=model4,cv=v,steps=24)
compute dec3241=%psdfactor(vsigma,||3,2,4,1||)
errors(model=model4,factor=dec3241,steps=24)
compute dec4321=%psdfactor(vsigma,||4,3,2,1||)
errors(model=model4,factor=dec4321,steps=24)
```

does orderings 1–2–3–4 (by default), 3–2–4–1 and 4–3–2–1.

7.5.8 Structural Residuals

The structural residuals are the observed values of $\mathbf{v}_t$. If the model produces a factor $\mathbf{F}$ of the covariance matrix, then $\mathbf{F}\mathbf{v}_t = \mathbf{u}_t$ or $\mathbf{v}_t = \mathbf{F}^{-1}\mathbf{u}_t$. In general, each component of $\mathbf{v}_t$ will be a linear combination of the full corresponding vector of VAR residuals $\mathbf{u}_t$. The easiest way to generate this is with the procedure `@StructResids`. This takes as input the factor matrix and the `VECT[SERIES]` of VAR residuals, and produces a `VECT[SERIES]` of the structural residuals

The syntax is:

```
@StructResids ( option ) u start end v
```

Parameters

<i>u</i>	<code>VECT[SERIES]</code> of VAR residuals (input)
<i>start end</i>	range to convert
<i>v</i>	<code>VECT[SERIES]</code> of structural innovations (output)

Option

factor=*factor of covariance matrix*

The factor matrix (decomposition) for which you want to compute the structural residuals. This doesn't have to be a full $N \times N$ matrix. If you have defined only the $N \times P$ loadings from a subset of innovations onto the variables, the procedure will produce a reduced size `VECTOR[SERIES]` of structural innovations.

Examples

To compute the structural residuals for a Blanchard–Quah factorization, do:

```
estimate(resids=resids)
compute bqfactor=%bqfactor(%sigma,%varlagsums)
@impactsignflip ||+1,+1||
@structresids(factor=bqfactor) resids $
    %regstart() %regend() sresids
```

For a Cholesky factorization, it would be

```
estimate(resids=resids)
@structresids(factor=%decomp(%sigma)) resids $
    %regstart() %regend() sresids
```

For a more general SVAR, you need to compute a factor of `%SIGMA` (the covariance matrix saved automatically by **ESTIMATE**) and use this factor as the argument for the **FACTOR** option. For example, if you use **CVMODEL** to compute a factor matrix `SFACTOR`, you would do something like:

```
@structresids(factor=sfactor) resids $
    %regstart() %regend() sresids
```

7.6 Using IMPULSE and ERRORS

IMPULSE computes the responses of the system to particular initial shocks. It has a large array of options for specifying the shocks, but most of these (such as **PATHS** and **MATRIX**) are used more naturally with **FORECAST**. The VAR methodology primarily works just with first period shocks.

ERRORS uses the same information as **IMPULSE**, but produces its output in a different form. It decomposes the forecast error variance into the part due to each of the innovation processes. While **IMPULSE** will accept any form of innovation process, **ERRORS** requires orthogonalization, as the decomposition is meaningless without it.

You should always use **IMPULSE** and **ERRORS** together. For instance, you may note an interesting and unexpected response using **IMPULSE**. Before you get too excited, examine the variance decomposition with **ERRORS**—you may find that it has a trivial effect.

As noted earlier, you can use the *Time Series—VAR (Forecast/Analyze)* wizard to analyze both the impulse responses and the variance decomposition.

Technical Information

If we look at the moving average representation of a vector time series:

$$(19) \quad \mathbf{y}_t = \hat{\mathbf{y}}_t + \sum_{s=0}^{\infty} \Psi_s \mathbf{u}_{t-s} = \hat{\mathbf{y}}_t + \sum_{s=0}^{\infty} \Psi_s \mathbf{F} \mathbf{v}_{t-s}, \quad E \mathbf{u}_t \mathbf{u}_t' = \Sigma \quad E \mathbf{v}_t \mathbf{v}_t' = \mathbf{I}$$

IMPULSE computes the Ψ_s or the $\Psi_s^* \equiv \Psi_s \mathbf{F}$ (for orthogonalized innovations). These are organized as N^2 series, although you *can* also do shocks to one innovation at a time.

The error in the H -step ahead forecast is:

$$(20) \quad \sum_{h=0}^{H-1} \Psi_h^* \mathbf{F} \mathbf{v}_{t-h}$$

As $\mathbf{v}$ has been assumed to be uncorrelated both across time and contemporaneously, the covariance matrix of the K -step forecast is

$$(21) \quad \sum_{s=0}^{K-1} \Psi_s^* \Psi_s^{*'} = \sum_{s=0}^{K-1} \Psi_s \mathbf{F} \mathbf{F}' \Psi_s'$$

This doesn't depend upon $\mathbf{F}$, as long as $\mathbf{F} \mathbf{F}' = \Sigma$. We can isolate the effect of a single component of $\mathbf{v}$ by rewriting the sum as

$$(22) \quad \sum_{s=0}^{K-1} \sum_{i=1}^N \Psi_s^* \mathbf{e}(i) \mathbf{e}(i)' \Psi_s^{*'} = \sum_{i=1}^N \sum_{s=0}^{K-1} \Psi_s^* \mathbf{e}(i) \mathbf{e}(i)' \Psi_s^{*'}$$

where $\mathbf{e}(i)$ is the i th unit vector. This decomposes the variance-covariance matrix of forecast errors into N terms, each of which shows the contribution of a component of $\mathbf{v}$ over the K periods.

Graphing Impulse Responses

The moving average representation (MAR) of a model is simply the complete set of impulse responses. These are most usefully presented graphically. There are three logical ways to organize graphs of the MAR.

- A single graph can have responses of all variables to a shock in one variable. If the variables are measured in different units, it is advisable to standardize the responses: if you divide the response of a variable by the standard deviation of its residual variance, all responses are in fractions of standard deviations. *When the variables are in comparable units and a comparison of actual values is important, it is better to graph unscaled responses.* For example, you would want to compare interest rates and the rate of inflation without scaling.
- A single graph can have responses of one variable to shocks in all the variables. There is no problem of scale in this case.
- You can use a matrix of small graphs, each with only a single response. This looks nicer, but is more difficult to set up, as you must be sure that all graphs showing the responses of a single variable use the same MAXIMUM and MINIMUM values. Otherwise, very small responses will be spread across the entire height of a graph box and look quite imposing.

Where possible, use the procedure **@VARIRF** to do the graphs. It takes an already estimated VAR, computes the impulse responses, and organizes the graphs in one of several formats. We use this twice in **IMPULSES.RPF**, once with each page having responses of all variables to a single shock (the **PAGE=BYSHOCKS** option), and once with each page having the responses of each variable to all shocks (**PAGE=BYVARIABLES**).

```
@VARIRF (model=canmodel , steps=nsteps , $  
         varlabels=implabel , page=byshocks)  
@VARIRF (model=canmodel , steps=nsteps , $  
         varlabels=implabel , page=byvariables)
```

Confidence Bands

Point estimates alone of impulse responses may give a misleading impression. You might note a response whose sign is unexpected. Is this truly interesting, or is it just a statistical fluke? This can be answered, in part, by examining the corresponding error decomposition. If the innovation you're examining in fact explains a trivial amount of the target variable, then it isn't really meaningful.

But many responses can't quite so easily be dismissed as uninteresting. **IMPULSE** produces a moving average representation from the point estimates of a VAR. Since the coefficients of the VAR aren't known with certainty, neither are the responses. There are three principal methods proposed for computing confidence bands or standard errors for impulse responses:

1. Monte Carlo integration
2. Delta method (linearization)
3. Bootstrapping

Chapter 7: Vector Autoregressions

The delta method is sometimes given the appealing description of “analytical”, but is based upon a linearization which becomes increasingly inaccurate as the number of steps grows and the response functions become increasingly non-linear. We have a very strong preference for Monte Carlo integration, as is done with the `@MONTEVAR` or the `@MCMCDODRAWS` procedures. See Sims and Zha (1999) for a discussion of these issues. The replication programs for the Sims and Zha paper include both Monte Carlo integration and bootstrapping.

Interpreting the Decomposition of Variance

Decomposition of Variance for Series CANRGDPS							
Step	Std Error	USARGDPS	CANUSXSR	CANCD90D	CANM1S	CANRGDPS	CANCPINF
1	0.004688318	13.062	2.172	2.321	0.604	81.842	0.000
2	0.007407510	21.727	2.495	1.291	0.943	73.505	0.040
3	0.008882190	19.386	4.086	0.977	8.445	67.062	0.044
4	0.010004321	15.284	4.194	3.754	12.355	64.256	0.156
5	0.010632775	14.403	4.704	6.786	12.090	61.583	0.434
6	0.011511805	17.409	5.516	13.026	10.328	53.250	0.472
7	0.013088464	21.646	5.594	21.954	8.939	41.435	0.432
8	0.014910398	23.338	5.798	29.436	8.936	31.964	0.529
9	0.016733054	23.104	6.219	35.950	8.549	25.436	0.742
10	0.018526192	22.059	6.274	41.696	8.106	20.916	0.949

This is part of one of the tables produced by an `ERRORS` instruction for a six-variable VAR. There will be one such table for each endogenous variable.

The first column in the output is the standard error of forecast for this variable in the model. Since the computation assumes the coefficients are known, it is *lower* than the true uncertainty when the model has estimated coefficients. The remaining columns provide the decomposition. In each row, they add up to 100%. For instance, in the sample above, 81.84% of the variance of the one-step forecast error is due to the innovation in CANRGDPS itself. However, the more interesting information is at the longer steps, where the interactions among the variables start to become felt. We have truncated this table to 10 lags to keep its size manageable, but ordinarily you should examine at least four years’ worth of steps.

The above table suggests the following:

- The three principal factors driving CANRGDPS are itself, USARGDPS and CANCD90D, which is a short interest rate.
- The importance of USARGDPS is fairly constant across the range. Because this variable was first in the ordering, it isn’t clear (from examining just this one ordering) whether this is an artifact of the ordering only.
- Innovations in CANCD90D take almost six periods to have an effect but quickly become the prime mover.
- The other three variables (CANM1S, CANUSXSR and CANCPINF) have negligible explanatory power for CANRGDPS.

If you want more information on *how* USARGDPS and CANCD90D affect CANRGDPS, you need to look at the impulse response functions.

Choosing Orderings

If you work with Cholesky factorizations, the orderings that you should examine depend upon the set of questions you want to answer, and upon the structure of the covariance matrix of residuals.

- Variables that you don't expect to have any predictive value for other variables should be put last: for instance, local variables in a model with national variables.
- By definition, the first variable in the ordering explains all of its one-step variance.
- The one-step variance will be nearly 100% due to own innovations if there is little correlation between the residuals of a variable and the residuals of variables that appear before it in the ordering.
- When there is substantial correlation among innovations in variables, the decomposition of one-step variance depends *strongly* on the order of factorization.

To determine whether a variable behaves exogenously, put the variable first in the ordering. The variance in an exogenous variable will be explained primarily by own innovations. The meaning of “primarily” depends upon the number of variables in the system—50% is quite high in a six variable system. Remember that if the covariance matrix is nearly diagonal, the decomposition of variance will be fairly robust to changes of order.

Focusing on Two Variables

When there is high correlation between innovations in two variables, run a pair of decompositions with the two variables placed next to each other, changing only the positions of those two variables from one ordering to the next. Since the combined explanatory power of the two variables is independent of which one comes first, how the variance is split between them can be examined.

Usually, most of the variance will be attributed to whichever variable comes first. If this is true for both orderings, we can draw no conclusions. If one variable does much better when second than the other, you have evidence that variable is the causative factor, and the other moves with it closely. If most of the power is attributed to whichever variable is second, some linear combination of the two variables, perhaps the sum or difference, is the truly important factor.

In the example shown for the decomposition of variance, there is the question of whether or not the influence of USARGDPS is mainly due to its placement in the ordering before CANRGDPS. That can be answered by switching the ordering around to put CANRGDPS earlier. This reordering pretty much wipes out the fraction of variance due to USARGDPS.

IMPULSES.RPF Example

`IMPULSES.RPF` computes and graphs impulse response functions (to Cholesky shocks) in several different formats. Two are done using `@VARIRF`, which does one page per shock (with six panes) and one page per variable (again with six panes). Then, direct programming is used to do single graphs with responses of all six variables to each shock, and single graphs with responses for each variable to all shocks. It also uses `@MONTEVAR` to do a basic set of confidence bands. If you're interested in a more involved setup for standard error bands or some other type of confidence limit on the estimates, see the examples in Chapter 16.

Five of the series are in logs, while interest rates are kept in levels. The logged series are all scaled by 100, which gives the responses a more natural scale—a response of 3 would be interpreted as 3% increase over the baseline due to the shock and -2 would be a 2% decrease. For the interest rate, which is in percent per annum, a response of 1 means a 1 percentage point increase (for instance, 5% to 6%).

Very little of `IMPULSES.RPF` needs to be changed to handle a different data set. Once the data are in and transformed, you need to set the number of lags and the number of steps of responses. Here, this is the recommended year's worth plus one on the lags, and six years' worth of impulse response steps:

```
compute nlags = 5
compute nsteps = 24
```

Then set up the VAR using the `SYSTEM` commands:

```
system(model=canmodel)
variables logusagdp logexrate can3mthpcp $
          logcanm1 logcangdp logcandefl
lags 1 to nlags
det constant
end(system)
```

To help create publication-quality graphs, this sets longer labels for the series. These are used both in graph headers and key labels.

```
compute implabel=|| $
  "US Real GDP", $
  "Exchange Rate", $
  "Short Rate", $
  "M1", $
  "Canada Real GDP", $
  "CPI" ||
```

Everything else (except the very last calculation) is independent of the data set. The system is estimated, and the `@VARIRF` procedure is used to do two standard sets of graphs: the first will generate six graph pages, with each showing the responses to a particular shock for the six variables, with each in a different “pane”. The second `@VARIRF` generates six graph pages each of which has the responses of one variable

to each of the six shocks, all in separate panes. You could also do **@VARIRF** with the option **PAGE=ALL** to get a single page with a six by six layout of all the responses to all the shocks, though with a six variable system, the panes in the graph are often too small to be readable.

```
estimate(noprint)
compute neqn=%nvar
@VARIRF(model=canmodel, steps=nsteps, $
    varlabels=implabel, page=byshocks)
@VARIRF(model=canmodel, steps=nsteps, $
    varlabels=implabel, page=byvariables)
```

@MONTEVAR does a quick-and-dirty set of the confidence bands—it does 16%-84% lower and upper bounds, which are “robust” versions of roughly one-standard error bands. This *does* generate a six by six table of graphs (like **@VARIRF** with **PAGE=ALL**), though those are a bit more useful with the confidence bands, since you can tell at a glance which ones are small enough (relative to their bounds) to be ignored.

```
@montevar(draws=4000, model=canmodel, $
    shocks=implabel, varlabels=implabel)
```

The remainder computes and generates specialized graphs of the impulse responses. The first step is to compute the full block of IRF's. These go into the **RECT[SERIES]** called **IMPBLK**. **IMPBLK(i, j)** is the response of variable *i* to a shock in *j*. **IMPULSE** with the **CV** option does Cholesky shocks.

```
declare rect[series] impblk(neqn, neqn)
declare vect[series] scaled(neqn)
impulse(model=canmodel, result=impblk, noprint, $
    steps=nsteps, cv=%sigma)
```

The first loop plots the responses of all series to a single series. The response of a series is normalized by dividing by its innovation variance. This allows all the responses to a shock to be plotted (in a reasonable way) on a single scale. Note that these graphs get a bit hard to read with more than five or six variables. The **GRAPH** uses the **NUMBER=0** option to handle the horizontal axis labeling, since that's the standard way to label IRF's.

```
do i=1, neqn
    compute header="Plot of responses to "+implabel(i)
    do j=1, neqn
        set scaled(j) = (impblk(j, i))/sqrt(%sigma(j, j))
    end do j
    graph(header=header, key=below, klabels=implabel, $
        number=0, series=scaled) neqn
end do i
```

The second loop graphs the responses of a variable to all shocks. These don't have to be normalized, since the appropriate size of the shocks is determined by the estimates.

Chapter 7: Vector Autoregressions

```
do i=1,neqn
  compute header="Plot of responses of "+implabel(i)
  graph(header=header,key=below,klabels=implabel,number=0,$
    series=%scol(impblk,i)) neqn
end do i
```

A common question has been how to set up a Cholesky or other calculated shock to have a specific impact on a variable—most commonly, a standard size interest rate shock. In this model, the impact of the interest rate shock on itself is .56522. In order to make that (say) -1 while keeping the same “shape” we need to divide by the .56522 (though we will use the program to handle that) and multiply by -1. You could do the standard shocks and rescale after the fact, but it’s easier to just rescale the shock. Both methods are equivalent because of the linearity of the IRF’s.

The following takes the Cholesky factor of %SIGMA and extracts the interest rate shock (shock number 3, since it’s the third series in the model). It then rescales it by multiplying by -1 and dividing by the 3rd element (which would be the impact on the interest rate, again, as the third variable).

```
compute factor=%decomp(%sigma)
compute [vector] rshock=%xcol(factor,3)
compute rshock=-1.0*rshock/rshock(3)
```

This uses **IMPULSE** with the **SHOCK** option to input just the one shock, putting the results in **TO_R** and graphs the response of Canadian RGDP (the fifth variable in the model). Note that **TO_R** is still a **RECT[SERIES]** even though there is just one shock, so you need the (5,1) subscripts to get the response.

```
impulse(model=canmodel,shock=rshock,steps=nsteps,$
  results=to_r,noprint)
graph(number=0,header=$
  "Response to Canadian RGDP to expansionary rate shock") 1
# to_r(5,1)
```

You need to be careful when doing this type of rescaling as it breaks the natural scale that the model has provided. It isn’t recommended that you do this unless this is the only one shock of interest.

CVMODEL.RPF example

CVMODEL.RPF estimates two structural models for the six-variable Canadian VAR. The first of these is a “B” type model which interprets the orthogonal shocks as two real shocks, two financial market shocks and two nominal shocks. The loadings of these shocks onto the innovations produces a highly overidentified model: there are only six free parameters instead of the fifteen for a just identified model.

$$(23) \quad \begin{bmatrix} u_u \\ u_c \\ u_r \\ u_x \\ u_m \\ u_p \end{bmatrix} = \begin{bmatrix} 1 & 0 & \beta_{uf1} & 0 & 0 & 0 \\ \beta_{cr1} & 1 & \beta_{cf1} & 0 & 0 & 0 \\ 0 & 0 & 1 & \beta_{rf2} & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ \beta_{mr1} & 0 & 0 & 0 & 1 & \beta_{mn1} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_{r1} \\ v_{r2} \\ v_{f1} \\ v_{f2} \\ v_{n1} \\ v_{n2} \end{bmatrix}$$

The second model is also overidentified, with nine parameters. This is an “A” type model, where non-orthogonalized shocks are mapped to orthogonalized ones.

$$(24) \quad \begin{bmatrix} 1 & 0 & \gamma_{ur} & 0 & 0 & 0 \\ \gamma_{cu} & 1 & \gamma_{cu} & 0 & 0 & 0 \\ 0 & 0 & 1 & \gamma_{rx} & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & \gamma_{mc} & \gamma_{mr} & 0 & 1 & \gamma_{mp} \\ 0 & \gamma_{pc} & \gamma_{pr} & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_u \\ u_c \\ u_r \\ u_x \\ u_m \\ u_p \end{bmatrix} = \begin{bmatrix} v_u \\ v_c \\ v_r \\ v_x \\ v_m \\ v_p \end{bmatrix}$$

For both of these, an **ERRORS** instruction is used to compute the decomposition of variance.

The setup of the basic VAR is the same as for **IMPULSE.RPF**, though with a slightly different order on the variables. (You can use any listing order for a structural model, through you have to make sure you define the model based upon the order you choose). These are the initial declarations for the first structural VAR. We need a FRML which evaluations to a RECTANGULAR matrix, which needs to be declared before it can be created. We also need to declare the free parameters:

```
dec frml[rect] bfrml
nonlin uf1 cr1 cf1 rf2 mf1 mm2
```

In this case, we’re using “descriptive” names for these (for instance, UF1 is for the loading of the F1 shock onto US GDP, CR1 for the loading of R1 onto the Canadian GDP). You could also use positions like B13, B21, etc. The following constructs the FRML describing the structural model. We strongly recommend using one row per line (with the \$ continuation) and using spaces or zero padding to make the columns line

Chapter 7: Vector Autoregressions

up. It will save you a lot of time if you have to make changes, or have some problem with the model. In this case, all parameters are initialized to zero, which often works OK for a model like this. The rows correspond to the dependent variables in the order in which you put them into the model, which the columns correspond to your intended shocks.

```
frml bfrml = ||1.0,0.0,uf1,0.0,0.0,0.0|$
           cr1,1.0,cf1,0.0,0.0,0.0|$
           0.0,0.0,1.0,rf2,0.0,0.0|$
           0.0,0.0,0.0,1.0,0.0,0.0|$
           mf1,0.0,0.0,0.0,1.0,mm2|$
           0.0,0.0,0.0,0.0,0.0,1.0||
compute uf1=cr1=cf1=rf2=mf1=mm2=pm2=0.0
```

This is estimated by using the genetic method first, then polishing the estimates with BFGS. In practice, you might want to repeat this several times to test whether there are global identification problems.

```
cvmodel(b=bfrml,method=bfgs,factor=bfactor,$
        pmethod=genetic,piters=50) %sigma
```

Because the shocks don't really correspond one-to-one with the variables, the labels option is used on **ERRORS** to give them the desired labels.

```
errors(model=canmodel,factor=bfactor,steps=28,window="BModel",$
        labels=||"Real 1","Real 2","Fin 1","Fin 2","Nom 1","Nom 2"||)
```

The same procedure is followed with the other model. It just uses the A option on **CVMODEL** rather than the B option:

```
dec frml[rect] afrml
nonlin rx ur cu cr pc pr mp mc mr

frml afrml = ||1.0,0.0,ur ,0.0,0.0,0.0|$
              cu ,1.0,cr ,0.0,0.0,0.0|$
              0.0,0.0,1.0,rx ,0.0,0.0|$
              0.0,0.0,0.0,1.0,0.0,0.0|$
              0.0,mc ,mr ,0.0,1.0,mp|$
              0.0,pc ,pr ,0.0,0.0,1.0||
compute ur=cu=cr=rx=mc=mr=mp=pc=pr=0.0
cvmodel(a=afrml,method=bfgs,factor=afactor,$
        pmethod=genetic,piters=50) %sigma
errors(model=canmodel,factor=afactor,steps=28,window="AModel")
```

7.7 Historical Decomposition and Related Techniques

Background

HISTORY decomposes the historical values of a set of time series into a base projection and the accumulated effects of current and past innovations. You can see, for instance, whether movements in variable x in 2009 were likely the result of innovations in variable z a year earlier.

The historical decomposition is based upon the following partition of the moving average representation

$$(25) \mathbf{y}_{T+j} = \sum_{s=0}^{j-1} \Psi_s \mathbf{u}_{T+j-s} + \left[\hat{\mathbf{y}}_{T+j} + \sum_{s=j}^{\infty} \Psi_s \mathbf{u}_{T+j-s} \right]$$

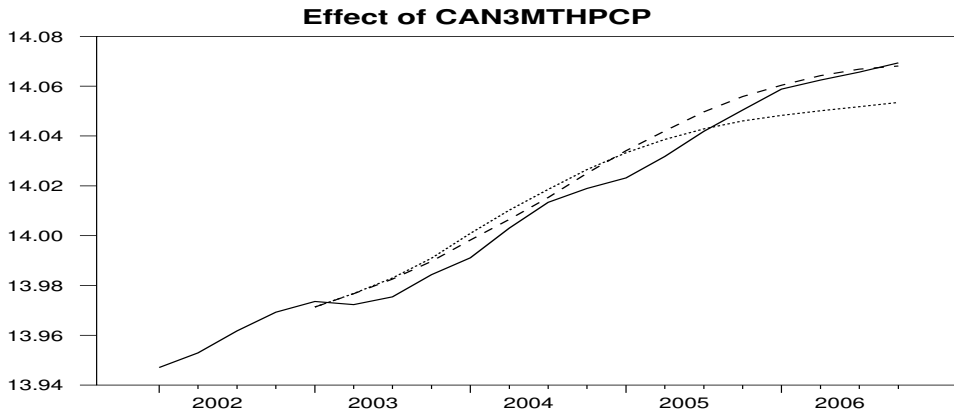
The first sum represents that part of $\mathbf{y}_{T+j}$ due to innovations in periods $T+1$ to $T+j$. The second is the forecast of $\mathbf{y}_{T+j}$ based on information available at time T .

If $\mathbf{u}$ has N components, the historical decomposition of $\mathbf{y}_{T+j}$ has $N+1$ parts:

- The forecast of $\mathbf{y}_{T+j}$ based upon information at time T (the term in brackets).
- For each of the N components of $\mathbf{u}$, the part of the first term that is due to the time path of that component.

Presenting Results

HISTORY generates a lot of series: $N(N+1)$ of them for a standard VAR. Graphs are the simplest way to present these. **HISTORY.RPF** (page UG–240) creates a separate page of graphs for each variable, with each page having a separate graph for the shocks to each of the other variables. Each of these subgraphs has the actual data, the forecast and the sum of the forecast and the effect of the residuals to that variable. This, for instance, shows the effect of the interest rate on Canadian GDP. We can see that almost the entire difference between actual GDP (the solid line) and the base forecast can be attributed to interest rate shocks.



HISTORY.RPF Example

HISTORY.RPF does a historical decomposition of a six-variable Canadian model, which includes GDP for Canada and the U.S., the Canadian-U.S. exchange rate, M1, GDP deflator and a short interest rate. (This is the same data set as was used in the previous section.) This is written to be generalized easily to other sets of variables. Other than the initial set up of variables, you just need to set the decomposition range (HSTART and HEND) change the model name, and, perhaps, change the layout of the array of graphs (the VFIELDS and HFIELDS are specific to six variables).

After the basic model setup, these lines set the range for the historical decomposition, so the base forecasts are computing using data through 2002:4 (the period before 2003:1).

```
compute hstart=2003:1
compute hend =2006:4
```

The next block pulls information out of the model. The VARLABELS are the labels of the dependent variables, while the SHOCKLABELS are the labels of the shocks. Here (with Cholesky shocks), they are the same. In practice, they might not be. And in practice, you might want to use more descriptive labels, which you can do using **COMPUTE**, for instance, something like

```
compute varlabels=||"Canadian Real GDP","GDP Deflator",...||

compute neqn=%modelsz(canmodel)
dec vect[int] depvar(neqn)
dec vect[labels] varlabels(neqn) shocklabels(neqn)
ewise varlabels(i)=%modellabel(canmodel,i)
ewise shocklabels(i)=%modellabel(canmodel,i)
compute depvar=%modeldevars(canmodel)
```

This does the decomposition. If you want non-Cholesky shocks, you can use a **FACTOR** option on the **HISTORY**.

```
history(model=canmodel,add,base=base,effects=effects,$
        from=hstart,to=hend)
```

This does a separate page of graphs for each dependent variable, with each page having a 3 x 2 layout with the graphs of the effects of the six shocks on that variable.

```
do j=1,neqn
  spgraph(vfields=3,hfields=2,window=varlabels(j),$
    header="Historical Decomposition of "+varlabels(j))
  do i=1,neqn
    graph(header="Effect of "+shocklabels(i)) 3
    # depvar(j) hstart-4 hend
    # base(j)
    # effects(j,i)
  end do j
  spgraph(done)
end do i
```


7.7.1 Counterfactual Simulation

The historical decomposition is a special case of a *counterfactual simulation*. The actual data can be recreated by adding the base forecast to the sum of contributions of *all* the components. If we omit some of those components, we're seeing what data would have been generated if some (linear combinations) of the residuals had been zero rather than what was actually observed.

The tool for doing more general counterfactual simulations is **FORECAST** with the option **PATHS**. With the **PATHS** option, **FORECAST** takes as input an additional **VECT [SERIES]** which has the paths of shocks to add in during the forecast period. Note that these have to be non-orthogonal shocks. If the restrictions on the residuals are in a transformed space (like orthogonalized shocks), these have to be transformed back into the space for non-orthogonal shocks. If $\mathbf{u}_t$ are the VAR residuals and $\mathbf{v}_t$ are the structural residuals, then $\mathbf{u}_t = \mathbf{F}\mathbf{v}_t$ and $\mathbf{v}_t = \mathbf{F}^{-1}\mathbf{u}_t$. The general procedure is:

1. Transform the VAR residuals into structural residuals.
2. Make the desired changes (zeroing, for instance) to those.
3. Transform back into the space for equation shocks.
4. Use **FORECAST (PATHS)** inputting the **VECT [SERIES]** of shocks computed in 3.

It's possible to combine several of these steps into a single matrix calculation. For instance, suppose that we use the "B" model in **CVMODEL.RPF** (page UG-237), which defines two "real shocks" (the first two of six), and we want to generate the path of GDP with all shocks other than those two zeroed out; this is one possible way to estimate potential (Canadian) GDP, which is the second endogenous variable. That can be done with the following (after the first **CVMODEL** in that example):

```
compute supplymask=||1.0,1.0,0.0,0.0,0.0,0.0||
compute supplyonly=bfactor*%diag(supplymask)*inv(bfactor)
dec vect[series] supplyshocks(6)
do t=hstart,hend
    compute %pt(supplyshocks,t,supplyonly*%xt(varresids,t))
end do t
forecast(paths,from=hstart,to=hend,model=canmodel,$
    results=withshocks)
# supplyshocks
set potgdp = withshocks(2)
```

The matrix **SUPPLYONLY** maps the original residuals to the set of non-orthogonal shocks which shut down the final four structural residuals—as you read it from right to left, **INV (BFACOR)** converts to structural residuals, **%DIAG (SUPPLYMASK)** zeroes out the last four elements, and **BFACOR** converts back to non-orthogonal shock space.

7.7.2 Conditional Forecasts

Background

The type of analysis done earlier in this section is relatively straightforward because it constrains the behavior of the *shocks*. It's much harder if you want to constrain the behavior of the endogenous variables themselves—that becomes a special case of *conditional forecasting*.

In a conditional forecast, certain values during the forecast period are fixed in advance. The simplest type of conditional forecast is where time paths for the *exogenous* variables are input, as is usually done in simultaneous equations models.

When forecasts are conditional upon future values of *endogenous* variables, the procedure is more complicated. Consider the formula for the error in a K -step forecast (forecasting t using $t-K$):

$$(26) \quad \sum_{s=0}^{K-1} \Psi_s \mathbf{u}_{t-s}$$

When we constrain a future value of the process, we are setting a value for this forecast error, since subtracting the unconstrained forecast from the constrained value produces the error of forecast. We have thus put a linear constraint upon the innovations of the periods between t and $t-K+1$.

With a one-step-ahead constraint, the forecast error in a variable consists of just one innovation since $\Psi_0 = \mathbf{I}$ (non-orthogonal innovations). However, *two* steps out, the forecast error in a variable depends upon the first step innovations in *all* variables plus its own second step. There are now many ways to achieve the particular value—we have only a single constraint on a linear combination of $N+1$ variables. Thus, it's much more difficult to achieve a counterfactual based upon the value of $\mathbf{Y}$.

Computing Conditional Forecasts

Conditional forecasting (in a linear model) is actually a special case of Kalman smoothing (Chapter 10). However, it's simpler to solve the problem directly than to cast the VAR into a state-space form. In calculating forecasts subject to constraints on the forecast error (26), it's most convenient to switch to orthogonalized residuals, thus, the forecast error is

$$(27) \quad \sum_{s=0}^{K-1} \Psi_s \mathbf{F} \mathbf{v}_{t-s}$$

where $\mathbf{F}$ is a factor of the covariance matrix. By stacking the orthogonalized innovations during the forecast period into a vector, we can write the constraints in the form

$$(28) \quad \mathbf{R} \mathbf{V} = \mathbf{r}$$

The conditional forecasts are then made by first computing the $\mathbf{V}$ vector which minimizes $\mathbf{V}'\mathbf{V}$ subject to the constraint. The solution to this is

$$(29) \mathbf{V}^* = \mathbf{R}'(\mathbf{R}\mathbf{R}')^{-1}\mathbf{r}$$

The $\mathbf{V}$'s are then translated back into non-orthogonalized shocks, and the model is forecast with those added shocks. This is similar to what is done in Section 7.7.1, the calculation is just complicated by the need to solve out the minimization problem rather than simply zero out shocks.

The @CONDITION Procedure

This can all be done using the procedure **@CONDITION**. Before you execute it, you must define the VAR and estimate it. **@CONDITION** can handle any number of constraints on single values of future endogenous variables. With the **GENERAL** option, you can use more general linear constraints on future values.

```
@condition(options)  nconstr
# series period value    (one for each constraint)
```

```
@condition(general,other options)  nconstr
# vector with linear combination of endogenous variables
# entry value
```

Where *nconstr* is the number of constraints.

Supplementary Cards (without GENERAL, one for each constraint)

<i>series</i>	Series constrained. This must be an endogenous variable of one of the equations.
<i>period</i>	Period of the constraint.
<i>value</i>	Target value for the constraint.

Supplementary Cards (with GENERAL, one pair for each constraint)

<i>vector</i>	VECTOR or real numbers/expressions with linear combination of endogenous variables to constrain.
<i>period</i>	Period of the constraint.
<i>value</i>	Target value for the constraint.

Options

model=*model to forecast*.

This should already be estimated.

from=*starting period of forecasts* [**end of estimation + 1**]

to=*ending period of forecasts* [**last constrained period**]

steps=*number of forecast steps*

STEPS can be used in place of the TO option if it's more convenient.

Chapter 7: Vector Autoregressions

results=VECTOR[SERIES] for constrained forecasts

Use this to save the forecasts to a VECTOR of SERIES.

cv=SYMMETRIC covariance matrix of residuals [from the MODEL]

factor=RECTANGULAR factor of the covariance matrix of residuals
[Cholesky factor of covariance matrix]

shocks=VECTOR[SERIES] for orthogonal shocks [not used]

You can use the SHOCKS option to retrieve the set of orthogonalized shocks which are used in the forecasts. The FACTOR option can provide the desired factorization of the covariance matrix. Note that the conditional forecasts don't depend upon the factor that you use; the only thing affected are the precise set of orthogonalized shocks needed to generate them.

general/[nogeneral]

Choose GENERAL if you want to apply constraints to a linear combination of the endogenous variables, rather than to individual variables.

simulate/[nosimulate]

If you choose SIMULATE, the procedure generates a draw from the constrained distribution of the forecasts, assuming normally distributed innovations. This result is saved into *forecasts* instead of the constrained forecasts. This generates just one draw per call to @CONDITION. It's a bit inefficient, since it goes through all the set up work each time, but, in practice, it seems to work adequately. If you need a very large number of simulations, or the simulations prove to be time-consuming, you can fairly easily adapt the procedure to generate more than one draw at a time. See Chapter 16 for information on processing the results of random draws.

Notes

When setting up constraints, it's a good idea to keep them as loose as possible. For instance, if you're trying to constrain a growth rate, it's better to do this as year over year changes, rather than constraining each quarter or month. That is, to constrain the series *Y* (in logs) to 4% annual growth, use

```
@condition(model=kpsw,from=2018:1,steps=8,results=cforecasts) 1
# y 2018:4 y(2017:4)+.04
```

rather than

```
@condition(model=kpsw,from=2014:1,steps=8,results=cforecasts) 4
# y 2018:1 y(2017:4)+.01
# y 2018:2 y(2017:4)+.02
# y 2018:3 y(2017:4)+.03
# y 2018:4 y(2017:4)+.04
```

The potential problem with the tighter constraints is that they might force some "whipsawing" innovations to hit all the values.

CONDITION.RPF Example

Example `CONDITION.RPF` demonstrates conditional forecasting, using the same data set as in `IMPULSES.RPF` and several other examples. It computes forecasts conditional on 5% year over year growth in the US GDP. This type of constraint is easy to handle because US GDP is in logs.

After the model is set up and estimated, this sets the forecast range

```
compute fstart=2007:1,fend=2009:4
```

This does the conditional forecast into the series `CONDFORE(1)` to `CONDFORE(6)`. Note that you use the name of the dependent variable on the constraint. The first constraint has `LOGUSAGDP` at 2007:4 hitting the value of the same series at 2006:4 plus .05 (for 5% year over year growth), and the second has a 10% change over two years to 2008:4.

```
@condition(model=canmodel,from=fstart,to=fend,results=condfore) 2  
# logusagdp 2007:4 logusagdp(2006:4)+.05  
# logusagdp 2008:4 logusagdp(2006:4)+.10
```

This computes the unconditional forecasts into `FORECAST(1)` to `FORECAST(6)`.

```
forecast(model=canmodel,results=forecasts,from=fstart,to=fend)
```

And this graphs the conditional and unconditional forecasts for each of the series.

```
do i=1,neqn  
  compute [label] l=%modellabel(canmodel,i)  
  graph(header="Forecasts of "+l>window=1,$  
    key=below,klabels=||"Unconditional","Conditional"||) 2  
  # forecasts(i)  
  # condfore(i)  
end do i
```

The remainder of the program is more technical. It does a set of 1000 simulations subject to the same constraint. The results of this for the Canadian GDP are analyzed on two graphs. Both of these show the .16 and .84 quantiles of the simulated values: these correspond roughly to one standard error. See Chapter 16 for more on the “bookkeeping” used to process the draws. Because we need a simulated series for each draw, and because we need the full history of the simulations (there are no sufficient statistics for order statistics, so you need all the generated data), the following sets up a `SERIES[VECTOR]`. At each entry from `FSTART` to `FEND`, this will have a 1000-vector with the simulated values for Canadian RGDP.

```
compute ndraws=1000  
dec series[vect] can  
gset can fstart fend = %zeros(ndraws,1)
```

Chapter 7: Vector Autoregressions

For each draw, this does a simulation subject to the constraints, and saves the simulated values for Canadian RGDP (variable 1 as this model is ordered).

```
do draw=1,ndraws
  @condition(simulate,model=canmodel,from=fstart,to=fend,$
    results=simfore) 2
  # logusagdp 2007:4 logusagdp(2006:4)+.05
  # logusagdp 2008:4 logusagdp(2006:4)+.10
  do t=fstart,fend
    compute can(t)(draw)=simfore(1)(t)
  end do t
end do draw
```

This computes the 84th and 16th percentiles of the distribution at each point in the forecast range and graphs the unconditional and conditional forecasts (again, the target variable is variable 1 in the model) along with the upper and lower bounds.

```
set upper fstart fend = %fractiles(can(t),||.84||)(1)
set lower fstart fend = %fractiles(can(t),||.16||)(1)
graph(key=below,$
  header="Canadian Real GDP with High Growth of US GDP",$
  klabels=||"Unconditional","Conditional","84%ile","16%ile"||) 4
# forecasts(1)
# condfore(1)
# upper / 3
# lower / 3
```

These transform the various GDP numbers into average (annualized) growth rates from FSTART-1 and does a combined graph.

```
set fgrowth fstart fend = $
  400.0*(forecasts(1)-logcangdp(fstart-1))/(t-fstart+1)
set cgrowth fstart fend = $
  400.0*(condfore(1)-logcangdp(fstart-1))/(t-fstart+1)
set ugrowth fstart fend = $
  400.0*(upper-logcangdp(fstart-1))/(t-fstart+1)
set lgrowth fstart fend = $
  400.0*(lower-logcangdp(fstart-1))/(t-fstart+1)
graph(key=below,$
  header="Canadian GDP Average Growth with High Growth of US GDP",$
  klabels=||"Unconditional","Conditional","84%ile","16%ile"||) 4
# fgrowth
# cgrowth
# ugrowth / 3
# lgrowth / 3
```

7.8 Cointegration and Error Correction Models

In Section 3.12, we discussed how to *test for* cointegration. In this section, we describe how to impose it on a (VAR) model. The standard VAR model

$$(30) \quad \mathbf{y}_t = \sum_{s=1}^L \Phi_s \mathbf{y}_{t-s} + \mathbf{u}_t$$

can always be rewritten in the form

$$(31) \quad \Delta \mathbf{y}_t = \sum_{s=1}^{L-1} \Phi_s^* \Delta \mathbf{y}_{t-s} + \Pi \mathbf{y}_{t-1} + \mathbf{u}_t$$

where $\Delta \mathbf{y}_t$ is the first difference of the $\mathbf{y}$ vector (that is, $\mathbf{y}_t - \mathbf{y}_{t-1}$). If Π is zero, then the VAR can be modelled adequately in first differences. If Π is *not* zero, then, even if each component of $\mathbf{y}$ has a unit root, a VAR in first differences is misspecified.

If Π is full-rank, there is nothing to be gained by writing the system in form (31) rather than (30): the two are equivalent. The interesting case is where Π is non-zero but less than full rank. In that case, we can write the matrix as

$$(32) \quad \Pi = \alpha \beta'$$

where α and β are $N \times r$ matrices (N is the number of components in $\mathbf{y}$ and r is the rank of Π). Note that the decomposition in (32) isn't unique: for any $r \times r$ non-singular matrix $\mathbf{G}$, it is also true that

$$(33) \quad \Pi = (\alpha \mathbf{G}) (\beta \mathbf{G}'^{-1})'$$

Where r is one, however, (32) is unique up to a scale factor in the two parts.

The instruction **ECT** allows you to estimate the VAR in the form (31) imposing a reduced rank assumption on the Π term. The $\beta' \mathbf{y}$ represent stationary linear combinations of the $\mathbf{y}$ variables. If the $\mathbf{y}$'s themselves are non-stationary, they are said to be *cointegrated*. The instruction name (which is short for *Error Correction Terms*) comes from the fact that (31) is sometimes called the error correction form of the VAR. See, for instance, Chapter 6 of Enders (2014) or Chapter 19 of Hamilton (1994).

Setting up the VAR

To estimate a VAR in error correction form, start out as if you were doing the standard form (30). That is, your variables aren't differenced, and you count the number of lags you want on the undifferenced variables. Create the error correction equations, and use **ECT** to add them to the system. You don't have to fill in the coefficients when setting up the VAR, but you at least need to have created the equations.

The VAR system works somewhat differently in error correction form. The regression output, coefficient vectors, and covariance matrices will be based upon the restricted form. When the **MODEL** is used in a forecasting instruction such as **IMPULSE** or **FORECAST**, it is substituted back into form (30), so that it analyzes the original variables and not the differenced ones.

Chapter 7: Vector Autoregressions

In general, you will use **ESTIMATE** to compute the coefficients for the model. You can use **KALMAN** (Section 7.14) to estimate the parameters sequentially, but the Kalman filter does *not* apply to the cointegrating vector β —it takes β as given by the coefficients in the error correction equation(s), and estimates only the Φ_s^* and α . If you are estimating these coefficients yourself (other than the least squares conditional on β that **ESTIMATE** does), use the function `%MODELSETCOEFFS(model, coeffs)`. The coefficient matrix `coeffs` is a RECTANGULAR matrix. Each column gives the coefficients for an equation. In a given equation, the first coefficients are the Φ_s^* . Note that there are $L-1$ lags for each endogenous variable, not L , since these equations are in differenced form. Next are the variables listed on the **DETERMINISTIC** instruction, if any. The final coefficients are the loadings α on the error correction terms.

Setting the Cointegrating Vectors

The cointegrating vectors β can either be set from theory or estimated. If you want to set them from theory, use the **EQUATION** instruction to create the equation, and use the `COEFFS` option on **EQUATION** or the `%EQNSETCOEFFS` function to set the coefficients. For instance, in the `COINTTST.RPF` example (page UG–105), we test whether Purchasing Power Parity is a cointegrating relationship. If (despite the results to the contrary in this dataset), we were to proceed with imposing this during estimation, we would do the following:

```
equation(coeffs=||1.0,-1.0,-1.0||) rexeq rexrate
# pusa s pita
system(model=pppmodel)
variables pusa s pita
lags 1 to 12
det constant
ect rexeq
end(system)
estimate
```

In this case, the error correction equation shows the desired linear combination of the endogenous variables. If, instead, the coefficients of the equilibrium equation are estimated by an auxiliary regression, the error correction equation will have one of the endogenous variables as its dependent variable. For the example above, you would just replace the **EQUATION** instruction (and supplementary card) with:

```
linreg(define=rexeq) pusa
# s pita
```

If you're estimating the cointegrating vector using a procedure which doesn't normalize a coefficient to one (such as `@JOHMLE`), define the equation *without* a dependent variable:

```
equation(coeffs=cvector) rexeq *
# pusa s pita
```


Error Correction Model (Example ECT.RPF)

The example file `ECT.RPF` analyzes a set of three interest rate variables, first testing for cointegration, then imposing it.

RATS does only a limited form of cointegration analysis. For a more complete treatment, the program CATS (page UG–107) is available from Estima as an add-on. CATS provides more sophisticated tests for cointegrating rank, model diagnostics and tests on the structures of both the cointegrating vectors and their loadings (and much more).

The first step is to test that the three series have unit roots. (It’s strongly accepted for all).

```
@dfunit(lags=6) ftbs3
@dfunit(lags=6) ftb12
@dfunit(lags=6) fcm7
```

There are several ways to estimate the cointegrating vector. The simplest is to run an “Engle-Granger” regression (one of the yields on the other two). While the estimates from that are “super-consistent” if the series are cointegrated, there are other ways to get better smaller sample performance. Here we’ll use the `@JOHMLE` procedure, which does maximum likelihood estimation. Other procedures for estimating a cointegrating vector or vectors are `@FM` (does *fully-modified least squares*) and `@SWDOLS` (does *dynamic OLS*). These are all included in the *Time Series—Cointegration Estimation* wizard.

Note that it is far from obvious (*a priori*) that the cointegrating rank will be one. It might very well be zero, or, if there were a single stochastic trend linking the yields, then the cointegrating rank would be two. The remainder of the analysis assumes rank one and extracts the cointegrating vector corresponding to the largest eigenvalue.

Because the series don’t have a trend, the appropriate choice for the deterministic component is `DET=RC`, which doesn’t include a constant in the individual equations (where it would cause the series to drift because of the unit roots), but restricts it to the cointegrating vector. The components of the cointegrating vector produced by `@JOHMLE` with this choice will have *four* components: the three variables + the constant. If we had used the default `DET=CONSTANT`, it would only be a 3-vector, but we would also include `DET CONSTANT` in the `SYSTEM` definition. Be careful with your choice for the `DET` option—it’s a common error to select the wrong one. Note that there is no reason that the error correction term(s) will have mean zero—the requirement is that they be stationary, which includes processes with non-zero means.

```
@johmle(lags=6, det=rc, cv=cvector)
# ftbs3 ftb12 fcm7
equation(coeffs=cvector) ecteq *
# ftbs3 ftb12 fcm7 constant
```

Chapter 7: Vector Autoregressions

This sets up the model with the error correction term. Note that, as described above, there is no `CONSTANT` as the constant is included in the cointegrating vector.

```
system(model=ectmodel)
variables ftbs3 ftb12 fcm7
lags 1 to 6
ect ecteq
end(system)
```

This estimates the model and computes the decomposition of variance:

```
estimate
compute sigma=%sigma
errors(model=ectmodel, steps=36)
```

ESTIMATE with a model with **ECT** elements defines the matrices `%VECMPI` and `%VECMALPHA`. `%VECMPI` is the estimated value of Π from (31) and `%VECMALPHA` are the α loadings from (32). Note well that the lag sums (used in the Blanchard-Quah and the short-and-long run restrictions of Section 7.5) are much more complicated for a VECM model than a regular VAR as the long-run response matrix isn't full rank. (If x and y are cointegrated, then if x has a zero long-run response, y must as well). See the `SHORTANDLONGRUNVECM.RPF` example for the proper way to handle this.

If we needed *two* cointegrating vectors rather than one (in this case, we really don't), we would need to use the `VECTORS` option on `@JOHMLE` to get the full set of eigenvectors, then `%XCOL` to pull out the coefficients for each of the two equations that we would need to define. For instance,

```
@johmle(lags=6, det=rc, vectors=cvectors)
# ftbs3 ftb12 fcm7
equation(coeffs=%xcol(cvectors,1)) ect1 *
# ftbs3 ftb12 fcm7 constant
equation(coeffs=%xcol(cvectors,2)) ect2 *
# ftbs3 ftb12 fcm7 constant
*
system(model=ect2model)
variables ftbs3 ftb12 fcm7
lags 1 to 6
ect ect1 ect2
end(system)
estimate
errors(model=ect2model, steps=36)
```

7.9 Near-VAR's

The systems analyzed so far will always have *precisely* the same set of variables on their right hand sides of each equation. You cannot use different sets of dummy variables in **DETERM** or different lag structures, or leave some endogenous variables out of some of the equations. A “near-VAR” is a model which is similar to a VAR, but has some slight differences among the equations.

To construct a near-VAR, you need to either set up the equations individually (using the instruction **EQUATION**), which are then bound into a model using **GROUP**, or you can set up two or more blocks using **SYSTEM** definitions, and then “add” the models. Using **EQUATION** and **GROUP** is the only choice if the equations have different lags, or otherwise have an irregular structure.

We'll look at two ways to set up a near-VAR where the equation for USA includes only the lags of USA itself, while the other two regions are full VAR equations.

With EQUATION

```
equation usaeq usagdp
# constant usagdp{1 to 4}
equation mexeq mexgdp
# constant usagdp{1 to 4} mexgdp{1 to 4} centgdp{1 to 4}
equation centeq centgdp
# constant usagdp{1 to 4} mexgdp{1 to 4} centgdp{1 to 4}
group gdpmodel usaeq mexeq centeq
```

With multiple SYSTEMS

This is likely to be more useful with larger models. You define systems using **DETERM** instructions for lags of the variables which are in the smaller block(s):

```
system(model=usa)
variables usagdp
lags 1 to 4
det constant
end(system)
system(model=centmex)
variables mexgdp centgdp
lags 1 to 4
det constant usagdp{1 to 4}
end(system)
compute gdpmodel=usa+centmex
```

Estimation

If you construct a near-VAR using either of these two methods, you can estimate it with **ESTIMATE** with the option **MODEL=GDPMODEL**. This will estimate the model with equation by equation least squares, and define all the standard summary statistics produced by **ESTIMATE** for a standard VAR: for instance %SIGMA is the covariance matrix of residuals and %VARLAGSUMS has the matrix of the lag sums.

Chapter 7: Vector Autoregressions

Least squares gives consistent estimates, but they aren't, in general, the maximum likelihood estimates. Instead, there may be *some* gain to using **SUR** (Seemingly Unrelated Regressions) to estimate the system instead of **ESTIMATE**. However, this is really only feasible for small systems because the number of parameters quickly gets excessive. RATS can accurately handle systems with over 200 parameters, but the improvement in efficiency will usually not be worth the trouble. The standard results on SUR vs OLS (Greene, 2012, p 294) tell us there can be a gain in efficiency only if

- the residuals are correlated across equations, and
- the regressors are not the same in each equation.

When the residual correlation is not large and there is much duplication in regressors, the improvement will be small. Even with high correlation, the regression on the smallest subset (the USA equation in the three-variable system) get the same estimates with SUR and OLS. If you want to estimate with SUR, you probably want maximum likelihood estimates, so you need iterated SUR. That would be done in our case with

```
sur (model=gdpmodel , iters=20)
```

Impulse Response Functions/FEVD

The point estimates of these work exactly the same way as they do for a full VAR, so you can use **IMPULSE**, **ERRORS** and **HISTORY**. However, you can't use the same simple techniques for drawing for the posterior for getting confidence bands that you can with a full (OLS) VAR. There are several reasons for this, but the most important is that the covariance matrix no longer has an unconditional density the way it does in a full (OLS) VAR. Instead, you need to do Gibbs sampling (Section 16.7). Because organizing a Gibbs sampler for a system of equations of differing forms is a bit complicated, there is a set of specialty procedures in **@SURGibbsSetup** for handling it.

Structural VAR's

You *can* estimate a structural near-VAR using the two-step procedure of estimating the lag model, then applying **CVMODEL** or some other technique for estimating the covariance matrix model. These will give consistent estimates. The main question is under what situations you can easily get maximum likelihood estimates. Unlike the case with a full VAR, restrictions on the covariance matrix now *have* an effect on the maximum likelihood estimates for the lag coefficients. Thus, there is no easy way to get maximum likelihood estimates for the model if the covariance model is overidentified—you would have to estimate the lag coefficients and covariance model jointly. However, if you have a *just-identified* structural model, then a two-step approach of estimating the lag model (using iterated **SUR** to get maximum likelihood), then estimating the structural model in a second step will give maximum likelihood for the full model.

7.10 VAR's for Forecasting: The Bayesian Approach

Problems with Unrestricted Models

Forecasts made using unrestricted vector autoregressions often suffer from the overparameterization of the models. The number of observations typically available is inadequate for estimating with precision the coefficients in the VAR. This overparameterization causes large out-of-sample forecast errors. For instance, see Fair (1979) for a comparison of forecasting performance of several models that includes an unrestricted VAR.

One possible way to handle this is to use some criterion to choose “optimal” lag lengths in the model. Examples are the Akaike or Schwarz Criterion. Example VARLAG.RPF (page UG–212) shows how these can be used for choosing an overall lag length. However, there are N^2 lag lengths which need to be chosen in a full model, which makes a complete search using such a criterion effectively impossible.

No method of applying direct restrictions (like PDL's for distributed lags) seems reasonable. Vector autoregressions are, in effect, dynamic reduced forms, not structural relations, so the meaning of the values for coefficients is not obvious, and no “shape” constraint suggests itself.

A Bayesian Approach

The Bayesian approach to this problem is to specify “fuzzy” restrictions on the coefficients, rather than “hard” shape or exclusion restrictions. Shrinkage estimators such as this have long been suggested for dealing with multicollinearity and similar problems.

In his dissertation, Litterman suggests the following: the usual approach to degrees of freedom problems is to reduce the number of regressors, which in autoregressive models means using fewer lags. Dropping a lag forces its coefficient to zero. Rather than adopting an all-or-nothing approach, we “suggest” that coefficients on longer lags are more likely to be close to zero than shorter lags. However, we permit the data to override our suggestion if the evidence about a coefficient is strong. Formally, we implement this by placing on the long lags Normal prior distributions with means of zero and small standard deviations. This allows us to estimate the coefficients using Theil's mixed estimation technique.

In a vector autoregression, we must concern ourselves not only with lags of the dependent variable, but also with lags of the other endogenous variables. Because of stability conditions, we have some pretty good information about the size of lag coefficients in a simple autoregression. However, it's not as clear what the sizes of coefficients on *other* variables should be, and these depend, in part, on the relative scales of the variables involved.

Specification of a complete Normal prior on a VAR would be intractable because the covariance matrix of the prior would have dimensions $N^2L \times N^2L$. Instead, we have

Chapter 7: Vector Autoregressions

put together a general form for the prior involving a few *hyperparameters* and, in effect, you estimate your model by choosing those.

You add the prior to your VAR by including the instruction **SPECIFY** in your system definition:

```
system(model=canusa)
variables usam1 usatbill canm1 cantbill canusxr
lags 1 to 13
det constant
specify (type=symmetric,tightness=.15) .5
end(system)
```

Standard Priors

In the discussion in this section, variable j refers to the j th variable listed on the **VARIABLES** instruction and equation i to the equation whose dependent variable is variable i .

The standard priors have the following characteristics:

- The priors put on the deterministic variables in each equation are non-informative (flat).
- The prior distributions on the lags of the endogenous variables are independent Normal.
- The means of the prior distributions for all coefficients are zero. The only exception is the first lag of the dependent variable in each equation, which has a prior mean of one, by default.

Because of these restrictions, the only information required to construct the prior is:

- the mean of the prior distribution for the first own lag in each equation.
- the standard deviation of the prior distribution for lag l of variable j in equation i for all i, j and l : denoted $S(i, j, l)$.

To simplify further the task of specifying the prior, the standard priors restrict the standard deviation function to the form:

$$(34) \quad S(i, j, l) = \frac{\{\gamma g(l) f(i, j)\} s_i}{s_j}; \quad f(i, i) = g(l) = 1.0$$

where s_i is the standard error of a univariate autoregression on equation i . We scale by the standard errors to correct for the different magnitudes of the variables in the system.

The part in braces is the *tightness* of the prior on coefficient i, j, l . It is the product of three elements:

1. The overall tightness $[\gamma]$, which, because of the restrictions on the f and g functions, is the standard deviation on the first own lag.
2. The tightness on lag l relative to lag 1 $[g(l)]$.
3. The tightness on variable j in equation i relative to variable i $[f(i,j)]$.

The Prior Mean

By default, we use a mean of zero for the prior on all coefficients except the first own lag in each equation. The prior on this coefficient is given a mean of one. This centers the prior about the random walk process

$$(35) \quad Y_t = Y_{t-1} + u_t$$

This seems to be a reasonable choice for many economic time series. For example, there are theoretical reasons for various asset prices to follow random walks, and a series growing exponentially at rate α can be represented by

$$(36) \quad \log Y_t = \log Y_{t-1} + \alpha$$

so its log follows a random walk with drift.

The only alternative which strongly suggests itself is a mean of zero on series which are likely to be close to white noise, such as a series of stock returns.

The **MEAN** and **MVECTOR** options of **SPECIFY** control the first own lag means. **MVECTOR** is more likely to be useful since it permits means to vary across equations:

```
specify (mvector=||1.0,0.0,1.0,1.0,1.0||,type=symmetric) .5
```

The **MVECTOR** option here puts a mean of 0.0 on the second equation and 1.0 on all the rest.

Lag Length and Lag Decay

Experience with VAR models has shown that it usually is better to include extra lags with a decaying lag prior rather than to truncate at an early lag and use the default prior. Unless you face severe data constraints, we would suggest using at least one year + one period of lags. Longer lags without any decay are a bad idea.

To tighten up the prior with increasing lags, use **TYPE=HARMONIC** or **TYPE=GEOMETRIC** with an appropriate value for **DECAY**. A **HARMONIC** with **DECAY=1.0** or **DECAY=2.0** commonly works well. **GEOMETRIC** tends to get too tight too fast.

Highly seasonal data are difficult to work with using these standard priors because you typically expect relatively large coefficients to appear on the lags around the seasonal. See Section 7.13 for some tips on dealing with seasonal data.

Overall Tightness

The default is the relatively “loose” value of 0.20. Practice has shown that a reasonable procedure is to set the `TIGHTNESS` parameter to something on the order of .1 or .2; then, if necessary, you can tighten up the prior by giving less weight to the other variables in the system. Since `TIGHTNESS` controls directly the important own lags, setting it too small will force the own lags too close to the prior mean.

Prior Type

The prior type determines the relative tightness function $f(i,j)$: the tightness on variable j in equation i relative to that on variable i . The two types of priors (selected using the `TYPE` option) are the `SYMMETRIC` and the `GENERAL`.

`TYPE=SYMMETRIC` is the simplest prior and is the default. There is only one free hyperparameter; indicated with the *other's weight* parameter on **`SPECIFY`**, it gives the relative weight (w) applied to all the off-diagonal variables in the system:

$$(37) \quad f(i,j) = \begin{cases} 1.0 & \text{if } i = j \\ w & \text{otherwise} \end{cases}$$

The symmetric priors generally are adequate for small systems: those with five or fewer equations. A combination of `TIGHTNESS=.20` and $w=0.5$ is a common choice. As you push w to zero, the system approaches a set of univariate autoregressions—coefficients on all variables other than the own lags of the dependent variable and the deterministic part are forced to zero. The following is an example of `SYMMETRIC`:

```
system(model=smallmod)
variables gnp m1 ipd tbill unemp bfi
lags 1 to 4
det constant
specify (type=symmetric, tight=.2) .5
end(system)
```

`TYPE=GENERAL` requires that you specify the entire $f(i,j)$ function. Obviously, it is unrealistic to think of fine-tuning the prior by picking all of these independently. In fact, such a strategy simply transfers the problem with overparameterization from estimating too many equation coefficients to estimating too many hyperparameters.

Instead, you should use a `GENERAL` prior in situations where

- the model is too large to apply safely a `SYMMETRIC` prior. The `TIGHTNESS=.2` and $w=.5$ recommended above tends to be too loose overall for a system with six or more equations. However, making w much smaller will cut out too much interaction. Use a `GENERAL` which puts moderate weight on variables which you see as being important and low weight on those you believe to be less important.
- the results of a `SYMMETRIC` show that you need to treat some equations more as univariate autoregressions than as part of the VAR. Use a `GENERAL` which is largely the same as the `SYMMETRIC`, but has small off-diagonal elements in the rows corresponding to these equations.

The $f(i,j)$ function is input in one of two ways: by a RECTANGULAR array (using the MATRIX option) or by supplementary cards.

```
system(model=ymp)
variables gnp m1 cpr ppi
lags 1 to 12
det constant
specify(type=general,tight=.2,decay=1.0)
# 1.0 0.5 0.5 0.5    0.1 1.0 1.0 0.1    0.1 1.0 1.0 0.1    $
   0.5 0.5 0.5 1.0
end(system)
```

or

```
declare rect priormat(4,4)
input priormat
  1.0 0.5 0.5 0.5
  0.1 1.0 1.0 0.1
  0.1 1.0 1.0 0.1
  0.5 0.5 0.5 1.0
specify(type=general,matrix=priormat,tight=.2,decay=1.0)
```

Estimation Methods

The VAR with a prior is estimated using the **ESTIMATE** instruction. This employs a variation on the mixed estimation procedure described in Section 1.2 of the *Additional Topics* PDF. It should be noted that, with this type of a prior, single equation techniques are not optimal except in the unlikely case that the residuals are uncorrelated. In the early 1980's, when Bayesian VAR's were introduced, system-wide techniques weren't feasible for any but the smallest models. And it's still the case that full system estimators, done properly (see, for instance, the GIBBSVAR.RPF program) can be done only for medium-sized models, because of the size of the matrices which must be inverted.

The simpler calculations done by **ESTIMATE** (and **KALMAN**, Section 7.14) still are quite valuable. The gains from more careful estimation are likely to be small, since it is only the combination of a prior and a non-diagonal covariance matrix that produces any gain at all. Our suggestion would be that you develop the model using the basic techniques and switch to the more computationally intensive methods only once the model has been built.

7.11 Differences When Using a Prior

Differences with END(SYSTEM)

When you use a **SPECIFY** in setting up the system, the **END (SYSTEM)** instruction causes RATS to print a synopsis of the prior. For example:

```
Summary of the Prior...
Tightness Parameter 0.100000
Harmonic Lag Decay with Parameter 0.000000
Standard Deviations as Fraction of Tightness and Prior Means
Listed Under the Dependent Variable
```

	LOGCANGDP	LOGCANDEFL	LOGCANM1	LOGEXRATE	CAN3MTHPCP	LOGUSAGDP
LOGCANGDP	1.00	0.50	0.01	0.01	0.20	0.50
LOGCANDEFL	0.50	1.00	0.01	0.01	0.20	0.50
LOGCANM1	0.50	0.50	1.00	0.01	0.20	0.50
LOGEXRATE	0.50	0.50	0.01	1.00	0.20	0.50
CAN3MTHPCP	0.50	0.50	0.01	0.01	2.00	0.50
LOGUSAGDP	0.50	0.50	0.01	0.01	0.20	1.00
Mean	1.00	1.00	1.00	1.00	1.00	1.00

Variables Defined by SPECIFY

%PRIOR **SPECIFY** stores the matrix of weights and means in **%PRIOR**, an $(N+1) \times N$ array, arranged as printed above. By making changes to **%PRIOR**, you can alter a standard prior without going through a complete redefinition of the **SYSTEM**.

Differences with ESTIMATE

RATS estimates the system of equations using the mixed estimation technique described in Theil (1971). The differences between **ESTIMATE** for systems with a prior and for systems without a prior are:

- The degrees of freedom reported are not $T-K$, where K is the number of regressors, but $T-D$, where D is the number of deterministic variables. This is a somewhat artificial way to get around the problem that, with a prior, K can exceed T .
- With the option **CMOM=SYMMETRIC** array for $X'X$, you can obtain the array **X'X** of the regressors.
- With the option **DUMMY=RECTANGULAR** array of dummy observations, RATS saves the dummy observations used in doing the mixed estimation procedure in the specified array.

7.12 Choosing a Prior

Influence of Model Size

The larger is a model relative to the number of data points, the more important the prior becomes, as the data evidence on the individual coefficients becomes weaker. “Average” situations are models with 9 parameters per equation for 40 data points, 30 for 120, and 70 for 400. These models call for moderately tight priors. Substantially larger models for a given size require greater tightness through either:

- a lower value for `TIGHTNESS`, or
- the downweighting of the “other” variables, either through a tighter `SYMMETRIC` prior or through use of a `GENERAL` prior.

Objective Function

In searching over the parameters governing the prior, we need to have, formally or informally, an objective function. Because we are generating a forecasting model, the best forms for this are based upon forecast errors. Three have been used:

- Theil U values (computed with the instruction **THEIL**), which can be used formally, by mapping them to a single value by a weighted average, or informally, by examining changing patterns in the values.
- likelihood function of the data conditional on the hyperparameters. This is a by-product of the Kalman filter procedure.
- log determinant of the covariance matrix of out-of-sample forecast errors.

The last two are discussed in Doan, Litterman and Sims(1984). The third was used in that paper and is the most difficult to compute. Our preference is the informal use of Theil U statistics.

In all cases, we calculate simulated “out-of-sample” forecasts within the data range. We do this by using the Kalman filter (Section 7.14) to estimate the model using only the data up to the starting period of each set of forecasts.

A Simple Procedure

A simple procedure which we have found to be effective is the following:

1. Run a system of univariate OLS models to get benchmark Theil U s.
2. Run a system of univariate models with a standard value for `TIGHTNESS`.
3. Run a standard `SYMMETRIC` prior.

Based upon these, adjust the prior (switching to a `GENERAL` prior):

- If the Theil U s in an equation are worse in 2 than in 1, loosen up on the own lags by setting the diagonal element to 1.5 or 2.0.
- If the Theil U s in an equation are worse in 3 than in 2, tighten up on the other variables by reducing the off-diagonal elements.

Chapter 7: Vector Autoregressions

Picking a Bayesian VAR: CANMODEL.RPF example

The example file CANMODEL.RPF uses the procedure RUNTHEIL (defined here, and needing modification for a specific model) to compute Theil U statistics for various priors. You need to change the *italicized* lines to accommodate your data set.

By setting TIGHTNESS to a large value (such as 2.00), you effectively eliminate the “Bayesian” part, and thus look at OLS. By setting OTHER to a very small value (such as .001), you effectively eliminate the “Vector” part, leaving a univariate model.

```
procedure runtheil
option choice      type      1      symmetric general
option rect        matrix
option vector      mvvector
option real         tightness .1
option real         other     .5
option string       window

local integer time

system(model=canmodel)
variables logcangdp logcandefl logcanm1 logexrate $
         can3mthpcp logusagdp
lags 1 to 4
det constant
specify(tight=tightness,type=type,matr=matrix,mvect=mvvector) other
end(system)

theil(model=canmodel,setup,steps=12,to=2006:4)
estimate(noprint) * 1998:4
theil
do time=1999:1,2006:3
    kalman
    theil
end do time
theil(dump>window=window)
end

*****
@runtheil(tightness=2.0,other=.001>window="Univariate OLS")
@runtheil(tightness=0.1,other=.001>window="Univariate BVAR")
@runtheil(tightness=0.1,other=0.5>window="Simple BVAR")
@runtheil(tightness=2.0,other=1.0>window="OLS VAR")
*
@runtheil(tight=.10,type=general>window="General")
# 1.00    0.50    0.50    0.50    0.50    0.50 $
   0.50    1.00    0.50    0.50    0.50    0.50 $
   0.01    0.01    1.00    0.01    0.01    0.01 $
   0.01    0.01    0.01    1.00    0.01    0.01 $
   0.20    0.20    0.20    0.20    2.00    0.20 $
   0.50    0.50    0.50    0.50    0.50    1.00
```

7.13 Selecting a Model

An Illustration

The selection of a Vector Autoregressive model for forecasting is more difficult than selection of an ARIMA or other univariate time series model. You have to make the following choices:

- which variables to include in the model
- lag length
- prior structure and hyperparameters

To look at a simple illustration, suppose that we are interested in the forecasts of the series, SALES. Previous tests have shown that SALES is closely related to national economic conditions. A first try at a model might be a two variable system with SALES and GDP. However, the model uses a one-step forecast of GDP to compute the two-step forecast of SALES. If the initial forecast of GDP is poor, then the forecasts of SALES derived from it are also likely to be unsatisfactory.

As a second attempt, you consider the addition of variables to improve the GDP forecasts. However, a problem arises as you add more variables: it seems that there are always still more variables which you could add to improve the forecasts of the existing variables. Obviously, you cannot include in the system every variable which *might* have an effect on SALES (through some route). At some point, you must decide which variables to include and which to exclude.

Choosing a Set of Variables

Although you can incorporate quite a few variables in a system through an appropriate choice of prior, it is still a good idea to restrict yourself to relatively small systems (3 to 5 variables) if you want to choose a model to forecast a single series. You should think of a list of candidate series. Usually, there are some obvious choices, such as GDP in the example above.

The instruction **ERRORS** can be very helpful in refining the forecasting model, especially when you have no strong prior information about which series will be important for the prediction of variables in question. There are two pieces of information it provides which are valuable:

The standard errors of forecast

You can use these for a quick comparison of the forecasting abilities of the VAR's from several sets of possible variables. Check the computed standard errors for the variable(s) of greatest interest. The set that produces the lowest values should be regarded as the most promising.

The decomposition of variance

This can indicate which variables you might replace to lower the errors. A variable which explains a very low fraction of the target variable is a good candidate for replacement. If you use the decomposition in this manner, remember to consider the possible effects of changes in the ordering.

Chapter 7: Vector Autoregressions

Using Partial VAR's

Often it is unrealistic to cast the forecasting model in the form of a full VAR. For instance, it is probably reasonable to assume that SALES (of a single company) has no real explanatory power for economy-wide variables, so a GDP equation could omit SALES entirely.

If we return to the situation on the previous page, a possible model would consist of

1. a single equation that explains SALES by lagged SALES, lagged GDP, and perhaps one or two other variables, and
2. a separate VAR system that forecasts GDP and the other variables.

We can combine the SALES equation with the other equations to form the forecasting model. This has several advantages over a VAR system which includes SALES:

- We can estimate the VAR subsystem using all the data available for its variables. This may be quite a bit longer than the data record for SALES.
- We can give special attention to the single equation for SALES, particularly if there is a seasonality problem (see below).

However, it is not possible to put a prior on the coefficients of the SALES equation using **SPECIFY**. Instead, you can use the procedure **@MIXVAR** to estimate this equation separately, then combine it with the VAR for the national variables. **@MIXVAR**, in effect, estimates a single equation using a prior of the same type used in full VAR's.

```
system(model=national)
variables orddurgd ipelect prime
lags 1 to 4
det constant
specify(type=symmetric) 1.0
end(system)
estimate

@mixonvar(define=saleseq,numlags=2) sales
# orddurgd ipelect prime

forecast(model=national+saleseq,results=forecasts,$
  steps=6,from=2007:3)
```

Dealing with Seasonality

Handling variables that exhibit strong seasonality can be somewhat tricky. There are several methods available, but none of them is guaranteed to work because seasonal effects can have very different forms.

- You can include seasonal dummies in the system using the **DETERM** instruction.
- You can model the VAR using seasonal differences (computed using **DIFFERENCE**), and use **MODIFY** and **VREPLACE** to rewrite the model prior to forecasting.

Including a long lag length allows the estimated coefficients to pick up the seasonal effect. However, you cannot use the **DECAY** factor with this method, since it will dampen the values of the important seasonal lags.

There are some types of non-diagonal priors (ones which don't put independent distributions on the lags) which might help with this. These would mainly be implemented using "dummy observations". See Section 7.14.4.

7.14 Using the Kalman Filter

Background

The Kalman filter is a fast, recursive algorithm for estimating and evaluating dynamic linear models. RATS has three instructions or sets of instructions which use it. **DLM** (Dynamic Linear Models), described in Chapter 10, is used for state-space models. These are usually small dynamic models which relate observed data to unobservable “states.” **RLS** (Section 2.1) uses the Kalman filter for Recursive Least Squares for a single linear model. This is generally used for examining the stability of a linear relationship.

For VAR’s, the “states” in the Kalman filter are the coefficients. The instruction **KALMAN** produces sequential estimates of these. **KALMAN** can be used to update coefficient estimates as the data set expands, or to estimate a model where coefficients are allowed to vary with time.

KALMAN works just one observation at a time, so it will generally be used in a loop. Because of the “step-at-a-time” design, you can do sequential calculations based upon each set of coefficients, such as producing forecasts or adjusting the model.

Assumptions

As mentioned above, the Kalman filter, as used by RATS for VAR’s, is a restricted form of the more general Kalman filter. It is specifically designed for estimating coefficients of a linear regression model, allowing for limited types of time-variation.

The following is the model we use. β_t is the vector of coefficients at time t :

- The measurement equation is $y_t = \mathbf{X}_t\beta_t + u_t$, where the variance of u_t is n_t .
- The state vector follows the process $\beta_t = \beta_{t-1} + \mathbf{v}_t$, with $\text{var}(\mathbf{v}_t) = \mathbf{M}_t$.
- u_t and $\mathbf{v}_t$ are independent.
- n_t and $\mathbf{M}_t$ are assumed to be known.

If $\mathbf{M}_t$ is equal to zero, then there is no time-variation.

Updating Formula

If we have an estimate of β_{t-1} using information through $t-1$ (denoted $\beta_{t-1|t-1}$) and its covariance matrix Σ_{t-1} , then the updated estimate given y_t and $\mathbf{X}_t$ is

$$(38) \quad \Sigma_{t|t-1} = \Sigma_{t-1} + \mathbf{M}_t$$

$$(39) \quad \Sigma_t = \Sigma_{t|t-1} - \Sigma_{t|t-1}\mathbf{X}_t'(\mathbf{X}_t\Sigma_{t|t-1}\mathbf{X}_t' + n_t)^{-1}\mathbf{X}_t\Sigma_{t|t-1}$$

$$(40) \quad \beta_{t|t} = \beta_{t-1|t-1} + \Sigma_{t|t-1}\mathbf{X}_t'(\mathbf{X}_t\Sigma_{t|t-1}\mathbf{X}_t' + n_t)^{-1}(y_t - \mathbf{X}_t\beta_{t-1|t-1})$$

Initial Values

To use this algorithm, we need to supply the following information:

- $\beta_{0|0}$, the initial state vector
- Σ_0 , the initial covariance matrix of the states
- n_t , the variance of the measurement equation
- M_t , the variance of the change in the state vector

7.14.1 Sequential Estimation

In the most common use of the Kalman filter in RATS:

- M_t is 0
- n_t is assumed to be the constant σ^2
- Σ_0 and $\beta_{0|0}$ are obtained using **ESTIMATE** through part of the sample.

This is a “fixed coefficients” setup. The true coefficient vector is assumed to stay the same throughout the sample, and the Kalman filter is used to estimate it using samples of increasing size.

If we look at the formulas on the last page, note that if we multiply M_t , Σ_t and n_t by the same constant, then that constant will drop out of the updating formula for β . We can take advantage of this by setting n_t to 1. We will then have a Σ that must be multiplied by σ^2 (which can be estimated separately using the residuals) to get the true covariance matrix. This makes Σ analogous to $(\mathbf{X}'\mathbf{X})^{-1}$ in a least-squares regression. **ESTIMATE** will initialize the Kalman filter and each **KALMAN** instruction will add one more observation to the sample.

This segment of code estimates the model through 1998:4, and does updates over the period from 1999:1 to 2006:3. Forecast performance statistics are compiled over that period using the instruction **THEIL**. Note that the estimates for a VAR without a prior will be the same whether you estimate through the whole period, or first estimate through a subperiod with the Kalman filter being used for the remainder. However, they won't match when the system has a prior. This is because the prior is rescaled using statistics computed using the sample on the **ESTIMATE**.

```
system(model=canmodel)
variables logcangdp logcandefl logcanm1 logexrate $
    can3mthpcp logusagdp
lags 1 to 4
det constant
specify(tightness=.1) .5
end(system)
```

Chapter 7: Vector Autoregressions

```
theil (model=canmodel, setup, steps=12, to=2006:4)

estimate(noprint) * 1998:4

theil
do time=1999:1,2006:3
    kalman
    theil
end do time
theil(dump,window="Forecast Statistics")
```

Recursive Residuals

The **KALMAN** instruction can also be used to compute *recursive residuals*, either for a VAR, or for a general linear regression. However, you will probably find the **RLS** instruction to be much more convenient for single equation models.

To compute recursive residuals with **KALMAN**, you first estimate the model over the first K observations (K = number of regressors), and then use the Kalman filter to run through the remainder of the sample. The $T-K$ recursive residuals have the convenient property that, if the model is the Standard Normal Linear Model, these residuals are independent Normal. They are the series of normalized one-step Kalman Filter forecast errors:

$$(41) \frac{(y_t - \mathbf{X}_t \boldsymbol{\beta}_{t-1|t-1})}{\sqrt{n_t + \mathbf{X}_t (\boldsymbol{\Sigma}_{t-1} + \mathbf{M}_t) \mathbf{X}_t'}}$$

The basic setup for a VAR model would be as follows:

```
system(model=recresvar)
variables list of variables
lags list of lags
deterministic list of deterministic variables
end(system)

compute nreg = number of regressors per equation
estimate(noprint) start start+nreg-1

do time=start+nreg,end
    kalman(rtype=recursive,resids=recresids)
end do time
```

7.14.2 Time-Varying Coefficients

Background

If you permit time-variation, you can't initialize the Kalman filter using **ESTIMATE**. Instead, you need to start by setting presample values for β and Σ , and then filter through the entire sample.

For *very* small models, you can set these as free parameters and optimize using **FIND**. However, this very quickly becomes infeasible. To run the simplest K variable time-varying parameters estimation, you need to set (or estimate) K (initial β) + $K(K+1)/2$ (initial Σ) + $K(K+1)/2$ (**M** matrix) + 1 (measurement equation variance) total parameters. With a mere 10 regressors, this is well over 100.

Fortunately, the initial values for β and Σ serve the same purpose as the mean and variance of a Bayesian prior. This suggests that, at least for VARs, we can just start with a standard prior.

We will be looking at estimating a single equation out of a two-variable VAR. See the example file `TVARYING.RPF` for a working example of the instructions in the next two sections. See also the procedure **TVARSET** which sets up an entire VAR using a structure similar to the one we describe here.

Initializing Coefficients

Use the instruction **ASSOCIATE** to initialize β for an equation.

```
equation kalmeq gdpq
# constant gdpq{1 to 4} fmla{1 to 4}
declare vector b(9)
ewise b(i)=(i==2)
associate kalmeq b
```

This sets a prior mean of 1 for `GDPQ{1}` (the second regressor) and 0 for all other coefficients.

Initializing Variances

You use the instruction **KFSET** to supply Σ_0 and n_t . **KFSET** sets n_t directly using one of the options **SCALE**, **CONSTANT** or **VECTOR**. List the Σ matrices as parameters on **KFSET**. You need to set their initial values at some point before you execute a **KALMAN**.

Most priors for VAR's have diagonal Σ_0 matrices, which cuts down dramatically on the number of "hyperparameters" which govern their values. The main point of the prior is to "tame" the coefficient estimates, as the severe collinearity among the regressors tends to make the data very weak at determining the precise contributions of the different lags.

Chapter 7: Vector Autoregressions

In Doan, Litterman and Sims (1984), we used a more complex variant of the following (continuing from above):

```
linreg gdpq
# constant gdpq{1 to 4}
compute gseesq=.9*%seesq
linreg fmla
# constant fmla{1 to 4}
compute mseesq=.9*%seesq
```

Set up the system. Use GSEESQ as the measurement equation variance

```
system kalmeq
kfset(constant,noscale,likelihood=likely) xxx
# gseesq
tvarying tvx
end(system)
```

This uses a HARMONIC decay with DECAY=.5

The hyperparameters (PIx) below are the following:

PI5 = Overall tightness. This corresponds to TIGHTNESS² on SPECIFY.

PI2 = Relative tightness on other variables. This corresponds to w^2 on the symmetric prior.

PI3 = Relative tightness on the constant. (No correspondence).

PI7 = Relative tightness on time variation.

```
compute pi5 = .20^2 , pi2 = .5^2 , pi3 = 10000.0 , pi7=1.e-8
dim xxx(9,9) tvx(9,9)
compute xxx=%const(0.0)
compute xxx(1,1)=pi5*pi3*gseesq
do i=1,4
  compute decayfac=i^(2*.5)
  compute xxx(i+1,i+1)=pi5*decayfac
  compute xxx(i+5,i+5)=pi5*pi2*decayfac*gseesq/mseesq
end do i
compute tvx=pi7*xxx
```

Executing the Filter

Once you have gone through all the initialization steps, you are ready to begin Kalman filtering. The first step must be done using **KALMAN** with the **STARTUP** option. Remaining steps use just **KALMAN**.

```
do time=1960:1,1998:4
  if time==1960:1
    kalman(startup=time)
  else
    kalman
  end do time
```

Likelihood Function

For the model with random walk coefficient variation, conditional upon previous information, y_t is distributed Normally with

$$\text{Mean} \quad \mathbf{X}_t \beta_{t-1|t-1}$$

$$\text{Variance} \quad \sigma_t^2 = n_t + \mathbf{X}_t (\Sigma_{t-1} + \mathbf{M}_t) \mathbf{X}_t'$$

Ignoring constants, (minus) the log of the likelihood element for this observation is

$$(42) \quad \log \sigma_t^2 + \frac{(y_t - \mathbf{X}_t \beta_{t-1|t-1})^2}{\sigma_t^2}$$

You can construct the sample likelihood by using the `LIKELIHOOD` option matrix on **KFSET**. This has dimensions $2 \times (\text{number of equations})$. The first element in each column is the cumulation of the first terms for an equation and the second element is the cumulation of the second terms.

As mentioned in Doan, Litterman and Sims (1984, p. 10), the filtering procedure above will produce the same series of coefficient estimates if n_t , Σ_0 and $\mathbf{M}_t$ are all multiplied by a constant. Concentrating out this “nuisance parameter” gives a pseudo-likelihood function for equation n which (disregarding constants) is

$$-.5 * (\text{LIKELY}(1,n) + \text{nobs} * \text{LOG}(\text{LIKELY}(2,n) / \text{nobs}))$$

where `LIKELY` is the `LIKELIHOOD` option matrix and `NOBS` is the number of observations.

While we could rely upon this by setting $n_t = 1$, when the constant multiplier for the variances just becomes σ^2 (which can be estimated as `LIKELY(2,n) / nobs`), we find that it's simpler in the case of VAR's to pick a reasonable value for n_t based upon the estimated variance from a preliminary regression. This is because we set Σ_0 directly based upon the standard VAR prior. The prior on the own lags is independent of the scale of the variable involved, and the priors on the lags of other variables are scaled up to take account of the scales. The nuisance parameter in the likelihood then becomes a correction factor to the variances.

7.14.3 More General Time-Variation

If we relax the random walk assumption for time-variation to allow the state vector to follow the process

$$(43) \quad \beta_t = \mathbf{A}_t \beta_{t-1} + \mathbf{v} \quad \text{with} \quad \text{var}(\mathbf{v}_t) = \mathbf{M}_t$$

then the KF updating formulas change to

$$(44) \quad \mathbf{S}_t = \mathbf{A}_t \Sigma_{t-1} \mathbf{A}_t' + \mathbf{M}_t$$

$$(45) \quad \Sigma_t = \mathbf{S}_t - \mathbf{S}_t \mathbf{X}_t' (\mathbf{X}_t \mathbf{S}_t \mathbf{X}_t' + n_t)^{-1} \mathbf{X}_t \mathbf{S}_t$$

$$(46) \quad \beta_{t|t} = \mathbf{A}_t \beta_{t-1|t-1} + \mathbf{S}_t \mathbf{X}_t' (\mathbf{X}_t \mathbf{S}_t \mathbf{X}_t' + n_t)^{-1} (y_t - \mathbf{X}_t \mathbf{A}_t \beta_{t-1|t-1})$$

The only difference between these formulas and those from before ((38) and (40)) is the addition of $\mathbf{A}_t$ to (44) and (46). Since the $\mathbf{A}$'s only apply to Σ_{t-1} and $\beta_{t-1|t-1}$, you can implement this in RATS by using **COMPUTE** to replace

- Σ_{t-1} by $\mathbf{A}_t \Sigma_{t-1} \mathbf{A}_t'$
- $\beta_{t-1|t-1}$ by $\mathbf{A}_t \beta_{t-1|t-1}$

This step needs to be done before each **KALMAN** instruction. To manipulate the coefficient vector, we set up the space for it ourselves using **ASSOCIATE (PERM)**.

Example

If we assume that the coefficient vector “shrinks” back towards the mean, following the process:

$$(47) \quad \beta_t = \pi_8 \beta_{t-1} + (1 - \pi_8)(\text{mean vector}) + \mathbf{v}_t$$

we need to make the following alterations to the example in the preceding section:

```
dec vector b(9) bmean(9)
ewise b(i)=(i==2)
compute bmean=b
associate(perm) kalmeq b
```

and

```
do time=1960:1,1998:4
  if time==1960:1
    kalman(startup=time)
  else {
    compute b=pi8*b+(1-pi8)*bmean
    compute xxx=(pi8^2)*xxx
    kalman
  }
end do time
```

This is a particularly simple example because $\mathbf{A}_t$ is just π_8 times the identity.

This example uses **FIND** to estimate two of the “hyperparameters” used in the example TVARYING.RPF, from Section 7.14.2. Note the following:

- The **IF** inside the **FIND** loop is how you reject illegal values. Just set the value to %NA and skip the remaining calculation.
- We eliminated the **PRINT** on **KALMAN** since that would give us regression output for each test setting of the hyperparameters.

```
compute pi5 = .20^2 , pi2 = .5^2 , pi3 = 10000.
compute pi7 = 0.00000001 , pi8 = 1.00
nonlin pi8 pi7
dec real loglikely
dim xxx(9,9) tvx(9,9)
find(trace) maximum loglikely
  if pi7<0.or.pi8>1.00 {
    compute loglikely=%na
    next
  }
  compute likely=%const(0.0)
  compute b=bmean
  compute xxx=%const(0.0)
  compute xxx(1,1)=pi5*pi3*gseesq
  do i=1,4
    compute decayfac=i^(2*.5)
    compute xxx(i+1,i+1)=pi5*decayfac
    compute xxx(i+5,i+5)=pi5*pi2*decayfac*gseesq/mseesq
  end do i
  compute tvx=pi7*xxx

  do time=1960:1,1985:4
    if time==1960:1
      kalman(start=time)
    else {
      compute b=pi8*b+(1-pi8)*bmean
      compute xxx=(pi8^2)*xxx
      kalman
    }
  end do time
  compute nobs=1985:4-1960:1+1
  compute loglikely=-.5*(likely(1,1)+nobs*log(likely(2,1)/nobs))
end find
```

7.14.4 Dummy Observation Priors

The priors handled using **SPECIFY** have the common property that the prior distributions are independent across coefficients. There are several ways of handling more general types of priors for VARs. You can apply single equation techniques with mixed estimation (described in Section 1.2 of the *Additional Topics* PDF). You can do system-wide estimation using the ideas in the `GIBBSVAR.RPF` program.

If, however, you want to combine a non-diagonal prior with time variation, you need to be able to code the prior information up into the initial mean and covariance matrix used by **KALMAN**. While it's possible to do this with matrix calculations, it will often be simpler to use **KALMAN** to do this. It has a special set of options for implementing “dummy observation” priors. These are priors that can be written in the form:

$$(48) \mathbf{y}_* = \mathbf{X}_* \boldsymbol{\beta} + \mathbf{v}_*$$

The options used on **KALMAN** to adjust the current coefficients and covariance matrix are

***x**=VECTOR of explanatory variables for dummy observation*
***y**=VECTOR of dependent variables for dummy observation*
***v**=VECTOR of equation variances*

When **KALMAN** is applied with these, it doesn't alter the likelihood matrix (page UG-269) or change the entry pointer.

Note that you can't apply this until after you've initialized the coefficients and covariance matrices for the VAR, generally with some form of diagonal prior. If you provide an **X** option, but not a **Y** option, it is assumed that the **Y** values are the ones predicted by current $\boldsymbol{\beta}$ for the input values of **X**. Because the filter “error” is zero, there will be no change to the coefficients, but the covariance matrix of the coefficients will change. If you don't provide a **V** option, the variance for the equations that were set up with **KFSET** are used.

For instance, the following is a dummy observation which says that the original prior coefficients are likely to be good predictors at the first data point (1949:3). This extracts into **XDUMMY** the “X” vector of the VAR model at that data point. This is scaled by a relative value **DUMWGHT**. This, in effect, scales down the equation variances by **DUMWGHT**. If **DUMWGHT** were one, the dummy observation would have the same level of information content as a standard observation. A higher value would mean it would be more precise than a typical observation.

```
compute xdummy = %eqnxvector(%modeleqn(sims9var,1),1949:3)*dumwght  
kalman(x=xdummy)
```

See Sims (1993) for more on the use of dummy observation priors.

8. Simultaneous Equations

We have not emphasized simultaneous equations in designing RATS over the years, leaving the field of the “big model” to specialized software. However, RATS can easily handle models of a more modest size, providing estimation and forecasting techniques.

Estimation Techniques

Formulas and Models

Gauss–Seidel Solver

Forecasting

Multipliers

Simulations

8.1 Estimation

Preliminary Transformations

Most simultaneous equations models require a sizable set of preliminary transformations of the variables: differences, averages, logs, percentage changes, etc. When these involve lags, some data points are lost. Fortunately, you do not need to keep track of this information—RATS handles it by marking as missing any entry which it cannot compute. For instance, in this example, the three constructed variables are actually defined over different periods.

```
set addshort = rs+rs{1}+rs{2}+rs{3}
set diffyd   = yd-yd{1}
set diffrate = r1-rs
```

We would strongly urge you to read the data as you receive it from the original source and to do any needed transformations as part of your program rather than saving the transformed data into a new file. The latter procedure would require someone recording what transformations were done, while the **SET** instructions are, in effect, self-documenting.

Instrument Lists

The estimation instructions (**LINREG**, **AR1**, **SUR**, **NLLS** and **NLSYSTEM**) take their instruments from the list maintained by the instruction **INSTRUMENTS**. With a small model, you can just set the list once and estimate the model. With a larger model, the available instrument list may be too large to be used in its entirety, as it will exhaust your degrees of freedom. Thus, during estimation, you may need to change it from one equation to the next. A combination of **INSTRUMENT** instructions with the **DROP** and **ADD** options will let you make small changes:

instrument(drop)	rw rw{1}	<i>Drops RW and RW{1} from the list</i>
instrument(add)	xz zw	<i>Adds XZ and ZW to the list</i>

Example

The example file `SIMULEST.RPF` demonstrates the various estimation techniques using a small model from Pindyck and Rubinfeld (1998), p. 390. Their model is

$$C_t = \alpha_C + \beta_C Y_t + \gamma_C C_{t-1} + u_{Ct}$$

$$I_t = \alpha_I + \beta_I (Y_{t-1} - Y_{t-2}) + \gamma_I Y_t + \delta_I R_{t-4} + u_{It}$$

$$R_t = \alpha_R + \beta_R Y_t + \gamma_R (Y_t - Y_{t-1}) + \delta_R (M_t - M_{t-1}) + \varepsilon_R (R_{t-1} + R_{t-2}) + u_{Rt}$$

$$Y_t \equiv C_t + I_t + G_t$$

The data set consists of the following variables, quarterly from 1947:1 to 1988:1. However, the estimates are done using a common estimation range of 1950:1 through 1985:4. If RATS were left to determine the range, estimation would start in 1948:1 as the earliest start of the range is determined by the presence of lag 4 on `RATE` in the instrument set.

CONS (<i>C</i>)	= real personal consumption
INVEST (<i>I</i>)	= real gross domestic investment
GNP (<i>Y</i>)	= real GNP net of exports and imports
GOVT (<i>G</i>)	= real government purchases of goods and services
MONEY (<i>M</i>)	= M1
RATE (<i>R</i>)	= 90 day Treasury bill rate

The variables Y_{t-1} (*YDIFF*), M_{t-1} (*MDIFF*) and R_{t-1} (*RSUM*) are created from these:

```
set ydiff = gnp-gnp{1}
set rsum  = rate+rate{1}
set mdiff = m-m{1}
```

Though we will start with the original set of equations, we will end up using a slightly different form for the investment equation, adding an I_{t-1} term to the second equation above. This is done because an estimate of the autocorrelation coefficient on the residuals in the original equation comes in very close to one.

The instrument set for all equations will be the full list of exogenous (*GOVT* and *MDIFF*) and pre-determined variables (lags and functions of lags of the four endogenous variables) that appear in the model. Note that you need to include the *CONSTANT*.

```
instruments constant cons{1} ydiff{1} gnp{1} govt $
           mdiff rsum{1} rate{4}
```

Two-Stage Least Squares

You do two-stage least squares by setting the instrument set (already done) and using **LINREG** with the *INST* option.

```
linreg(inst) cons 1950:1 1985:4
# constant gnp cons{1}
linreg(inst) invest 1950:1 1985:4
# constant ydiff{1} gnp rate{4}
linreg(inst) rate 1950:1 1985:4
# constant gnp ydiff mdiff rsum{1}
```

Autocorrelated Errors

To correct for first-order autocorrelated errors, use **AR1** with the *INST* option. If you choose *METHOD=CORC*, RATS uses Fair's (1970) procedure. This requires, for consistency, that you include as instruments the lags of the variables involved in the regression. The alternative is *METHOD=HILU*, which does not have this requirement.

With the speed of current computers, there is no practical advantage to using the "Cochrane–Orcutt" style iterative procedures. Particularly in simultaneous equations, it is quite possible to have the function being optimized show multiple peaks when graphed against the autoregressive parameter. The search method of *HILU* will

Chapter 8: Simultaneous Equations

make it much more likely that you will find the best estimates. In fact, in the example model, we found a difference between the two techniques, and altered the model as a result.

A common criticism of simultaneous equations models where many of the equations have to be estimated with autocorrelation corrections is that much of the explanatory power of the “model” actually comes not from the structural relationship, but from the autoregressive errors. If your original equations are showing very low Durbin-Watsons, it would probably be a good idea to rethink them as dynamic equations.

The following expands the instrument set as required by Fair’s procedure and re-estimates the investment equation, using both `HILU` and `CORC`.

```
instruments constant cons{1 2} ydiff{1 2} gnp{1} govt{0 1} $
    mdiff{0 1} rate{1 to 5}
ar1(method=hilu,inst) invest 1950:1 1985:4
# constant ydiff{1} gnp rate{4}
ar1(method=corc,inst) invest 1950:1 1985:4
# constant ydiff{1} gnp rate{4}
```

The two **AR1** instructions produce dramatically different results even though they are (theoretically) minimizing the same function. Because of this, all further estimates use an investment equation which includes lagged investment as an explanatory variable. This incorporates the dynamics into the equation rather than tacking it on to the error term. The change in the model requires a change in the instrument set, since `INVEST{1}` is now in the model:

```
instruments constant cons{1} ydiff{1} gnp{1} invest{1} $
    govt mdiff rsum{1} rate{4}
```

Three Stage Least Squares

You can estimate the system by three stage least squares (3SLS) using either **SUR** or **NLSYSTEM** with the `INST` option.

To use **SUR**, you must do the following:

1. Define the equations in the system using a set of **EQUATION** instructions.
2. Set up the instruments list using **INSTRUMENTS**.
3. Estimate the model using **SUR (INST)**.

To use **NLSYSTEM**, you need to

1. Set the list of parameters to be estimated using the instruction **NONLIN**. If you might be doing a good deal of experimenting with your equations, it would be a good idea to set up a separate `PARMSET` for each equation, and then combine them when you estimate.
2. Create formulas for the equations with **FRML** instructions.

3. Set up the instruments list using **INSTRUMENTS**.
4. Estimate the model with **NLSYSTEM(INST)**.

NLSYSTEM and **SUR** should produce almost identical results for any linear model which has no restrictions on the parameters. This uses **SUR**:

```
equation consleq cons
# constant gnp cons{1}
equation investleq invest
# constant invest{1} ydiff{1} gnp rate{4}
equation rateleq rate
# constant gnp ydiff mdiff rsum{1}
*
group prmodel consleq investleq rateleq
sur(inst,model=prmodel,iterations=10) * 1950:1 1985:4
```

and this uses **NLSYSTEM**:

```
nonlin(parmset=structural) c0 c1 c2 i0 i1 i2 i3 i4 r0 r1 r2 r3 r4
frml consnl cons = c0+c1*gnp+c2*cons{1}
frml investnl invest = i0+i1*invest{1}+i2*ydiff{1}+$
i3*gnp+i4*rate{4}
frml ratenl rate = r0+r1*gnp+r2*ydiff+r3*mdiff+r4*rsum{1}
nlsystem(inst,parmset=structural,cvout=v) 1950:1 1985:4 $
consnl investnl ratenl
```

Limited Information Maximum Likelihood (LIML)

LIML actually was used before the now much more popular 2SLS as a single equation (limited information) estimator for simultaneous equation models. For a description of it, see, for instance, Greene (2012). While it has some theoretical advantages over 2SLS, it doesn't appear to provide superior estimates. It can be done with RATS using the procedure **@LIML**. Just as with **LINREG(INST)** for 2SLS, set the instrument list in advance using **INSTRUMENTS**. The syntax is then familiar:

```
@liml depvar start end
# list of explanatory variables

@liml cons 1950:1 1985:4
# constant gnp cons{1}
@liml invest 1950:1 1985:4
# constant invest{1} ydiff{1} gnp rate{4}
@liml rate 1950:1 1985:4
# constant gnp ydiff mdiff rsum{1}
```

Chapter 8: Simultaneous Equations

Full Information Maximum Likelihood (FIML)

RATS has no specific instruction to do FIML for simultaneous equations. However, for small systems, you can implement FIML using the instruction **NLSYSTEM**. This assumes that you have Normal residuals with an unrestricted covariance matrix. You would set everything up as you would for 3SLS, except that, instead of using the **INSTRUMENTS** option, you include the option *JACOBIAN=FRML for the Jacobian*. For the linear model $\mathbf{Y}_i\Gamma = \mathbf{X}_i\mathbf{B} + \mathbf{u}_i$, this will provide a formula which computes $|\Gamma|$. For the example model, the Jacobian reduces to $1 - C1 - I3$, but we write out the full expression.

```
frml jacobian = %det(||1.0 ,0.0 ,0.0 , -c1|$
                    0.0 ,1.0 ,0.0 , -i3|$
                    0.0 ,0.0 ,1.0 ,0.0|$
                    -1.0,-1.0 ,0.0 ,1.0||)
nlsystem(parmset=structural,cvout=v,jacobian=jacobian,itors=200,$
        title="FIML Estimates") 1950:1 1985:4 consnl investnl ratenl
```

PRSETUP.SRC

For the forecasting examples later in the chapter, we have created a “source” file called **PRSETUP.SRC** which estimates the (revised) model using 2SLS, and performs all other operations which must be done before doing forecasts. The **LINREG** instructions on that include the *FRML* option to save the estimated equation in a form usable for forecasting, as described in the next section.

```
instruments constant cons{1} ydiff{1} gnp{1} invest{1} $
    govt mdiff rate rate{4}
linreg(inst,frml=conseq) cons 1950:1 1985:4
# constant gnp cons{1}
linreg(inst,frml=investeq) invest 1950:1 1985:4
# constant invest{1} ydiff{1} gnp rate{4}
linreg(inst,frml=rateeq) rate 1950:1 1985:4
# constant gnp ydiff mdiff rsum{1}
```

We also need to define the accounting and definitional identities needed to close the model:

```
frml(identity) gnpid gnp = cons+invest+govt
frml(identity) rsumid rsum = rate+rate{1}
frml(identity) ydiffid ydiff = gnp-gnp{1}
```

8.2 Setting Up a Model for Forecasting

The MODEL Data Type

A **MODEL** is a special data type in RATS which describes a collection of equations or formulas. For a simultaneous equations model, this is usually created with the instruction **GROUP**. The **MODEL** data type is also used in VAR analysis (Chapter 7)—in that case, the **SYSTEM** instruction is the standard creator.

There is no hard limit on the number of equations in a model. However, if you have a 100+ equation model, you may be better off with a more specialized program.

There are a number of functions and operations you can apply to the **MODEL** data type. For instance, the “+” operator applies to models to combine two smaller models into a larger one. The function `%MODELSIZE(model)` returns the number of equations. `%MODELDEPVARS(model)` returns a **VECTOR** of **INTEGERS** which lists the dependent variables of the equations. `%MODELLABEL(model, i)` returns the label for the dependent variable of equation *i*.

There are quite a few other functions for manipulating the **MODEL** data type, but most of those apply mainly to vector autoregressions or similar types of multivariate time series models.

Creating Formulas

A model which is linear in its endogenous variables can be forecast using methods similar to those for VAR's and ARIMA models. However, our emphasis will be on more general models which could have non-linear aspects, such as mixed logs and levels, or even explicitly non-linear equations. Your model can include linear equations defined using the **DEFINE** option of an instruction like **LINREG** or **BOXJENK**, but any non-linearities have to be handled using **FRMLs**. A **FRML** is a description of a (possibly) non-linear relationship.

There are five ways to create the **FRMLs** for use in simultaneous equations:

- **LINREG (FRML=*formula to define*)** or **AR1 (FRML=*formula to define*)** defines a formula from the estimated equation.
- **FRML** defines formulas with free parameters (specified by **NONLIN**) to be estimated by **NLLS** or **NLSYSTEM**. Or, with fixed parameters, it allows you to input directly a structural equation, if, for instance, you estimated it outside of RATS.
- **FRML (EQUATION=*equation to convert*)** or **FRML (LASTREG)** converts an already estimated (linear) equation to a formula.
- **FRML (IDENTITY)** creates an identity.
- **EQUATION (FRML=*formula to define*)** associates the named formula with the equation being defined. If the equation is estimated (for instance with **SUR** or **LINREG (EQUATION=*equation*)**) the formula is automatically redefined. This association is broken if you ever change the equation in such a way that something changes besides its coefficients.

Chapter 8: Simultaneous Equations

The GROUP Instruction

You put a model together using the instruction **GROUP**. This creates a model from a set of **FRML**'s and **EQUATION**'s. *Each endogenous variable must be on the left side of one and only one of these.* See the discussion below for tips on bringing your model into compliance. If you are doing any type of simulation which adds shocks to the equations, it will also be necessary to list identities last.

The **GROUP** instruction also allows you to specify the series which are to receive the forecasted values. This is done by appending **>>series** to a formula on the list. If you are planning to do simulations with Normally distributed shocks, you will also provide the covariance matrix for those shocks on **GROUP** using the **CV** option.

The models created with **GROUP** can be “added” using the notation *model1+model2*, which appends model 2 to model 1. This allows you to define separate “sector” models and combine them for forecasting.

The model used in the forecasting examples later is created using the following **GROUP** instruction (on the **PRSETUP.SRC** file):

```
group prsmall conseq>>f_cons investeq>>f_invest $
rateeq>>f_rate gnpid>>f_gnp rsumid ydiffid
```

Swapping Endogenous Variables

Because RATS uses a Gauss–Seidel algorithm to solve models, each endogenous variable must be on the left side of one and only one of the equations. If you estimate an equation with an exogenous variable on the left (such as a money demand equation), or a pair of equations with the *same* variable on the left, you must rearrange an equation to make it compatible with **GROUP**. If the equation is linear and is estimated by **LINREG**, **AR1** or **SUR**, you can accomplish this by using **MODIFY** and **VREPLACE** with **SWAP**:

```
linreg(inst,define=mtemp) money
# constant gnp rate
modify mtemp
vreplace money by rate swap
```

This rewrites the equation with **RATE**, rather than **MONEY**, as the dependent variable. You can see the results by using the **PRINT** option on **VREPLACE**.

Definitional Identities

If you have a model with mixed logs and levels, or other transformations of endogenous variables, you need to add to your model identities which define them. Usually, these will just mimic the **SET** instructions used to make the transformations:

```
set rsum = rate+rate{1}
frml(identity) rsumid rsum = rate+rate{1}
```

If you don't do this, the forecasting procedure will treat **RSUM** and **RATE** as unrelated variables.

8.3 Solving Models: Various Topics

Gauss–Seidel Algorithm

RATS solves models using the Gauss–Seidel algorithm. This is a simple, and usually effective, algorithm which requires no complex calculations. Simply put, it solves

$$\begin{aligned}y_1 &= f_1(y_2, y_3, \dots, y_n) + u_1 \\y_2 &= f_2(y_1, y_3, \dots, y_n) + u_2 \\&\vdots \\y_n &= f_n(y_1, y_2, \dots, y_{n-1}) + u_n\end{aligned}$$

by setting y_1 using the first equation, using initial values for the other y 's, then setting y_2 using the second equation, using the initial values of y_3 to y_n and the just computed value for y_1 , etc. From this, it's clear why each endogenous variable can be on the left side of one and only one equation.

This process continues until convergence, which means that the y 's change little from one iteration to the next. Convergence is not guaranteed, however, even if a solution exists, and even if the system is linear. For instance, the system below won't converge as written. It *will* converge if the equations are renormalized, and it will also converge with the use of a damping factor of .5 or less—see the next page.

$$\begin{aligned}y_1 &= y_2 + 5 \\y_2 &= -2y_1 + 3\end{aligned}$$

Note the following:

- The initial values for the y 's come from the data series themselves, if available; otherwise the last historic value is taken. In multiple step forecasts, the step k solution initializes step $k+1$.
- Convergence is considered to be achieved when all endogenous variables satisfy

$$\min \left(\left| \frac{y_k - y_k^0}{y_k^0} \right|, |y_k - y_k^0| \right) < \varepsilon$$

where y_k^0 is the value from the previous iteration. That is, the percentage change is less than ε for larger values, and the absolute change is less than ε for smaller ones (less than 1.0). You control ε with the CVCRIT option on the instructions: **FORECAST**, **STEPS**, **THEIL** or **SIMULATE**.

These instructions also have the options `ITERS=number of iterations`, and `DAMP=damping factor` which apply to the Gauss–Seidel solution procedure.

Chapter 8: Simultaneous Equations

Achieving Convergence

If you have problems getting convergence, there are several adjustments which you can make. Obviously, if a (non-linear) system has no solution, nothing will work.

- Increase the `ITER` option on the **FORECAST**, **STEPS**, **THEIL**, or **SIMULATE** instruction from the default of 50. This can help if the procedure is converging, but just moving very slowly.
- Reorder the equations. This is probably a hit-or-miss strategy, since it may be hard to identify the source of the poor behavior.
- Use the `DAMP` option. `DAMP=λ` causes RATS to use

$$y_k = (1 - \lambda)y_k^0 + \lambda(f_k(\dots) + u)$$

a weighted average of the old and new values. The default is $\lambda=1$. Taking a value of $\lambda < 1$ will slow the solution process, but a small enough value of λ will eliminate the explosive behavior of systems like the one on the previous page.

If you're getting NA's for some or all of your forecasts, it is almost certainly because you have some exogenous variables which aren't defined during your forecast period. See the "Out-of-Sample Forecasts" segment below.

Out-of-Sample Forecasts

To compute forecasts out-of-sample, you need to make some assumptions about the future of the exogenous variables. There are two ways to handle these:

- Add to the model simple equations, such as autoregressions, to forecast them.
- Set paths for them over the forecast period, using **SET** for systematic paths and **DATA** otherwise.

You can, of course, combine these two, treating different variables differently. This short example (`SIMULFORE.RPF`) uses the first method, while the add factor example later in this section uses the second. Our example has the slight complication that `MDIFF`, not `MONEY` itself, is needed in the interest rate equation, thus requiring an additional identity to define this to the model. (See "`PRSETUP.SRC`" for a description of the setup file).

```
source prsetup.src
linreg(define=moneyeq) m
# constant m{1 2}
linreg(define=govteq) govt
# constant govt{1 2}
frml(identity) mdiffid mdiff = m-m{1}
group exogs moneyeq govt eq mdiffid
forecast(model=exogs+prsmall,print,from=1986:1,to=1990:4)
```

Multipliers

You compute multipliers by running the model with and without a set of changes to the paths of the exogenous variables. Unless the model is linear (this one is), the computed multipliers apply only to the particular context—they will change when you solve the model at a different period, or with different assumptions on the other exogenous variables.

You must compute the two sets of forecasts and subtract them to get the multipliers, which can involve some rather complicated bookkeeping. The easiest way to do this is with the **RESULTS** option on **FORECAST** which creates a **VECTOR** of **SERIES** for the results. The **DO** loop in this example loops over the number of equations in the model and creates labels of “M_dependent variable” for the multipliers.

This is example `SIMULMULT.RPF`.

```
source prsetup.src
smp1 1984:1 1985:4
forecast(model=prsmall,results=base)
compute govt(1984:1)=govt(1984:1)+1.0      increase govt
forecast(model=prsmall,results=mults)
compute govt(1984:1)=govt(1984:1)-1.0      reset govt

do i=1,%rows(mults)
    set mults(i) = (mults(i)-base(i))/2.0
    labels mults(i)
    # # "M_"+"%modellabel(prsmall,i)
end do i

print(picture="*.###") / mults
```

Historical Simulations

You can use the instruction **THEIL** to compute statistics on the accuracy of the model using the historical data. Because this solves the model for each time period you loop over, it can take a long time with a big model.

This looks at forecasts for 1 to 4 steps ahead over the period from 1982:1 to 1985:4. The `TO=1985:4` on **THEIL(SETUP,...)** indicates the end of the historical data.

This is example `SIMULTHEIL.RPF`.

```
source prsetup.src
theil(setup,model=prsmall,steps=4,to=1985:4)
do time=1982:1,1985:4
    theil time
end do time
theil(dump>window="Forecast Performance")
```

Chapter 8: Simultaneous Equations

Add Factors

Add factors adjust the solution of the model by altering one or more intercepts, or, equivalently, adding a non-zero error term to one or more equations. You can implement add factoring in RATS using either the **SHOCKS** or **INPUT** options of **FORECAST**. **INPUT** (which uses a supplementary cards) is more convenient for small models and **SHOCKS** (which uses a **VECTOR**) is better for larger ones. If you need to add factor more than one period, use the **PATHS** option.

This example (**SIMULADD.RPF**) does three forecasts, bumping the first period interest rate by 0.5 and 1.0 points for the second and third. (Adding x to the interest rate (3rd) equation adds x to R_t , since R_t appears only in that equation. Generally, the effect of a change will not be quite as predictable).

```
source prsetup.src
smpl 1986:1 1988:4
set m      1986:1 1988:4 = m{1}*1.01
set govt   1986:1 1988:4 = govt{1}*1.008
set mdiff  1986:1 1988:4 = m-m{1}
declare real fudge
dofor fudge = 0.0 0.5 1.0
    forecast(model=prsmall,input,print)
    # 0.0 0.0 fudge
end dofor
```

A similar setup using **SHOCKS** would use (following the **SET MDIFF**)

```
declare vector shocks(3)
compute shocks=%const(0.0)
declare real fudge
dofor fudge = 0.0 0.5 1.0
    compute shocks(3)=fudge
    forecast(model=prsmall,shocks=shocks,print)
end dofor
```

9. ARCH and GARCH Models

This chapter describes ARCH (AutoRegressive Conditional Heteroscedasticity) and related models. We look at the basic univariate models and their variations, and describe how to estimate these using the built-in **GARCH** instruction and Wizard, as well as the more general **MAXIMIZE** instruction. We then discuss various multivariate models.

The emphasis here is on estimation of the models by maximum likelihood. Chapter 16, uses Bayesian methods, in particular, see examples `GARCHGIBBS.RPF` (page UG-537) and `GARCHIMPORT.RPF` (page UG-525). In addition, example `SWARCH.RPF` (page UG-390) estimates a “switching” ARCH, or SWARCH, model.

While this is a long chapter for describing what is, largely, a single instruction, it still can’t cover the breadth of such an important topic. If you are deeply interested in these models, we would suggest that you get our *ARCH/GARCH and Volatility Models* course. For more information on that, see

https://estima.com/courses_completed.shtml

Univariate Models
The GARCH Instruction
Using MAXIMIZE
Forecasts
Multivariate Models

9.1 ARCH and Related Models

ARCH (AutoRegressive Conditional Heteroscedasticity), GARCH, and related models have become very popular. Thanks to the flexible maximum likelihood estimation capabilities of RATS, it has proven to be an excellent tool for estimating standard ARCH and GARCH models, as well as the many complex variants on them.

The instruction **GARCH** can handle most of the more standard ARCH and GARCH models. However, because refinements on these models are being developed, it's still useful to know how to do the calculations using more basic instructions.

In this section, we briefly introduce the basic theory underlying these models, and demonstrate the standard programming set ups required to estimate them. For additional examples, see, for instance, Enders (2014) or Martin, Hurn and Harris(2013). Campbell, Lo and MacKinlay (1997) has an extensive discussion of the theory, particularly for univariate models and Tsay (2010) has quite a few examples of less standard models.

Autoregressive conditional heteroscedasticity (ARCH) was proposed by Engle (1982) to explain the tendency of large residuals to cluster together. A very general form for an ARCH or GARCH model is

(1) $y_t = X_t\beta + u_t$, where the residual is mean zero and serially uncorrelated,

(2) $h_t \equiv \text{var}(u_t) = h(u_{t-1}, u_{t-2}, \dots, u_{t-q}, h_{t-1}, h_{t-2}, \dots, h_{t-p}, X_t, X_{t-1}, \dots, X_{t-k}, \alpha)$

where α is a set of unknown parameters for the h function. Assuming the residuals are Normally distributed, the log likelihood term for entry t (omitting additive constants, although they're included in output from **GARCH**) takes the form

$$(3) \quad -\frac{1}{2} \log h_t - \frac{1}{2} (y_t - X_t\beta)^2 / h_t$$

where h_t is $h(\cdot)$ evaluated at t . This has a standard regression model for the mean of the series, and a separate model for the variance. We'll refer to these as the "mean model" and "variance model." These can be, and often are, analyzed separately. The parameters for (1), for instance, can be estimated consistently by least squares—they just won't be efficient if they don't take into account the heteroscedasticity.

Alternative Models

ARCH (or most commonly) GARCH models aren't the only models proposed to explain volatility "clustering". The stochastic volatility model (page UG-340) is, at first glance, similar to the GARCH model, but, unlike (2), where the variance is a computable function of past data, in the stochastic volatility model, it's unobservable, which creates certain technical problems. By contrast, the ICSS algorithm of Inclan and Tiao(1993) (@**ICSS** procedure) doesn't "model" variance changes but attempts to identify discrete changes in the variance from the data. However, this doesn't permit any type of fore-

casting of volatility since the variance changes can only be identified from observed data.

In ICSS, the assumption is that there are (or at least may be) different variance “regimes” in the data set with no real connection among them. There are other models which also are based upon volatility regimes, but with some form of “switching” models for the changes. either unobserved variance regimes (such as Markov Switching models) or observable regimes (threshold models).

True Volatility is Unobservable

One (unfortunate) fact that affects not just GARCH but all the alternative models is the true volatility isn’t observable—we’re trying to infer it from the observable data. For instance, the h_t in (2) is just the model’s estimate of the variance at time t , not the actual volatility. A common belief is that the squared residuals can be used to substitute for volatility, but, while, if the model is correct, that will have the correct expected value, it’s a (very) poor proxy for the truth (see Andersen and Bollerslev(1998)).

Because the true value isn’t observable, it makes it much more difficult to evaluate models. Other things being equal, a higher log likelihood means that the model explains the data (the y) better. But explaining y better doesn’t necessarily mean that the model has done a better job of capturing the volatility. For instance, in almost any type of GARCH model, you will get a much better log likelihood from using a Student- t error process rather than a conditionally normal one. And the difference is usually due to a handful of “outliers”, which provide a natural experiment for the GARCH model—if GARCH is a correct description of the data-generating process, then the outliers should be followed by a period of high volatility. With a t error process, the estimates of the volatility following the outlier will generally be lower than they are with the normal errors, as the t likelihood doesn’t penalize large standardized residuals as severely.

In part because true volatility isn’t observable, and in part because it is (by definition) a positive number, evaluating volatility forecasts is much more difficult than evaluating forecasts of the data itself. Andersen and Bollerslev used high-frequency data to provide a second source for volatility information, which often isn’t available. Patton(2011) shows that some commonly used forecast error statistics (such as the mean absolute error or MAE) fail badly when used for evaluating volatility—a model which systematically underestimates volatility will generate smaller MAE statistics.

9.2 Data Preparation/Preliminaries

Check Your Data!

Before you do anything, check your data. More time has been lost fitting GARCH models to bad data than to anything else. What are you looking for? Data which are have long flat spots with occasional changes (generally price-controlled or otherwise managed data) will not admit a GARCH model—in fact, that’s the exact opposite of how a GARCH process works, as a big change is followed by a period with no change at all. If controls were removed at some point in your data range, you may be able to restrict your attention to the later part of the data set, but there is no way to include the controlled stretch in a single model.

Assets prices whose trading is subject to “circuit-breakers” or other types of interventions can break the expected behavior of a GARCH model. You may need to end up either “dummying” out such periods, or somehow removing a range of data from the data set.

Prices on thinly traded assets again won’t follow a standard GARCH process. There are much more complicated models which can jointly model both frequency of trades and prices, but for a standard GARCH, you may need to go to a coarser frequency (such as weekly rather than daily).

You may have daily data with non-trading days. You can either use the irregular data (so Monday follows Thursday if Friday is a non-trading day) or you can make the non-trading day’s price equal to the preceding day (so Friday’s return is zero). In practice, both are used. If there are quite a few non-trading days, you might also find that going to weekly data (with weeks ending on, for instance, Wednesday) is a better approach.

Finally, a standard GARCH model is a description of a “stationary” process. If the data look quite different at the start of the data set than at the end, you may need to think about whether a single model will do what you want.

Data Scaling

The most common application of GARCH models is to a series (or set of series) of asset returns. If these are computed as log returns over a relatively short period of time (such as daily), the scale and spread of the data is quite small. For the numerical reasons described in “Convergence and Scale of Parameters” on page UG–110, it’s a good idea to scale those up by a factor of 100, as in

```
set dlogdm = 100*log(dm/dm{1})
```

Non-linear parameters which measure a variance are scaled up by 10000 by this, which generally brings them up to a more manageable magnitude.

Serial Correlation in the Residuals

Since (by far) the most common application of GARCH models is to asset returns, the “mean model” is often largely ignored. The default for the RATS GARCH model is just a constant (fixed non-zero mean) for each series. And in most applications to asset returns, that is probably fine. However, increasingly, GARCH models are being applied to “non-traditional” data, such as macroeconomic data, where the data being serially uncorrelated can’t be assumed. If you have a model such as that, you need to choose a mean model which does at least a reasonable job of producing serially uncorrelated residuals in (1); if you don’t—you might find it very difficult to get a GARCH model to fit properly. The one difficulty with this is that the standard tests for serial correlation generally assume away the possibility of the higher-order dependence shown by the GARCH model, so if you pick an AR or ARMA model (for a single variable) or a VAR for a multivariate system based upon standard methods, you might pick the “wrong” model. For univariate diagnostics for serial correlation, you can use the West-Cho test (page UG–82) rather than the more standard Q test.

The best strategy is generally to use standard methods with a fairly “tight” criterion to select the mean model (for instance, BIC rather than AIC) and then test the results of the GARCH model to see if there is any correctable serial correlation.

Testing for ARCH/GARCH Effects

@ARCHTEST for univariate ARCH (or GARCH) is described on page UG–80. The extension of that to multiple series is given by **@MVARCHTEST**. Note that both of these assume they are being given mean zero serially uncorrelated data, that is, the residuals from (1), *not* the raw data.

Both these tests have a null of absence of ARCH. However, don’t assume that because your test rejects lack of ARCH, that you can get a working GARCH model. All the test has shown is that the data show a certain clustering of large residuals which is consistent with ARCH or GARCH, not that it was created by an ARCH or GARCH process. Any of the alternative models described on page UG–286 will also create a situation where you would almost certainly reject homoscedasticity in favor of “ARCH/GARCH”. In fact, even a single variance change of the type for which the Goldfeld-Quandt test (page UG–79) is designed will (if the change is large enough) trigger a significant value for an ARCH test.

9.3 Standard Univariate Models

GARCHUV.RPF example

We'll use the example file `GARCHUV.RPF` to demonstrate the different forms of standard univariate GARCH models. Note that, in practice, you wouldn't use all of these (or even all but one or two) in a single analysis—this is just for illustration. `GARCHUV.RPF` is based upon an example from Verbeek (2008). All the examples look at the returns on the Deutsche Mark-Dollar exchange rate, which (as described on page UG-288) is scaled by 100.

```
open data garch.asc
data(format=free,org=columns) 1 1867 bp cd dm jy sf
set dlogdm = 100*log(dm/dm{1})
```

ARCH Models

The simplest “variance model” is the $\text{ARCH}(q)$ model:

$$(4) \quad h_t = c_0 + \alpha_1 u_{t-1}^2 + \alpha_2 u_{t-2}^2 + \dots + \alpha_q u_{t-q}^2$$

Assume that the dependent variable is the daily appreciation of the Deutsche Mark vs the dollar, and the model to be estimated is an $\text{ARCH}(6)$. You can estimate this using **GARCH** by:

```
garch(q=6) / dlogdm
```

The **GARCH** instruction is designed to handle univariate or multivariate models, so the syntax has the estimation range first (hence the `/`) followed by the list of dependent variable series.

GARCH Models

Even Engle in his original paper ran into some problems implementing the ARCH model with real world data. Volatility seems to have a different type of persistence than can be explained by an $\text{ARCH}(1)$ model, where both the short- and long-term clustering depend upon a single parameter. It's possible to add additional lags of the squared variance, as was done in the example above, but unconstrained estimates of the lags will often show some negative coefficients. Engle used four lags, but had to constrain the shape of the lag distribution.

Bollerslev (1986) proposed the GARCH model as an alternative. In a GARCH model, the variance term depends upon the lagged variances as well as the lagged (squared) residuals. This allows for a more flexible persistence in volatility with a relatively small number of parameters. The variance model for the standard $\text{GARCH}(p,q)$ is

$$(5) \quad h_t = c_0 + \alpha_1 u_{t-1}^2 + \alpha_2 u_{t-2}^2 + \dots + \alpha_q u_{t-q}^2 + b_1 h_{t-1} + b_2 h_{t-2} + \dots + b_p h_{t-p}$$

To estimate a $\text{GARCH}(1,1)$ on the same series as above, use

```
garch(p=1,q=1) / dlogdm
```

9.3.1 Exponential Models

Nelson (1991) introduced a number of refinements on the GARCH model, which we'll examine separately in this section, as you can choose them separately on the **GARCH** instruction. The first was to model the log of the variance, rather than the level. This avoids any problem that could arise because of negative coefficients in standard ARCH and GARCH models, and gives rise to an EGARCH model. The form for the variance that we use in RATS is

$$(6) \quad \log h_t = c_0 + a_1 |u_{t-1}| / \sqrt{h_{t-1}} + a_2 |u_{t-2}| / \sqrt{h_{t-2}} + \dots + a_q |u_{t-q}| / \sqrt{h_{t-q}} + b_1 \log h_{t-1} + \dots + b_p \log h_{t-p}$$

A few things to note about this. First, it doesn't include "asymmetry" effects, where positive and negative u 's have different effects. That will be described in Section 9.3.4. Nelson also used an alternative distribution to the normal, which is discussed in Section 9.3.3. Because the lagged residuals come into the model in "standardized" form, the variance persistence of the EGARCH comes solely through the b coefficients.

EGARCH models are often parameterized using

$$(7) \quad |u_{t-i}| / \sqrt{h_{t-i}} - E(|u_{t-i}| / \sqrt{h_{t-i}})$$

which, by definition, has mean 0. However, the expected value can be a complicated function of the distribution, and has no effect on the volatility as it just washes into the the constant term in (6).

To estimate a GARCH(1,1) with an exponential variance model, use

garch (p=1 ,q=1 ,exp) / dlogdm

9.3.2 IGARCH

We can rearrange the GARCH(1,1) model as

$$(8) \quad h_t = c_0 + a_1 (u_{t-1}^2 - h_{t-1}) + (a_1 + b_1) h_{t-1}$$

where (if the model is correct) the a_1 term now has expected value zero. So $a_1 + b_1$ is the proper measure of the persistence of volatility. In the IGARCH model (Nelson (1990)), this is constrained to be one. In general, in (5)

$$(9) \quad a_1 + a_2 + \dots + a_q + b_1 + \dots + b_p = 1$$

To estimate an IGARCH model, include the I=NODRIFT or I=DRIFT option on **GARCH**. I=NODRIFT imposes the constraint (9) and zeros out the constant (c_0) in the variance equation. I=DRIFT also imposes (9) but leaves the constant free. This can be added as an option to ARCH, GARCH or EGARCH models (for EGARCH, it constrains only the sum of the b 's). To estimate an IGARCH (1,1) model (without variance drift), use

garch (p=1 ,q=1 ,i=nodrift) / dlogdm

9.3.3 Fat-Tailed Distributions

In many cases the assumption of conditional normality cannot be maintained. RATS offers two choices as alternatives: the Student- t and the Generalized Error Distribution (GED). You choose which to use with the `DISTRI`B option, which has choices `NORMAL` (the default), `T` and `GED`. Both the Student- t and GED have a shape parameter, which determines their kurtosis, and a scale parameter, which determines the variance given the shape parameter. Since ARCH or GARCH models a variance, the t or GED density selects the scale to give that variance. For instance, the GED has density function

$$(10) \quad \frac{\exp\left[-|x|/b\right]^{2/c}/2}{b\left(2^{c/2+1}\right)\Gamma(1+c/2)}$$

where c is the shape and b the scale. The variance, given b and c , is

$$(11) \quad 2^c b^2 \Gamma(3c/2) / \Gamma(c/2)$$

Given a variance from the model and the value of the shape parameter, (11) is solved for b , and (the log of) (10) is used in maximum likelihood estimation.

Both densities have the normal as a special case. For the GED, that's with shape parameter $c=1$; for the t , it's with infinite degrees of freedom. The GED family includes both fat-tailed densities ($c>1$) and thin-tailed ones ($c<1$). The t with finite degrees of freedom is always fatter-tailed than the normal. Note that, because the t has no variance if the degrees of freedom are less than or equal to 2, the density function (as rescaled by RATS) won't be defined for those values.

You can either set a value for the shape parameter using the option `SHAPE`, or have it estimated, which is the default if you don't include `SHAPE`. Continuing with our example, the following two instructions would estimate a GARCH(1,1) with a t distribution for the errors. In the first, the degrees of freedom are pegged at 5, while in the second, they're estimated.

```
garch(p=1,q=1,distrib=t,shape=5) / dlogdm  
garch(p=1,q=1,distrib=t) / dlogdm
```

This can be combined with any of the other models and options. Note that while Nelson(1991) used the GED with his proposed EGARCH model, there is no reason you have to use it, and, in fact, most empirical work done with EGARCH uses the Normal and t distributions used with other model types.

As noted on page UG-287, while the t distribution *has* to produce a higher log likelihood than the normal (as the normal is a special case), in practice, it usually is *substantially* higher. However, that doesn't necessarily mean that it gives a superior estimate of the volatility, which is why it isn't used as much as you would expect.

9.3.4 Asymmetry

It has long been recognized that equity returns exhibit asymmetrical conditional variance behavior, that is, that positive values of the residuals have a different effect than negative ones. That won't be captured by any of the models so far, since the residual always enters the variance as a square or absolute value. The EGARCH model from Nelson (1991) adds an extra term to (6) to provide for this; an EGARCH(1,1) would have the variance evolving according to

$$(12) \quad \log h_t = c_0 + a_1 |u_{t-1}| / \sqrt{h_{t-1}} + b_1 \log h_{t-1} + d_1 u_{t-1} / \sqrt{h_{t-1}}$$

There are many equivalent ways to introduce the asymmetric effect into the model—we choose this one to maintain a similar form to that used in other models you can estimate with **GARCH**. Note that, with this parameterization, a *negative* value of d_1 means that negative residuals tend to produce higher variances in the immediate future. (This sign convention is used only with the EGARCH).

An analogous change to the standard GARCH model was proposed by Glosten, et. al. (1993) and is known as GJR, after the originators. Again, looking at a GARCH(1,1), the GJR variance model is

$$(13) \quad h_t = c_0 + a_1 u_{t-1}^2 + b_1 h_{t-1} + d_1 u_{t-1}^2 I_{u_{t-1} < 0} (u_{t-1})$$

where I is an indicator function, in this case, for $u < 0$. With this formulation, a *positive* value of d_1 means negative residuals tend to increase the variance more than positive ones.

To estimate with asymmetric effects, just add the **ASYMMETRIC** option:

```
garch(p=1,q=1,exp,asymmetric) / dlogdm
garch(p=1,q=1,asymmetric) / dlogdm
```

9.3.5 The Mean Model

While the need for a proper choice of the mean model was discussed up front (page UG–289), the examples so far have used only an intercept. There is one model for the mean that is even simpler than this, where the dependent variable is already thought to be a mean zero process. To estimate the model under that assumption, use the option **NOMEAN**.

For a less trivial model, you can include the option **REGRESSORS** and add a supplementary card with the explanatory variables for the mean. Include **CONSTANT** if it's one of the explanatory variables. For instance, the following will do a GARCH(1,1) with an AR(1) process for the mean:

```
garch(p=1,q=1,regressors) / dlogdm
# constant dlogdm{1}
```

If you want an ARMA model for the mean, use lags of **%MVGAVGE** in your list of regressors for any moving average terms. For instance, the same model as above, but with an ARMA(1,1) is:

Chapter 9: ARCH and GARCH

```
garch(p=1,q=1,regressors) / dlogdm
# constant dlogdm{1} %mvgavge{1}
```

With the ARMA model, the pre-sample values for the moving average process are set to zero. The residuals are generated recursively conditioned on that. Note that an ARMA model is usually a poor choice—if the autocorrelation in the data is fairly weak (which is usually the case), the AR and MA coefficients will not be well-determined by the model, and you are likely to see (unreasonably) large coefficients for both with similarly large standard errors.

You can also use the option `EQUATION` to indicate both the dependent variable and the mean model. For instance,

```
linreg(define=ar1) dlogdm
# constant dlogdm{1}
garch(p=1,q=1,equation=ar1)
```

estimates an AR(1) model first by least squares, then with GARCH errors.

ARCH-M, GARCH-M

ARCH-M models (Engle, Lilien, Robins, 1987) generalize the ARCH model by allowing a function of the variance to enter the regression function itself. The most common form of this uses the variance itself as a regressor. You can estimate an ARCH-M or GARCH-M model by using the special name `%GARCHV` among the regressors. This refers to the variance generated by the model. It behaves like a series, so you can include lags of it with the standard `{...}` notation. To add the current variance to an AR(1)-GARCH(1,1) model:

```
garch(p=1,q=1,regressors) / dlogdm
# constant dlogdm{1} %garchv
```

It's important to note that if there is no “GARCH” there is no “M”—if the GARCH effect is weak, the GARCH variance is nearly constant so the “M” effect isn't identified.

9.3.6 ARCH-X, GARCH-X Models

If you add other explanatory variables to your *variance* model besides those formed from lagged residuals and lagged variances, you get an ARCH-X or GARCH-X model. To do this, use the option `XREGRESSORS` and add a supplementary card listing the extra explanatory variables in the variance equation. (*Don't* include `CONSTANT`, which will always be in the variance model). If you need both regressors for the mean model and for the variance model, put the card for the mean model regressors first. Suppose you have a dummy variable for Mondays (note that there *isn't* one in this data set), and you think that the variance is systematically different on Mondays. This adds the “Monday” effect to the variance model:

```
garch(p=1,q=1,regressors,xreg) / dlogdm
# constant dlogdm{1}
# monday
```

9.3.7 Estimation

The models are all estimated using maximum likelihood. Because the Normal is a special case of both the t and the GED, the log likelihood used by **GARCH** includes all the integrating constants so the function values produced under different choices for the distribution will be comparable.

Pre-sample Variance

These types of models have a technical issue that needs to be addressed in estimating them: the variance is generated recursively and the pre-sample values of it are unobservable, and, unfortunately, the estimates can be somewhat sensitive to the choice made for these.

For the basic ARCH model, it's possible to start the estimation at entry $q+1$, as the variance depends only upon the lagged u 's, which are (in general) computable given the set of parameters. If you wish to do this, use the option `CONDITION`. However, that isn't available for a GARCH model, because the required lag of h can't be computed no matter how many data points are used.

By default, the **GARCH** instruction handles both the pre-sample lagged squared u 's and the lagged variances by setting them to the unconditional estimates: the variance from the least squares estimate of the mean model. If you wish to use a different value, you can include the option `PRESAMPLE=pre-sample variance`. Because of this method of handling the pre-sample information, any ARCH or GARCH model with the same mean model can be estimated over the same range, that is, the only constraint on the range is from the dependent variable and any regressors or "x"-regressors.

Estimation Methods

The **GARCH** instruction provides for several estimation algorithms, which you select with the `METHOD` option. For final estimation the default is `BFGS` while `BHHH` is a common choice, since the model is in the form that can be estimated using that (Section 4.2). However, `BHHH` isn't necessarily a good choice as the main estimation method, as the curvature far from the maximum can be quite different from that calculated by `BHHH`. If you want the `BHHH` covariance matrix, it's better to use `BFGS` as a preliminary method, then switch to `BHHH` for a final iteration once it's converged.

The derivative-free methods (Section 4.3) are very useful in *multivariate* GARCH models, but aren't quite as important with the univariate ones. However, if you're having convergence problems with `BFGS` or `BHHH`, you can use the `PMETHOD` option with `SIMPLEX` (most common) or `GENETIC` and combined with `PITERS` to refine the initial guesses. (The "annealing" algorithms will almost never be needed).

An alternative to using one of the non-normal distributions (Section 9.3.3) is to obtain quasi-Maximum Likelihood estimates (page UG-116) by using the log likelihood function from the conditional normal specification. As this is not the true likelihood, it requires calculating a robust estimate of the covariance of the parameter estimates. `METHOD=BFGS` with the `ROBUSTERRORS` option produces the required calculation:

Chapter 9: ARCH and GARCH

`garch (method=bfgs, robust, other options) ...`

Under fairly weak conditions, the resulting estimates are consistent even when the conditional distribution of the residuals is non-normal (Bollerslev and Wooldridge, 1992). The covariance matrix calculated by RATS is not numerically identical to the one referred to in that paper, but is asymptotically the same..

Output

The output from a **GARCH** instruction is similar to that from a **LINREG**, but with fewer summary statistics; for instance, no R^2 since it isn't minimizing the sum of squared residuals.

```
GARCH Model - Estimation by BFGS
Convergence in    18 Iterations. Final criterion was  0.0000093 <=  0.0000100

Dependent Variable DLOGDM
Usable Observations                      1866
Log Likelihood                      -2068.1265

Variable                      Coeff      Std Error      T-Stat      Signif
*****
1.  Mean (DLOGDM)              -0.020635802  0.015767968   -1.30872   0.19063035
2.  C                          0.016180175  0.005181947    3.12241   0.00179375
3.  A                          0.110125419  0.015859223    6.94394   0.00000000
4.  B                          0.868369919  0.018692362   46.45587   0.00000000
```

The coefficients are listed in the order:

1. Mean model parameters in the standard order for regressions. If you use the default mean model, the coefficient will be labeled as `Mean(series)`. If you used the `REGRESSORS` option, these will be labeled as they would for any other regression.
2. Constant in the variance equation (labeled as `C`).
3. “ARCH” (lagged squared residuals) parameters, in increasing order of lag (labeled as `A`, or `A(lag)` if you used more than one lag)
4. “GARCH” (lagged variance) parameters, if any, in increasing order of lag (labeled as `B` or `B(lag)`)
5. Asymmetry coefficients if any (labeled as `D` or `D(lag)`)
6. `XREGRESSORS` variables, labeled as they would in any other regression.

This is also the order in which the coefficients appear in the `%BETA` vector defined by **GARCH**, and is the order for the `INITIAL` vector, if you want to use that.

RESIDS and HSERIES options and Standardized Residuals

For a univariate model, you can get the residuals and series of variance estimates using the RESIDS and HSERIES options. (Different options are needed for multivariate models). The following, for instance, will produce in the series USTD the *standardized residuals* from a GARCH(1,1).

```
garch(p=1,q=1,resids=u,hseries=h) / dlogdm
set ustd gstart gend = u/sqrt(h)
```

GARCH also defines the %RESIDS series when you estimate a univariate model.

If you need an “M” term (page UG–294) other than the current variance, you can use the combination of HSERIES and HADJUST options to generate a new series. You first have to initialize the series (either to zero or perhaps to a series generated from a preliminary GARCH estimate), and use HADJUST to set the current value of that series. For instance, the following uses the square root of the variance which is generated “on the fly” by the **GARCH** instruction by taking the square root of current HS (the series into which the variances are being saved by HSERIES) and saving it into the current entry of the series SQRTH:

```
set sqrth = 0.0
garch(p=1,q=1,hseries=hs,hadjust=(sqrth=sqrt(hs)), $
    regressors) / dlogdm
# constant sqrth
```

Graphing Variances/Standard Deviations

If you want to graph volatility estimates, it’s usually better to graph the estimated standard deviations rather than the variances themselves—detail on much of the data range (where the variance is relatively low) will be lost in comparison to the expected peaks if you do the variances directly. (100-1 ratios from high to low in variance aren’t that uncommon, which converts to a more manageable 10-1 in standard deviation form). Also, graphing variances across an entire (sometimes very long) data range can also lose detail. GARCHUV.RPF computes and graphs standard deviations from three models over roughly 100 data points.

```
garch(p=0,q=6,hseries=hh06) / dlogdm
garch(p=1,q=1,hseries=hh11) / dlogdm
garch(p=1,q=1,exp,asymm,hseries=hha) / dlogdm

set h06 = sqrt(hh06)
set h11 = sqrt(hh11)
set hha = sqrt(hha)
graph(key=below,header="Estimated Standard Deviations", $
    klabels=| | "ARCH6", "EGARCH11", "GARCH11" | | ) 3
# h06 1770 *
# hha 1770 *
# h11 1770 *
```

9.3.8 Diagnostics

If the model is adequate, the standardized residuals (see above) should be serially uncorrelated (if the mean model is chosen correctly), and their squares should be as well (if the variance model is chosen correctly). The first can be tested with Ljung-Box (1978) and the second with the McLeod-Li (1983). Both are included in the **@BDINDTESTS** procedure, along with several other independence tests, and McLeod-Li has its own procedure named (not surprisingly) **@MCLEODLI**. You can also use the **@REGCORRS** procedure which not only computes the test statistic, but graphs the autocorrelations. For the McLeod-Li test, you need to apply this to the squared standardized residuals. The appropriate degrees of freedom correction for McLeod-Li is the number of “GARCH” parameters ($p+q$).

```
garch(p=1,q=1,resids=u,hseries=h) / dlogdm
set ustd = u/sqrt(h)
set ustdsq = ustd^2
@regcorrs(qstat,number=40,dfc=1,title="GARCH-LB Test") ustd
@regcorrs(qstat,number=40,dfc=2,title="GARCH-McLeod-Li Test") $
    ustdsq
```

When you have a very large number of observations, it’s unlikely that both tests will pass at a conventional .05 level. That’s where the autocorrelation graph can help, since it will often be clear, even if the test doesn’t pass at the conventional level, that there isn’t really much (correctable) residual autocorrelation. If the statistically significant residual autocorrelation is on the short lags (like 1 and 2), it may be helpful to add another lag to your autoregression if that’s what you’re doing with the mean model. If the short lags are generally small and what shows as “significant” is on longer lags, then there probably is no simple model change which can fix it—with very large data sets, even quite a small autocorrelation can be significant at the .05 level.

A standard **STATISTICS** instruction on the standardized residuals can be used to see if there is kurtosis that might be helped by using a non-normal distribution (Section 9.3.3). However, if you are already *using* one of those, the standardized residuals will still show kurtosis, and generally more than before.

Another test which can be helpful in spotting problems is a structural stability (fluctuations) test. For more on that see page UG–357. In this case, it would be done by adding the **DERIVES** option to the **GARCH** instruction and using the **@FLUX** procedure:

```
garch(p=1,q=1,derives=dd) / dlogdm
@flux(title="Stability Test for GARCH model")
# dd
```

Example **GARCHFLUX.RPF** (page UG–357) shows how you can look more carefully at the test information to decide whether there are any changes required to the model.

9.3.9 Extensions (Using MAXIMIZE)

While the **GARCH** instruction can handle a wide range of standard GARCH models, there will always be recent refinements which don't fit into its framework. In most cases, these can be estimated using **MAXIMIZE** by making adaptations to one of a few basic models. We've found that it's easiest to do this if the key parts of the log likelihood are computed by separate **FRMLs**. Thus, there are seven steps used in specifying ARCH, GARCH, and like models in RATS:

1. use the **NONLIN** instruction to specify the parameters to be estimated
2. use **FRML** to specify the conditional mean(s)
3. use **FRML** to specify the conditional variance(s)
4. use **FRML** to specify the log likelihood using the mean and variance formulas
5. set the initial values of the parameters
6. set the pre-sample values of the residuals and the conditional variance
7. use the **MAXIMIZE** instruction to compute the estimates.

Using separate **PARMSETS** for the mean and variance models is also a good practice. See Section 4.6 for more on that.

The following program segment implements a GARCH(1,1) model, with a variance function of $h_t = c + au_{t-1}^2 + bh_{t-1}$. The values for the residuals, the squared residuals and the variances are generated (recursively) and saved into the series U, UU and H. H is generated first to allow for the possibility that your mean model depends upon its current value. The contents of these series will change with each function evaluation. The use of the UU series simplifies some of the calculations, since the model is generating its expected value. This, along with many other variants on GARCH models, are demonstrated on the example file **GARCHUVMAX.RPF**.

It's important to note that when you use **MAXIMIZE**, you *must* provide an explicit start for the estimation range. Because of the recursive nature of the calculation, **MAXIMIZE** can't easily figure out which entry is the first one computable. In this case, the first entry is 3, allowing for one lag due to the AR(1), and another because the input series was generated as a log first difference.

```
linreg dlogdm
# constant dlogdm{1}
```

Steps 2, 1 (for the mean), 5 (for the mean)

```
frml (lastreg,vector=beta) meanf
nonlin (parmset=meanparms) beta
```

Step 5

```
set uu = %seesq
set h  = %seesq
set u  = 0.0
```

Step 1 (for the variance model)

```
nonlin (parmset=garchparms) c a b
```

Chapter 9: ARCH and GARCH

Step 5 (for the variance model)

```
compute a=.05,b=.75,c=%seesq*(1-a-b)
```

Step 3

```
frml varf = c+a*uu{1}+b*h{1}
```

Step 4

```
frml logl = (h(t)=varf(t)),(u=dlogdm-meanf),$(  
  (uu(t)=u^2),%logdensity(h,u)
```

Step 7

```
maximize (parmset=meanparms+garchparms) logl 3 *
```

If you try to estimate a GARCH model using **MAXIMIZE** and get the error message:

Missing Values And/Or SMPL Option Leave No Usable Data Points

your estimation range probably starts too soon for your data. Check the range of your data and adjust the **MAXIMIZE** instruction appropriately. If you still have problems, you can test the settings of your formulas with (for instance),

```
set testlog start end = logl(t)  
print start end testlog u h
```

where *start* and *end* are the starting and ending dates you want to use for your estimation. See if the printed values make sense. If you see NA's (missing values) in some or all of the series, work your way back and check the calculations in question to find where they're failing. In a GARCH model, once there's a single bad data point, all later time periods will also be dropped since the variance term isn't computable.

If the **MAXIMIZE** seems to work, but you're losing observations unexpectedly, do the combination of **SET** and **PRINT** above. Check your data to make sure there isn't a missing value. If nothing turns up, do the same thing *before* the **MAXIMIZE** (right after setting your initial guess values). RATS will throw out any data points which produce NA's given the initial parameter settings. So, if you see NA's in the output, you may need to change your initial values.

The GARCH model has one numerical problem not shared by ARCH. It is possible for the variance to “explode” for certain values of the parameters. In estimating the unknown parameters, RATS may examine some of these bad values with uncertain consequences. We have two suggestions for dealing with this:

1. Use **PMETHOD=SIMPLEX** for a small number of preliminary iterations to (in effect) improve your initial estimates. Then use **BFGS** or **BHHH** as your main estimation method.
2. Replace the **VARF** formula with the following, which forces an NA value if the variance gets near an overflow:

```
frml varf = c + a*uu{1} + b*%ovcheck(h{1})
```

9.3.10 Out-of-Sample Variance Forecasts

For the simple GARCH(1,1) model, the variance evolves according to

$$(14) \quad h_t = c + ah_{t-1} + bu_{t-1}^2$$

The forecast of h_{t+k} given information at time $t-1$ (denoted Ω_{t-1}) can be derived as

$$(15) \quad E(h_{t+k} | \Omega_{t-1}) = c + aE(h_{t+k-1} | \Omega_{t-1}) + bE(u_{t+k-1}^2 | \Omega_{t-1}) \\ = c + (a + b)E(h_{t+k-1} | \Omega_{t-1})$$

(as long as $k > 0$) which uses the fact that the expectation of the squared residuals is h by definition. Thus, multiple step forecasts can be obtained recursively. Note that the coefficient on the lagged term in the forecasting equation is the sum of the coefficients on the lagged variance term and the lagged squared residuals. When $k=0$, we would be able to use sample values for both h_{t-1} and u_{t-1}^2 .

Even though there are parallels in the structure, forecasting an ARCH or GARCH model is more complicated than forecasting an ARMA model, because the out-of-sample “residuals” can’t simply be set to zero, as they enter as squares. The following does the forecasts by defining two formulas: one produces a recursive estimate of the variance, the other produces the expected value of the squared residuals. Out-of-sample, these are, of course, the same. They differ, however, in their values at the end of the sample, where the lagged UU series are the sample values of the squared residuals, while the lagged HT series are the estimated variances. This estimates a GARCH(1,1) model and forecasts 120 periods from 1992:1 on. We need both the HSERIES and RESIDS options on the **GARCH** instruction, since both pieces are needed in constructing the forecasts.

```
garch(p=1,q=1,hseries=ht,resids=at) / sp500
```

Create the historical UU series

```
set uu = at^2
```

Copy the coefficients out of the %BETA vector

```
compute vc=%beta(2), vb=%beta(4), va=%beta(3)  
frml hfrml ht = vc + vb*ht{1} + va*uu{1}  
frml uufrml uu = ht  
group garchmod hfrml>>ht uufrml>>uu  
forecast(model=garchmod,from=1992:1,steps=120)
```

While this type of recursive calculation works for the simple ARCH and GARCH models, it doesn’t work for an EGARCH model. Forecasts for those have to be done by simulation or bootstrapping; methods described in Chapter 16.

9.4 Multivariate GARCH Models

The main problem with extending univariate models to a set of variables is that the covariance matrix must be positive-definite at each time period in order for the likelihood to be defined. Even if the *variance* of each equation stays positive, if the cross terms stray out of bounds for just one data point, a set of parameters gives an undefined function value. Also, the number of parameters can grow quite quickly as you increase the number of variables and the estimation time grows as well. Except for some highly restricted parameterizations, the practical limit is three or four variables.

A multivariate GARCH model uses a recursion in the lagged covariance matrices and lagged residuals to generate a value for the current covariance matrix. There have been many formulas proposed to do that—the model type for a multivariate GARCH is chosen with the `MV` option. They mainly fall into two broad classes:

1. VEC models (Section 9.4.1) generate direct recursions for each component of a covariance matrix, both variances and covariances. The two most common choices here are the diagonal VEC or DVEC (which is the default) and the (full) BEKK.
2. Restricted correlation models (Section 9.4.2) generate the variances directly but then derive the covariances from them using a separate calculation. The two main choices here are the CC and DCC. For these, there is a separate `VARIANCES` option (page UG–307) which is used to choose the model used for the variances.

The basic syntax of **GARCH** is

```
garch (options) start end list of series
```

The arrangement with the range first allows for the variable length list of series names for the multivariate models that we’re doing here.

For our example (`GARCHMV.RPF`), which includes almost all the code fragments for this section, we’ll be working with a set of three exchange rates of Japan, France and Switzerland versus the U.S. dollar, with a bit over 6000 daily observations:

```
open data g10xrate.xls  
data(format=xls,org=columns) 1 6237 usxjpn usxfra usxsui  
*  
set xjpn = 100.0*log(usxjpn/usxjpn{1})  
set xfra = 100.0*log(usxfra/usxfra{1})  
set xsui = 100.0*log(usxsui/usxsui{1})
```

All of the short examples will use the default mean model of a separate constant or each series. Because there are multiple equations, the procedure for allowing for a more general mean is different from that for the univariate model (Section 9.4.5).

Note that almost every type of multivariate GARCH model has a different set of parameters, so the arrangement of the output changes. Examples of the output with descriptions of the parameters are included in the details for `GARCHMV.RPF`.

9.4.1 VECH form models

Standard (Diagonal VECH)

A first attempt at an extension to several variables would model each variance or covariance term with a separate recursion:

$$(16) \mathbf{H}_{ij,t} = c_{ij} + a_{ij} u_{i,t-1} u_{j,t-1} + b_{ij} \mathbf{H}_{ij,t-1}$$

This is generally known as multivariate GARCH(1,1) (or MGARCH or diagonal VECH) and is what the **GARCH** instruction will estimate if you do no other options besides picking p and q . The option is `MV=DVECH` or `MV=STANDARD`.

While this is the most “straightforward” extension of a univariate model, the lack of connection among the variance terms is implausible, and it may be hard for the parameter estimates to stay clear of regions where the covariance matrix is on the verge of being deficient rank at some data points.

This, and the next two, methods in particular often need a bit of help to convergence with some preliminary simplex iterations. *Do not* overdo this: 10 simplex iterations is usually enough.

```
garch (p=1,q=1,pmethod=simplex,piters=10) / xjpn xfra xsui
```

BEKK, Diagonal BEKK (DBEKK), Triangular (TBEKK)

The BEKK formulation (Engle and Kroner, 1995) directly imposes positive definiteness on the variance matrix:

$$(17) \mathbf{H}_t = \mathbf{C}\mathbf{C}' + \mathbf{A}'\mathbf{u}_{t-1}\mathbf{u}_{t-1}'\mathbf{A} + \mathbf{B}'\mathbf{H}_{t-1}\mathbf{B}$$

As each term is positive semi-definite by construction, this will avoid bad regions. However, while positive-definiteness is assured, it comes at the cost of a poorly behaved likelihood function. With all the parameters entering through quadratic forms, they aren’t globally identified—changing the signs of all elements of $\mathbf{C}$, $\mathbf{B}$ or $\mathbf{A}$ will have no effect on the function value. $\mathbf{C}$ can have only $n(n+1)/2$ free parameters—**GARCH** parameterizes it to be lower triangular. This term is often written as the equivalent $\mathbf{C}'\mathbf{C}$ where $\mathbf{C}$ is *upper* triangular. We have chosen to retain the positioning of the transposes for $\mathbf{A}$ and $\mathbf{B}$ in (17) that were used in the original papers and are now standard in the literature. However, note that that switches the meanings of subscripts in those matrices from what is common in most multivariate analysis (not just GARCH, but VAR’s as well): for the BEKK, $\mathbf{A}(i,j)$ will have the effect of residual i on variable j , so the target is in the column, not the row.

Choose the BEKK model with `MV=BEKK`. DBEKK and TBEKK are restricted forms of this. DBEKK (diagonal BEKK, chosen with `MV=DBEKK`) makes $\mathbf{A}$ and $\mathbf{B}$ diagonal. TBEKK (triangular BEKK, chosen with `MV=TBEKK`) makes the pre-multiplying matrices $\mathbf{A}'$ and $\mathbf{B}'$ in (17) lower triangular. (The coefficients will actually be reported for the lower triangles). This creates a recursive ordering among the variables, and (unlike almost all other model types) makes the model order important. (TBEKK is

sometimes defined with the pre-multiplying matrix being *upper* triangular, which produces a counter-intuitive situation where the first variable in the model is actually *last* in the ordering of effects.

Note that the **A** and **B** parameters are sensitive to the relative scales of the variables. If the variables (or their variances) are different scales, it's possible for an off-diagonal element to be larger than a diagonal one. It's also quite possible for some of the **A** or **B** coefficients to be negative if differences between contemporaneous residuals have some effect on variances.

```
garch(p=1,q=1,mv=bekk,pmethod=simplex,piters=10) / xjpn xfra xsui
```

Because of the structure of the BEKK model, extensions to allow asymmetry (page UG–308) or variance shifts (page UG–309) are more complicated than they are for most other model forms.

VECH (Full)

RATS also offers the VECH model, which allows complete interaction among the terms

$$(18) \text{ vech}(\mathbf{H}_t) = \mathbf{C} + \mathbf{A} \text{vech}(\mathbf{u}_{t-1} \mathbf{u}_{t-1}') + \mathbf{B} \text{vech}(\mathbf{H}_{t-1})$$

The *vech* operator takes a symmetric matrix and returns a vector with only its lower triangle. In RATS, symmetric arrays are stored by rows, so that's how the coefficients are arranged. You implement this with the option `MV=VECH`. However, this has a large number of free parameters (even in a bivariate case), and it's clearly unwieldy with more than two variables. Model (16) is known as a diagonal VECH because it is (18) with **A** and **B** constrained to be diagonal. The VECH takes a VERY long time to estimate, and is rarely used, so we didn't include it in the example file.

EWMA

EWMA (Exponentially Weighted Moving Average) is a *very* tightly parameterized variance model. There is just a single real parameter (α) governing the evolution of the variance:

$$(19) \mathbf{H}_t = (1 - \alpha) \mathbf{H}_{t-1} + \alpha (\mathbf{u}_{t-1} \mathbf{u}_{t-1}')$$

This is chosen with `MV=EWMA`.

9.4.2 Restricted Correlation Models (CC, DCC and others)

The remaining methods all use GARCH models for the individual variances, but generate the covariances in a more restricted fashion. There are several choices for modeling the variances, which are discussed beginning on page UG–306.

MV=DIAG

The simplest of the restricted models is MV=DIAG. This estimates separate univariate GARCH models on each dependent variable. The “model” for the covariances between variables is that they are all zero. This allows you to do “two-step” procedures, which model the correlations based upon the standardized residuals from the univariate models. Using **GARCH** to handle all the variables simultaneously (rather than doing separate univariate **GARCH** instructions) ensures that they are estimated over a common range. Other than that, it has relatively little use, since there is typically quite a bit of contemporaneous correlation among the residuals, which this ignores.

MV=CC (Constant Correlation)

The next step up in complexity is the *Constant Correlation* specification, which you can estimate with the option MV=CC. The covariances are given by

$$(20) \quad H_{ij,t} = R_{ij} \sqrt{H_{ii,t} H_{jj,t}}$$

where the off-diagonal (lower triangular) elements of **R** are estimated parameters.

garch (p=1,q=1,mv=cc) / xjpn xfra xsui

MV=DCC (Dynamic Conditional Correlation)

While CC generally has a well-behaved likelihood function, and can handle a bigger set of variables than the more fully parameterized models of Section 9.4.1, it does have the drawback of requiring the correlation to be constant. In some applications, time-varying correlations are essential. Engle (2002) proposed a method of handling this which he dubbed *Dynamic Conditional Correlations*. This adds two scalar parameters which govern a “GARCH(1,1)” model on the covariance matrix as a whole.

$$(21) \quad \mathbf{Q}_t = (1 - a - b)\bar{\mathbf{Q}} + a\mathbf{u}_{t-1}\mathbf{u}_{t-1}' + b\mathbf{Q}_{t-1}$$

where $\bar{\mathbf{Q}}$ is the unconditional covariance matrix.

However, **Q** isn’t the sequence of covariance matrices. Instead, it is used solely to provide the correlation matrix. The actual **H** matrix is generated using univariate GARCH models for the variances, combined with the correlations produced by the **Q**.

$$(22) \quad H_{ij,t} = Q_{ij,t} \sqrt{H_{ii,t} H_{jj,t}} / \sqrt{Q_{ii,t} Q_{jj,t}}$$

garch (p=1,q=1,mv=dcc) / xjpn xfra xsui

The parameters are labeled DCC (A) and DCC (B) in the output.

There's a separate DCC option which chooses the form of the submodel (21). The default is DCC=COVARIANCE, which is (21) as written, using the model residuals. DCC=CORRELATION is Engle's original idea of

$$(23) \quad \mathbf{Q}_t = (1 - a - b)\bar{\mathbf{Q}} + a\varepsilon_{t-1}\varepsilon'_{t-1} + b\mathbf{Q}_{t-1}$$

where ε_t is the vector of *standardized* residuals divided by the model estimate for its standard deviation) and $\bar{\mathbf{Q}}$ is the sample covariance matrix of the those. Engle's recursion is really designed for “two-step” estimation procedures which estimate univariate models first and take the standardized residuals as given in a second step of estimating the joint covariance matrix. However, it can't be applied easily with any of the VARIANCE models which have interactions among the variances from different equations. It also, in practice, rarely fits better than DCC=COVARIANCE, and often fits quite a bit worse. (It tends to be more easily dominated by outliers.) DCC=COVARIANCE is the calculation that **GARCH** has been doing since DCC was added and remains the default.

DCC=CDCC (corrected DCC) is based upon the suggestion in Aielli(2013). Aielli shows the Engle's DCC recursion isn't internally consistent, in the sense that $E(\varepsilon_t\varepsilon'_t) \neq \mathbf{Q}_t$ as it would be if the secondary model were actually “GARCH”, and he proposes an alternative which fixes that. Aielli's model has the same problem as Engle's that it doesn't really apply except to models which can be done in two steps (mainly because the calculation of the implied $\bar{\mathbf{Q}}$ requires that the individual variances be known), so DCC=CDCC uses a “feasible” estimator of the sample correlation matrix.

MV=ADCC

This is similar to DCC, but adds to the $\mathbf{Q}$ recursion an asymmetry term as described on page UG–308 with a scalar multiplier (labeled as DCC (G) in the output).

MV=CHOLESKY

The Cholesky model has some similarities to the others in the section, but one important difference—while the other models apply a univariate model to observed data, the Cholesky model uses ideas from structural VAR modeling to map the observable residuals ($\mathbf{u}$) to uncorrelated residuals ($\mathbf{v}$) using $\mathbf{u}_t = \mathbf{F}\mathbf{v}_t$, where $\mathbf{F}$ is lower triangular. The difference with the VAR literature is that the components of $\mathbf{v}$ are assumed to follow (univariate) GARCH processes rather than having a fixed (identity) covariance matrix. As in the VAR literature, it's necessary to come up with some normalization between the $\mathbf{F}$ and the variances of $\mathbf{v}$. For the Cholesky model in the VAR, the obvious choice is to fix the variances at 1. However, here the variances aren't fixed, so it's simpler to make the diagonals of $\mathbf{F}$ equal to 1 and leave the component GARCH processes free, so the free parameters in $\mathbf{F}$ will be the elements below the diagonal.

The Cholesky model is sensitive to the order of listing—by construction, the first $\mathbf{v}$ is identical to the first $\mathbf{u}$.

```
garch(p=1,q=1,mv=cholesky) / xjpn xfra xsui
```

VARIANCES option

The methods in this section all require models for the variances of the individual processes. The default is a standard GARCH model (VARIANCES=SIMPLE), each with the same number of GARCH parameters. You can make four other choices for this.

The first is VARIANCES=EXP, which gives the multivariate E-GARCH model. The individual variance models are E-GARCH models as described in equation (6) without the asymmetry term. (Asymmetric effects are permitted as described below).

Another is VARIANCES=VARMA. This was proposed by Ling and McAleer (2003). The variance terms here take the form (for a 1,1 model):

$$(24) \quad H_{ii,t} = c_{ii} + \sum_j a_{ij} u_{j,t-1}^2 + \sum_j b_{ij} H_{jj,t-1}$$

This allows large shocks in one variable to affect the variances of the others. This tends to get overparameterized when the number of variables gets too large. Note, by the way, that the phrase VARMA-GARCH also gets applied to a GARCH model with a VARMA mean model (page UG-312). Make sure that you understand which you intend.

VARIANCES=SPILLOVER is similar to VARIANCES=VARMA, but doesn't include the lagged variances of "other" variables, that is,

$$(25) \quad H_{ii,t} = c_{ii} + \sum_j a_{ij} u_{j,t-1}^2 + b_i H_{ii,t-1}$$

Note that VARMA is an extension of SPILLOVER, and SPILLOVER is an extension of SIMPLE (which doesn't include the "other" squared lagged residuals).

Finally, VARIANCES=KOUTMOS is similar to SPILLOVER, but uses an EGARCH setup. This incorporates asymmetry of a very specific form (with just a single controlling term for each variable) by construction. The variance recursion takes the form

$$(26) \quad \log H_{ii,t} = c_{ii} + \sum_j a_{ij} \left(\frac{|u_{j,t-1}|}{\sqrt{H_{jj,t-1}}} + d_j \frac{u_{j,t-1}}{\sqrt{H_{jj,t-1}}} \right) + b_i \log H_{ii,t-1}$$

This is from Koutmos(1996). This extends VARIANCES=EXP, which doesn't include the "other" variable standardized residuals.

The following estimates a CC-VARMA-GARCH.

```
garch(p=1,q=1,mv=cc,variances=varma,pmethod=simplex,piters=10) / $  
xjpn xfra xsui
```

9.4.3 Other Model Options

Fat-Tails

To allow for a fat-tailed distribution, you can choose `DISTRIB=T` option which uses a multivariate Student- t distribution or `DISTRIB=GED` which uses a multivariate generalization of the GED. As with univariate models, you can choose a fixed “shape” (degrees of freedom for the t) with the `SHAPE` option, or, if you don’t use that option, the degrees of freedom will be estimated. This estimates an EWMA model with t distributed errors.

```
garch(p=1,q=1,mv=ewma,distrib=t) / xjpn xfra xsui
```

Asymmetry

The `ASYMMETRIC` option adds asymmetry effects to multivariate GARCH models. The precise form will depend upon the variance model you choose. For models where only the variances are directly governed by a GARCH model (Section 9.4.2), each variance just has a term added as in (13).

For the VECH models, define

$$(27) \quad \mathbf{v}_{t-1} = \mathbf{u}_{t-1} \circ I_{u < 0}(\mathbf{u}_{t-1})$$

where $\circ$ denotes the elementwise (Hadamard) product of the vectors. $\mathbf{v}$ will be a copy of $\mathbf{u}$ with positive elements zeroed out. With this, for the standard MV-GARCH model, the asymmetry terms in a GARCH(1,1) add the following to the formula for $H_{ij,t}$:

$$(28) \quad d_{ij} v_{i,t-1} v_{j,t-1}$$

for the BEKK, it’s

$$(29) \quad \mathbf{D}' \mathbf{v}_{t-1} \mathbf{v}_{t-1}' \mathbf{D}$$

and for the VECH, it’s

$$(30) \quad \mathbf{D} \text{vech}(\mathbf{v}_{t-1} \mathbf{v}_{t-1}')$$

Use the `SIGNS` option to change the behavior of $\mathbf{I}$ for each component. The default behavior is $\mathbf{I}=1$ when $\alpha < 0$ and zero otherwise as noted above. This gives the behavior described on page UG–293, where negative residuals increase variance given positive $\mathbf{D}$ values. Use `SIGNS` to supply +1 values rather than –1 values for any components where you want the opposite behavior: $\mathbf{I}=1$ for $\alpha > 0$ such that *positive* residuals increase variance (for positive $\mathbf{D}$).

Note that, unlike univariate GARCH models, the choice for sign on the definition of $\mathbf{v}$ matters because of the effect on the cross terms.

This estimates a CC-EGARCH with asymmetry.

```
garch(p=1,q=1,mv=cc,asymmetric,variances=exp) / xjpn xfra xsui
```

XREGRESSORS

As with univariate models, you add exogenous variance shifts by using the `XREGRESSORS` option and including a supplementary card listing the variables. (Again, don't include the `CONSTANT`.) The same shift is applied to all variables. For the models in Section 9.4.2, `XREG` adds the shifts to each variance equation separately. For the `DVECH` and `VECH`, it adds for each X-regressor an extra "variance constant" term with the same layout as the model's `C` term, but multiplied by the regressor.

Those are all fairly obvious. However, the proper way to handle an X-regressor with the `BEKK` models isn't as clear. Employing the same idea as used for the `DVECH` and `VECH` runs into the problem that $\mathbf{E}\mathbf{E}'$ is forcibly positive semi-definite. For instance, if that method were used and you add a Monday dummy, the model would force the variance to be higher (or at least no lower) for Monday than for any other day, even if the effect were, in fact, just the opposite. Instead, what `RATS` does by default is to replace the original variance constant with

$$(31) \quad (\mathbf{C} + \mathbf{E}x_t)(\mathbf{C} + \mathbf{E}x_t)'$$

where x_t is the X-regressor and $\mathbf{E}$ (like $\mathbf{C}$) is a lower triangular matrix. This preserves the restriction that `BEKK` generates a positive definite matrix, and doesn't prejudge the signs of the effects if you use a set of (mutually exclusive) dummies. You *can* use separate terms by including the option `XBEKK=SEPARATE` (the default is `XBEKK=COMBINED`) which replaces the constant term in the recursion with

$$(32) \quad \mathbf{C}\mathbf{C}' + \mathbf{E}\mathbf{E}'x_t^2$$

9.4.4 Fetching the Residuals and Covariance Matrices

The `HSERIES` and `RESIDS` options described before for univariate models won't do for a multivariate model, as we need a covariance *matrix* and a *vector* of residuals at each entry. Since there are situations where you would need to be able to work with the full matrix at a given entry, and others where you are more interested in the time series of a component of them, each of these now has two options providing the information in different forms.

Covariance Matrices (HMATRICES and MVHSERIES)

The `HMATRICES` option provides the covariance matrices as a `SERIES` of `SYMMETRIC` arrays. The `MVHSERIES` option which gives them as a `SYMMETRIC` of `SERIES`. The latter is more convenient if you want to graph one of the components, or use it as an input to (for instance) a regression. The former is more useful for things like forecasting the variance, where you need to be able to work with the covariance matrix as a whole.

The following uses `HMATRICES` and the `%CVTOCORR` function to generate and graph the conditional correlations. (Note: this works the same way with any type of MV GARCH model, not just DCC). `%CVTOCORR` transforms a covariance matrix to a correlation matrix, then the (1,2) etc. extracts a particular element from that. Note that the process of extracting and analyzing the covariances or correlations is the same for *all* types of multivariate GARCH models.

```
garch(p=1,q=1,mv=dcc,variances=koutmos,hmatrices=hh) / $
    xjpn xfra xsui

set jpnfra = %cvtocorr(hh(t))(1,2)
set jpnsui = %cvtocorr(hh(t))(1,3)
set frasui = %cvtocorr(hh(t))(2,3)

spgraph(vfields=3,footer="Conditional Correlations")
graph(header="Japan with France",min=-1.0,max=1.0)
# jpnfra
graph(header="Japan with Switzerland",min=-1.0,max=1.0)
# jpnsui
graph(header="France with Switzerland",min=-1.0,max=1.0)
# frasui
spgraph(done)
```

Residuals (RSERIES and RVECTORS)

RSERIES generates a VECTOR[SERIES] with the residuals. This serves the same purpose as the RESIDS options on instructions like **ESTIMATE** and **SUR**. The alternative is RVECTORS, which creates a SERIES[VECTOR].

The following saves the residuals (using RSERIES) and the covariances (using MVHSERIES) and creates a dummy variable which is 1 if the residual for Japan is in the left .05 tail of the conditional distribution. (The FIXT is to correct for the difference between the standard t distribution and the rescaled t with unit variance used by GARCH).

```
garch(p=1,q=1,mv=bekk,pmethod=simplex,piters=10,distrib=t,$
      rseries=rs,mvhseries=hhs) / xjpn xfra xsui
*
compute fixt=(%shape-2)/%shape
set trigger = %tcdf(rs(1)/sqrt(hhs(1,1)*fixt),%shape)<.05
sstats(mean) / trigger>>VaRp
disp "Probability of being below .05 level" #.#### VaRp
```

9.4.5 The Mean Model

As with the univariate GARCH models, the default for the mean model is the constant. In the output, these are shown as (for instance) `MEAN(XJPN)`, ..., `MEAN(XSUI)`. And, as in the univariate case, you can estimate the models with mean zero processes by adding `NOMEAN`.

It's important to remember that main task of the mean model is to produce (jointly) serially uncorrelated residuals so you can fit a GARCH model properly. The model which will do that with one data set may be different from the one that will do that with a similar but different data set. Don't blindly copy a mean model from another paper.

MODEL option

If you need a more complicated mean model, and you have a common set of regressors, you can use the `REGRESSORS` option (just as with univariate model) with a single supplementary card. If you have extra mean variables which are *different* among equations, you need to create a `MODEL` variable, generally using `GROUP`. Because the equations in the model include the information about the dependent variable, you don't need to list dependent variables on the `GARCH` instruction. Suppose that we want AR(1) models for each of the three variables and a DCC model:

```
equation(constant) jpneg xjpn 1
equation(constant) fraeq xfra 1
equation(constant) suieq xsui 1
group ar1 jpneg fraeq suieq
garch(p=1,q=1,model=ar1,mv=dcc,pmethod=simplex,piter=10)
```

You can use the `SYSTEM` definition to create a VAR (or VECM if you have cointegrated variables) for the mean model. This does a one lag VAR with a BEKK variance model:

```
system(model=var1)
variables xjpn xfra xsui
lags 1
det constant
end(system)
garch(p=1,q=1,model=var1,mv=bekk,pmethod=simplex,piters=10)
```

For the more complicated mean models, the output splits these up by the dependent variables, such as

Mean Model (XJPN)				
1. Constant	0.004012392	0.005915957	0.67823	0.49762456
2. XJPN{1}	0.025541591	0.011287801	2.26276	0.02365045
Mean Model (XFRA)				
3. Constant	-0.003121697	0.005914162	-0.52783	0.59761444
4. XFRA{1}	0.005553417	0.009898923	0.56101	0.57478919
Mean Model (XSUI)				
5. Constant	-0.003058263	0.006944297	-0.44040	0.65964795
6. XSUI{1}	-0.001890160	0.009596114	-0.19697	0.84384988

GARCH-M

To do GARCH-M with a *multivariate* model, you have to plan ahead a bit. First of all, you have to decide what “M” effects you want in the model (or in any given equation). In an N variable model, there are $N(N+1)/2$ variances and covariances, and rarely will all of them be included in each equation.

On your **GARCH** instruction, you need to use the option `MVHSERIES=SYMM[SERIES]` (Section 9.4.4) which will save the paths of the variances (example: `MVHSERIES=HHS`). Your regression equations for the means will include references to the elements of this array of series. Since those equations need to be created in advance, you need the `SYMM[SERIES]` first as well. The code segment below includes as explanatory variables in each equation all the covariances which involve the residuals from that equation.

```
dec symm[series] hhs(3,3)
clear(zeros) hhs
equation jpneq xjpn
# constant hhs(1,1) hhs(1,2) hhs(1,3)
equation fraeq xfra
# constant hhs(2,1) hhs(2,2) hhs(2,3)
equation suieq xsui
# constant hhs(3,1) hhs(3,2) hhs(3,3)
group garchm jpneq fraeq suieq
garch(model=garchm,p=1,q=1,pmethod=simplex,piters=10,$
      mvhseries=hhs)
```

If you need some *function* of the variances or covariances, you need to use the `HADJUST` option to define it based upon the values that you save either with `MVHSERIES` or with `HMATRICES`. The following, for instance, adds the conditional standard deviation (generated into the series `COMMONSD`) of an equally weighted sum of the three currencies:

```
set commonsd = 0.0
system(model=customm)
variables xjpn xfra xsui
det constant commonsd
end(system)
*
compute %nvar=%modelsz(customm)
compute weights=%fill(%nvar,1,1.0/%nvar)
garch(model=customm,p=1,q=1,mv=cc,variances=spillover,$
      hmatrices=hh,hadjust=(commonsd=sqrt(%qform(hh,weights))) ) )
```

VARMA Mean Model

To create a VARMA model for the mean (not to be confused with VARMA variance calculations, page UG–307), create a `VECT[SERIES]` for the residuals, add the lags to the model using the **DET** instruction in the system, and use the `RSERIES` option to save the residuals (which are saved as they are calculated). The following will do a VMA mean model, with an asymmetric BEKK GARCH model, saving the residuals into `U` and adding lags of `U(1)`, `U(2)` and `U(3)` to the mean model. (The dimension and the list would have to be adjusted if you have a different number of variables).

```
dec vect[series] u(3)
clear(zeros) u
system(model=varma)
variables xjpn xfra xsui
det constant u(1){1} u(2){1} u(3){1}
end(system)
*
garch(model=varma,p=1,q=1,mv=bekk,asymmetric,rseries=u,$
      pmethod=simplex,piters=10,iters=500)
```

Note that VARMA models (with both lagged dependent variables *and* lagged residuals) can be subject to cancellation problems. For instance, with this data set, if we add `LAGS 1` to the system definition to convert this from a VMA(1) to a VARMA(1,1) mean model, it won't converge properly and the AR and MA parameters will show up as being (almost) equal and opposite sign—there have been papers published with all the signs of overparameterized mean models. A more complete example that demonstrates the problem is available as `VARMAGARCH.RPF`.

If you need a model to handle serial correlation, start small and start with a VAR, and (perhaps) go to VARMA only if the VAR leaves serial correlation in the residuals (Section 9.4.6). The **@VARLAGSELECT** procedure can be helpful in picking a starting lag length for a VAR model to handle the serial correlation. You're generally best off using the more stringent BIC as the selection criterion for that—starting with the smallest reasonable model and increasing the lags only if necessary.

9.4.6 Diagnostics

Univariate Diagnostics

If the model is correctly specified, each set of standardized residuals should pass the same types of tests as they do for univariate models, that is, there should be no autocorrelation either in the residuals or their squares (Section 9.3.8). If we used the options `rseries=rs` and `mvhseries=hhs`, we can get the (univariate) standardized residuals with

```
set z1 = rs(1)/sqrt(hhs(1,1))
set z2 = rs(2)/sqrt(hhs(2,2))
set z3 = rs(3)/sqrt(hhs(3,3))
```

We can do the same types of diagnostics as for univariate models; in this case, we'll use `@BDINDTESTS`:

```
@bdindtests(number=40) z1
@bdindtests(number=40) z2
@bdindtests(number=40) z3
```

If you don't want to hard-code a specific set of variables, you can write this more flexibly as: (note: **GARCH** defines `%NVAR` as the number of dependent variables)

```
dec vect[series] z(%nvar)
do i=1,%nvar
    set z(i) = rs(i)/sqrt(hhs(i,i))
    @bdindtests(number=40) z(i)
end do i
```

While it's not uncommon for a published paper to include the univariate diagnostics, unless you're doing a model which relies heavily on the univariate models themselves (such as a `MV=CC` or `MV=DCC` with `VARIANCES=SIMPLE` or `VARIANCES=EXP` rather than one of the variance forms with interactions, page UG-307), they won't really tell you much.

Multivariate Diagnostics

There are multivariate generalizations of the Ljung-Box *Q* to test for serial correlation in the mean, one of which is in the `@MVQSTAT` procedure. The problem with applying these to a multivariate GARCH model is that they assume a constant covariance matrix. We can easily standardize the *variances*, but, except for the `CC` model, the correlations will change from period to period. Instead, the entire residual vector needs to be "standardized" by pre-multiplying by the inverse of a factor of that period's covariance matrix to produce (if the model is correct) a set of series which have the identity as their covariance matrix. Note that this says *a* factor. There are an infinite number of ways to factor a covariance matrix (with two or more variables), that is, produce an **F** that satisfies $\mathbf{FF}'=\mathbf{H}$. Different choices for the factoring process will produce different standardized residuals and (as a result) different diagnostics. Two obvious ways to do this are to use a Cholesky factor or an eigenvector-based factor.

Note that, unlike the case with a VAR, you aren't trying to come up with a “model” for factoring—you just want something mechanical so you can apply the diagnostics. You can generate a `VECT[SERIES]` of the jointly standardized residuals with the `STDRESIDS` option and choose the method of factoring with the `FACTORBY` option, which offers the choice of `CHOLESKY` (default) and `EIGEN`.

The following estimates an asymmetric BEKK model, with t distributed errors, and saves the residuals and variances, standard residuals and derivatives, and uses these to do a multivariate Q test, a test for multivariate (residual) ARCH and a fluctuations test.

```
garch(model=varma,mv=bekk,asymmetric,p=1,q=1,distrib=t,$
      pmethod=simplex,piters=10,rseries=rs,mvhseries=hhs,$
      stdresids=zu,derives=dd)
@mvqstat(lags=5)
# zu
@mvarchtest(lags=5)
# zu
@flux
# dd
```

Note that the `@MVARCHTEST` in particular often fails to be acceptable at conventional levels, even if there is no clearly correctable problem with the model. See “Effect of Large Sample Sizes” on page UG–70.

9.4.7 Hypothesis Tests and Coefficient Interpretation

“Spillover” Tests

“Spillover” in the context of multivariate GARCH models is a rather vaguely defined term. In general, the term spillover in economics refers to an event producing an effect elsewhere despite there being no obvious connection. However, it's hard to argue that there is no obvious connection between, for instance, two exchange rates (which presumably have the same numeraire currency, and thus are both directly affected by any event which touches the numeraire). Spillover has come to mean a model-based “Granger-style” causality, either in the mean or (more commonly) in the variance, that is, a question of whether a set of coefficients on “other” variables is zero. The more straightforward case is for causality in the mean.

Causality Tests in the Mean (Granger Causality)

If you are estimating a VAR with a GARCH error process, Granger causality tests for the mean involve the same restrictions as they do in a standard VAR. However, there is no simple way to estimate restricted and unrestricted models as is done in the `VARCAUSE.RPF` example (page UG–214). Instead, you need to do a Wald test on the full VAR-GARCH model. That's most easily set up using the *Statistics—Regression Tests* wizard, selecting the “Exclusion Restrictions”. The `VECMGARCH.RPF` example file shows how to do that in the context of a VECM mean model.

Causality/Spillover in Variance

Testing for some type of relationship in the *variances* is complicated by the fact that the volatility is unobservable (page UG–287). Granger causality *for the mean* is defined in terms of the improvement in forecasts of the observable data. With volatility being unobservable, we can’t make a similar requirement for detecting causality in the variance. In fact, all we can really do is test whether a model gives (significantly) *different* estimates of the volatility, not significantly better ones. On top of that, only certain model forms even permit such a test. The DVECH, by construction, doesn’t allow variance “spillover”—each variance depends only upon its own lags and its own lagged residuals. In the BEKK model, if **A** or **B** is non-diagonal, the model has variances which depend upon other equations’ residuals (if **A** isn’t diagonal) or variances and covariances (if **B** isn’t diagonal). However, testing the off-diagonals (either all or some) is testing the BEKK against the *very* restricted DBEKK. In practice, you can fit a BEKK, which permits some form of “spillover” test, and a DVECH, which doesn’t, to the same data set and get similar likelihoods and even similar estimates from the variances and covariances.

If you wish to do this type of test in a model which allows for it (a BEKK or TBEKK model or a restricted covariance model with VARMA, spillover or Koutmos variance model), as with the causality test in the mean, you will generally have to do a Wald test on the full model, and it’s easiest to set that up with *Statistics—Regression Tests*. You can do a likelihood ratio test by comparing (for instance) a BEKK with a DBEKK, but that tests for overall relationship, and not any specific direction.

Interpreting Coefficients

The coefficients in a univariate GARCH model generally are fairly easy to interpret, particularly if we look at the rearranged recursion (8). Something similar can be done with each component of the DVECH, and with the simpler forms of variance models in 9.4.2. However, it’s very difficult to say much of anything about the individual coefficients in a BEKK model. First off, they’re part of a complicated quadratic form for the recursion—since everything gets “squared”, the values often seem unnaturally high, particularly the **A** coefficients, where diagonal values of .3 or larger are not uncommon and **A** plus corresponding **B** coefficients can be well above one, again, because they enter as squares so the sum of just the values doesn’t really say anything. It’s *possible* to expand the model out into the equivalent VECH representation, but for a three variable system like the one in **GARCHMV.RPF**, that would fill up most of a page and would *still* not be easy to interpret, since it has coefficients on a large number of (typically) highly correlated variables—the estimated variances and covariances of what are usually fairly closely related series. If the model allows for a Variance Impulse Response Function (Section 9.4.8) that usually provides a better way to summarize the information in the model.

9.4.8 Forecasting

There are two different schemes for calculating out-of-sample forecasts for the variances. The DVECH and BEKK models can be converted into special cases of the VECH, which can be forecast using a multivariate extension of (16). The CC and DIAG models use (16) to forecast the variances, which are then converted to a full covariance matrix by using the estimated correlation pattern. Use the **@MVGARCHFore** procedure to do the calculations. This requires that you save the covariance matrices and residuals using the **HMATRICES** and **RVECTORS** options (Section 9.4.4) and input them to the procedure. An example is:

```
@MVGarchFore(steps=100) hh rd
```

Note that there are quite a few combinations of options which do not allow simple recursive forecasts. For instance, the **DCC** model, or **VARIANCE=EXP** or **VARIANCE=KOUTMOS** are excluded from the ones that **@MVGARCHFore** can handle. You also cannot do simple forecasts of asymmetric multivariate models because the sign-restricted interaction terms have no closed-form expected values—while the sign-restricted *squares* have 1/2 the expected value of the unrestricted squares, that's not the case for the interactions between two *different* residuals. For a model with no simple extrapolation, forecasts have to be approximated by simulations or bootstrapping. The types of models which admit closed-form variance forecasts also allow for the calculation of a Variance Impulse Response Function (VIRF)—see Hafner and Herwartz(2006).

The syntax for **@MVGARCHFore** is:

```
@MVGARCHFore (options) h u
```

Parameters

- H* (input & output) is the **SERIES [SYMM]** which has the in-sample estimates of the covariances (produced with the **HMATRICES** option on **GARCH**), and will be extended to hold the forecasts of the covariances
- U* (input) is the **SERIES [VECT]** of the residuals. This is produced by the **RVECTORS** option on **GARCH**.

Options

steps=*number of out-of-sample forecast steps* [12]

mv=[**standard**]/**bekk**/**diagonal**/**cc**/**dcc**/**vech**

This should be the same choice that you made on the **GARCH** instruction.

9.4.9 Extensions (Multivariate)

If you want to estimate a GARCH model which doesn't fit into the forms given by **GARCH**, you can use **MAXIMIZE** instead. The setup for this follows the same steps as it does for a univariate model, but all the steps are more complicated because of the need to deal with matrices rather than scalars. The task is usually easier if you write it specifically for two variables. If you need to allow for three or more, it makes more sense to write it very generally (as is done here) rather than expanding out terms.

This shows the code for estimating a standard GARCH(1,1) model and a CC. For most variations, all you would typically need to do is to have an alternative FRML for HF. In this case, HF is easy to write as a simple FRML evaluating to a SYMMETRIC array, but remember that, if the calculation is more complicated, you can use a **FUNCTION**.

This is from example file GARCHMVMAX.RPF. Note that it takes about four times as long to estimate a model this way as it does a similar model done with **GARCH**.

The initial few lines of this set the estimation range, which needs to be done explicitly, and the number of variables. Then, vectors for the dependent variables, residuals and residual formulas are set up. The **SET** instructions copy the dependent variables over into the slots in the vector of series.

```
compute gstart=2,gend=6237
compute n=3
dec vect[series] y(n) u(n)
dec vect[frml] resid(n)
set y(1) = xjpn
set y(2) = xfra
set y(3) = xsui
```

This is specific to a mean-only model. It sets up the formulas (the &i are needed in the formula definitions when the FRML is defined in a loop), and estimates them using **NLSYSTEM**. This both initializes the mean parameters, and computes the unconditional covariance matrix. If you want more general mean equations, the simplest way to do that would be to define each FRML separately.

```
dec vect b(n)
nonlin(parmset=meanparms) b
do i=1,n
    frml resid(i) = (y(&i)-b(&i))
end do i
nlsystem(parmset=meanparms,resids=u) gstart gend resid
compute rr=%sigma
```

The paths of the covariance matrices and UU are saved in the SERIES[SYMM] named H and UU. UX and HX are used to pull in residuals and H matrices.

```
declare series[symm] h uu
declare symm hx(n,n)
declare vect ux(n)
```

Chapter 9: ARCH and GARCH

These are used to initialize pre-sample variances.

```
gset h * gend = rr
gset uu * gend = rr
```

This is a standard (normal) log likelihood formula for any multivariate GARCH model. The difference among these will be in the definitions of HF and RESID. The function %XT pulls information out of a matrix of SERIES.

```
declare frml[symm] hf
frml logl = $
    hx = hf(t) , $
    %do(i,1,n,u(i)=resid(i)) , $
    ux = %xt(u,t) , $
    h(t)=hx, uu(t)=%outerxx(ux) , $
    %logdensity(hx,ux)

dec symm vcs(n,n) vas(n,n) vbs(n,n)
compute vcs=rr,vas=%const(0.05),vbs=%const(0.80)
nonlin(parmset=garchparms) vcs vas vbs
frml hf = vcs+vas.*uu{1}+vbs.*h{1}
nlpar(derives=second)
maximize(parmset=meanparms+garchparms,pmethod=simplex,piters=10,$
    iters=400) logl gstart gend
```

CC. The correlations are parameterized using an $(n-1) \times (n-1)$ matrix for the sub-diagonal. The (i,j) element of this will actually be the correlation between $i+1$ and j .

```
dec symm qc(n-1,n-1)
dec vect vcv(n) vav(n) vbv(n)
*
function hfcccgarch time
type symm hfcccgarch
type integer time
do i=1,n
    compute hx(i,i)=vcv(i)+$
        vbv(i)*h(time-1)(i,i)+vav(i)*uu(time-1)(i,i)
    do j=1,i-1
        compute hx(i,j)=qc(i-1,j)*sqrt(hx(j,j)*hx(i,i))
    end do j
end do i
compute hfcccgarch=hx
end
*
frml hf = hfcccgarch(t)
nonlin(parmset=garchparms) vcv vav vbv qc
compute vcv=%xdiag(rr),vav=%const(0.05),$
    vbv=%const(0.80),qc=%const(0.0)
maximize(parmset=meanparms+garchparms,pmethod=simplex,piters=10) $
    logl gstart gend
```


10. State Space and DSGE Models

In this chapter, we discuss Dynamic Linear (State Space) models and Dynamic Stochastic General Equilibrium models. Among the textbooks for which we provide worked examples, four of them are devoted to state-space models: Commandeur and Koopman (2007), Durbin and Koopman (2012), Harvey (1989), and West and Harrison (1997). Commandeur and Koopman is intended as a more practical text in state-space methods, the others are more theoretical. State-space models are also included as topics in Brockwell and Davis (2002), Hamilton (1994), Kim and Nelson(1999), Koop(2003), Lütkepohl(2006), Martin, Hurn and Harris(2013) and Tsay(2010).

We also have a State-Space/DSGE E-Course which goes into greater detail on the application of these methods using RATS. For more information, see:

https://estima.com/courses_completed.shtml

State Space Models
Filtering and Smoothing
The DLM Instruction
State Space Simulations
Initialization and Estimation
DSGE Models
Setup and Solving
Applications
Estimation

10.1 Dynamic Linear (State Space) Models

Dynamic linear or State-space models (DLM's) form a very broad class which can describe most of the models commonly used in time series work, either directly in their linear form, or indirectly, as an approximation to non-linear dynamics. The instruction **DLM** is the primary tool for dealing with these models in RATS.

The general form of a state-space model has quite a few components: unobservable states, observable data, shocks, mapping matrices. The **DLM** instruction has as many as twelve inputs, and almost as many potential outputs.

We will write the model in the following form:

$$(1) \quad \mathbf{X}_t = \mathbf{A}_t \mathbf{X}_{t-1} + \mathbf{Z}_t + \mathbf{F}_t \mathbf{W}_t$$

$$(2) \quad \mathbf{Y}_t = \alpha_t + \mathbf{C}_t' \mathbf{X}_t + \mathbf{V}_t$$

The $\mathbf{X}$'s are the (unobservable) *state variables*; the $\mathbf{Y}$ are the observable data. (1) is known as the *state equation*, *transition equation* or *system equation*. (2) is the *measurement* or *observation equation*. The $\mathbf{Z}$, if present, are exogenous variables in the evolution of the states. The $\mathbf{W}$ are shocks to the states; the $\mathbf{F}$ matrix has the loadings from those shocks to states, which allows for there to be fewer shocks than states (which will generally be the case). The μ are any components in the description of the observable which depend upon exogenous variables (such as a mean in the form of a linear regression). The $\mathbf{V}$ are measurement errors. The $\mathbf{W}$ and $\mathbf{V}$ are assumed to be mean zero, normally distributed, independent across time and independent of each other at time t as well.

The model (1) and (2) is too broad for many purposes, and too narrow for others. It is broader than often needed since many of the components will be either absent or at least won't be time-varying. For instance, in (1), the transition matrices $\mathbf{A}_t$ are usually time-invariant, as are the variances of shocks $\mathbf{W}_t$. And most models don't need the exogenous shift $\mathbf{Z}_t$.

It is too narrow since it allows for only one lag of the state, and both equations are linear. The first restriction is easily overcome; allowing for non-linearity is harder, but doable, at least approximately. Contemporaneous correlation between the shocks in the state equation $\mathbf{W}_t$ and those in the measurement equation $\mathbf{V}_t$ can be handled by adding $\mathbf{V}_t$ to the set of states; serial correlation in $\mathbf{V}_t$ is handled the same way.

What can state-space models do for us? In some cases, our main interest is in the unobservable states. In the typical application, the state-space model generates a decomposition of the observable $\mathbf{Y}_t$ into two or more *unobservable components*. The state-space model (known as a UC model) allows estimates of these. In other cases, the states are only a means to an end—we're interested in doing inference on some parameters governing the process (such as variances of the shocks, or free parameters in the $\mathbf{A}$ or $\mathbf{C}$ matrices), and the state-space model is the most convenient way to analyze this. In either case, we need to be able to estimate the states given the data.

10.2 Filtering and Smoothing

Let the data be represented as $\{\mathbf{Y}_1, \mathbf{Y}_2, \dots, \mathbf{Y}_T\}$. For any random variable ξ , we'll use the following abbreviation for the conditional density on a subsample of the $\mathbf{Y}_t$:

$$f(\xi | t) \equiv f(\xi | \mathbf{Y}_1, \mathbf{Y}_2, \dots, \mathbf{Y}_t).$$

There are three “natural” types of inference for $\mathbf{X}_t$:

1. Prediction: $f(\mathbf{X}_t | t-1)$
2. Filtering: $f(\mathbf{X}_t | t)$
3. Smoothing: $f(\mathbf{X}_t | T)$

When our main interest is in the $\mathbf{X}_t$ themselves, we'll usually want to do smoothing, putting all the information to use. Prediction and filtering are most useful when the states are only of indirect interest, since they maintain the sequencing of the data.

Note, by the way, that there is no reason that we need a value of $\mathbf{Y}_t$ for every $t = 1, \dots, T$. The state-space framework can handle missing data in a very simple fashion—it's one of its greatest advantages as a computational tool.

Ignoring (for now) the issue of what to do about $f(\mathbf{X}_1 | 0)$, the model (1) and (2), combined with the assumption that the shocks are normal, generates a joint normal density for $\{\mathbf{X}_1, \dots, \mathbf{X}_T, \mathbf{Y}_1, \dots, \mathbf{Y}_T\}$. The prediction, filtering and smoothing densities are thus normal with a mean that's a linear function of the $\mathbf{Y}$. A direct solution of that problem, however, can be quite slow and messy, requiring an inverse of a matrix the size of $\{\mathbf{Y}_1, \mathbf{Y}_2, \dots, \mathbf{Y}_T\}$. The *Kalman filter* and *Kalman smoother* are algorithms which solve this in a very efficient fashion, never needing an inverse larger than the dimension of just the single $\mathbf{Y}_t$.

The Kalman Filter

The Kalman filter is relatively simple. Since the filtered density $f(\mathbf{X}_{t-1} | t-1)$ is normal, we'll write its mean as $\mathbf{X}_{t-1|t-1}$ and variance as $\Sigma_{t-1|t-1}$. Solving (1) forward from $t-1$ to t gives the predictive density for $\mathbf{X}_t | t-1$ as normal with mean and variance

$$(3) \quad \mathbf{X}_{t|t-1} = \mathbf{A}_t \mathbf{X}_{t-1|t-1} + \mathbf{Z}_t, \Sigma_{t|t-1} = \mathbf{A}_t \Sigma_{t-1|t-1} \mathbf{A}_t' + \mathbf{F}_t \mathbf{M}_t \mathbf{F}_t'$$

where we're defining $\mathbf{M}_t \equiv \text{var}(\mathbf{W}_t)$. The predictive density for $\mathbf{Y}_t | t-1$ is then normal with mean and variance

$$(4) \quad \alpha_t + \mathbf{C}_t' \mathbf{X}_{t|t-1}, \mathbf{C}_t' \Sigma_{t|t-1} \mathbf{C}_t + \mathbf{N}_t$$

which is using $\mathbf{N}_t \equiv \text{var}(\mathbf{V}_t)$. These two equations give the predictive densities. Because $\{\mathbf{X}_t, \mathbf{Y}_t\} | t-1$ is jointly normal, the filtered density for $\mathbf{X}_t | t$ can be computed using standard results for conditional normals as having mean and variance

$$(5) \quad \mathbf{X}_{t|t} = \mathbf{X}_{t|t-1} + \Sigma_{t|t-1} \mathbf{C}_t (\mathbf{C}_t' \Sigma_{t|t-1} \mathbf{C}_t + \mathbf{N}_t)^{-1} (\mathbf{Y}_t - \alpha_t - \mathbf{C}_t' \mathbf{X}_{t|t-1})$$

$$(6) \quad \Sigma_{t|t} = \Sigma_{t|t-1} - \Sigma_{t|t-1} \mathbf{C}_t (\mathbf{C}_t' \Sigma_{t|t-1} \mathbf{C}_t + \mathbf{N}_t)^{-1} \mathbf{C}_t' \Sigma_{t|t-1}$$

(5) and (6) are sometimes known as the *correction* step. We correct our estimates based upon the prediction error for $\mathbf{Y}_t$. The matrix $\Sigma_{t|t-1} \mathbf{C}_t' (\mathbf{C}_t' \Sigma_{t|t-1} \mathbf{C}_t + \mathbf{N}_t)^{-1}$ from (5), which shows how we adjust the mean, is known as the *Kalman gain*.

The Kalman Smoother

The derivation of the Kalman smoother is much more complicated than the Kalman filter. The smoothed estimates for the end-of-sample period T are just the filtered estimates; the correction for previous time periods then requires a backwards pass through the data. The algorithm used in RATS is the one given in Durbin and Koopman (2012), which requires less calculation, generally is more accurate, and provides more information as a side effect than the one in (for instance) Hamilton (1994).

Limit Calculations

In most applications, the system matrices $\mathbf{A}$, $\mathbf{C}$, $\mathbf{F}$ and $\mathbf{Z}$ and the variance matrices $\mathbf{N}$ and $\mathbf{M}$ are constant—the only inputs that are time-varying are $\mathbf{Y}$ and (if present) μ . If that's the case, then it can be shown that (as long as $\mathbf{A}$ has no explosive roots) $\Sigma_{t|t}$ approaches a limit. If *that* approaches a limit, then so does the Kalman gain. Once that limit has been reached, the calculations can be simplified considerably by just using the previous values for the variance and gain matrices.

In applications where $\mathbf{Y}$ is univariate (which is probably 90% of the time), there isn't that much to be gained from taking advantage of this since the calculations are on relatively small matrices. However, in some situations $\mathbf{Y}_t$ is quite large (in a FAVAR model, it might have over 100 components), and there could be quite a savings in time to be made by exploiting this. The `LIMIT` option on `DLM` can be used to good effect in that case. It gives the number of entry points after which $\Sigma_{t|t}$ is to be considered converged. Note that this is not an absolute time period, but the number of entries relative to the start of the filter. You may have to do some experimenting to find the best value for the limit entry, but if you need to solve a large DLM repeatedly, it can be well worth the extra effort. Note, however, that you can't use this if you have any missing values.

Missing Values

When $\mathbf{Y}$ is univariate, the handling of missing values is quite simple: if $\mathbf{Y}_t$ (or μ_t) is missing, do the prediction calculation (3), but not the corrections (5) and (6). The “filtered” mean and variance if t isn't observable are simply the predicted values given $t-1$.

The same is done if $\mathbf{Y}_t$ is multivariate and no part of it is observable. It's a bit trickier if $\mathbf{Y}_t$ is multivariate and some components are observable and some are not. In some papers, you will see a recommendation to make the diagonal element of $\mathbf{N}_t$ very large for the missing components, which is how you would simulate the proper handling of this if you had relatively primitive Kalman filtering and smoothing software. Instead, however, RATS simply removes the components of $\mathbf{C}$, $\mathbf{N}$, $\mathbf{Y}$ and μ corresponding to the missing $\mathbf{Y}$ and calculates (4), (5) and (6) using those.

10.3 Setting up the System

The simplest non-trivial state-space model is the *local level* model, or random walk with noise. The single state variable follows the random walk:

$$(7) \quad x_t = x_{t-1} + w_t$$

and the measurement equation is

$$(8) \quad y_t = x_t + v_t$$

The interpretation of this model is that x_t is an (unobservable) local level or mean for the process. It's local in the sense that it can evolve from period to period because of the shock process w_t . The observable y_t is the underlying process mean contaminated with the measurement error v_t . If we compare this with (1) and (2), we can read off the following:

$$(9) \quad \mathbf{X}_t = [x_t], \mathbf{A}_t = [1], \mathbf{F}_t = [1], \mathbf{W}_t = [w_t], \mathbf{Y}_t = [y_t], \mathbf{C}_t = [1], \mathbf{V}_t = [v_t]$$

where all other components are zero. (Note, by the way, that the definition of $\mathbf{X}$ is only for interpretation and organization—you will never actually input it).

The AR(2) process

$$(10) \quad x_t = \varphi_1 x_{t-1} + \varphi_2 x_{t-2} + w_t$$

can be converted to the state equation

$$(11) \quad \mathbf{X}_t \equiv \begin{bmatrix} x_t \\ x_{t-1} \end{bmatrix} = \begin{bmatrix} \varphi_1 & \varphi_2 \\ 1 & 0 \end{bmatrix} \mathbf{X}_{t-1} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} w_t$$

Handling lags in the underlying model by adding x_{t-1} to the state vector is known as *state augmentation*. If x_t is the observable, note that the only shock is shifted into the state equation. The (non-zero) system matrices are

$$(12) \quad \mathbf{A}_t = \begin{bmatrix} \varphi_1 & \varphi_2 \\ 1 & 0 \end{bmatrix}, \mathbf{F}_t = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \mathbf{W}_t = [w_t], \mathbf{Y}_t = [x_t], \mathbf{C}_t = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \mathbf{V}_t = [0]$$

If $y_t = \tau_t + x_t + v_t$, where τ_t follows a random walk as in (7) and x_t is the AR(2) process (10), the combined state equation is

$$(13) \quad \mathbf{X}_t \equiv \begin{bmatrix} \tau_t \\ x_t \\ x_{t-1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \varphi_1 & \varphi_2 \\ 0 & 1 & 0 \end{bmatrix} \mathbf{X}_{t-1} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} w_t^{(\tau)} \\ w_t^{(x)} \end{bmatrix}$$

so we get

$$(14) \quad \mathbf{A}_t = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \varphi_1 & \varphi_2 \\ 0 & 1 & 0 \end{bmatrix}, \mathbf{F}_t = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, \mathbf{W}_t = \begin{bmatrix} w_t^{(\tau)} \\ w_t^{(x)} \end{bmatrix}, \mathbf{C}_t = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \mathbf{V}_t = [v_t]$$

This is example of the common practice of “adding” independent components to form a larger model. When you do this, the **A** and **F** matrices for the submodels are concatenated diagonally, and the **C** matrix is concatenated vertically. You can see, for instance, that the **A** matrix in (14) is just a block diagonal matrix with blocks from (9) and (12). The **F** matrix might be a bit harder to see because of the blocks aren’t square, but it is also formed by taking the **F** from (9) and (12) and blocking them diagonally. The $\sim\backslash$ diagonal concatenation can be helpful in building up the **A** and **F** matrices and the $\sim\sim$ vertical concatenation can be used for the **C**.

Getting Information into DLM

DLM has quite a few inputs: at a minimum, the **A**, **C** and **Y**, the variances for **V** and **W**, and there are often others. Some of those (such as the **Y**) will almost certainly be changing with time, while others will be fixed, and some, while they are fixed over time, might depend upon parameters that you’re trying to estimate, and thus won’t be known in advance.

These all go into **DLM** using options with names based upon the notation given in (1) and (2); the **A** matrix is input with the **A** option, **C** with the **C** option, etc. The options for the two variance matrices are **SV** and **SW**. In general, you can give the values for these in the most natural way. For instance, if you want the series **LOGGDP** to be your **Y**, the option would be **Y=LOGGDP**. If **C** is the vector $[1,1,0]$, the option could read **C=||1.0,1.0,0.0||**. (Note that if you have just one observable, the **C** can be input either as a row or a column vector.)

DLM will automatically detect whether the input information is time-varying so you don’t have to take special steps as you might with other state-space software.

F is handled by the **F** option, and defaults to the identity matrix, which is appropriate if the shocks are the same dimension as the states. Many implementations of state-space models don’t have the equivalent of the **F** matrix—if that’s the case, when the set of fundamental shocks is smaller dimension than the number of states, the covariance matrix of **W** will be singular. Either way works, but it’s generally much simpler to work with the **F** mapping matrix.

The two system matrices used the least are **Z** and μ —the options for these are called **Z** and **MU** and they default to zeros of the proper dimensions.

The options are simplest for a model with system matrices like (9). With just one state and one observable, everything is a scalar. The options on **DLM** to describe that model are **A=1**, **C=1**, **Y=series with data**, **SW=shock variance** and **SV=measurement variance**. If the variances aren’t known constants, and you want to estimate them, you can just use variable names; for instance, if the shock variance is represented by the variable **SIGSQETA** and the measurement variance by **SIGSQEPS**, you would use **SW=SIGSQETA** and **SV=SIGSQEPS**.

Once you have a model with more than one state, it’s generally easier to create the system matrices using separate instructions. For instance, with the model with ma-

trices (12), you can set up **C** and **F** with

```
compute c=||1.0|0.0||,f=||1.0|0.0||
```

(While we're giving the matrices the same name as the option, that's not required). More generally, you could use the `%UNITV` function:

```
compute c=%unitv(2,1),f=%unitv(2,1)
```

where you can change the number of states (lags) easily by replacing the 2 in these with the number of lags in the representation.

The proper handling of the **A** matrix for this model will depend upon whether the φ_i are known constants, or need to be variables. If they're known (say 1.1 and -.3), you can create the matrix with something like:

```
compute a=$  
||1.1,-0.3|  
1.0, 0.0||
```

Note that this could also be done as the more compressed (but harder to read):

```
compute a=||1.1,-0.3|1.0,0.0||
```

Suppose instead that they need to be variables (call them `PHI1` and `PHI2`). In that case, we need to create a `FRML[RECT]`, that is, a **FRML** that evaluates to a `RECTANGULAR` matrix. You can do that with:

```
dec frml[rect] af  
frml af =$  
||phi1,phi2|  
1.0, 0.0||
```

The **FRML** instruction can create formulas which evaluate to something other than a real, but the target has to have its type **DECLARED** first as is done here. With this definition, the option on **DLM** would read `A=AF`.

A more flexible alternative to defining a `FRML[RECT]` is to create a **FUNCTION** (Section 4.12) which returns a `RECTANGULAR`:

```
function adlm  
type rect adlm  
compute adlm=$  
||phi1,phi2|  
1.0, 0.0||  
end
```

The **A** matrix information would then go into **DLM** with `A=ADLM()`.

The matrices in (14) are more typical of an actual application of a state-space model, where two (or more) component models are added together to create the overall model.

Note that, even if the φ_i are unknown, the **F** and **C** matrices are fixed. So the matrices could be set up with

```
compute f=$
||1.0,0.0|$
  0.0,1.0|$
  0.0,0.0||
compute c=||1.0,1.0,0.0||
function adlm
type rect adlm
compute adlm=$
  1.0~\$
||phi1,phi2|$
  1.0, 0.0||
end
```

with options **A=ADLM()**, **F=F** and **C=C**. The **SW** matrix will need to be a 2×2 . In most cases, the components of **W** are assumed to be independent of each other, so **SW** will be diagonal. Diagonal matrices are created most easily using the **%DIAG** function, which takes a **VECTOR** and creates a diagonal matrix from it. If we call the two component variances **SIGSQETA** and **SIGSQC**, the option could read **SW=%DIAG(||SIGSQETA,SIGSQC||)**.

Getting Information from DLM

DLM has many uses and generates quite a bit of information. You can, for instance, get the estimated states and their variances, predicted observables and prediction errors and variances, state shock estimates and variances, and more. Most of these are retrieved as a **SERIES** of matrices. For instance, the estimated states are a **SERIES[VECTOR]**—at each time period analyzed by **DLM**, there's a complete **VECTOR**. In most cases, you will be interested in the time path of only one or two components of this. You can pull items out easily with **SET** instructions. If **STATES** is the name of your output, the following will pull the first component out of this:

```
set xstate = states(t)(1)
```

Note that you have to use the two sets of subscripts: the **(t)** is used to pick the entry out of the **SERIES** of **VECTORS** and the **(1)** pulls the first element out of that vector. The variances of the states (and most other variances) are produced as a **SERIES** of **SYMMETRICS**. Again, **SET** is generally the easiest way to pull information out of this. For instance, if **SIGMAS** is the name assigned to the series of covariance matrices, to get the standard deviation of the first state, do

```
set xstddev = sqrt(sigmast(t)(1,1))
```

The first element in either a **VECTOR** or **SYMMETRIC** (the **(1)** or **(1,1)**) can also be extracted using the **%SCALAR** function. Using this, the two **SET** instructions here could also have been done as:

```
set xstate = %scalar(states)
set xstddev = sqrt(%scalar(sigmast))
```


@LOCALDLM Procedure

One of the most important basic state-space models is the *local trend* model. One way to write this is

$$\begin{aligned} x_t &= x_{t-1} + \tau_{t-1} + \xi_t \\ (15) \quad \tau_t &= \tau_{t-1} + \zeta_t \\ y_t &= x_t + \varepsilon_t \end{aligned}$$

This has two states: a local trend rate (τ_t) and the local trend itself (x_t). If the variance of ζ_t is zero, the trend rate is constant, and the series is a linear trend plus noise. If that variance is non-zero, the trend rate can evolve. The model is small enough that it isn't hard to write down the system matrices, which are just matrices with 1's and 0's. However, for convenience, we have a procedure **@LocalDLM** which constructs these, which can then be used with **DLM**. The full model above is done with

```
@localdml(type=trend,shocks=both,a=at,c=ct,f=ft)
```

SHOCKS=BOTH allows for the shocks in each of the state equations. More commonly, the shock in the first equation is suppressed, so the default is SHOCK=TREND. The SHOCKS option changes the F matrix, but not the others. See page UG–339 for another example using **@LocalDLM**.

@SEASONALDLM Procedure

The procedure **@SeasonalDLM** sets up the system matrices for one of two representations of an evolving seasonal. The “sum” of a seasonal model and a trend model (such as is created by **@LocalDLM**) creates what Harvey calls the *basic structural model* which can model fairly well many series with a strong seasonal. The two options for the seasonal type are TYPE=FOURIER and TYPE=ADDITIVE. The Fourier gives a smoother pattern, and so is the best choice when the seasonal is linked to weather. If the seasonal is more due to the secular calendar (dominated by holidays, school year and the like), the additive seasonal is a better choice. An example is

```
@seasonaldml(type=additive,a=as,f=fs,c=cs)
```

We use **@SeasonalDLM** and **@LocalDLM** in many textbook examples included with RATS, including several from Commandeur and Koopman (2007) and Harvey (1989).

@ARMADLM Procedure

ARMA equations have, in general, many possible state-space representations. If there are any MA components, they can be fairly complicated to set up. Instead, you can use the **@ARMADLM** procedure which converts an EQUATION (with coefficients already defined) into state-space form. In addition to A and F, the example below also defines the Z vector as the constant state shift due to the constant in the equation.

```
boxjenk(ma=||2,4||,maxl,constant,define=dyeq) dy
@armadlm(a=adlm,f=fdlm,z=zdlm) dyeq
```

10.4 The DLM Instruction

The syntax for **DLM** is

```
dlm( options )   start   end   state vectors   state variances
```

Most of the input and output comes through the options. The two primary outputs, however, are the *state vectors* and *state variances* which give the estimated mean and variance of the states.

The **TYPE** option controls the type of analysis done on the model. Choose **TYPE=FILTER** (which is the default) for Kalman filtering, and **TYPE=SMOOTH** for Kalman smoothing. (Choices for simulations are in Section 10.5).

The Examples

The four examples in the next two sections show various techniques applied to the Nile river flow data (100 years of annual data) used at the beginning of Durbin and Koopman(2012). The state-space model for this is the local level model (9). The **A** and **C** option settings are from (9). We're using specific values for the variances from the book. **PRESAMPLE=DIFFUSE** handles the $f(\mathbf{X}_1 | 0)$ that we ignored earlier—we'll describe that in Section 10.6.

```
open data nile.dat  
calendar 1871  
data (format=free,org=columns,skips=1) 1871:1 1970:1 nile
```

TYPE=SMOOTH: DLMEXAM1.RPF example

DLMEXAM1.RPF does Kalman smoothing over the full sample (thus the / for the range). The smoothed means for the underlying level are extracted into the series **A** and the variances into **P**. The mean level and actual data are then graphed with a 90% confidence interval.

```
dlm(a=1.0,c=1.0,sv=15099.0,sw=1469.1,y=nile,presample=diffuse,$  
  type=smooth) / xstates vstates  
set a      = %scalar(xstates)  
set p      = %scalar(vstates)  
set lower  = a+sqrt(p)*%invnormal(.05)  
set upper  = a+sqrt(p)*%invnormal(.95)  
  
graph(footer=$  
  "Figure 1. Smoothed state and 90% confidence intervals") 4  
# nile  
# a  
# lower / 3  
# upper / 3
```

TYPE=FILTER: DLMEXAM2.RPF example

Example file DLMEXAM2.RPF uses the same data set, doing predictions for 10 years beyond the sample. That can be done by simply running the Kalman filter (TYPE=FILTER) from the start of the sample to the end of the forecast period. Once there is no longer data in the **Y** series, the Kalman filter just does the prediction step and not the correction step. This uses several other options for output. The **YHAT** option gives the predictions for **Y** and **SVHAT** gives the forecast variance.

```
dln(a=1.0,c=1.0,sv=15099.0,sw=1469.1,y=nile,presample=diffuse,$
    type=filter,yhat=yhat,svhat=svhat) * 1980:1

set forecast 1971:1 1980:1 = %scalar(yhat)
set stderr   1971:1 1980:1 = sqrt(%scalar(svhat))

set lower 1971:1 1980:1 = forecast+%invnormal(.25)*stderr
set upper 1971:1 1980:1 = forecast+%invnormal(.75)*stderr

graph(footer="Out-of-sample Forecasts with 50% CI") 4
# nile      1931:1 1970:1
# forecast  / 2
# lower    / 3
# upper    / 3
```

10.5 Simulations

It's easy to generate an *unconditional* draw for the states because of the sequential independence. Given $\mathbf{X}_{t-1}$, we draw $\mathbf{W}_t$ and compute

$$(16) \quad \mathbf{X}_t = \mathbf{A}_t \mathbf{X}_{t-1} + \mathbf{Z}_t + \mathbf{F}_t \mathbf{W}_t$$

We repeat this as necessary. This is what is done with **DLM** with the option `TYPE=SIMULATE`. That, however, isn't very interesting since the generated states will have no relationship to the observed data. The one real use of this is for out-of-sample simulations, allowing for analysis of more complicated functions of forecasts than just the mean and variance.

Of greater interest is a draw for the states *conditional* on the data. The Kalman smoother gives us the mean and covariance matrix for each state individually, but that isn't enough to allow us to do draws, since conditional on the data, the states are highly correlated. Conditional simulation is even more complicated than Kalman smoothing, but can be done with a couple of Kalman smoothing passes, one on simulated data. Conditional simulation is chosen in RATS with the option `TYPE=CSIMULATE` (conditional simulation).

The main use of conditional simulation is in Bayesian techniques such as Markov Chain Monte Carlo. In fact, it is sometimes known as *Carter-Kohn*, after the algorithm described in Carter and Kohn (1993) as a step in a Gibbs sampler. Conditional on data and other parameters, it gives a draw for the states. The next step would then be to produce a draw for the other parameters, conditional on the states (and data). See Chapter 16 for more on these (advanced) techniques.

TYPE=SIMULATE: DLMEXAM3.RPF example

As a simple example, we'll work with the Nile data again. This is from example file `DLMEXAM3.RPF`. We first Kalman filter through the data period, which will give us the end-of-data mean and variance for the state. We then simulate (10000 times) the model for 50 periods beyond the end of data observed data, keeping track of the maximum value achieved by the flow in each simulation. The **DLM** instruction that does the simulation uses the `X0` and `SX0` options for inputting the initial mean and variance as the final values from the Kalman filter.

```
dlim(a=1.0,c=1.0,sv=15099.0,sw=1469.1,presample=diffuse,y=nile,$
    type=filter) / xstates vstates
compute ndraws=10000
set maxflow 1 ndraws = 0.0
do draw=1,ndraws
    dlim(a=1.0,c=1.0,sv=15099.0,sw=1469.1,$
        x0=xstates(1970:1),sx0=vstates(1970:1),$
        type=simulate,yhat=yhat) 1971:1 2020:1 xstates
    set simflow 1971:1 2020:1 = %scalar(yhat)
    ext(noprint) simflow
    compute maxflow(draw)=%maximum
end do reps
```

TYPE=CSIMULATE: DLMEXAM4.RPF example

Example file DLMEXAM4.RPF again uses the river flow data. Most uses of TYPE=CSIMULATE will be more complicated, but this gives some idea of how to get information out of the simulations. This Kalman smooths through the data, and uses the WHAT and VWHAT options to save smoothed estimates of the $\mathbf{W}$ and $\mathbf{V}$.

There are many (an infinite number) of combinations of $\mathbf{W}$ and $\mathbf{V}$ shocks that will produce the observed data. Kalman smoothing computes the “most likely” (highest density) set of shocks. Conditional simulation draws from the joint distribution for which the smoothed estimates are the mean. With TYPE=CSIMULATE, the WHAT and VWHAT options give the simulated values for the shocks.

This saves the values for one specific data point (1913, which is a fairly large outlier) and graphs the density for $V(1913:1)$ of the simulated draws.

```
dln(a=1.0,c=1.0,sv=15099.0,sw=1469.1,presample=diffuse,$
    y=nile,type=smooth,what=what,vhat=vhat) / xstates vstates
compute ndraws=10000

set what_h 1 ndraws = 0.0
set vhat_h 1 ndraws = 0.0

do draw=1,ndraws
    dln(a=1.0,c=1.0,sv=15099.0,sw=1469.1,presample=diffuse,$
        y=nile,type=csimulate,what=what_s,vhat=vhat_s)

    compute what_h(draw) = what_s(1913:1)(1)
    compute vhat_h(draw) = vhat_s(1913:1)(1)

end do draw

density(smoothing=1.5) vhat_h 1 ndraws xv fv

scatter(hgrid=vhat(1913:1)(1),style=lines,footer=$
    "Conditional Density of V at 1913 with Smoothed Estimate")
# xv fv
```

10.6 Initialization

Inputting the Initial Mean and Variance

The calculations require one other piece of information: the initial distribution of the states. In the earlier examples, we've had to use some of the options from this section.

The initial distribution requires a mean and a variance. For models with just one or two states, it might be possible to come up with a reasonable setting for this, with a mean in the right range, and a variance large enough to cover any likely value. If you *have* pre-sample settings, use the options `X0` (for the mean) and `SX0` (for the covariance matrix) to input them to **DLM**. We used these in the example of unconditional simulation on page UG–332 where we had the filtered mean and variance from the end of the sample. The default for `X0` is a vector of zeros.

One thing to note is that **DLM** wants the pre-sample information to give $\mathbf{X}_{0|0}, \Sigma_{0|0}$. Some books and papers use a slightly different description of the state-space model where the pre-sample information is more naturally in the form $\mathbf{X}_{1|0}, \Sigma_{1|0}$ (in our notation). In general, these *aren't* the same. If you want your input values to be interpreted in this other way, also include the option `PRESAMPLE=X1`.

Ergodic Solution

If $\mathbf{A}$, $\mathbf{Z}$, $\mathbf{F}$ and $\text{var}(\mathbf{W})$ are all time-invariant (which is usually the case), we can think about solving the model for the steady-state or *ergodic* distribution. If we take expected values in the state equation and assume we have a common expectation, we get

$$(17) \quad \mathbf{E}\mathbf{X} = \mathbf{A}\mathbf{E}\mathbf{X} + \mathbf{Z}, \text{ or } \mathbf{E}\mathbf{X} = (\mathbf{I} - \mathbf{A})^{-1} \mathbf{Z}$$

If there is no $\mathbf{Z}$ component, the presample mean will just be zero, which is the default. If we take variances (assuming the variance of $\mathbf{W}$ is constant), we get the equation:

$$(18) \quad \Sigma_{\mathbf{x}} = \mathbf{A}\Sigma_{\mathbf{x}}\mathbf{A}' + \Sigma_{\mathbf{w}}$$

The variance equation, while taking a somewhat odd form, is, in fact, a linear equation in the elements of $\Sigma_{\mathbf{x}}$. By expanding this, it can be solved by standard linear algebra techniques. The standard textbook solution for this (see, for instance, Hamilton(1994), page 378) is to rearrange it into the linear system:

$$(19) \quad [\mathbf{I} - \mathbf{A} \otimes \mathbf{A}] \text{vec}(\Sigma_{\mathbf{x}}) = \text{vec}(\Sigma_{\mathbf{w}})$$

As written, this has some redundant elements, since $\Sigma_{\mathbf{x}}$ is symmetric. Still, with those eliminated, it requires solving a linear system with $n(n+1)/2$ components. The solution procedure for this requires $O(n^6)$ arithmetic operations. This starts to dominate the calculation time for the entire Kalman filter process for even fairly modest values of n . Instead of this “brute force” solution, RATS uses a more efficient technique described in Doan (2010), which has a solution time $O(n^3)$. Select this calculation for the initial mean and variance using the option `PRESAMPLE=ERGODIC`. If you need to do the calculation separately, you can use the function `%PSDINIT(A, SW)`, which returns the solution to (18) for the input values for $\mathbf{A}$ and the variance for $\mathbf{W}$.

Diffuse Prior

The ergodic solution described above only exists if all the eigenvalues of $\mathbf{A}$ are inside the unit circle. The “solution” of (18) when $\mathbf{A}=1$ is $\Sigma_{\mathbf{x}} = \infty$. The approach generally taken in state-space modelling to deal with this has been to set the pre-sample mean to zero and use a diagonal matrix with “large” elements for the covariance. One problem with this approach is that a value which is large enough to be effectively infinite for one of the states might not be large enough for another. There are also round-off error problems in the calculation of the state covariance matrix.

An alternative to this approximation was provided by Koopman (1997), which is known as *exact diffuse initialization*. This was later refined in Durbin and Koopman (2012). This does the actual limits as the variances go to infinity. It is implemented by writing the covariance matrices as a sum of “infinite” and “finite” parts, which are updated separately in the Kalman filter and Kalman smoother. If you use the option `PRESAMPLE=ERGODIC`, **DLM** will recognize that you have non-stationary roots and adjust the calculation to use exact diffuse initialization. This will also correctly handle the case where there are a mix of unit and stationary roots. See Doan (2010) for more information on how this works.

If you have a model where all roots are known to be unit roots, you can use the option `PRESAMPLE=DIFFUSE`. That will give the same result as `PRESAMPLE=ERGODIC`, but won’t require the extra step (internally) to analyze the roots, and so will work somewhat faster. We used this several times in the short examples earlier in the chapter.

Conditioning on Early Values

If you have a non-stationary model, the initial state vector is usually represented by a diffuse prior. If, rather than using the recommended `PRESAMPLE=ERGODIC` or `PRESAMPLE=DIFFUSE` to do exact handling of this, you use large finite values, the likelihood could be dominated by the first few observations where the variance is high. If you do this, you can use the option `CONDITION=initial periods` to indicate the number of observations which should *not* be incorporated into the likelihood function. You still include these in the estimation range—the likelihood actually used will just condition on them. Typically, you would condition on the number of states, but it might be a smaller number if the initial state vector is non-stationary only in a reduced number of dimensions.

The calculated likelihood (Section 10.7) is the only thing affected by the use of the `CONDITION` option. Note that this is only slightly different from what happens in the exact handling of the diffuse prior, which omits from the likelihood any observation which has an “infinite” predictive variance. Because the likelihood with the exact diffuse prior depends on the number of unit roots, `CONDITION` also has value where the number of unit roots isn’t known; for instance, if you have an AR component that might or might not be stationary.

10.7 Estimating Parameters

The Kalman filter gives us the predictive density for $\mathbf{Y}_t$ given $\{\mathbf{Y}_1, \dots, \mathbf{Y}_{t-1}\}$ as

$$(20) \quad \mathbf{Y}_t \sim N(\alpha_t + \mathbf{C}'_t \mathbf{X}_{t|t-1}, \mathbf{C}'_t \Sigma_{t|t-1} \mathbf{C}_t + \mathbf{N}_t)$$

We can use this with the standard time-series trick of factoring by sequential conditionals to get the full sample likelihood:

$$(21) \quad p(\mathbf{Y}_1, \dots, \mathbf{Y}_T) = p(\mathbf{Y}_1) p(\mathbf{Y}_2 | \mathbf{Y}_1) p(\mathbf{Y}_3 | \mathbf{Y}_1, \mathbf{Y}_2) \dots p(\mathbf{Y}_T | \mathbf{Y}_1, \mathbf{Y}_2, \dots, \mathbf{Y}_{T-1})$$

The log likelihood function for the full sample is thus (omitting additive constants, although those will be included in the values reported by **DLM**)

$$(22) \quad -\frac{1}{2} \sum_t \log |\mathbf{C}'_t \Sigma_{t|t-1} \mathbf{C}_t + \mathbf{N}_t| - \frac{1}{2} \sum_t (\mathbf{Y}_t - \alpha_t - \mathbf{C}'_t \mathbf{X}_{t|t-1})' (\mathbf{C}'_t \Sigma_{t|t-1} \mathbf{C}_t + \mathbf{N}_t)^{-1} (\mathbf{Y}_t - \alpha_t - \mathbf{C}'_t \mathbf{X}_{t|t-1})$$

This is the objective function used for Kalman filtering (and smoothing). Whether you estimate parameters or not, **DLM** will set the variables %LOGL and %FUNCVAL to this or a similar value as discussed later.

To estimate a model, use **NONLIN** to set up a PARMSET with the free parameters you want to estimate. On **DLM** itself, you have a choice of several estimation methods. Of those, only **BFGS** and **GAUSSNEWTON** can compute standard errors for the coefficients, and **GAUSSNEWTON** can only be applied if you have just one observable. The derivative-free methods (**SIMPLEX**, **GENETIC**, **ANNEALING** and **GA**) are generally used only as preliminary methods (**PMETHOD** option) to refine initial guesses.

Variance Scale Factors

There are three variance terms in the model: Σ_w , Σ_v , and $\Sigma_{0|0}$. By default, **DLM** calculates the likelihood assuming that these are known (or being conditioned upon). However, as with linear regressions, it's more common that the variance in the measurement equation isn't known. The simplest way to handle this is to assume that all three variance (matrices) are known up to a single (unknown) constant multiplier—this is often a reasonable assumption since, if **C** is fixed, the scale of the **X**'s will have to adjust to match the scale of **Y** anyway.

There are two options available for estimating this multiplier: it can be concentrated out, or it can be given a prior distribution. This is controlled by the option **VARIANCE**. This scale factor is usually, though not always, the unknown variance in the measurement equation, hence the option name. **VARIANCE=CONCENTRATED** requests that this value be concentrated out, while **VARIANCE=CHISQUARED** gives it an (inverse) chi-squared prior. If, as is typical, this is the variance in the measurement equation, you use the option **SV=1**. Note well: if you choose either one of these, *all* the estimated variance matrices will need to be scaled by the estimated variance. The default for the **VARIANCE** option is **VARIANCE=KNOWN**, which means that all the variances are to be used as given.

With VARIANCE=CONCENTRATED, if we write $\mathbf{N}_t = \lambda n_t$, the log likelihood element can be rewritten as (for simplicity, omitting μ_t)

$$(23) \quad -\frac{1}{2} \sum_t \log |\mathbf{C}'_t (\lambda \Sigma_{t|t-1}) \mathbf{C}_t + \lambda n_t| - \frac{1}{2} \sum_t (\mathbf{Y}_t - \mathbf{C}'_t \mathbf{X}_{t|t-1})' (\mathbf{C}'_t (\lambda \Sigma_{t|t-1}) \mathbf{C}_t + \lambda n_t)^{-1} (\mathbf{Y}_t - \mathbf{C}'_t \mathbf{X}_{t|t-1})$$

where λ is that unknown scale factor. λ can be isolated to give

$$(24) \quad \frac{-rank}{2} \log \lambda - \frac{1}{2\lambda} \sum_t \frac{(\mathbf{Y}_t - \mathbf{C}'_t \mathbf{X}_{t|t-1})' (\mathbf{Y}_t - \mathbf{C}'_t \mathbf{X}_{t|t-1})}{(\mathbf{C}'_t \Sigma_{t|t-1} \mathbf{C}_t + n_t)} - \text{terms without } \lambda$$

where rank is the sum of the ranks of the $\mathbf{C}'_t \Sigma_{t|t-1} \mathbf{C}_t + n_t$. This will just be the number of observations if you have one observable and no missing values. The maximizing value for λ is

$$(25) \quad \hat{\lambda} = \frac{1}{rank} \sum_t (\mathbf{Y}_t - \mathbf{C}'_t \mathbf{X}_{t|t-1})' (\mathbf{C}'_t \Sigma_{t|t-1} \mathbf{C}_t + n_t)^{-1} (\mathbf{Y}_t - \mathbf{C}'_t \mathbf{X}_{t|t-1})$$

DLM sets the variable %VARIANCE to this value. Substituting this in gives the concentrated likelihood (again, omitting additive constants)

$$(26) \quad -\frac{rank}{2} \log \lambda \mathbb{E} \frac{rank}{2} - \frac{1}{2} \sum_t \log |\mathbf{C}'_t \Sigma_{t|t-1} \mathbf{C}_t + n_t|$$

which is what **DLM** will maximize if you are estimating parameters. You *should* get the same results by estimating all the variances directly. However, the likelihood function is often very flat in the common scale factor, so the full set of variances may be close to being perfectly correlated. If this happens, you can run into problems with the optimization routine stalling out short of the maximum.

With VARIANCE=CHISQUARED, you need to provide two pieces of information in order to complete the prior: the “mean” (PSCALE=*prior scale*) and the degrees of freedom (PDF=*prior degrees of freedom*). When you Kalman filter through the data, the estimates of the variance will also be updated. If you need to track that, you can use the options SIGHISTORY=*series of estimated variances* and DFHISTORY=*series of degrees of freedom of variance estimates*. When you use VARIANCE=CHISQUARED, the conditional distribution of $\mathbf{Y}$ is a t , not a Normal, and the likelihood is computed using the t density. See West and Harrison (1997) if you’re interested in this approach.

Estimating Variances

In many models, the only unknown parameters are the variances of the components. It seems straightforward to estimate these by maximizing the log likelihood. What complicates this is that there is nothing in the likelihood itself that constrains the individual variances to be non-negative. The only requirement for evaluating the log

likelihood element at t is that the predictive covariance matrix from (20) be positive-definite. That means that $\mathbf{N}_t$ could be negative if the first term is positive enough and vice versa. And, unfortunately, this problem comes up all too often.

There are two main approaches for dealing with this. The first is to estimate the model unconstrained; if a variance comes in negative, zero it out and re-estimate. If that works, it should be the global maximum. If you have enough components that *two* variances come in negative, it's not as clear how to proceed. In that case, it's probably best to use RATS' ability to put inequality constraints on parameters (Section 4.4)

The other method is to parameterize the variances in logs. Although that doesn't allow a true zero value, a variance which really should be zero will show up as something like -30 in logs.

There's one other numerical problem that can come up when you estimate the variances directly. Even non-zero variances can sometimes be *very* small. It can be very hard for numerical optimizers to deal with parameters that have such naturally small scales—that is, it can be very hard to distinguish between a parameter which is naturally small and one that is small because it's really supposed to be zero. You can sometimes fix this fairly easily by just multiplying the data (*after* such things as log transformation) by 100. That increases all the component variances by a factor of 10000, which is usually enough to fix this problem. Parameterizing the variances in logs will also take care of this type of problem. Concentrating out a variance also can help, since it often makes the only really small variance the one that is estimated constructively using the expression in (24).

Example

This estimates the MA(1) model in the form shown below. We'll concentrate out the variance of ε , so SW is 1. (SV is zero, since there is no measurement error in this form).

$$(27) \quad \mathbf{X}_t = \begin{bmatrix} \varepsilon_t \\ \varepsilon_{t-1} \end{bmatrix}, \mathbf{A} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \mathbf{F} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \mathbf{w}_t = [\varepsilon_t], \mathbf{C}' = [1 \quad \theta], \mathbf{v}_t = 0$$

```
nonlin theta
compute a=||0.0,0.0||1.0,0.0||
compute f=%unitv(2,1)

compute theta=0.0
dlm(a=a,f=f,c=||1.0,theta||,y=ldiff,sw=1.0,var=concentrate,$
presample=ergodic,method=gauss) 1959:2 2008:4
```

Hodrick-Prescott Filter: HPFILTER.RPF example

The Hodrick-Prescott (1997) filter has been used extensively to extract a growth component from macroeconomic series, leaving the cyclical element. The HP filter is a special case of the local trend model (15).

This is available in RATS using the **FILTER** instruction with the option **TYPE=HP**. However, **FILTER(TYPE=HP)** returns just the smoothed trend. If we want to do more than that, we need to set up the state-space model and use **DLM**.

In addition to omitting the “level” shock, in the local trend, the HP filter pegs the ratio of the variances between the measurement error and the trend shock to 1600 for quarterly data. For illustration, in **HPFILTER.RPF**, we estimate the pair of variances. Before you do this, you should first ask whether you really want to. Attempts to estimate the variances in local trend models often produce results which look “wrong”. In particular, the common problem is that maximum likelihood produces a trend which isn’t stiff enough to fit our understanding of what a trend should be. This is a particularly extreme case, since with this data set, the estimated “trend” ends up almost exactly reproducing the data, to the point that you can’t even see both lines on a graph. The root cause of this is that HP proposed a calculation which, in effect, specified how stiff the trend was. Their calculation translates into Kalman smoothing on a local trend model, but *not* a local trend model with freely estimated variances.

Set up the local trend model using only the trend shock. Do Kalman smoothing using the HP pegged ratio for the variances. Extract the local trend rate, which is the second components out of **HPSTATES**.

```
@localdml(type=trend,shocks=trend,a=ahp,c=chp,f=fhp)
compute lambda = 1600.0

dml(a=ahp,c=chp,f=fhp,sv=1.0,sw=1.0/lambda,presample=diffuse,$
    type=smooth,var=concentrate,y=lgdp) / hpstates
set hprate = hpstates(t) (2)
```

Now estimate the two variances. These are parameterized in logs, so the **SV** and **SW** options use **exp(..)** functions to transform back to levels:

```
nonlin lsv lsw
compute lsv=-3.0,lsw=-7.0
dml(a=ahp,c=chp,f=fhp,y=lgdp,sv=exp(lsv),sw=exp(lsw),$
    presample=diffuse,type=smooth,method=bfgs) / ltstates
```

Extract the trend rate from **LTSTATES** and graph vs the HP estimate.

```
set ltrate = ltstates(t) (2)
graph(header="Comparison of Local Trend Rates",key=below,$
    klabels=||"From HP Filter","From Estimated Local Trend"||) 2
# hprate
# ltrate
```

Stochastic Volatility Model: SV.RPF example

A direct competitor with the GARCH model (Chapter 9) for modeling the variances of financial time series is the stochastic volatility model. In its simplest form, this is represented by

$$(28) \quad y_t = \varepsilon_t \sqrt{h_t}$$

$$(29) \quad \log h_t = \gamma + \varphi \log h_{t-1} + w_t$$

where $\varepsilon_t \sim N(0,1)$, $w_t \sim N(0,\sigma^2)$. Estimating the free parameters of this model is somewhat tricky because the variances (h_t) aren't observable and are subject to the random shock w_t . (29) is a perfectly good state equation for $\log h_t$, but (28) isn't in the proper form for a measurement equation. However, if we square and log (28), we get

$$(30) \quad \log y_t^2 = 1 \times \log h_t + \log \varepsilon_t^2$$

which would be usable, except that $\log \varepsilon_t^2$ isn't Normally distributed. It is, however, a distribution with a known mean and variance, so the model's parameters can be estimated by quasi-maximum likelihood using **DLM**. This is from example SV.RPF.

MEANX2 is the (exact) mean of log chi-square, VARX2 is the variance

```
compute meanx2 = %digamma(0.5) - log(0.5)
compute varx2  = %trigamma(0.5)
```

GAMMA and PHI are very highly correlated when PHI is near 1, which makes estimation by general hill-climbing procedures somewhat tricky. Instead, we reparameterize this to use GAMMAX=GAMMA (1-PHI) in place of GAMMA.

```
set dlogp = log(usxuk{0}/usxuk{1})
diff(center) dlogp / demean
nonlin phi sw gammax
set ysq = log(demean^2) - meanx2
```

Get initial guess values from ARMA(1,1) model. PHI is taken directly as the AR(1) coefficient The variance SW is backed out by matching first order autocorrelations in the MA term. GAMMA is chosen to reproduce the mean of the YSQ series

```
boxjenk(ar=1,ma=1,constant,noprint) ysq
compute phi = %beta(2), $
      sw = -phi*varx2*(1+%beta(3)^2)/%beta(3) - (1+phi^2)*varx2
compute sw = %if(sw<0,.1,sw)
compute gammax = %mean
```

Estimate the unconstrained model

```
dlim(method=bfgs,sw=sw,sv=varx2,y=ysq,type=filter,c=1.0, $
      sx0=sw/(1-phi^2),x0=gammax,a=phi,z=gammax*(1-phi)) 2 * states
```

Business Cycle Model: DLMCYCLE.RPF example

This is an example with more than one observable. It's a simple case of a common cycle model, but demonstrates many of the requirements for handling a more involved dynamic factor model. The model takes the form

$$\begin{aligned} C_t &= \phi_c C_{t-1} + w_{ct} \\ (31) \quad y_{it} &= \mu_i + \gamma_i C_t + \chi_{it} \\ \chi_{it} &= \phi_i \chi_{i,t-1} + w_{it} \end{aligned}$$

where C is the (unobservable) common cycle, the y are observable macroeconomic series (in growth rates) and the χ are the measurement errors in the observation equations. Because the measurement errors are assumed to be serially correlated (with AR(1) processes), we need to add those to the state equation. With two observables, the state-space model takes the form

$$\begin{aligned} (32) \quad \mathbf{X}_t &\equiv \begin{bmatrix} C_t \\ \chi_{1t} \\ \chi_{2t} \end{bmatrix} = \begin{bmatrix} \phi_c & 0 & 0 \\ 0 & \phi_1 & 0 \\ 0 & 0 & \phi_2 \end{bmatrix} \begin{bmatrix} C_{t-1} \\ \chi_{1,t-1} \\ \chi_{2,t-1} \end{bmatrix} + \begin{bmatrix} w_{ct} \\ w_{1t} \\ w_{2t} \end{bmatrix} \\ y_t &\equiv \begin{bmatrix} y_{1t} \\ y_{2t} \end{bmatrix} = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix} + \begin{bmatrix} \gamma_1 & 1 & 0 \\ \gamma_2 & 0 & 1 \end{bmatrix} \mathbf{X}_t \end{aligned}$$

One issue with any model such as this is that the scale of C isn't identified—if you multiply C by λ and divide all the γ by λ , the model is unchanged. There are two ways to handle this: either fix the scale of C (most easily by pegging the variance of w_c to 1) or fix one of the two loadings, such as pegging γ_1 at 1. The former is safer, since it's possible (though unlikely) that you could be pegging a loading which is actually zero theoretically.

Example file `DLMCYCLE.RPF` estimates this with growth rates of income and consumption as the two observables. This sets up the free parameters, which include the autoregressive coefficients, the mean shifts in the observations equations, the loadings on the cycle (called g), and the three variances. (The shocks are assumed to be independent). As discussed above, the variance of the cycle shock is fixed at 1. We include it in the parameter set with the pegged value as it makes the state-space representation easier to understand.

```
dec rect g(1,2)
nonlin phic phi1 phi2 mu1 mu2 g sigc=1.0 sig11 sig22
```

The matrices in the state-space representation are all time-invariant (other than the data), but several depend upon the free parameters. The most convenient way to handle these is to create a “startup” function, which gets executed at the start of each function evaluation to copy the current parameter settings into matrices:

```
function %%DLMStart
compute a=%diag(||phic,phi1,phi2||)
compute sw=%diag(||sigc,sig11,sig22||)
compute c=g~~%identity(2)
end %%DLMStart
```

Coming up with good guess values for state-space models can be difficult. The most important ones to get at least within an order of magnitude or so are the variances—the others are usually more robust to poor guess values. With a complicated model, it's often a good idea to start with a more restricted submodel and “bootstrap” the estimates of the simpler model to start the complex model. What's included in the example may be more than is needed in practice, but gives an idea of how you can construct guess values from more straightforward calculations. First is an estimate of an AR(1) on the sum of the two series to get a guess for the PHIC.

```
set combined = ygr+cgr
linreg combined
# constant combined{1}
compute phic=%beta(2)
```

The residual variance in the cycle equation is supposed to be one. So get a preliminary estimate of the cycle by taking the fitted values and scaling them down by the observed standard error:

```
prj testc
set testc = testc/sqrt(%seesq)
```

Estimate AR(1) regressions on the two observables and pull out guess values.

```
ar1 ygr
# constant testc
compute mu1=%beta(1),g(1,1)=%beta(2),phi1=%rho,sig11=%seesq
ar1 cgr
# constant testc
compute mu2=%beta(1),g(1,2)=%beta(2),phi2=%rho,sig22=%seesq
```

Estimate the parameters by maximum likelihood. Given the estimated model, use Kalman smoothing to estimate the business cycle. The way this works, at each function evaluation the %DLMSTART() function is executed. That resets the SW, A and C matrices to the values needed for the current set of parameters. With the multiple observable, you need to use the ||...|| notation to feed in both values.

```
dlim(start=%DLMStart(),type=smooth,y=||ygr,cgr||,sw=sw,c=c,a=a,$
      mu=||mu1,mu2||,presample=ergodic,pmethod=simplex,piters=10,$
      method=bfgs) 1947:2 1989:3 xstates
```

The cycle series is the first component of the state vector. A positive value means above “average” growth.

```
set cycle 1947:2 1989:3 = xstates(t)(1)
graph(footer="Estimated State of Business Cycle")
# cycle
```

10.8 DSGE: Setting Up and Solving Models

DSGE stands for **D**ynamic **S**tochastic **G**eneral **E**quilibrium. The **DSGE** instruction takes a set of formulas and solves them (either exactly or approximately) to create a state-space model. Input to it are the formulas (in the form of a **MODEL**), and list of endogenous variables. Output from it are the system matrices for the state-space representation: **A**, **F** and **Z**. The algorithm used is the QZ method from Sims (2002). However, internally RATS uses a “real” Schur decomposition, rather than the complex Schur decomposition described in that paper.

We’ll start with a simple example which has no expectation terms.

$$x_t = \rho x_{t-1} + \varepsilon_{xt}$$

$$c_t = c_{t-1} + \mu_c + x_{t-1} + \varepsilon_{ct}$$

The equations are input using a set of **FRML**’s which are grouped into a model. We need to define the variables **X** and **C** as series so they will be recognized as such in analyzing the model. We also will need to provide actual values for the parameters ρ and μ_c . **DSGE** can’t do symbolic analysis in terms of the deep parameters; it can only solve for the representation given a specific set of values. The initial setup is:

```
dec series x c
dec real rho mu_y
frml f1 x = rho*x{1}
frml f2 c = c{1} + mu_y + x{1}
group BansalYaron f1 f2
```

Not all **FRML**’s in a model will have a specific dependent variable. If you have an implicit relationship, write it as an expression in the form $f(\dots) = 0$ if the **IDENTITY** option is used, or $f(\dots) = \varepsilon_t$ if it has a shock.

To solve for a specific set of parameter settings, we can do something like:

```
compute rho=.98,mu_c=.005
dsge(model=BansalYaron,a=a,f=f,z=z) x c
```

The series listed on the **DSGE** instruction line are the endogenous variables in the model. The number of these *must* match the number of equations. Because the model has two non-identities, there are two shocks. The first two states will be x_t and c_t ; the endogenous variables in the order listed. **DSGE** will examine the formulas, looking for occurrences of the endogenous variables. These will be slotted into the form:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_t \\ c_t \end{bmatrix} = \begin{bmatrix} \rho & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_{t-1} \\ c_{t-1} \end{bmatrix} + \begin{bmatrix} 0 \\ \mu_c \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \varepsilon_{xt} \\ \varepsilon_{ct} \end{bmatrix}$$

where, while we’ve written them symbolically, these will all have numeric values. In this case, the **A**, **F** and **Z** can be simply read off.

Let's look at a more complicated example. The first order conditions for the problem:

$$(33) \quad \max E_0 \sum \beta^t (u_1 - u_2 c_t)^2 \text{ subject to } y_t = f k_{t-1} + \varepsilon_t, k_t + c_t = y_t$$

can be written:

$$(34) \quad E_t \beta f \times (u_1 - u_2 c_{t+1}) = (u_1 - u_2 c_t) \\ k_t + c_t = f k_{t-1} + \varepsilon_t$$

In the formula, you write the expectation of c_{t+1} given t using the standard series lead notation $C\{-1\}$. The first equation is an identity. We rearrange that to make the model:

```
dec real beta f u1 u2
dec series c k
frml(identity) f1 = beta*f*(u1-u2*c{-1}) - (u1-u2*c)
frml          f2 = k+c-f*k{1}
```

An example setting up and solving this is:

```
compute beta    =.99
compute f       =1.02
compute u1      =1.0
compute u2      =0.5
group bliss f1 f2
dsge(model=bliss,a=a,f=f,z=z) c k
```

This generates the intermediate representation:

$$\begin{bmatrix} u_2 & 0 & -\beta f u_2 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_t \\ k_t \\ E_t c_{t+1} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} c_{t-1} \\ k_{t-1} \\ E_{t-1} c_t \end{bmatrix} + \begin{bmatrix} \beta f u_1 - u_1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \eta_t + \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \varepsilon_t$$

where η_t is a process satisfying $E_t \eta_{t+1} = 0$. Sims' method does a decomposition of the system which, as part of the solution, eliminates the η_t term, leaving only a backwards looking state-space model in the three state variables: the two endogenous variables plus $E_t c_{t+1}$.

Now this problem is simple enough that it can be solved exactly (and symbolically) by, for instance, the method of undetermined coefficients. **DSGE** produces the equivalent representation (for any given setting of the parameters), but it ends up with the redundant state $E_t c_{t+1}$, which (in the solved version) just passively reacts to the other two states. You should look at **DSGE** as being something of a "black box", which produces a solution to the problem, just not necessarily in the most obvious form. However, because the first block of states are *always* the dependent variables in the order you specify, you can easily determine the dynamics for the series of interest—you do the calculations with the full state-space model, and then extract out that top block.

10.8.1 Requirements

While **DSGE** operates numerically on the parameters, it works symbolically with the series. In order to do this, it needs to be able to recognize when a series is used and what (specific) lag or lead is needed. Series have to be referenced by individual names. You can't use elements of `VECT[SERIES]`. The lag or lead has to be a hard number, not a variable: that is, `C{N}` isn't permitted.

The model has to be a “general equilibrium” model—any series used in the equations of the model have to be dependent variables for the model. The only “exogenous” processes are the shocks which are implicit (in the `FRML`'s input as non-identities).

All expectations use information through t . These are represented as leads, so `C{-1}` is $E_t c_{t+1}$ and `C{-2}` is $E_t c_{t+2}$. Expectations through other information sets (such as $t-1$) can be handled by adding states, as we'll see below.

Autoregressive Shocks

Convert the original equation into an identity by adding a new series which represents the shock, and add a new (non-identity) describing the AR for the shock. For instance:

```
frml(identity) eqn8 yhat = (1-tau*ky-gy)*chat+tau*ky*i_hat+gy*eps
frml eqneps g epsg = rho_g*epsg{1}
```

Note that you now need to include `EPSG` among the target series, since the model's solution must include it. You might find it easier to handle *all* shocks this way, even if they aren't autocorrelated.

Date Convention for Capital Stock

In the example above, the capital stock is “time-stamped” so that k_t is determined at time t (by the decision regarding c_t). In many descriptions of this type of model, the capital stock is instead dated so k_t is what is used in production at t . The resource constraint with this type of date scheme will usually be written:

$$k_{t+1} + c_t = y_t + (1 - \delta)k_t$$

(δ is the depreciation rate). You *can* keep that date scheme as long as you re-date that specific equation. If included as written, the k_{t+1} will be interpreted as an expected value rather than a realized number. Instead, you want to use this in the form:

$$k_t + c_{t-1} = y_{t-1} + (1 - \delta)k_{t-1}$$

All other occurrences of k in the model will naturally be dated t and earlier.

Expectations at $t+k$, $t-k$

These can be handled as described in Sims (2002). Consider

$$(35) \quad m_t = p_t + y_t - \alpha(E_{t-1}p_{t+1} - E_{t-1}p_t)$$

Define $e_{1t} = E_t p_{t+1}, e_{2t} = E_t p_{t+2}$. Then (35) can be represented using the combination of three formulas:

```
frml(identity) md = m-p-y+alpha*(e2{1}-e1{1})
frml(identity) e1f e1 = p{-1}
frml(identity) e2f e2 = e1{-1}
```

10.8.2 Non-linear models

The optimization (33) produces a set of linear equations only because it uses a quadratic utility function and a linear production function. With a log utility function and production function $y_t = f k_{t-1}^\alpha \theta_t$, the first order conditions become:

$$(36) \quad E_t \beta R_{t+1} c_t / c_{t+1} = 1$$

$$R_t = \alpha f k_{t-1}^{\alpha-1} \theta_t$$

$$y_t = f k_{t-1}^\alpha \theta_t$$

$$k_t + c_t = y_t$$

$$\log \theta_t = \varepsilon_t$$

where R_t is the gross rate of return on capital and θ_t is a productivity shock. It's possible to substitute out for R and y , but at a substantial loss in clarity. Those types of substitutions are often necessary if you're trying to simplify the model to a small enough set of states to solve analytically. However, if **DSGE** is doing the hard work, you're better off writing the model in short, easy-to-read formulas with extra variables.

Because the equations aren't linear, the Sims solution method can't apply directly. Instead, we can get an approximate solution by linearizing the model. The standard way of handling this in the literature is to log-linearize around the steady state. However, **DSGE** offers the possibility of doing a simple linearization. You need to include either `EXPAND=LOGLINEAR` or `EXPAND=LINEAR` to get one of the expansions.

In order to log-linearize, all series need to be strictly positive. With this in mind, the technology shock in (35) is multiplicative rather than additive. Log-linearization involves writing each series in the form $x_t = \bar{x} \exp(\tilde{x}_t)$, where $\bar{x}$ is the expansion point. We end up with a linearized system written in terms of $\tilde{x}_t$.

The log-linearization is quite straightforward once you have the expansion point. RATS has a symbolic differentiation routine which can handle almost any of its (scalar) functions, including probability functions like `%CDF`. You *can*, however, run into problems calculating the steady state. By default, **DSGE** starts with guess values of 1.0 for each variable and uses a log-linear version of Newton's method to solve the system of non-linear equations. That generally works acceptably if the variables aren't bounded above. There are, however, models where, for instance, labor is parameterized in terms of fraction of available time, with `labor + leisure = 1`. Under

those circumstances, the initial guess values aren't even valid. There are two ways to feed in alternative guess values. The simplest usually is to use `<<` on an individual series to override the default value of 1. For instance:

```
dsge (model=growth, a=a, f=f, z=zc, expand=loglinear, trace) $  
  c k y lab<<.5 xi
```

uses values of 1 for C, K, Y and XI, but .5 for LAB. You can use the TRACE option on **DLM** to see how the iterations are going. That will sometimes help you isolate any series which are causing problems.

You can also use the INITIAL option to input the full vector (one per series). That will mainly be used in situations where we need to repeatedly solve a system with changes in the parameters, when, for instance, we're estimating them. Once you've solved the system once, you can fetch the steady state values for that with the option STEADY=VECTOR for steady state estimates. You can then feed that back into future **DSGE** instructions using the INITIAL option. Since the steady state generally won't change much with small changes to the parameters, you'll reduce the amount of time required to locate the new expansion point.

If you want to set the expansion point rather than have it estimated, you can use either the `<<` or INITIAL option to put in the guess values, then include the option `ITERS=0`.

10.8.3 Unit Roots

Unit roots aren't a problem for the **DLM** instruction once you have the model solved out to state space form—it can handle a mixture of unit and stationary roots using the PRESAMPLE=ERGODIC option. They do, however, pose several problems for solving a model into state-space form.

First, the solution procedure is to solve “unstable” roots forwards and “stable” roots backwards. For the purpose of solution, a unit root is considered to be stable—only roots greater than 1 are solved forwards. In the world of computer arithmetic, however, a test like $\lambda \leq 1$ can be dangerous. Due to roundoff error, it's possible for a value which should be *exactly* 1 to come out slightly larger (or smaller, though that won't be a problem). **DSGE** has a CUTOFF option which determines the boundary between the blocks of roots. The default value for that is 1.0, but it actually is $1.0 + \text{a small positive number}$ (10^{-7}). This gives a cutoff which is close enough to 1.0 that it's highly unlikely for a model to have a root that is, in fact, unstable, but less than that value, but is still far enough from 1 that true unit roots will generally have a computed value lower than it. It's possible (though unlikely) that you'll have a model where the “fuzziness” added by **DSGE** isn't enough. If your model doesn't solve correctly, you can put in something like `CUTOFF=1.001` to give a much higher tolerance for detection of a unit root.

A more significant problem occurs in expanding a non-linear model. A model with unit roots has no single steady state. For instance, if you have a persistent technology shock such as:

$$\log \theta_t = \rho \log \theta_{t-1} + \varepsilon_t$$

in computing the steady state, when $\rho=1$, we get $0=0$ —the model does nothing to tack down a value for $\bar{\theta}$. Instead of having *one* steady state, the model has a different solution for the other variables for each value of $\bar{\theta}$. To get an expansion point, you'll have to do one of two things:

1. Input the entire expansion point as described on page UG–346.
2. Provide values for the series which produce the unit root(s), and add the option `SOLVEBY=SVD` to **DSGE**.

`SOLVEBY` controls the way Newton's method solves the system of equations at each iteration. With the unit root(s), the system of equations is singular. `SOLVEBY=SVD` lets this solve a partial system which is non-singular, while leaving unchanged the part that's singular. If you try to solve a non-linear system with a unit root without using `SOLVEBY=SVD`, you'll get a message like:

```
## FO16. Newton's method had singularity. If model is non-stationary, use
    option SOLVEBY=SVD. Error was in FRML F7
On iteration 1 of solution for steady state
```

This suggests the fix and points out the equation that produced the singularity.

10.8.4 DSGE: Applications

Given an input model, **DSGE** solves it into a state-space model representation. So what can we do with this? Many models aren't designed to be estimated with actual data. In order to take a state-space model to data, the model needs at least as many shocks as there are observable series. Almost all the models we've seen have (many) fewer shocks than endogenous variables, even if we don't count series like capital that really aren't easily observed. Instead, most solved models are used to generate simulated economies and study the serial correlation patterns implied by the model. Simulations can be done easily using **DLM**; the calculation of responses and serial correlation require some additional procedures. There are many examples of this in the Novales, Fernandez and Ruiz(2009) textbook replications.

In order to solve the model in the first place, you need to provide values for the deep parameters of the model. One thing that you did not need before, but will need now, are the variances of the shocks. If you have more than one exogenous shock, they will be ordered based upon the listing on the **GROUP** instruction which created the model. The first formula in that list that isn't an identity defines the first shock, the second defines the second shock, etc. The shock is signed to give a positive shift to the function as written. If you write a formula:

```
frml f2 = (k+c)-f*k{1}
```

the interpretation will be $k_t + c_t - f k_{t-1} = \varepsilon_t$, which means that it's a positive shock to output. The same formula written in the opposite order:

```
frml f2 = f*k{1}-(k+c)
```

would produce a shock that's actually negative to output. It's a good idea to "endogenize" any shock that occurs in more complicated formulas so you can make sure it has the sign that you want. For instance,

```
frml(identity) f2 = (k+c)-(f*k{1}+e)
frml f3 = e
```

Simulations

DLM with the option `TYPE=SIMULATE` can create a simulated path for the complete state vector. Now as we've seen above, the full state vector can include quite a few states which are of no direct interest, having been added to the model solely to allow a solution in the proper form. However, since the original endogenous variables are in a known position in the state vector, we can extract them from the `SERIES[VECT]` that **DLM** produces.

One thing you must keep in mind is that if you log-linearize the model, the states generated must be interpreted as simulated values for $\log(x_t/\bar{x})$. If you're interested in simulated series for the data itself, you'll have to save the steady state (using the `STEADY` option on **DSGE**) and transform back.

Impulse Responses

You can compute and graph impulse response functions for a state-space model using the procedure `@DLMIRF`. This takes as input the **A** and **F** matrices from the state-space model. (The state shifts **Z** don't affect the results). It then computes and graphs the responses to the variables of the system to the shocks. Note that it uses a unit size for each shock which might give responses that are unnaturally large. For instance, a unit shock to ε_t in $\log \theta_t = \rho \log \theta_{t-1} + \varepsilon_t$ will increase θ by a factor of e , with correspondingly enormous changes to the variables. You can change the scale of the shocks by multiplying **F** by a small number. If you have just one shock, or would like the same scaling to apply to all, you can just use the option `F=.01*FDLM`, for instance. If you want different scales to apply to each shock, you can do `F=FDLM*%DIAG(scale factors)`. The scale factors here should be standard errors, not variances.

There are several options for controlling the organization of the graphs. Aside from standard `HEADER` and `FOOTER` options, the `PAGE` option allows you to arrange pages `BYSHOCK` (one shock per page, all variables), `BYVARIABLE` (one variable per page, all shocks), `ALL` (everything on one page) and `ONE` (one combination of shock and variable per page). There's also a `COLUMNS` option which allows you to better arrange the graphs on a single page. The shocks are given labels using the `SHOCKS` option and the endogenous variables are labeled using the `VARIABLES` option. Because the

state vector usually includes augmenting variables (and often endogenized shocks), you generally don't want to include the entire state vector in the graphs. If you just restrict the `VARIABLES` option to list a subset of the states, only those will be shown. Since you have freedom to list the variables in any order on the **DSGE** instruction, you can arrange the variables there so the ones of interest come first.

Solving a DSGE Model: DSGEKPR.RPF example

Watson (1993) analyzes a special case of the model below (which some minor renaming of variables) whose equilibrium is described by the following equations:

$$(37) \quad \frac{\theta}{1 - N_t} = C_t^{-\eta} \alpha \frac{Y_t}{N_t}$$

$$(38) \quad 1 = \beta \gamma^{1-\eta} E_t R_{t+1} (C_t / C_{t+1})^\eta$$

$$(39) \quad \gamma R_t = (1 - \alpha) \frac{Y_t}{K_{t-1}} + 1 - \delta$$

$$(40) \quad C_t + I_t = Y_t$$

$$(41) \quad \gamma K_t = I_t + (1 - \delta) K_{t-1}$$

$$(42) \quad Y_t = Z_t K_{t-1}^{1-\alpha} N_t^\alpha$$

$$(43) \quad \log Z_t = (1 - \rho) \log(\bar{Z}) + \rho \log(Z_{t-1}) + \varepsilon_t$$

There is only one exogenous shock in this model (equation (43)—the technology shock). This is a non-linear model, which will be analyzed using a log-linearization. Because N can't take the value 1, we can't use the default initial guess values for all variables, so the *initial values* parameters are used for some. `SOLVEBY=SVD` is included to deal with the unit root in the technology shock.

This is example file `DSGEKPR.RPF`. (This is the King-Plosser-Rebelo model, hence the name).

```
open data watson_jpe.rat
calendar(q) 1948:1
data(format=rats) 1948:1 1988:4 y c invst h
```

Several of the series used in the model are unobservable and so aren't in the data set (which is never actually used in this example). They're included in a `DECLARE SERIES` instruction so they can be used in defining the equations.

```
declare series n r z k
declare real theta eta alpha beta gamma delta psi
```

These are the calibration values used by Watson.

<code>compute eta=1.0</code>	<i>Log utility</i>
<code>compute alpha=.58</code>	<i>Labor's share</i>
<code>compute nbar=.2</code>	<i>Steady state level of hours</i>
<code>compute gamma=1.004</code>	<i>Rate of technological progress</i>
<code>compute delta=.025</code>	<i>Depreciation rate for capital (quarterly)</i>
<code>compute rq=.065/4</code>	<i>Quarterly steady state interest rate</i>
<code>compute rho=1.00</code>	<i>AR coefficient in technical change</i>
<code>compute lrstd=.01</code>	<i>Long-run standard deviation of output</i>
<code>compute zbar=1.0</code>	<i>Mean of technology process</i>
<code>compute theta=3.29</code>	<i>Preference parameter for leisure</i>
<code>compute beta=gamma/(1+rq)</code>	

Define the equations. F7 is the only one with a shock attached. F2 is the only one with an expectational term (for future C).

```
frml(identity) f1 = theta/(1-n)-c^(-eta)*alpha*y/n
frml(identity) f2 = 1-beta*gamma^(1-eta)*c/c{-1}*r{-1}
frml(identity) f3 = gamma*r-(1-alpha)*y/k{1}-1+delta
frml(identity) f4 = c+invst-y
frml(identity) f5 = gamma*k-invst-(1-delta)*k{1}
frml(identity) f6 = y-z*k{1}^(1-alpha)*n^alpha
frml          f7 = log(z)-(1-rho)*log(zbar)-rho*log(z{1})
```

```
group swmodel f1 f2 f3 f4 f5 f6 f7
```

Solve the model to state-space form, overriding several settings for the guess values.

```
dsge(model=swmodel,expand=loglinear,a=a,f=f,solveby=svd) $
y c<<0.5 invst<<0.5 n<<nbar r<<1+rq k z
```

Graph the impulse responses using the state-space matrices. The state-space model will have more than four states, but by limiting the VARIABLES option, you only see the four main ones, and not R, K, Z and any augmenting states.

```
@dlmirf(a=a,f=f,graph=byvar,shocks=||"Technology"||,$
variables=||"Output","Consumption","Investment","Hours"||)
```

10.8.5 DSGE: Estimation

There's a relatively thin literature on formal estimation of the deep parameters in a DSGE using actual data. Conceptually, estimation shouldn't be *that* difficult. Given the model, we have a method which generates either an exact state-space model (for linear models) or an approximate one (for non-linear models). For any collection of parameters, we can generate a history of states, which can be compared with actual data.

Unfortunately, it's not that easy. The main problem can be seen by looking at example `DSGEKPR.RPF` (page UG–350). This is a perfectly reasonable looking model, which generates predictions for output, consumption, labor input, investment. However, it has only one exogenous shock. As a result, the errors in the four predictions have a rank one covariance matrix: once you know one, you should be able to compute the others exactly. In sample, that will *never* happen. In order to do a formal estimation, we will have to do one of the following:

1. Reduce the number of observables to match the number of fundamental shocks.
2. Add measurement errors to increase the rank of the covariance matrix.
3. Add more fundamental shocks.

Reducing the number of observables makes it much more likely that some of the deep parameters really can't be well estimated. For instance, because the capital stock is effectively unobservable, the depreciation rate δ can't easily be determined from the data—a high value of δ with a high level for the capital stock produces almost the same predictions for the observable variables as a low value for both. Adding fundamental shocks requires a more complex model.

For models which admit a complete description of the observables, there are two main estimation techniques: maximum likelihood and Bayesian methods. For this, Bayesian methods are more popular than straight maximum likelihood. In some sense, all estimation exercises with these models are at least partially Bayesian—when parameters such as the discount rate and depreciation rate are pegged at commonly accepted values, that's Bayesian. Those are parameters for which the data have little information, so we impose a point prior.

The heart of both maximum likelihood and Bayesian methods is the evaluation of the likelihood of the derived state-space model. So what accounts for the popularity of the latter? Variational methods of optimization (like the BFGS algorithm used by **DLM**) often try to evaluate the likelihood at what are apparently very strange sets of parameters. This is part of the process of a “black box” routine discovering the shape of the likelihood. For simpler functions, there is usually no harm in this: if a test set of parameters requires the log or square root of a negative number, the function returns an NA and the parameter set is rejected. If a root goes explosive, the residuals get really large and the likelihood will be correspondingly small. However, the function evaluation in a DSGE is quite complicated.

If the model is non-linear, we must first do the following:

1. solve for a steady-state expansion point
2. solve for the backwards representation
3. solve for the ergodic mean and covariance of the state-space model

before we can Kalman filter through to evaluate the likelihood. On the first few iterations (before much is known about the shape of the function), it's quite easy for a test set of parameters to go outside the range at which those steps are all well-behaved. By imposing a prior which excludes values we know to be nonsensical, we can avoid those types of problems.

The other main reason is that the likelihood function can be quite flat with respect to parameters besides β and δ ; those are just the two most common parameters that are poorly identified from the data. Including a prior allows for those other parameters to be restricted to a region that makes economic sense.

Both maximum likelihood and (especially) Bayesian estimation are quite complicated. We provide an example of maximum likelihood for a small model with two observables. Even this model (which doesn't have expectational terms, so all **DSGE** is doing is organizing the state space representation) requires some fussing with guess values.

A couple of things to note: first, the DSGE solution is inside a function, here called **EvalModel**:

```
function EvalModel
dsge (model=cagan,a=adlm,f=fdlm) x mu a1 a2 eps eta
end EvalModel
```

Because each function evaluation needs a new set of system matrices, we need to use a **START** option on **DLM** to call **EvalModel** to create them. Note that the function doesn't actually return anything; instead, it sets the matrices **ADLM** and **FDLM** which are used in the **DLM** instruction.

Second, right after the model is set up comes the following:

```
compute EvalModel()
compute cdlm=%identity(2)~~%zeros(%rows(adlm)-2,2)
```

We have two observables, which match up with the first two variables in the model. However, the **C** matrix needs to have the same number of rows as there are states in the model. Since **DSGE** is a bit of a black box, it might be hard to tell in advance just how many states there are in the expanded model. This takes care of that by counting the number of rows in the **A** matrix and adding a block of zeros at the bottom of **C** to cover all but the first two.

DSGE Estimation: CAGAN.RPF example

CAGAN.RPF estimates the parameters for Cagan's model of hyperinflation. (MU is money growth, X is inflation).

```
open data cagan_data.prn
data(format=prn,org=cols) 1 34 mu x

declare series a1 a2 eps eta
declare real    alpha lambda sig_eta sig_eps
```

While it's possible to substitute out A1 and A2, there's no advantage to it, and it makes the formulas unreadable.

```
frml(identity) f1 = x -(x{1}+a1-lambda*a1{1})
frml(identity) f2 = mu-$
                ((1-lambda)*x{1}+lambda*mu{1}+a2-lambda*a2{1})
frml(identity) f3 = a1-1.0/(lambda+(1-lambda)*alpha)*(eps-eta)
frml(identity) f4 = a2-(1.0/(lambda+(1-lambda)*alpha)*$
                ((1+alpha*(1-lambda))*eps-(1-lambda)*eta))

frml          d1 = eps
frml          d2 = eta

group cagan f1 f2 f3 f4 d1 d2
```

$LAMBDA + (1 - LAMBDA) * ALPHA$ needs to stay clear of its zero point. It appears that it needs to be negative, so ALPHA must be less than $-LAMBDA / (1 - LAMBDA)$ on the guess values.

```
compute alpha=-3.00, lambda=.7, sig_eta=.001, sig_eps=.001
*****
function EvalModel
dsge(model=cagan,a=adlm,f=fdlm) x mu a1 a2 eps eta
end EvalModel
*****
compute EvalModel()
compute cdml = %identity(2)~~%zeros(%rows(adlm)-2,2)
```

Because we really have no idea what the scale is on the variances, we first estimate the model with the alpha and lambda fixed. This uses only a small number of simplex iterations to get the sigmas into the right zone.

```
nonlin sig_eta sig_eps
dlm(start=%(EvalModel()), sw=%diag(||sig_eps^2,sig_eta^2||), $
    a=adlm, f=fdlm, y=||x,mu||, c=cdml, sw=sw,presample=ergodic,$
    method=simplex, iters=5, noprint)
*
nonlin alpha lambda sig_eta sig_eps
dlm(start=%(EvalModel()), sw=%diag(||sig_eps^2,sig_eta^2||), $
    a=adlm, f=fdlm, y=||x,mu||, c=cdml, sw=sw,presample=ergodic,$
    pmethod=simplex, piters=5, method=bfgs)
```

11. Thresholds, Breaks, Switching

This chapter describes various threshold and structural break models and tests, as well as Markov switching models. Testing for or modeling breaks is a current topic of great interest in the profession. However, many of these models are technically quite demanding, so you need to understand the issues.

There is a section on threshold autoregression (TAR) models in Walter Enders' *Applied Econometric Time Series* (2014), as well as in the *RATS Programming Manual* (included with RATS as a PDF). The Tsay (2005) and Martin, Hurn and Harris(2012) textbooks also includes several examples of the models described here—you can find RATS code for these in the **TextbookExamples** subdirectory.

We also have a *Structural Breaks and Switching Models* e-Course which goes into greater detail on the application of these methods using RATS. For more information, see:

https://estima.com/courses_completed.shtml

Structural Breaks and Stability

Fluctuations Test

Rolling Regressions

Bai-Perron Algorithm

Unit Roots and Breaks

Markov/Hamilton Switching Models

SWARCH Model

11.1 Stability Testing and Structural Breaks

We discuss basic stability tests (tests for the lack of a structural break) beginning on page UG–86. Here, we’ll look at more advanced ways of testing for breaks, and consider models that incorporate one or more structural breaks. In most cases, we are actually testing a proposed model for specification errors. Does a single specification seem to work across the entire data set? Are the errors homoscedastic, or are they higher in one part of the data set? If the model fails to pass the test, it’s very rare that the alternative model with the break is actually what we would think is “correct.” Instead, it simply points us in the direction of a better model. Most of the newer techniques are likewise specification tests on a simpler model. Let’s look more closely at what we’ve seen earlier:

Recursive Least Squares (page UG–91)

Why would we be interested in the estimates from a subset (particularly a very small subset) of the data if we were convinced that the data were accurately described with a time-invariant model? With recursive least squares, the alternative to the null hypothesis of a time-invariant model with i.i.d. residuals is the vague alternative that they aren’t. The CUSUM test looks for breaks in the model itself, while CUSUMSQ examines breaks in the variance.

For least squares, **RLS** can do the calculations of the sequential estimates very quickly using the Kalman filter. The same types of calculation can be done (less efficiently) for other types of models with direct sequential estimation as we’ll see in Section 11.2.

Goldfeld-Quandt Test (page UG–79)

This computes a set of linear regressions with a complete break at a relatively arbitrary location. Clearly, the sample split used is not intended as an alternative “model”; it’s chosen to maximize the power of the test in case the variance does indeed change systematically with the variable which controls the split.

Fluctuation (Nyblom) Tests (page UG–90)

Similar to tests based upon the recursive residuals, these use the fact that under the null of a correct and time-invariant model, various subsample statistics should “fluctuate” in a particular way. If they don’t, that is taken as evidence against the model, without necessarily suggesting a specific alternative. The use of the **@FLUX** procedure for testing a *non-linear* model is demonstrated on the next page (**@STABTEST** is for linear models only).

Chow Tests (page UG–87)

A standard Chow test uses a sample split at a known location, testing for a complete break in the specification at that point. A full structural break is unlikely to be of much use—are there *no* coefficients that we think are the same across the sample?

Intervention Modeling (page UG–200)

This is the one situation seen so far in which the break is part of a specific alternative model.

Fluctuations Test: GARCHFLUX.RPF example

The procedure **@FLUX** is the more general variant of the **@STABTEST** procedure covered on page UG–90. For a model whose parameters are estimated by optimizing a sum across the data set (such as the log likelihood function), the derivatives of the log likelihood elements with respect to each parameter should (at the final estimates) sum to zero, or at least something close to it numerically. If the model is stable across the data set, the partial sums of the derivatives should stay at least reasonably close to zero. If, on the other hand, there is a structural break, the partial sums of the derivatives would be expected to stray from zero in one direction since one part of the data set would prefer a higher or lower value for at least one of the parameters. **@FLUX** takes as input the derivatives (typically produced by the **DERIVES** options on the estimation instruction) and does a Nyblom(1989) fluctuations test. This is especially useful in cases like GARCH models where there is really no good way to do a more formal sample split test. That’s what is done by the **GARCHFLUX.RPF** example.

```
set x = 100.0*log(usxjpn/usxjpn{1})
```

Estimate the GARCH model and save the derivatives.

```
garch(p=1,q=1,derives=dd) / x
@flux(title="Full Sample Fluctuations Test")
# dd
```

This does a closer examination of the variance intercept (coefficient 2). If there are no breaks, this should be (when rescaled) a “Brownian Bridge” (Brownian motion tied to zero at both ends), which should wander around zero. The fact that it drops rather sharply for the first 1200 or so observations, then climbs back to zero indicates that the gradient is consistently negative at the start of the sample and consistently positive after that, suggesting that the first (roughly 20%) of the data needs a much lower value for this parameter than the remainder.

```
acc dd(2) / fluxstat
graph(footer="Cumulated Gradient for Variance Intercept")
# fluxstat
```

While it would be possible to adjust the model to add an **XREG** dummy for the first part of the sample, given the size of the data set, simply dropping those first data points probably makes more sense.

The estimation is redone, as is the fluctuation test, which this time passes.

```
garch(p=1,q=1,derives=ddx) 1200 * x
@flux(title="Partial Sample Fluctuations Test")
# ddx
```

11.2 Rolling Regressions

Running a sequence of estimations over a moving window of data is very easy to do in RATS. We touch on this briefly on page UG–4 where we note that you can loop over a range of dates, and use dates in mathematical expressions. We also discuss computing forecasts for a rolling regression on page UG–159. Here, we'll look in more detail at the tools available and the issues that can arise.

Loops and the @ROLLREG Procedure

Rolling regressions typically take one of three forms:

1. estimations done using a moving (fixed-size) window of data
2. estimations where one or more data points are added at the end of the sample
3. estimations where the starting period is incremented but the ending period remains fixed (so that the sample size decreases).

Recursive least squares is an example of (2). The most commonly used of these, however, is the moving fixed window. This isn't so much a *test* of a structural break as an admission that no single specification is likely to be valid across the full data set. There's no model of structural change under which this technique is exact, but it can be approximately justified by an assumption that the change in the model takes place slowly. Note that while moving fixed-width analysis is increasingly common, it often is poorly-motivated, particularly when applied to rolling window hypothesis tests. `ROLLINGCAUSALITY.RPF` shows how use of a rolling window can easily produce incorrect inferences.

Any of these can be done using simple loops. For example, the first of these does moving windows of width 120, with samples ending in 1999:12 through 2017:12. The second fixes the start of a sample at 1990:1, then moves the end period from 1999:12 to 2017:12. The first regression is the same in both cases; the difference is that the moving window shifts the start point of the range along with the end point.

Moving Window:

```
compute width=120
do end=1999:12,2017:12
    linreg(noprint) depvar end-width+1 end
    # regressors
end do
```

Moving end period

```
compute start=1990:1
do end=1999:12,2017:12
    linreg(noprint) depvar start end
    # regressors
end do
```

The Help—Pseudo-Code Generator—Rolling Analysis wizard can help set up the overall structure of the first two types of loops. (The third type is rarely used.)

In addition to the loop and the estimation instructions, you would normally need to add some bookkeeping instructions to collect the desired data. For least squares rolling regressions, we recommend using the **@ROLLREG** procedure, originally written by Simon van Norden and others at the Bank of Canada. It can do all three types of rolling regression, and has options for saving the coefficients, coefficient standard errors, and regression standard errors from each estimation, as well as options for graphing the coefficients over time, estimating with robust standard errors, and more. For example:

```
@rollreg(graph,move=32) y
# constant x2 x3
```

does a rolling regression with a moving window of 32 observations, and produces three graphs showing the evolution of each of the three coefficients with standard error bands. To collect the three graphs into a single-page **SPGRAPH**, you could do:

```
spgraph(vfields=3,header="Moving Window Estimates (width=32)")
  @rollreg(graph,move=32) y
  # constant x2 x3
spgraph(done)
```

Rolling Nonlinear Estimations

For simple non-linear estimations that converge easily, such as those performed using **NLLS** and **BOXJENK**, the process for doing a rolling regression is essentially the same as for linear regressions. That is, just enclose the nonlinear estimation inside a loop that controls the starting and/or ending period(s) of the estimation, and add whatever bookkeeping instructions are necessary to collect the desired information. For instance, in **ARIMA.RPF**, we have the following, which does rolling **BOXJENK** instructions (adding to the end of the sample) on two models, and computes the one-step forecasts.

```
do time=1995:4,2008:1
  boxjenk(noprint,constant,define=ar7eq,ar=7) spread * time-1
  boxjenk(noprint,constant,define=ar2ma17eq,ar=2,ma=||1,7||) $
    spread * time-1
  uforecast(equation=ar7eq,static) forecast_ar7 time time
  uforecast(equation=ar2ma17eq,static) forecast_ar2ma17 time time
end do
```

More care is required for cases where the estimation may be sensitive to initial parameter values, or where the validity of the standard errors depends on how the nonlinear estimation process converges (for instance in the **BFGS** algorithm, page UG–116). If only the point estimates are important, you don't have to worry about problems with standard errors, but it will still help to speed up the calculations by feeding in already converged estimates. If you're using an instruction which uses **PARMSET**'s, that will already happen if you don't repeat your guess value calculations inside the loop. With an instruction like **GARCH**, you can use the **INITIAL** option. With either type of instruction, you can use the **HESSIAN** option to feed in the inverse Hessian to initialize **BFGS**.

Rolling GARCH Estimates: GARCHBACKTEST.RPF example

GARCHBACKTEST.RPF is an example of rolling GARCH estimates. It uses a rolling set of estimates to produce a one-step out-of-sample estimate of the Value-at-Risk (VaR) which is used to backtest the validity of this particular VaR calculation.

This uses the entire sample up to a given point to estimate the GARCH model, which is the standard procedure for simulated out-of-sample analysis. It's *possible* to adjust this for a moving fixed-width window, but you have to be very careful about making an estimation window too narrow. Remember that with any type of rolling non-linear model, you are hoping that the estimator will work successfully every period because you're not estimating one "model" but possibly hundreds. If you have a data set with long quiet stretches interrupted by some noisy patches, the moving window GARCH estimates can get very unstable if the window moves out of a quiet range. It's also not clear what the point is of using the fixed-width window rather than using *all* of the previous data—given a set of estimated parameters, the calculated value of the GARCH variance rarely depends on more than about the last 100 data points anyway, so the only effect of the rolling sample is to change the parameters, not (necessarily) for the better.

This is the tail probability to evaluate. This is higher than would be typical, but at least for this sample, the Markov test done later doesn't work if this is smaller.

```
compute alpha=.10
```

Period of back test (roughly the last two years of the data, excluding the last period).

```
compute tstart=%allocend()-520  
compute tend   =%allocend()-1
```

TRIGGER will be a dummy variable which is 1 if the observed data is below the model's estimate for the VaR.

```
set trigger tstart tend = 0.0
```

INFOBOX (page UG-497) is useful here because it takes a while to do over 500 GARCH estimates on 6000 data points. It puts up a progress bar which notifies you of the progress (and approximate time until completion).

```
infobox(action=define,lower=tstart,upper=tend,progress) $  
"GARCH Model Backtest"
```

This estimates the GARCH model over the full sample and saves the estimates (into FULLBETA) and the inverse Hessian (into FULLXX) to be fed back into **GARCH** for the other samples. Note the PRINT option. This is to contrast with the NOPRINT inside the loop. Make sure before you go on that you actually have a model which you can fit successfully. (In practice, you shouldn't even write the rest of this until you can get a GARCH model which fits for the full sample).

```
garch(p=1,q=1,resids=u,hseries=h,print) / x  
compute fullbeta=%beta,fullxx=%xx  
do end=tstart,tend
```


This is the **GARCH** instruction inside the loop. The *end* parameter on the range is the END that's being used for the **DO** index. This puts the **NOPRINT** option on and increases the iteration limit to 400 (just in case). If you're having problems with a loop like this, cut the loop range to a more manageable length and *take the NOPRINT option off*.

```
garch (p=1,q=1,resids=u,hseries=h,noprint,$
      initial=fullbeta,hessian=fullxx,itors=400) * end x
```

This computes the one-step forecast for the variance (for END+1 given END).

```
compute hhat=%beta(2)+%beta(3)*u(end)^2+%beta(4)*h(end)
```

And this computes the model's prediction for the VaR. Because the model allows for a non-zero mean, the one-step prediction is Normal with mean %BETA(1) and variance given by HHAT. This computes the lower ALPHA tail value for that distribution.

```
compute limit=%beta(1)+%invnormal(alpha)*sqrt(hhat)
```

This computes the observed value at END+1 with the model's VaR calculation. If it's smaller, TRIGGER is set to 1 (otherwise 0).

```
compute trigger(end)=(x(end+1)<limit)
```

This updates the progress bar and also puts a 1 (for good) or 0 (for bad) on the second line. You should check that that's staying at 1. If not, you should take the advice above, and cut down the loop range and get rid the **NOPRINT** to see what's happening.

```
infobox(current=end) "Model converged "+%converged
```

```
end do end
```

```
infobox(action=remove)
```

The remainder of the program is just number-crunching the values of the TRIGGER dummy. The first computes Kupiec's likelihood ratio test for the observed alpha matching the chosen one:

```
sstats(mean) tstart tend trigger>>alphahat
compute pof=2*%nobs*((1-alphahat)*log((1-alphahat)/(1-alpha))+$
      alphahat*log(alphahat/alpha))
disp "POF measure" pof
```

This computes Christofferson's Markov measure of independence.

```
sstats tstart+1 tend .not.trigger.and..not.trigger{1}>>n1 $
      .not.trigger.and.trigger{1}>>n2 $
      trigger.and..not.trigger{1}>>n3 $
      trigger.and.trigger{1}>>n4
compute p00=n1/(n1+n3),p10=n2/(n2+n4),p0=(n1+n2)/(n1+n2+n3+n4)
*
compute markov=n1*log(p00)+n3*log(1-p00)+n2*log(p10)+$
      n4*log(1-p10)-((n1+n2)*log(p0)+(n3+n4)*log(1-p0))
disp "Markov test" markov
```

11.3 A General Structure for Analyzing Breaks

We now consider the case where the overall sample is fixed, the basic form of the model is also fixed, but we want to allow for additional regressors to handle possible breaks at unknown locations. (A break at a *known* location is generally fairly simple).

Isolated outliers are typically not considered to be “breaks” in the sense considered here. However, if you’re not careful, some of these techniques can give a “false positive” for a break when all that is present is a simple outlier. With the break at an unknown location, one way to control that is to restrict the possible break points to a central range in the data, typically excluding the first and last 10–15% of the data set. By requiring a certain minimal amount of data to be included in each partition, you make it very hard for an outlier near either end to be misclassified as a break. The symbol π is commonly used to represent the excluded fraction of the data.

Single Breaks

The following is “pseudo-code” for searching for a single break with an excluded range on either end. As you can see, the controlling code is quite simple—most of the work is setting up and estimating the model with the break. This assumes that we’re looking for the smallest *t*-statistic, and that the (to-be-added) code is producing a value for the *t*-statistic with these settings as TSTAT. If you’re looking to *maximize* something, just change the *<* to *>* in the **IF** inside the loop. The following are assumed to have been created already:

PI (fractional) excluded zone at ends
LOWER lowest possible entry
UPPER highest possible entry

FIX (x) takes the real-valued results of multiplying the number of observations by PI and rounds down to the nearest integer. The resulting BSTART and BEND are the range where breaks are permitted.

```
compute nobs    =upper-lower+1
compute bstart  =lower+fix(pi*nobs)
compute bend    =upper-fix(pi*nobs)
```

The minimum *t* value is initialized as an NA, so we know to keep the first value we see. We prefer this (with a check for %VALID in the test inside the loop) to setting a “high” value and just doing the size check, as there is no chance of starting with a value which isn’t high enough.

```
compute mint    =%na
do time=bstart,bend
    Do calculation here with breakpoint at TIME.
    if .not.%valid(mint).or.tstat<mint
        compute mint=tstat,bestbreak=time
end do time
```

At this point, the best value will be in MINT and the optimal break is in BESTBREAK.

Multiple Breaks

Needless to say, this is quite a bit more complicated than the single break. The typical programs that you'll find for multiple breaks are written for the case of two breaks only. However, the following general structure will handle any number of breaks from one on up, with no additional programming.

With multiple breaks, you not only have to exclude breakpoints near the end of the data, but you must also make sure the breaks themselves don't get too close together. This uses the same control parameter (**PI**) at the ends and in the middle, though that can easily be changed. In addition to **PI**, **LOWER** and **UPPER** as before, we also need:

BREAKS number of breaks

With **BREAKS** set to the appropriate value, you can do:

```
compute nobs =upper-lower+1
compute pinobs=fix(pi*nobs)
```

Set up the starting break points (the leftmost legal values) into **BPS** and the upper bounds (rightmost legal values) into **UPPERBOUND**.

```
local vect[int] bps(breaks) upperbound(breaks) bestbreaks(breaks)
do i=1,breaks
    compute bps(i) = (lower-1) + pinobs*i
    compute upperbound(i) = endl+1-pinobs*(breaks+1-i)
end do i
compute mint=%na
compute done=0
while .not.done {
```

Do calculation here with breakpoints at BPS (1) ,...,BPS (BREAKS) .

```
if .not.%valid(mint).or.tstat<mint
    compute mint=tstat,bestbreaks=bps
```

Update the break points. Add to the final slot, until we hit its upper bound. When we do, add one to the second to last slot, and reset the final one to its smallest possible value. When the second to last is exhausted, add one to the one earlier, etc. If DONE is zero at the end, we still have a valid combination that we haven't checked.

```
compute done=1
do i=breaks,1,-1
    compute bps(i) = bps(i) + 1
    if bps(i)>=upperbound(i)
        next
    do j=i+1,breaks
        compute bps(j) = bps(j-1) + pinobs
    end do j
    compute done=0
break
```

```
    end do i
}
```

At this point, the best value will be in MINT and the optimal set of breaks is in BEST-BREAKS.

Note that when the number of breaks is two or more, this can require a *very* large number of calculations, particularly when the data set is large. The number of combinations with k breaks goes up with the power of T^k ; since the calculation itself (given a combination) is also likely $O(T)$, the calculations go up with T^{k+1} . If the base model is a quickly computed linear regression, two breaks with 400 data points won't take an unreasonable amount of time. Two breaks with 2000 data points will take roughly 125 times as long. If the base model is non-linear and requires iterated calculations, you might even find that two breaks with 400 data points takes too long.

For linear models estimated with least squares, the Bai-Perron algorithm (Section 11.4) reduces the computational effort with two or more breaks. However, that's not available for any type of non-linear model.

The search procedures for breaks shown above are included in quite a few RATS procedures, such as **@APBreakTest** (for a single break in a linear regression) and **@LSUnit** (for multiple breaks in a unit root test). The following example shows it being used in a test procedure which can't be handled by the existing procedures.

Breakpoint at Unknown Location: ONEBREAK.RPF example

ONEBREAK.RPF is based on an example from Stock & Watson (2011), Chapter 15. The model being estimated is a long distributed lag of the percentage change in the relative price of Orange Juice (OJ price divided by the overall PPI) on the "Freezing Degree Days", which is a measure of low temperatures in the growing region.

The model is estimated by least squares. If we assumed homoscedastic errors, we could use the **@APBreakTest** procedure (which the example uses as well). However, the authors want to correct the covariance matrix for possible serial correlation using a Newey-West window. There's no short-cut way to do the sequential tests under those conditions—you have to estimate the model with both the original variables and the dummied-out versions.

This uses the one-break structure (page UG–362), searching for the maximal F -statistic. It also graphs the series of break statistics, as does the **@APBreakTest** procedure. As you can see, there is quite a bit of difference between the patterns with and without correction for serial correlation.

This does the **@APBreakTest**, which looks for a single break (in the time sequence) for a standard linear regression:

```
@apbreaktest(graph) drpoj 1950:1 2000:12
# constant fdd{0 to 18}
```

This estimates the target regression over the full data set with the desired Newey-West standard errors and checks for serial correlation in the residuals:

```
linreg(lags=7,lwindow=newey,define=baseeqn) drpoj 1950:1 2000:12
# constant fdd{0 to 18}
@regcorrs(number=36)
```

This is the “one-break” setup code described on page UG–362. The upper and lower bounds are pulled out from the full sample regression using %REGSTART() and %REGENGEND().

```
compute lower=%regstart(),upper=%regend()
compute pi=.15
```

```
compute nobs =upper-lower+1
compute bstart=lower+fix(pi*nobs)
compute bend =upper-fix(pi*nobs)
```

FSTAT will be used for a graph of the test statistics.

```
set fstat lower upper = %na
```

DUMMIES will have the dummied-out copies of the original regressors.

```
compute k=%nreg
dec vect[series] dummies(k)
```

```
compute maxf=%na
do time=bstart,bend
```

Set up the dummies (for the subsample greater the TIME). This uses %EQNXVECTOR to extract the full x_i vector, then pulls out element i from it.

```
do i=1,k
  set dummies(i) = %eqnxvector(baseeqn,t)(i)*(t>time)
end do i
```

Do the regression and get the F -statistic on the dummies.

```
linreg(noprint,lags=7,lwindow=newey) drpoj lower upper
# constant fdd{0 to 18} dummies
exclude(noprint)
# dummies
```

EXCLUDE will give a chi-squared test with K degrees of freedom. This is transformed into an approximate F statistic.

```
compute fac = float(%ndf)/float(%nobs)
compute fstat(time) = %cdstat*fac/k
if fstat(time)>maxf
  compute maxf=fstat(time),bestbreak=time
end do
```

```
graph(grid=(t==bestbreak), $  
      header="Robust F-Statistics for Breakpoints")  
# fstat
```

11.4 The Bai-Perron Algorithm

In a series of papers, most notably Bai and Perron (2003), Bai and Perron studied ways to analyze multiple (unknown) change points in the linear model. Many of these are implemented in a procedure called **@BaiPerron**. The syntax is:

```
@BaiPerron( options ) depvar start end
# list of regressors
```

Main Options

```
minspan=shortest distance between two breaks [no. of regressors]
maxbreaks=maximum breaks allowed [2]
nfix=number of regressors which are fixed over time [0]
iters=maximum number of iterations [20] (matters only if nfix>0)
```

```
[print]/noprint
tests/[notests]
```

These control printing of the final regression and breakpoints (PRINT) and of the table of BIC, LWZ and F tests for choosing the number of breaks.

Supplementary Card

list of regressors with the fixed regressors (if any) listed first

Description

Bai-Perron is an algorithm for efficiently finding the least squares break points in a linear regression model of the form $y_t = X_t\beta + u_t$. From a computational standpoint, the key result here is that the level of effort of calculation can be reduced substantially when there are two or more breaks. This is (in general) specific to the case of the linear model. In the general framework described in Section 11.3, for a fixed value T_1 for the first breakpoint, there are a large number of partitions for the range $[T_1 + 1, \dots, T]$. For a linear model with complete coefficient breaks, there is no need to re-estimate the regression over $[1, \dots, T_1]$ for every choice for breaking the later sample—the coefficients and sums of squared residuals aren't affected by how those later breaks are done. The sums of squared residuals can be computed just once for each legal subset of the range, after which the partitions can be examined efficiently without redoing any of the actual estimations.

Note that this is just a way of doing the calculations efficiently. It applies regardless of the behavior of the variables—if you have a linear regression, are estimating by least squares, and want to find the break points which minimize the sum of squared residuals, you can use **@BaiPerron** to do it. Note, however, that while the estimates are correct even if the data are non-stationary, the tabled critical values for the number of breaks don't apply for non-stationary data.

The basic idea can also be applied if the threshold variable isn't "time". The procedure **@MultipleBreaks** is similar to **@BaiPerron**, but uses an input variable to partition the sample. And the same *idea* (though not the **@BaiPerron** procedure

itself) could be applied to, for instance, a simple non-linear least squares model with a closed-form function. However, ARMA and GARCH models (as examples) have functions which are generated recursively from the start of the data, so they wouldn't benefit from this.

The main point of these multiple break point tests is *to check a specification of a fixed coefficient model*. If the change point analysis shows a break in the specification, it is highly unlikely that you would respond to that information by replacing your original model with the same specification estimated in two subsamples. Instead, you would look for a change in specification, maybe finding a different proxy, or transformation of a variable, or addition of some regressor which would somehow model the break.

You have to be a bit careful when using this procedure. In particular, you need to set the `MINSPAN` and `MAXBREAKS` options with care. The default for `MINSPAN` is the number of regressors. If you are just using a single regressor (a `CONSTANT` for example), then one of the partitions can be a single point. Under those circumstances, it is quite easy for the BIC to keep going down rather steadily as you increase the number of breaks. Increasing `MINSPAN` will tend to force it to find only those situations would seem to fit with a more conventional notion of a structural break rather picking out outliers.

With Fixed (Non-Breaking) Regressors

The dynamic programming algorithm described in Bai and Perron is *guaranteed* to find the partition of the sample which minimizes the sum of squared residuals when the coefficients are allowed to break completely. The procedure also allows for a certain number of coefficients to take a single value in all partitions. This is controlled by the `NFIX` option. That is much more complicated: because all the estimates are linked, the sum of squared residuals over one subsample will depend upon which other subsamples are used. Write the model as:

$$(1) \quad y_t = X_t\beta + Z_t\delta_{(i)} + u_t \text{ if } t \in P_{(i)}$$

The algorithm used in the `NFIX` case fixes the common coefficients (β), applies the full-break algorithm to $y_t - X_t\beta$ to get new partitions, re-estimates (1), and repeats until convergence. While this takes quite a bit less time than applying Section 11.3, it doesn't *guarantee* that the best partition for the model is located.

If you're interested in "broken trends", note that Bai-Perron can handle only certain types of those—either a complete break with both intercept and trend rate changing, or (using the `NFIX` option) a fixed trend rate, but with breaks in the intercept. It can't handle a spline situation where the trend rate changes but the function value doesn't.

11.5 Threshold Autoregression Models

A standard autoregression (or, more generally, an ARMA model) can, with a sufficiently large number of lags and the right parameters, generate cycles of any frequency. However, any such cycle is symmetric, in the sense that the period in “up” cycles must be identical to that in the “down” cycles. Due to randomness, the observed cycles aren’t, in practice, identical, but on average the lengths will be the same. However, it’s a stylized fact that in GDP, the down cycles are shorter than the up—recessions are shorter and steeper than expansions.

The threshold autoregression is an alternative model that can produce asymmetric cycles. Instead of a single autoregression, it uses two (or more) branches with some form of trigger which determines which of the two applies. If that trigger is based on a lag of the series itself, the threshold autoregression remains a self-contained description of the process. Not only will the branches have different coefficients, but they might even have completely different lag structures. The general form is:

$$(2) \quad y_t = \begin{cases} \phi_{11}y_{t-1} + \dots + \phi_{1p}y_{t-p} + u_t & \text{if } z_{t-d} < c \\ \phi_{21}y_{t-1} + \dots + \phi_{2q}y_{t-q} + u_t & \text{if } z_{t-d} \geq c \end{cases}$$

SETAR Model

SETAR stands for **S**elf-**E**xiting **T**hreshold **A**uto**R**egression. The Self-Exciting refers to the fact that the threshold variable (z in (2)) is a lag of the dependent variable. c (and possibly d) are unknown. The estimates of (2) can be done by least squares on the two branches separately, so the evaluation of the overall sum of squares is fairly quick for a given pair of d and c . The problem is that the sum of squares isn’t a continuous function of either: d for the obvious reason that it’s an integer parameter, c because Z_{t-d} takes only a finite number of values in sample, and (2) only changes at those values. As a result, a SETAR must be estimated using a grid search, over c given d , then (if d is unknown as well) over values of d . The grid in c will be over the observed values of Z_{t-d} . This grid search generally ignores a percentage (10–15%) of the most extreme values on either end in order to ensure that all the branches can be estimated with a sufficient number of data points.

The RATS procedure for estimating a SETAR is called **@TAR**. This does the analysis as described in Bruce Hansen (1996). This estimates the optimal break, and includes an option for bootstrapping the significance level, since the maximal F -statistic has a non-standard distribution. The following estimates a SETAR on **GGROWTH** using only lags 1, 2 and 5, with the threshold chosen from those lags as well. It does 1000 bootstrap replications.

```
@tar(laglist=|1,2,5|,nreps=1000) ggrowth
```

STAR Models

The sharp cutoff in the SETAR model is, in many cases, unrealistic, and the lack of continuity in the objective function causes other problems—you can't use any asymptotic distribution theory for the estimates, and without changes, they aren't appropriate for forecasting since it's not clear how to handle simulated values which fall between the observed data values near c .

An alternative is the STAR model (**S**mooth **T**ransition **A**uto**R**egression). Instead of the sharp cutoff, this uses a smooth function of Z_{t-d} :

$$(3) \quad y_t = X_t\beta_{(1)} + X_t\beta_{(2)}G(Z_{t-d}, \gamma, c) + u_t$$

The transition function $G(Z_{t-d}, \gamma, c)$ is bounded between 0 and 1, and depends upon a location parameter c and a scale parameter γ . (There are other, equivalent, ways of writing this. The form we use here lends itself more easily to *testing* for STAR effects, since it's just least squares if the G function is pushed to zero).

The two standard transition functions are the logistic (LSTAR) and the exponential (ESTAR). The formulas for these are:

$$(4) \quad G(Z_{t-d}, \gamma, c) = \begin{cases} 1 - [1 + \exp(\gamma(Z_{t-d} - c))]^{-1} & \text{for LSTAR} \\ 1 - \exp(-\gamma(Z_{t-d} - c)^2) & \text{for ESTAR} \end{cases}$$

LSTAR is more similar to the SETAR model: for values well to the left of c , the G value is near zero, so the coefficient vector is $\beta_{(1)}$, while for values well to the right, G is near one, so the coefficients are $\beta_{(1)} + \beta_{(2)}$. As $\gamma \rightarrow \infty$, LSTAR converges to a standard threshold model with a break at c . As $\gamma \rightarrow 0$, it converges to least squares.

ESTAR treats the tails symmetrically, with values near c having coefficients near $\beta_{(1)}$, while those farther away (in either direction) being close to $\beta_{(1)} + \beta_{(2)}$. ESTAR is often used when it is assumed that there are costs of adjustment in both directions.

STAR models, at least theoretically, can be estimated using non-linear least squares. This, however, requires a bit of finesse: under the default initial values of zero for all parameters used for **NLLS**, both the parameters in the transition function and the autoregressive coefficients that they control have zero derivatives. As a result, if you do **NLLS** with the default `METHOD=GAUSS`, it can never move the estimates away from zero. A better way to handle this is to split the parameter set into the transition parameters and the autoregressive parameters, and first estimate the autoregressions conditional on a pegged set of values for the transition parameters. If you're not familiar with the use of multiple `PARMSETS`, see page UG-129.

γ depends upon the scale of the threshold variable, and it may be hard to think of reasonable guess values. For that reason, it's helpful to replace γ with γ/σ in the formula, where σ is some estimate for the standard deviation for Z . (Standardizing Z itself gives the identical behavior, but it's generally better to keep the threshold

series in its natural scale.) In that form γ will have a common meaning regardless of the scale of Z . (For an ESTAR, use γ / σ^2).

The following shows these two ideas and is taken from `TARMODELS.RPF` (page UG–373). Lag 3 of x is used as the transition variable. This uses the (reciprocal) sample standard deviation of x to re-scale the exponent, and uses guess values of the sample mean for c and 2 for γ . The first **NLLS** estimates the autoregressive coefficients only (those in the `REGPARMS` parmset), then the second estimates all the parameters.

Note that the LSTAR transition function uses the `%LOGISTIC` function. If you use (4) directly, the exp in the denominator might overflow for large positive values of Z_{t-d} . `%LOGISTIC` does the same calculation, but in a “safe” way. The ESTAR function doesn’t have this problem, since very large values will “underflow”, but that just gives the desired zero value for the exp.

```
stats x
compute scalef=1.0/sqrt(%variance)
nonlin(parmset=starparms) gamma c
frml glstar = %logistic(scalef*gamma*(x{3}-c),1.0)
compute c=%mean,gamma=2.0
frml star x = g=glstar,phi1f+g*phi2f
nonlin(parmset=regparms) phi1 phi2
nonlin(parmset=starparms) gamma c
nlls (parmset=regparms,frml=star) x
nlls (parmset=regparms+starparms,frml=star) x
```

Testing

There are LM tests for threshold effects which can be applied to a standard regression. The **@THRESHTEST** and **@TSAYTEST** procedures test for threshold effects given a specific threshold series. **@THRESHTEST** does this by doing a standard structural break test, but with the observations added in order of the threshold series. As the maximum F -statistic has a non-standard distribution, the procedure provides for bootstrapped p -values. **@TSAYTEST** does an “arranged autoregression” test, which uses recursive residuals for the model with sample ordered by the threshold variable. If those are correlated with the regressors, it is evidence of a threshold effect.

@STARTEST tests for general non-linearity which includes either LSTAR or ESTAR. As with the others, it requires a specific threshold series, so, in practice, it is usually applied with several choices for the delay parameter d .

Forecasting/Impulse Responses

TAR models are multi-branch linear models tied together by a non-linear function. Because of the non-linearity, there really is no single “impulse response function.” For instance, the effect of a shock will be quite different when you are near a transition point than when you aren’t. Responses are also very sensitive to scale. In a purely linear model, the effect of doubling the shock is to double the response—that’s not true with a TAR model. What provides much the same information as a standard

IRF is the *eventual forecast function*. This just calculates the point forecasts from some initial situation. How it looks will depend upon what that the initial situation is (as will be true with almost any non-linear model), but it should give some idea of what types of cycles the model can generate.

However, if you want to use the model for *actual* forecasts, the non-stochastic calculation of the eventual forecast function can be highly misleading. It maps out just one set of many possible sets of transitions. Instead, doing random simulations and averaging is the proper approach. Simulation methods in general are covered in Chapter 16 and the simulations specifically for threshold models are described in detail as part of the *Structural Breaks and Switching Models e-course*.

Also requiring simulation is the calculation of a Non-Linear Impulse Response Function (sometimes known as a GIRF for Generalized Impulse Response Function). This is similar to the calculation of forecasts described above, but layers upon that the simulation of a random shock at period 0 (or sometimes a random positive or random negative shock if the two are likely to have quite different effects). Again, note that there is no single “impulse response function” in these models—the response to shocks changes with the initial conditions, and with the shock size and sign.

Identification Problems

Both “sharp” and smooth transition models have the property that, if there really *aren’t* two regimes, an entire set of parameters isn’t identified. For the threshold models, this isn’t a major problem—because the models are estimated by least squares given a test value of the break value, it will always be possible to get the estimates, it’s just that the difference between the fit with and without the breaks is relatively small. For the STAR models, this is much more serious. Because of the non-linear interaction between the threshold function and the model, there are several different ways to get rid of the unnecessary second branch: $\beta_{(2)}$ can be zero, which makes γ and c unidentified or c can be off the end of the data, making $\beta_{(2)}$ unidentified. Either one is a problem for a non-linear estimation method.

The STAR model has another identification problem: if there is a threshold effect, but it’s sharp and not smooth, the optimal value of γ is infinite. And when γ is very large, the function is no longer differentiable with respect to c which is why you have to do a grid search for the sharp transition in the first place—this is more common than one would think. You might find that you need to shift to a TAR model instead.

Multivariate Models

Both TAR and STAR models have extensions to multiple observables. For the TAR model, this requires using **SWEEP** rather than **LINREG** to do the calculations, and for STAR, **NLSYSTEM** rather than **NLLS**. Multivariate TAR models (or threshold VAR as a special case) have a better developed literature and there are several paper replication programs for those. The identification problems for STAR models described above are much more of a problem for multivariate models.

Estimating and Testing TAR Models: TARMODELS.RPF example

TARMODELS.RPF is based on example 3 from Terasvirta (1994). The Canadian lynx population is a standard data set in the non-linear time series literature. The population fluctuates because a high population causes depletion of their food source (lynx are a small wild cat), causing a rapid die-off due to starvation. When the population is low, it can only be rebuilt through reproduction. Thus the up cycles and down cycles are driven by completely different mechanisms, suggesting that a threshold effect is likely.

The **@YuleLags** procedure does a quick, efficient examination of a range of AR models for stationary data. Because this is for a standard AR, with no threshold effects, it might not give a good estimate of the lag length in the presence of a threshold (it would probably overshoot). Here it picks 11 as the minimum AIC.

```
@yulelags (max=20) x
```

Standard practice is to pick the delay by choosing the one which gives the most significant test statistic. The tests are done on the demeaned data using 11 lags and up to a 9 period delay. D=3 is by far the strongest test.

```
diff(center) x / xc  
do d=1,9  
    @StarTest(p=11,d=d) xc  
end do d
```

This initially fixes the C and GAMMA at the mean and two times the reciprocal of the standard error, respectively. The logistic exponent is rescaled by 1/s as shown in the article, so GAMMA is, with that parameterization, put at 2. This gives a fairly sharp split between larger and smaller values, so two branches should have decidedly different coefficients if there is evidence of STAR. With C and GAMMA fixed, NLLS is actually just OLS.

```
stats x  
compute scalef=1.0/sqrt(%variance)  
nonlin(parmset=starparms) gamma c  
frml glstar = %logistic(scalef*gamma*(x{3}-c),1.0)  
compute c=%mean, gamma=2.0
```

These are the two branches. Putting these in as linear equations makes specification more flexible—you can easily change the lag lists and the rest goes through as is.

```
equation standard x  
# constant x{1 to 11}  
equation transit x  
# constant x{1 to 11}
```

Chapter 11: Switching/Break Models

Convert the linear equations into FRMLs with PHI1 and PHI2 as the coefficient vectors. Put them together with the transition function to make the STAR formula.

```
frml(equation=standard,vector=phi1) phi1f
frml(equation=transit ,vector=phi2) phi2f
frml star x = g+glstar,phi1f+g*phi2f
```

```
nonlin(parmset=regparms) phi1 phi2
nonlin(parmset=starparms) gamma c
```

Estimate the model with the STARPparms left out.

```
nlls(parmset=regparms,frml=star) x
```

Based upon the initial results, the standard equation is trimmed to just X{1} and TRANSIT to X{2 3 4 10 11}.

```
equation standard x
# x{1}
equation transit x
# x{2 3 4 10 11}
frml(equation=standard,vector=phi1) phi1f
frml(equation=transit ,vector=phi2) phi2f
```

The new specification is re-estimated with just the regression parameters, then including the STAR parameters.

```
nlls(parmset=regparms,frml=star) x
nlls(parmset=regparms+starparms,frml=star) x
```

To forecast the non-linear equation, we need to create a MODEL with the one formula. This uses SFORECAST for the results for all forecasting calculations, so we need to pull information out that we need as we do forecasts.

```
group starmodel star>>sforecast
```

Compute eventual forecasting function starting in 1925:1.

```
forecast(model=starmodel,steps=40,from=1925:1)
set eforecast 1925:1 1964:1 = sforecast
```

Computes forecasts for 40 steps beginning in 1925 by taking the mean of simulated values over 10000 replications.

```
set meanf 1925:1 1964:1 = 0.0
compute nreps=10000
do reps=1,nreps
    simulate(model=starmodel,from=1925:1,to=1964:1)
    set meanf 1925:1 1964:1 = meanf+sforecast
end do reps
set meanf 1925:1 1964:1 = meanf/nreps
```

Graphs the eventual forecast, the mean forecast and the last few years of actual data. While the process has a fairly regular cycle of roughly eight years, shocks will shift the timing enough that even 16 years out, the probability is fairly high that it will be in a somewhat different phase than it is today. The mean forecasts thus eventually converge to the series mean.

```
graph(footer="Forecasts of Lynx Data with LSTAR Model", $
    key=loright, klabels=$
    ||"Eventual Forecast","Mean Forecast","Actual"||) 3
# eforecast
# meanf
# x 1920:1 1934:1
```

11.6 Unit Roots and Breaks

In a unit root process, the “trend” is the accumulation of a large number of small shocks. Perron (1989) showed that the standard procedures for detecting unit roots could be fooled by one (or more) sharp changes to the process. Since then, there have been a large number of procedures proposed to test for unit roots allowing (under the null, alternative or both) for breaks in the underlying process.

While it is *possible* that one of these tests will uncover the true data generating process, the main point of these is to examine the correctness of a basic unit root testing procedure. Do a standard unit root test (Section 3.11). If that *rejects* the unit root, then there really isn’t much of a point to applying a fancy test that allows for breaks. It’s when you *accept* the unit root with the standard test that it would be a good idea to use one of the procedures described here.

Perron’s original paper picked specific dates for the break based on the historical record. Most subsequent work, however, has allowed for the break to be data-determined. The general framework of the tests is:

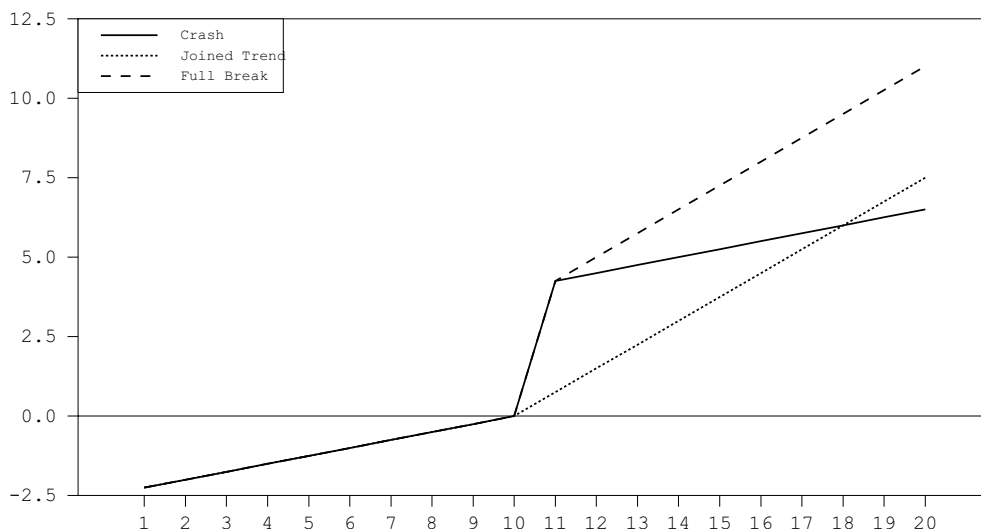
```
Loop over possible breaks
  Do unit root test given the breaks
  If this is least favorable to the unit root hypothesis
    Save it
End
```

The resulting test statistic is the value produced by the set of breaks which is least favorable to the unit root hypothesis. The distribution of that statistic will be non-standard and will depend upon the number and form of the breaks allowed. A common misconception is that this is “estimating the location of the breaks.” Again, these are procedures for testing for unit roots, *not* for choosing a model. The breaks are chosen (in general) to be those that are least favorable to the unit root, not to be the best fitting.

The embedded unit root test will, in general, depend upon “nuisance” parameters such as the augmenting lags of the differences. In most testing procedures, these are chosen separately for each candidate set of breaks. This means that there is quite a bit of number-crunching required. Though any testing procedure can easily be extended from one to two to three or more breaks, once you’re beyond two breaks, the time to do the calculations may get too burdensome. The time complexity of the calculation goes up with $T^{breaks+1}$. With $T=400$, doing two breaks takes (roughly) 400 times as long as one break, so even two breaks might take too long with a large enough data set. However, if three or more breaks are a real possibility, it’s not clear that that will generate the same level of persistence that causes the false acceptance of a unit root that one or two does.

While a break in the mean can create a false acceptance for a unit root in a non-trending series, the main interest is in the behavior of trending series, such as log GDP. There are three main types of breaks considered for trending series:

- Crash:** the trend rate remains the same before and after, but there is an immediate change in the level.
- Join:** the trend rate changes, but there is no immediate change to the level.
- Full Break:** both the trend rate and the level change



Perron (1990) also distinguishes between Additive Outliers (AO) and Innovational Outliers (IO). The AO are as described above: direct changes to the level of the process. The IO add a shift to the error process, so the change works into the data more slowly.

There are many special-purpose RATS procedures for doing unit root tests with breaks. Of those, we have written three to handle any number of breaks. While these can do the computations for three or more breaks, there aren't at this point tables of critical values for those. We've provided for a large number of breaks (as no extra code was required), but aren't sure whether there is any value to this in practice.

The general procedures are **@LSUnit** (based upon Lee and Strazicich, 2003), **@PerronBreaks** (based on Perron, 2006) and **@LPUNIT** (based upon Lumsdaine and Papell, 1997). **@LSUnit** is an LM testing procedure, so it estimates under the null of a unit root process with a particular (broken) trend. **@PerronBreaks** estimates the model under the alternative and examines the restriction to the unit root. Even with the same assumption regarding the form of the break, the testing procedures might produce different values for the location of the break. **@LPUNIT** is based upon a Wald test, estimating the model under the more general alternative hypothesis.

All three procedures use the same options for handling the added lags (LAGS, METHOD and SLSTAY). **@LSUNIT** and **@PERRONBREAKS** use the BREAKS option to

Chapter 11: Switching/Break Models

choose the number of breaks. However, **@LPUNIT** uses the **NBREAKS** option for that.

@PerronBreaks allows for more general types of breaks, with an **AO** option for choosing additive outliers and **IO** for innovational outliers. You must use one (and only one) of these two options:

```
ao=[none]/mean/crash/join/break
io=[none]/mean/crash/break
```

@LSUnit has a **MODEL** option which does only the **AO** type.

```
model=[crash]/break
```

while **@LPUNIT** has a **BREAK** option to choose the break, which, for it, is an **IO** break.

```
break=[intercept]/trend/both
```

All three procedures are included in the *Time Series—Unit Root Test* wizard.

Unit Root Tests with Breaks: UNITROOTBREAK.RPF example

UNITROOTBREAK.RPF applies to the log of U.S. real wages over a 100 year span the one-break versions of the three unit root tests. Graphically, the real wage seems to have had a trend break sometime in the late 1930's (lower growth before, higher after). The first step is a standard Dickey-Fuller test which accepts the unit root. All three breaking tests allow for the full break, both rate and level. Because they use different methods and models for the broken trend, they identify slightly different break points (1939 for **@LSUNIT**, 1940 for **@LPUNIT** and 1936 for **@PERRONBREAKS**). They do, however, all reject the unit root hypothesis once you allow for a break.

```
open data nelsonplosser.rat
calendar(a) 1871
data(format=rats) 1871:1 1970:1 realwages stockprice
*
set logwage = log(realwages)
*
@dfunit(det=trend,maxlag=8,method=gtos) logwage
@lsunit(model=break,lags=8,method=gtos,breaks=1) logwage
@lpunit(break=both,maxlags=8,method=gtos,nbreaks=1) logwage
@perronbreaks(ao=breaks,breaks=1,lags=8,method=gtos) logwage
```

Figures with detrended data

```
@glsdetrend(break=both,tb=1939:1) logwage / wagedetrend
set wagedetrend = logwage-wagedetrend
graph(footer="Figure 9. Logarithm of Real Wages (1900-1970)") 2
# logwage
# wagedetrend
```

11.7 Markov Switching Models

Markov switching models for time series have become quite popular, largely as a result of the work of James Hamilton (see Chapter 22 of Hamilton, 1994). These combine a switching model (with unobservable regimes) for the description of data, with a Markov chain for the regimes.

While popular, these models are quite demanding technically. Before you decide to employ one, make sure that you understand how these work. The regimes are hidden, latent, unobservable; whatever word you want to describe them, there is and cannot be any formula which can tell you why period t is (apparently) in a particular regime. You can try to interpret the results, but in many cases (particularly in models with many switching parameters) there is no clear way to describe the differences. If you think you need multiple regimes, but have a good idea about what determines whether the data are in one regime or the other, you may be better off with a “hard” threshold model such as a TAR or STAR (Section 11.5).

We’ll start with the simpler case of the non-Markov switching model (called a *mixture* model) to introduce many of the concepts.

11.7.1 Mixture Models

Suppose that we have a series y which can be in one of two possible (unobservable) regimes at each time period. (We will use the word “regime” rather than “state” to avoid possible confusion with state-space model.) We’ll use S_t to represent the regime. The likelihood function when $S_t = i$ is written in the very general form $f_{(i)}(y_t | X_t, \Theta)$, where X_t are exogenous and Θ are parameters. Most often, the two regimes are linear regressions with different coefficients, but it’s also possible to apply this with state space or ARCH models. The unconditional probability for $S_t = 1$ is the (unknown) p . The log likelihood element for observation t is

$$(5) \quad \log \{ p f_{(1)}(y_t | X_t, \Theta) + (1 - p) f_{(2)}(y_t | X_t, \Theta) \}$$

Each likelihood element is a probability-weighted average of the likelihoods in the two regimes. This produces a sample likelihood which can show very bad behavior, such as multiple peaks. In the most common case where the two regimes have the same structure but with different parameter vectors, the regimes become interchangeable. Unlike the case with the threshold models where you design the regimes to be the one left of the break and the one right of it, the “labeling” of the two regimes isn’t defined by the model itself.

This model is not particularly useful by itself, as the regimes are independent across time. As such, it’s more often used in modelling the residual process in a more complicated model. However, the estimation methods carry over to more realistic models.

There are three main ways to estimate a model like this: conventional maximum likelihood, expectation-maximization (EM), and Bayesian Markov Chain Monte Carlo (MCMC). These have in common the need for the values for $f_{(i)}(y_t | X_t, \Theta)$. Our sugges-

tion is that you create a **FUNCTION** to do this calculation, which will make it easier to make changes. The return value of the **FUNCTION** should be a **VECTOR** with size equal to the number of regimes. *Remember that these are the likelihoods, not the log likelihoods.* If it's more convenient doing the calculation in log form, make sure that you exp the results before you return. As an example:

```
function RegimeF time
type vector RegimeF
type integer time
local integer i
dim RegimeF(2)
ewise RegimeF(i)=$
    exp(%logdensity(sigsq,%eqnrvalue(xeq,time,phi(i))))
end
```

Maximum Likelihood

The log likelihood function is just the sum of (5) across observations. The parameters can be estimated by **MAXIMIZE**. There are two main issues: first, as mentioned above, the two regimes aren't really defined by the model if the density functions take the same form. Generic guess values thus won't work—you have to force a separation by giving different guess values to the two branches. Also, there's nothing in the log likelihood function that forces p to remain in the interval $[0,1]$. With two regimes, that usually doesn't prove to be a problem, but constraining the p parameter might be necessary in more complicated models. The usual way of doing that is to model it as a logistic, with $p = 1 - (1 + \exp(\theta))^{-1}$.

Maximum likelihood is always the simplest of these to set up. However, in most cases, it's the slowest (unless you use a very large number of draws on the Bayesian method). The best idea is generally to try maximum likelihood first, and go to the extra trouble of EM only if the slow speed is a problem.

EM

EM (page UG–153) is able to simplify the calculations quite a bit in these models. The augmenting parameters are the regimes. Instead of (5), we need to maximize in the M step the simpler

$$(6) \quad E_{\{S_t|\Theta_0\}} \log f(y_t, S_t | X_t, \Theta) = E_{\{S_t|\Theta_0\}} \{ \log f(y_t | S_t, X_t, \Theta) + \log f(S_t | X_t, \Theta) \}$$

For the typical case where the underlying models are linear regressions, the two terms on the right don't interact. Maximizing the sum of the first just requires a probability-weighted linear regression; while maximizing the second estimates p as the average of the probabilities of $S_t = 1$. p is constructed to be in the proper range, so we don't have to worry about that as we might with maximum likelihood. The value of the E step is that it allows us to work with sums of the more convenient log likelihoods rather than logs of the sums of the likelihoods as in (5).

The probabilities of the two regimes used in the M step are not the unconditional p and $1 - p$, but the estimated probabilities using the data and the previous parameter values. The E step requires computing those probabilities of the regimes. That can be done quite simply by Bayes rule:

$$(7) \quad P(S_t = 1 | y_t, X_t, \Theta) = \frac{pf_{(1)}(y_t | X_t, \Theta)}{pf_{(1)}(y_t | X_t, \Theta) + (1 - p)f_{(2)}(y_t | X_t, \Theta)}$$

The application of EM to this problem just requires repeating those two steps. As is typical of EM, it can get estimates close to the likelihood maximizers fairly quickly. However, because it doesn't estimate the p and Θ jointly, the progress of the iterations slows down near the optimum, and it also can't get a full covariance matrix of the estimates including both p and Θ . A useful strategy with larger models where maximum likelihood is too slow is to use EM for a certain number of iterations, then switch over to maximum likelihood to "polish" the estimates.

Bayesian MCMC

As with EM, the regime variables are now treated as parameters. However, now we draw values for them as part of a Gibbs sampler (Section 16.7). The repeated steps are (in some order) to draw Θ given S_t and p , then S_t given Θ and p , then p given S_t . Given the actual values of S_t , the Θ are computed using standard regressions across the subsamples determined by the values of S_t . Given Θ , the data information about $P(S_t = 1)$ is given by (7). That's a draw from a Bernoulli which can be done with %RANBRANCH. Finally, p can be drawn from a beta distribution (with %RANBETA) using the counts of the number of observations currently in each regime, combined with a (weak) prior to prevent p from collapsing to 0 or 1.

As with EM, the p must stay in range by construction. However, the "re-labeling" problem is a major concern here—the sampler can switch the interpretations of the regimes (sometimes many times). When the mixture model is being used for the residual process only, that is of little concern, but when it's the actual model, it will be.

11.7.2 Markov Chains

As mentioned above, a model where the regimes are independent across time will generally be unrealistic for the process itself, and is generally used only for modelling residuals. The Markov Chain offers an alternative where the regime is still unobservable, but dependent on previous regimes. If we're in regime 1, the next period we can either stay in 1, or move to 2. Let's call the probability of staying p_{11} ; the probability of moving thus is $1 - p_{11}$. If we're in regime 2, we could move to 1 or stay in 2. Let's call the probability of moving p_{12} . The probability of staying is $1 - p_{12}$. This is describing an *invariant* chain, with transition probabilities that are the same at all time periods. That *can* be relaxed, generally with only minor changes.

Note that papers and books which concentrate only models with two regimes generally use p_{11} and p_{22} as the two free parameters, and derive the probabilities of switching

regimes from those. However, the parameterization that we are choosing generalizes easily to more than two regimes.

As before, we will have $f_{(i)}(y_t | X_t, \Theta)$ giving the likelihoods at t conditional on the regime. Let $p_{t-1|t-1}$ represent the probability of regime 1 at time $t-1$ given data through $t-1$. Applying the transitions, the probability of regime 1 at t given data through $t-1$ is $p_{t|t-1} = p_{11}p_{t-1|t-1} + p_{12}(1 - p_{t-1|t-1})$. This is the *prediction* step. The log likelihood is now computed as in the simpler model, except with the sequential estimate of the probability (compare with (5)):

$$(8) \quad \log \left\{ p_{t|t-1} f_{(1)}(y_t | X_t, \Theta) + (1 - p_{t|t-1}) f_{(2)}(y_t | X_t, \Theta) \right\}$$

Next is the *update* step, which computes the probability given data through t :

$$(9) \quad p_{t|t} = \frac{p_{t|t-1} f_{(1)}(y_t | X_t, \Theta)}{p_{t|t-1} f_{(1)}(y_t | X_t, \Theta) + (1 - p_{t|t-1}) f_{(2)}(y_t | X_t, \Theta)}$$

This *filtering* procedure for computing the log likelihood works because of the properties of conditional likelihoods. The one remaining problem is what to do with $t=1$. What is the meaning of $p_{0|0}$, which is needed to compute the likelihood for $t=1$? There are two common ways to handle this. The first is to compute the *ergodic* or *stationary* probability of the chain. As long as all the transition probabilities are non-zero, there is a unique probability that doesn't change when you apply the transitions to it. That can be computed fairly easily. The alternative is to add the pre-sample probability to the parameter set. It turns out that using the ergodic probability is simplest if you use maximum likelihood and the parameterized probability is simplest if you use EM.

@MSSETUP functions

Computing a log likelihood element for a Markov switching model takes three steps: the prediction step, the calculation of the log likelihood, then the update step to get the probabilities for use in the next entry. And the formulas above are specific to the two regime model. If you move to three regimes, the number of free parameters in the transition matrix goes up to six, and with four, to twelve. To allow for more than two regimes, it makes sense to switch to matrix operations and break the calculations into manageable pieces. We now describe a set of functions to do this that are provided on the procedure file `MSSETUP.SRC` (and are pulled in with a call to `@MSSETUP` or one of the more specific setup procedures).

The representation for the transition matrix which simplifies the calculations most is an $(n-1) \times n$ matrix, where $\mathbf{P}(i, j)$ is the probability of moving to regime i from regime j . The missing $\mathbf{P}(n, j)$ is just one minus the sum of the other elements in column j . (The transition matrix is sometimes written at the transpose of this—with the target regime in the column and source regime in the row. That convention requires post-multiplying by the transition matrix, rather than premultiplying with the convention used here.)

In the following, we will use the VECTOR $\mathbf{p}^*$ to represent the most “current” estimate of the probabilities of the regimes.

The function **%MCSTATE (P, PSTAR)** takes such a transition matrix, combined with $\mathbf{p}^*$, and returns the VECTOR of predicted probabilities.

%MSUPDATE (F, PSTAR, FPT) returns the VECTOR of updated regime probabilities given the vector of likelihoods (levels, not logs) of the regimes in F and the predicted probabilities of the regimes in the vector PSTAR. FPT is a REAL returned by the function which gives $\mathbf{F} \bullet \mathbf{p}^*$. If F is a vector of likelihoods given the regimes, FPT will be the overall likelihood for this entry so the log of it will be the log likelihood element.

%MCERGODIC (P) takes the transition probability matrix (parameterized as described above) and returns as a VECTOR the ergodic (stationary) distributions of the regimes. In most applications, this will be the appropriate initialization for $\mathbf{p}^*$. To feed this into the PSTAR vector, use the option `START=(PSTAR=%MCERGODIC(p))` on the **MAXIMIZE** which estimates the model.

Two other functions on the file are used for handling a parameterization of $\mathbf{P}$ using logistic indexes. **%MSLOGISTICP (THETA)** returns the result from transforming an $(n-1) \times n$ matrix of logistic indexes into an $(n-1) \times n$ matrix of transition probabilities. **%MSPLOGISTIC (P)** is the inverse of %MSLOGISTICP—it takes the $(n-1) \times n$ matrix of transition probabilities and returns the implied logistic indexes. That mapping, which takes the unbounded θ_{ij} to the bounded p_{ij} , is:

$$(10) \quad p_{ij} = \frac{\exp(\theta_{ij})}{1 + \exp(\theta_{1j}) + \dots + \exp(\theta_{n-1,j})}$$

Using these functions, the basic setup for maximum likelihood estimation for a Markov switching model is:

```
compute n=2
dec rect p(n-1,n)
dec vect f(n) pstar(n)
nonlin p a01 a02 a11 a12 sigma
frml markov = f=SimpleRegimeF(t), $
    pstar=%msupdate(f,%mcstate(p,pstar),fpt),log(fpt)
maximize(start=(pstar=%mcergodic(p))) markov start end
```

The final line in the definition of the MARKOV formula will look the same regardless of the structure of the model. It’s the definition of the “F” vector that changes from model to model.

While these three functions are often useful, the calculations for some models don’t easily fit into this framework. For instance, the model in HAMILTON.RPF (page UG–388) has a large set of regimes with a very sparse (that is, mostly zero) transition matrix, so it does the same types of calculations but uses specialized methods to avoid a lot of multiplies by zero.

Regime Probability Smoothing

The sequence of $p_{t|t}$ gives the probability that the data are in regime 1 at time t given information only through time t . The smoothed estimates of the probabilities are the computed using *all* the data. Computing these requires keeping the entire history of both the filtered $p_{t|t}$ and predicted $p_{t|t-1}$ probabilities, then doing an extra sweep through the data (in reverse) at the end. See Hamilton (1994) for technical details.

To do this, add the following to the set up:

```
dec series[vect] pt_t pt_t1 psmooth
gset pt_t      = %zeros(nstates,1)
gset pt_t1     = %zeros(nstates,1)
gset psmooth   = %zeros(nstates,1)
```

and adjust the MARKOV formula to

```
frml markov = f=SimpleRegimeF(t), $
              pt_t1=%mcstate(p,pstar), $
              pt_t=pstar=%msupdate(f,pt_t1,fpt), log(fpt)
```

and add

```
@%mssmooth p pt_t pt_t1 psmooth
```

after the model has been estimated. %MSSMOOTH is also on MSSETUP.SRC.

Models with Lags

The model which we've been using so far has an observation equation which depends upon the current regime, not lagged regimes—the only dynamic connection among regimes comes through the Markov switching process for them. The analysis gets more complicated if the observation equation also depends upon lagged regimes. There have been several proposals for Markov switching models for GDP growth (or, more generally for Markov switching VAR's) in increasing order of complexity:

$$(11) \quad x_t = \alpha(S_t) + \varphi_1 x_{t-1} + \dots + \varphi_p x_{t-p} + u_t$$

$$(12) \quad x_t = \alpha(S_t) + \varphi_1(S_t)x_{t-1} + \dots + \varphi_p(S_t)x_{t-p} + u_t$$

$$(13) \quad x_t - \mu(S_t) = \alpha + \varphi_1(x_{t-1} - \mu(S_{t-1})) + \dots + \varphi_p(x_{t-p} - \mu(S_{t-p})) + u_t$$

$$(14) \quad x_t - \mu(S_t) = \varphi_1(x_{t-1} - \mu(S_{t-1})) + \dots + \varphi_p(x_{t-p} - \mu(S_{t-p})) + (1-L)u_t$$

In (11) and (12) the autoregression changes with the regime. In (11), only the intercept changes, while in (12), it's the entire equation. Either of these can be handled using the methods described so far. The `StateF` function just needs to compute the residual and likelihood in each regime.

(13) is the Hamilton switching model. With $n=2$, there are high mean growth and low mean growth periods with a regime-invariant autoregression linking them to the

data. The measurement equation depends upon current and p lagged values of the regimes. This is most easily handled by expanding the number of “regimes” to n^{p+1} to cover all the combinations that could occur in (13). The augmented regimes at time t have the underlying regimes at $(t, t-1, \dots, t-p)$. The transition probabilities between most of the augmented regimes is zero, since we can only move from $(S_t, S_{t-1}, \dots, S_{t-p})$ to $(S_{t+1}, S_t, \dots, S_{t-p+1})$ if $S_t, \dots, S_{t-p+1}$ remain the same.

(14) is the model from Lam (1990). While apparently trivially different from (13) (with the $(1-L)$ in the error term), that makes the likelihood at t dependent upon the regimes stretching all the way back to the start of the data set since u is unobservable. While it’s possible to estimate (14) by maximum likelihood, it’s quite a complicated process. This is an ideal situation for Gibbs sampling, since the model only needs to be computed for the single current draw for the history of the regimes, rather than being a probability average across regimes.

@MSVARSETUP

The procedure **@MSVARSetup** sets up a Markov switching VAR (or autoregression if there is only one variable). It creates matrices for the state transition probabilities (**P**), the means or intercepts (**MU**) and the lag coefficients (**PHI**) and defines several functions and formulas needed for estimating the model, both using maximum likelihood and EM.

Syntax

```
@MSVARSetup ( options )
# list of dependent variables
```

Options

```
lags=# of VAR lags [1]
states=# of states [2]
switch=[m]/i/mh/ih/c/ch
```

In each of these, **M** indicates a Hamilton-type model where the mean of each series is regime-dependent, **I** indicates the intercept is regime-dependent, and **C** means that the entire coefficient vector switches—those are mutually exclusive. **H** indicates variances (covariance matrices for multivariate systems) are regime-dependent.

It should be noted that great care is required to estimate a model with switching variances using maximum likelihood (or EM) as the likelihood function is unbounded. To see why, take the mixture model with both means and variances switching. The log likelihood for observation t is

$$(15) \log \left\{ p f_N(x_t - \mu_1, \sigma_1^2) + (1-p) f_N(x_t - \mu_2, \sigma_2^2) \right\}$$

where $f_N(x, \sigma^2)$ is the normal density at x with variance σ^2 . If we take $\alpha_i = x_i$ (we could peg it at *any* data value), the likelihood can be made arbitrarily high by mak-

ing σ_1 very close to 0. The other data points will, of course, have a zero density for the first term, but will have a non-zero value for the second, and so will have a finite log likelihood value. As a result, the likelihood function has many very high “spikes” around μ values equal to the data points.

This can be fixed by using a more complicated Bayesian estimation, since even a weak prior on the variances can exclude these very narrow regions. You can also add a REJECT option to prevent either variance from getting too small. In the case of a VAR, the REJECT test has to be a more complicated test for a nearly singular covariance matrix.

You also may find the results from allowing for switching variances to be disappointing. While you may *hope* that (for instance) SWITCH=MH will give you high and low mean regimes, but with different variances, the problem with latent regimes is that the data may favor high variance and low variance regimes and not really care much about the means—in fact, that *tends* to be the case. Markov models generally work best when the number of switching parameters is sharply constrained.

Example (Maximum Likelihood)

This sets up and estimates a one lag, two regime “Hamilton” (switching mean) VAR model by maximum likelihood. Note that you need to give an explicit estimation range. This uses the logistic parameterization for the likelihood, which requires also computing `P=%MSLOGISTICP(THETA)` during the START processing.

MU will be a VECTOR[VECTOR] with the outer vector for the regimes and inner for the variable. So MU(1) (5) is the mean in the first regime for the fifth variable (DCAN). PHI will be a VECT[RECT] with one matrix per lag. In this case (since there’s only one lag), there will just a single 6×6 matrix.

The %MSVARPROB(T) function does the whole set of calculations for computing the likelihood for the regimes, and doing the prediction and update steps on the probabilities. It returns the likelihood, so we just need to define a **FRML** to give its log. @MSVARINITIAL does a standard set of guess values, aimed at making regime 1 the high mean and regime 2 the low mean regime. (The complete example is on the Krolzig MSVAR replication files as `wbc_multivariate_pt1.rpf`).

```
compute gstart=1962:1,gend=1991:4
@msvarsetup(lags=1,states=2)
# dusa djap dfrg duk dcan daus
nonlin(parmset=varparms) mu phi sigma
nonlin(parmset=msparms) theta
frml msvarf = log(%MSVARProb(t))
@msvarinitial gstart gend
maximize (parmset=varparms+msparms,$
  start=(p=%mslogisticp(theta),pstar=%MSVARInit()),$
  method=bfgs,iters=400) msvarf gstart gend
```

Example (EM Estimation)

The M step of EM for the VAR coefficients (the means, lag coefficients and covariance matrix) is just a probability weighted multivariate linear regression, with the probabilities being the smoothed estimates for each combination of regimes. The M step for the transition matrix requires smoothed estimates not just of the probabilities at t , but of the joint probabilities of all combinations of $\{S_t, S_{t-1}\}$ in order to estimate the probabilities of moving from S_{t-1} to S_t . This requires no extra calculations for the Hamilton mean switching model, since it already needs smoothed estimates of all combinations of regimes from t to $t-p$. The `MSVARSETUP` file includes several support functions for EM estimation. `@MSVAREMGENERALSETUP` sets up the work arrays needed and `@MSVAREMSTEP` does the combination of E and M steps. For the example above, this does 50 iterations of EM.

```
@msvarinitial gstart gend
@msvarEMgeneralsetup
do emits=1,50
    @msvarEMstep gstart gend
    disp "Iteration" emits "Log Likelihood" %log1
end do emits
```

The likelihood functions used by ML and EM aren't strictly comparable. As written, the ML likelihood function uses the ergodic initialization for the pre-sample probabilities. If you try that with EM, the (otherwise) relatively simple M step for the transition probabilities becomes quite complex. Instead, the natural way to handle the pre-sample probabilities with EM is to estimate them, which simply requires taking the probability smoother and running it backwards one extra period. It's possible to add the pre-sample probabilities to the parameter set of ML, but that adds more parameters to an already often quite large set. It remains true that the best approach is generally to use EM up to a point, then switch to ML, but the final estimates won't converge as quickly as they might in a model where the target likelihoods match.

@MSREGRESSION, @MSSYSREGRESSION, procedure groups

The `@MSVARSETUP` procedures are for the specific case of a VAR (or univariate AR) where the only variables allowed are the lagged dependent variables and the intercept (in some form). In many cases, however, that might be too restrictive. There are two other groups of procedures for more general models: `@MSRegression` is for univariate regressions while `@MSSysRegression` is for multivariate regressions (multiple equations with the same explanatory variables). The use of these is covered in detail as part of the *Structural Breaks and Switching Models e-course*.

Hamilton Switching Model: HAMILTON.RPF example

HAMILTON.RPF uses @MSVARESETUP to estimate Hamilton's switching model for GDP growth (Hamilton, 1994, Chapter 22). Despite the fact that it's just one variable, @MSVARESETUP is used because the mean-only switch is included in it. (The mean switch makes little sense except in a self-contained model like a VAR or AR, and so isn't included in @MSRegression).

Set up a mean-switching model with just the one variable and four lags. (The desired two basic regimes is the default).

```
@msvarsetup(lags=4,switch=m)
# g
```

Because everything is recursively defined, Markov switching models need a "hard" start and end for their range, which this sets. The MSVARF is the log likelihood formula, using the function brought it when you execute @MSVARESETUP:

```
compute gstart=1952:2, gend=1984:4
frml msvarf = log(%MSVARProb(t))
```

This is parameterized using the P matrix for the probabilities. MU has the regime-specific means. It's a VECTOR[VECTOR], with the primary selector picking the regime, while the secondary picks the "variable" (which is just 1). So MU(1)(1) is the mean in regime 1 and MU(2)(1) is the mean in regime 2. PHI is what is used when the lag coefficients aren't regime-dependent. It's a VECT[RECT] where the VECTOR is over the lags and RECT over the variables. Again, with just a single variable, the latter has just (1,1) subscripts. So PHI(1)(1,1) is the first lag coefficient, PHI(4)(1,1) the last. SIGMA (again, fixed across regimes) is a 1×1 SYMMETRIC matrix.

```
nonlin(parmset=msparms) p
nonlin(parmset=varparms) mu phi sigma
```

This does the guess values by estimating the base VAR, and copying out the lag coefficients and variance from that, and giving separated values to the means (lower for regime 1, higher for regime 2). This generally works for the Hamilton model with just the switching mean. If the entire coefficient vector is switching, it may not work as well, since it is really designed with the assumption that it's means that differ.

```
@msvarinitial gstart gend
```

This does the estimation. This uses a REJECT option to avoid doing a function evaluation if the transition matrix has any values out of range, which is possible if we are estimating the P form.

```
maximize(parmset=varparms+msparms,$
  start=(pstar=%MSVARInit()),$
  reject=%MSVARInitTransition()==0.0,$
  pmethod=simplex,piters=5,method=bfgs,iters=300) $
```

```
msvarf gstart gend
```

This computes the smoothed estimates of the states, and puts into `PCONTRACT` the probability of the lower mean regime (the first, if our guess values held up).

```
@msvarsmoothed gstart gend psmooth  
set pcontract gstart gend = psmooth(t) (1)
```

To create the shading marking the recessions, create a dummy series which is 1 when the `RECESSQ` series is 1, and 0 otherwise. `RECESSQ` is 1 for NBER recessions and -1 for expansions—it's included on the data set, but more generally can be created (for the U.S.) using the `@NBERCYCLES` procedure.

```
set contract = recessq==1
```

Graph the data on the top and the probability of contraction (with the recession shading) on the bottom.

```
spgraph(vfields=2)  
graph(header="Quarterly Growth Rate of US GNP",shade=contract)  
# g %regstart() %regend()  
graph(style=polygon,shade=contract,$  
    header="Probability of Economy Being in Contraction")  
# pcontract %regstart() %regend()  
spgraph(done)
```

Compute the predictive probability-weighted standardized residuals and check for serial correlation.

```
@MSVARStdResiduals %regstart() %regend() stdu  
@regcorrs(title="MS Predictive Residuals",qstat) stdu(1)
```

Predictive probability-weighted standardized residuals is quite a long description, but they are the proper choice for doing any diagnostics. The regime-specific residuals (the model residuals across time for a specific regime) won't work because the model says nothing about their behavior in the time periods which aren't part of that regime. The residuals have to be computed using the model's predictions, weighted by the predictive probabilities for them to be serially uncorrelated. They are standardized because if the variance were switching, the regimes would be predicting difference variances, so the residuals have to be standardized before being probability-weighted. If the model is correct the residuals would be expected to be serially uncorrelated. They would not, however, be expected to be Normal.

SWARCH (Switching ARCH): SWARCH.RPF example

Switching ARCH/GARCH models were first proposed by Hamilton and Susmel (1994) as a possible alternative to the standard ARCH and GARCH models. Instead of persistence in volatility being a function of the size of past residuals and the variance, it comes through a Markov model for two or more volatility regimes. Technically, switching models are much easier to handle for ARCH than GARCH, as the arch has a “finite” memory, so that’s what this does. The persistence that the lagged variance term in GARCH provides is handled in this model by the Markov process.

SWARCH.RPF estimates a three regime Markov model. There are several possible ways to model the effect of the different regimes on volatility: the one we choose here (from Cai, 1994) multiplies the constant term in the variance equation by the regime dependent multiple. The variances are all relative to that in regime 1, so the “HV” vector has one less element than the number of regimes. Hamilton and Susmel handle it in a slightly different fashion that’s a bit more complicated—see the replication files for that paper. If you’re interested in switching *GARCH*, there are several paper replications for different treatments of that. Note that all of those require approximating the log likelihood, and the accuracy of those approximations is suspect. Of the two “filter” methods, we would recommend Dueker(1997). For greater coverage of switching ARCH and GARCH, see the *Structural Breaks and Switching Models e-course*.

These set the number of regimes, and the number of ARCH lags. Because the likelihood depends only upon the current regime, we don’t need a positive value for LAGS on @MSSETUP.

```
@MSSetup(states=3)
compute q=2
```

HV will be the relative variances in the regimes. This will be normalized to a relative variance of 1 in the first regime, so it’s dimensioned $N-1$.

```
dec vect hv(nstates-1)
```

This will be the vector of ARCH parameters

```
dec vect a(q)
```

We have three parts of the parameter set: the mean equation parameters (here, just an intercept), the ARCH model parameters, and the Markov switching parameters, for which we’ll use the logistic indexes.

```
nonlin(parmset=meanparms) mu
nonlin(parmset=archparms) a0 a hv
nonlin(parmset=msparms) theta
```

uu and u are used for the series of squared residuals and the series of residuals needed to compute the ARCH variances.

```
clear uu u
```

ARCHRegimeF returns a vector of likelihoods for the various regimes at time given residual E. The likelihoods differ in the regimes based upon the values of HV, where the intercept in the ARCH equation is scaled up by HV. Again, the elements of HV are offset by 1 since they're normalized with the first regime at 1.

```
function ARCHRegimeF time e ht
type vector    ARCHRegimeF
type real      e
type integer   time
type vector    *ht
*
local integer  i j
local real     vi
*
dim ARCHRegimeF(nstates) ht(nstates)
do i=1,nstates
  compute vi=a0*%if(i>1,hv(i-1),1)
  do j=1,q
    compute vi=vi+a(j)*uu(time-j)
  end do i
  compute ht(i)=vi
  compute ARCHRegimeF(i)=%if(vi>0,$
    %density(e/sqrt(vi))/sqrt(vi),0.0)
end do i
end
```

As is typically the case with Markov switching models, there is no global identification of the regimes. By defining a fairly wide spread for HV, we'll hope that we'll stay in the zone of the likelihood where regime 1 is low variance, regime 2 is medium and regime 3 is high.

```
compute hv=||10,100||
```

These are our guess values for the P matrix. We have to invert that (using %MSPLOGISTIC) that to get the guess values for the logistic indexes.

```
compute p=||.8,.2,.05|.2,.6,.4||
compute theta=%msplogistic(p)
```

These are the guess values for the other parameters. The ARCH lag parameters are started close to zero, A0 at the series variance. UU is initialized to the unconditional variance.

```
stats x
compute mu=%mean,a0=%variance,a=%const(0.05)
set uu = %variance
```

This will hold the regime-specific variances at each time period:

```
dec series[vect] regimeh
```


12. Cross Section and Panel Data

While the primary focus of RATS is on time series data and time series techniques, it also offers strong support for analysis of cross section and panel data. The emphasis of this chapter is on specialized techniques which are applied mainly (or solely) to these types of data sets.

The textbook replication files for Wooldridge's (2010) *Econometric Analysis of Cross Section and Panel Data, 2nd Edition*, Baltagi's (2008) *Econometric Analysis of Panel Data, 4th Edition* and Greene's (2012) *Econometric Analysis, 7th Edition* include many examples of the techniques described here and others that are more advanced.

For greater coverage on the specific subject of panel data, check out our *Panel and Grouped Data* course. For more information on that, see

https://estima.com/courses_completed.shtml

Probit and Logit Models

Censored and Truncated Samples

Hazard Models

Panel Data Sets

Fixed and Random Effects

12.1 Overview

This chapter describes special techniques used in the analysis of cross-section and panel data sets. The emphasis is on techniques which are used almost exclusively for such data sets, such as probit models. Cross-section data sets are also more likely than time series data sets to require use of weighted least squares (Section 2.3), tests for heteroscedasticity (Section 3.5), and subsample selections (Section 1.6.2 of the *Introduction*).

Below is a brief description of cross-sectional and panel data sets. Topics relating to cross-sectional analysis begin on the next page. For information on working with panel data, see Sections 12.5 and 12.6.

Cross-Sectional Data

Cross-sectional data sets consist of a *single* observation for a number of individuals (or firms or countries, etc.). *Do not* use a **CALENDAR** instruction for cross-sectional data. Set the number of observations to process either with **ALLOCATE** or on the **DATA** instruction or let **DATA** figure it out for you. For instance, suppose you have data on the number of males and number of females in fifty states. You could either do something like:

```
allocate 50
open data statepop.dat
data(format=free,org=obs) / males females
```

or

```
open data statepop.dat
data(format=free,org=obs) 1 50 males females
```

or even

```
open data statepop.dat
data(format=free,org=obs) / males females
```

By omitting the **CALENDAR** instruction, you are telling RATS that your data set has no time series properties. RATS will label entries with the entry number alone. Note that on a **LINREG** (or other least squares estimator), RATS will still calculate a Durbin–Watson statistic. This is likely to have little value, but if your data set is ordered on a particular variable, the Durbin–Watson may be able to detect a specification error related to that variable. (It can't be "serial" correlation, but there can be correlation in the ordering variable).

Panel Data

Panel data sets have *several* observations, collected over time, for a number of individuals. They share properties with both time series data sets and cross-sectional data sets. RATS has a special form of the **CALENDAR** instruction to describe a panel data set. See Section 12.5 on page UG–413 for more information.

12.2 Probit and Logit Models

Background

You can use the instruction **DDV** (discrete dependent variables) to estimate probit and logit models. (The instructions **PRB** and **LGT** from earlier versions of RATS are still available). If there are only two choices, the models take the form:

$$(1) \quad P(Y_i = 1 | X_i) = F(X_i \beta)$$

where the dependent variable Y takes either the value 0 or 1, X is the set of explanatory variables and

- $F(z) = \Phi(z)$ (the standard normal CDF) for the probit
- $F(z) = \exp(z) / (1 + \exp(z))$ for the logit.

See, for example, Wooldridge (2010, section 15.3). Please note that RATS requires a numeric coding for Y . RATS treats any non-zero value as equivalent to “1”.

It’s very useful to look at a framework from which logit and probit models can be derived. Suppose that there is an unobservable variable y_i^* which measures the “utility” of choosing 1 for an individual. The individual is assumed to choose 1 if this function is high enough. Thus,

$$(2) \quad y_i^* = X_i \beta + u_i \text{ with}$$

$$(3) \quad Y_i = \begin{cases} 1 & \text{if } y_i^* \geq 0 \\ 0 & \text{if } y_i^* < 0 \end{cases}$$

If the u_i are assumed to be independent standard normal, the result is a probit model; if they’re independent logistic, we get a logit. (RATS also allows use of the asymmetric extreme value distribution). This type of underlying model is used to derive many of the extensions of the simple binary choice probit and logit.

Estimation

Probit and logit models require fitting by non-linear maximum likelihood methods, as discussed in Chapter 4. However, their likelihood functions are usually so well-behaved that you can simply let the program take off from the standard initial estimates (all zeros). You may, in bigger models, need to boost the **ITERS** option. But usually all you need is

```
ddv(dist=probit)  depvar      (use DIST=LOGIT for logit)
# list of regressors
```

See the instruction **DDV** in the *Reference Manual* for technical details. **DDV** includes the standard options for computing robust covariance matrices, but keep in mind that the estimates themselves may not be consistent if the assumption about the distribution isn’t correct.

Using PRJ

The instruction **PRJ** has many options designed to help with diagnostics and predictions for logit and probit models. In its most basic use, after estimating a model by **DDV**, you can compute the series of the “index” $X_i\beta$ by just

```
prj index
```

With the option **CDF**, you can generate the series of predicted probabilities of the “1” choice. Use the option **DIST=LOGIT** if you want these to be calculated for the logit or **DIST=EXTREME** for the extreme value—the default is to compute these for the normal, regardless of your choice on the **DDV**. Some additional statistics can be obtained using the options **DENSITY**, **MILLS** and **DMILLS**.

PRJ will also let you compute the index, density and predicted probability for a single input set of X 's. The values are returned as the variables **%PRJFIT**, **%PRJDENSITY** and **%PRJCDF**. The two options which allow this are **XVECTOR** and **ATMEAN**. The **ATMEAN** option requests that the calculation be done at the mean of the regressors over the estimation range, while with **XVECTOR** you provide a vector at which you want the values calculated. This example computes the “slope coefficients” for a probit, giving the derivatives of the probability with respect to the explanatory variables evaluated at the mean.

```
ddv(dist=probit) grade
# constant gpa tuce psi
prj(atmean,dist=probit)
disp "Slope Coefficients for probit"
disp %prjdensity*%beta
```

Generalized Residuals

With **DDV**, you can compute the *generalized residuals* by including the **GRESIDS** option. If the log likelihood element for an observation can be written $g(X_i\beta)$, the generalized residuals are the series of derivatives $g'(X_i\beta)$. This has the property that

$$(4) \quad \partial/\partial\beta \sum g(X_i\beta) = \sum g'(X_i\beta)X_i = 0$$

that is, the generalized residuals are orthogonal to the explanatory variables, the way the regular residuals are for a least squares regression. They crop up in many diagnostic tests on logit and probit models. For instance, the following (from Greene, 2012, example 17.10) computes an LM test for heteroscedasticity related to the series **KIDS** and **FAMINC**. The test statistic determines whether two constructed variables are also orthogonal to the generalized residuals.

```
ddv(gresids=gr) lfp
# constant wa agesq faminc we kids
prj fit
set z1fit = -fit*kids
set z2fit = -fit*faminc
```

```
mcov(opgstat=lm) / gr
# %reglist() z1fit z2fit
cdf(title="LM Test of Heteroscedasticity") chisqr lm 2
```

Ordered Probit and Logit

In an ordered probit model, there are three or more choices, with the possible choices having a natural ordering. A single index function combined with a partitioning of the real line is used to model the choice process. If we have m choices, let $a_1, \dots, a_{m-1}$ be the upper bounds on the intervals (the top choice is unbounded above). If an individual's index is less than a_1 , she chooses 1; if between a_1 and a_2 , she chooses 2, etc. An ordered probit occurs if the index function I takes the form

$$(5) \quad I_i = X_i' \beta + u_i$$

where u_i is a standard normal, and an ordered logit if u is a logistic. The probability that an individual chooses j is

$$(6) \quad P(\text{choose } j | X_i) = \begin{cases} \Phi(a_1 - X_i' \beta) & \text{if } j = 1 \\ 1 - \Phi(a_{m-1} - X_i' \beta) & \text{if } j = m \\ \Phi(a_j - X_i' \beta) - \Phi(a_{j-1} - X_i' \beta) & \text{otherwise} \end{cases}$$

To estimate an ordered probit using **DDV**, add the option **TYPE=ORDERED**. *Do not* include **CONSTANT** in the regressors—a non-zero intercept would simply shift the cutpoints, leaving an unidentified model.

This is a part of an example from Wooldridge (2010, 16-2):

```
ddv(type=ordered,dist=probit,cuts=cuts) pctstck
# choice age educ female black married $
  finc25 finc35 finc50 finc75 finc100 finc101 wealth89 prftshr
```

The **CUTS** option returns the vector of cut points. Only the standard regression coefficients are included in the **%BETA** vector. Note that RATS does not require that the dependent variable have a specific set of values for an ordered probit; they just have to have a numeric coding in which each choice has a unique value, and these increase in the natural order of the choices. For instance, you could use 1,2,..., m , or 0,1,..., $m-1$. However, the labeling on the output for the cutpoints will be **CUT (1)**, **CUT (2)**, ... regardless of the coding that you use.

Multinomial Logit

There are two main formulations of the logit model for multiple (unordered) choices: the multinomial logit and the conditional logit, though the dividing line between the two is sometimes a bit unclear. The common structure of these is that for each individual there are (linear) functions $f_1, \dots, f_m$ of the explanatory variables and it's assumed that

$$(7) \quad P(\text{choose } j) = \exp(f_j) / \sum_{i=1}^m \exp(f_i)$$

In the classic multinomial logit, the f functions for individual i take the form

$$(8) \quad f_j = X_i \beta_j$$

that is, each choice uses the same explanatory variables, but with a different set of coefficients. With this structure, the coefficients on one of the choices can be normalized to zero, as subtracting (say) $X_i \beta_1$ from all the f values will leave the probabilities unchanged. Thus, the model needs to estimate $m-1$ sets of coefficients. To estimate a multinomial logit model, add the option `TYPE=MULTINOMIAL`. (RATS doesn't do multinomial probit, so you don't need the `DIST` option for logit). For instance, the following is a part of an example from Wooldridge (2010, example 15.4).

```
ddv(type=multinomial, smpl=smpl) status
# educ exper expersq black constant
```

The choices are (numerically) coded into the dependent variable. You can choose whatever scheme you want for this, as long as each choice uses a single number distinct from all other choices. For instance, you can use 0, 1, 2, ... or 1, 2, 3, ... However you code it, the coefficients are normalized to be zero for the lowest numbered choice.

The alternative to the multinomial logit is the conditional logit model of McFadden (1974). Here the f 's take the form

$$(9) \quad f_j = X_{ij} \beta$$

where X_{ij} is the set of attributes of choice j to individual i . Note that the coefficient vector is the same. The idea behind the model is that there is a common "utility function" over the attributes which is used to generate the probabilities. Theoretically, this model could be used to predict the probability of an alternative which hasn't previously been available (a proposed transit system, for instance), as the probabilities depend only upon the attributes of the choice. If you want to do this, however, you have to be careful about the choice of variables—any choice-specific dummies would render the calculations involving new choices meaningless.

The “X” variables for the conditional logit take quite a different form from those for the multinomial logit and the other models considered earlier. To keep the overall form of the instruction similar to the rest of **DDV**, the data are input differently: there is one observation in the data set for each combination of an individual and choice. If, for instance, there are four possible choices and 500 individuals, the data set should have 2000 observations. Internally, **DDV** will compute likelihoods on an individual by individual basis, and will report the number of individuals as the number of usable observations. The dependent variable will be a zero-nonzero coded variable which is non-zero if the individual made the choice indicated by that entry. In order for **DDV** to calculate the likelihood, you need a separate series which distinguishes the data for the individuals. Use the **INDIVS** option to show what series that is. The **MODES** option isn’t required to estimate the model, but allows additional diagnostics; it provides the name of the series which shows which choice an entry represents.

The following is a part of an example from Greene (2012, section 18.2.9). The data set is 210 individuals, 4 choices (air, train, bus, car) for each. The data set itself doesn’t include the individual and mode identifiers, so they are constructed as a function of the entry number. This uses choice-specific dummies, which are also generated. Note that you don’t include **CONSTANT**, as it would wash out of the probabilities. This also estimates a separate model for the individuals who didn’t choose “air” as part of a Hausman test. In order to do this, you just need to knock out all the “air” entries using a **SMPL** option. If **DDV** finds an individual who chose none of the options in the data set being used in the estimation, that observation will be dropped from the calculations, which is what we want.

```
set indiv  = %block(t,4)
set which  = %clock(t,4)
set chosen = mode

set dair   = which==1
set dtrain = which==2
set dbus   = which==3
set airhinc = dair*hinc

*
* Conditional logit model
*
ddv(type=conditional,indiv=indiv,modes=which) chosen
# dair dtrain dbus gc ttime airhinc
ddv(type=conditional,indiv=indiv,modes=which,smpl=which<>1) chosen
# dair dtrain dbus gc ttime airhinc
```

Using MAXIMIZE

You can also use the **MAXIMIZE** instruction to estimate probit and logit models that don't fit into one of the forms that can be handled by **DDV**. The key for doing this is the function **%IF**:

$$\%IF(x,y,z) = \begin{cases} y & \text{if } x \neq 0 \\ z & \text{if } x = 0 \end{cases}$$

If you create a **FRML** called **ZFRML** which computes the equivalent of $X_i\beta$, then the log likelihood **FRMLS** for probit and logit are

```
frml probit = (z=zfrml(t)) , %if(y,log(%cdf(z)),log(%cdf(-z)))
frml logit = (z=zfrml(t)) , %if(y,z,0.0)-log(1+exp(z))
```

The easiest way to create **ZFRML** is by doing a **FRML** instruction with either the **LASTREG** option (after doing a regression) or with the **REGRESSORS** option.

Samples with Repetitions

In econometric work, we occasionally have a data set where there are repetitions on each set of values for the explanatory variables. Ideally, you will have one series with the number of repetitions on the X values, and another series indicating either the total number or the fraction of "successes." You can estimate a probit or logit for such a data set by using **MAXIMIZE**. If series **REPS** and **SUCCESS** are the number of repetitions and the number of successes, the log likelihood for probit is

```
frml probit = (z=zfrml(t)) , $
              success*log(%cdf(z))+(reps-success)*log(1-%cdf(z))
```

If the data are limited to the fraction of successes *without* total counts, use

```
frml probit = (z=zfrml(t)) , $
              fract_success*log(%cdf(z))+(1-fract_success)*log(1-%cdf(z))
```

Heteroscedastic Probit

This estimates a probit model allowing for possible heteroscedasticity in the residual of the "index" process. The scedastic model takes the form

$$(10) \exp(b_0 \times \text{KIDS} + b_1 \times \text{FAMINC})$$

```
frml(regress,vector=bp) zfrml
# constant wa agesq faminc we kids
frml(regress,vector=bh) hfrml
# kids faminc
frml hprobit = (z=zfrml/sqrt(exp(hfrml))),$
              %if(lfp,log(%cdf(z)),log(%cdf(-z)))
nonlin bp bh
maximize(pmethod=simplex,piters=10,method=bhhh) hprobit
```


Logit and Probit Models: PROBIT.RPF example

PROBIT.RPF computes several logit and probit models for exercise 11.7 in Pindyck and Rubinfeld (1998). It tests the significance of the “Children in School” dummies using both a Wald test (by **EXCLUDE**) and likelihood ratio.

```
open data probit.dat
data(org=obs) 1 95 public1_2 public3_4 public5 private $
  years teacher loginc logproptax yesvm
```

Linear probability model

```
linreg yesvm
# constant public1_2 public3_4 public5 private $
  years teacher loginc logproptax
```

```
ddv(dist=logit) yesvm
# constant public1_2 public3_4 public5 private $
  years teacher loginc logproptax
ddv(dist=probit) yesvm
# constant public1_2 public3_4 public5 private $
  years teacher loginc logproptax
```

Test whether “Children in School” dummies are significant. Use “Wald” test first.

```
exclude(title="Wald Test of Children in School Dummies")
# public1_2 public3_4 public5 private
```

Likelihood ratio test. We already have the unrestricted model.

```
compute logunres = %logl

ddv(dist=probit) yesvm
# constant years teacher loginc logproptax
compute logres=%logl

compute lratio=2*(logunres-logres)
cdf(title="LR Test for Children in School Dummies") chisqr $
  lratio 4
```

Chapter 12: Cross Section/Panel Data

Probit and Linear Probability Models: UNION.RPF example

UNION.RPF is taken from Johnston and DiNardo (1997), pp. 415-425. It analyzes the probability of union membership using a sample of 1000 individuals with probit, logit and linear probability models.

```
open data cps88.asc
data(format=prn,org=columns) 1 1000 age exp2 grade ind1 married $
  lnwage occ1 partt potexp union weight high
```

Fit a linear probability model. Do a histogram of the predicted “probabilities”

```
linreg union
# potexp exp2 grade married high constant
prj fitlp
density(type=histogram) fitlp / gx fx
scatter(style=bargraph,$
  header="Histogram of Predicted Values from a LPM")
# gx fx
```

Probit model. Show a scatter graph of the predicted probabilities of the probit vs. the LPM. Include the 45 degree line on the graph.

```
ddv(dist=probit) union
# potexp exp2 grade married high constant
prj(dist=probit,cdf=fitprb)
scatter(style=dots,lines=||0.0,1.0||,$
  header="Probit vs Linear Probability Model",
  hlabel="Predicted Probability(Probit)",
  vlabel="Predicted Probability(LP)")
# fitprb fitlp
```

Evaluate the predicted effect of switching industries on individuals currently in low union industries. This is done by evaluating the predicted probabilities with the value of “high” turned on for all the individuals, then knocking out of the sample those who were already in a high union industry. The predicted probabilities are obtained by “dotting” the coefficients with the required set of variables, then evaluating the normal CDF (%CDF) at those values.

```
set z = %dot(%beta,||potexp,exp2,grade,married,1,1||)
set highprb = %if(high,%na,%cdf(z))
scatter(style=dots,vmin=0.0,lines=||0.0,1.0||,header=$
  "Effect of Affiliation on Workers in Low-Union Industries")
# fitprb highprb
```

12.3 Censored and Truncated Samples

Background

Consider the basic regression model:

$$(11) \quad y_i = X_i\beta + u_i$$

Suppose you wish to analyze this model using a cross-sectional sample. If the sample is chosen randomly from all available individuals, you can just use standard regression techniques. Similarly, there are no special problems if you select your sample based upon values of the *exogenous* variables.

In this section, we examine situations in which the sample is limited in some manner based upon the values of the *dependent* variable y . The two simplest types of samples with *limited dependent variables* are truncated and censored samples.

In a *truncated* sample, an observation is left out of the observable sample if the value of y does not meet some criterion. For example, suppose you want to use payroll data to study the number of hours worked. You will have a truncated sample because your study will exclude people who work zero hours and are thus not on a payroll.

A *censored* sample (“tobit” model) has some observations for which we do not observe a true value of the dependent variable. For instance, in a study of unemployment duration, we will not see the *true* duration for an individual still unemployed at the end of the survey period.

More generally, if the value of a variable which is determined simultaneously with the dependent variable influences whether an observation is in the sample, then the sample suffers from *selection bias*.

We can see the statistical problem produced by such samples in the following example, where the sample is truncated at zero:

$$(12) \quad y_i = X_i\beta + u_i; \text{ observe } i \text{ only if } y_i > 0$$

If $X_i\beta$ is small (less than 0), observation i can only be in the sample if u is large enough that y_i is positive. If you estimate by least squares, u_i and $X_i\beta$ will be negatively correlated and $\hat{\beta}$ will be biased towards zero. The mere fact that an observation is in the sample gives us at least some information about its residual.

Estimation

For the simplest forms of these, you can use the instruction **LDV** (limited dependent variables). Some other models can be done using **MAXIMIZE**, while others have a likelihood too complex for attack with maximum likelihood but can be estimated consistently by two-step methods.

Maximum Likelihood

Start with the normal linear model:

$$(13) \quad y_i = X_i\beta + u_i, \quad u_i \sim N(0, \sigma^2) \quad \text{i.i.d.}$$

If y_i is truncated below at TR (that is, only observations for which $y_i \geq TR$ are in the sample), the (log) likelihood element for entry i is

$$(14) \quad K - \frac{1}{2} \log \sigma^2 - \frac{1}{2\sigma^2} (y_i - X_i\beta)^2 - \log \Phi\left(\frac{X_i\beta - TR}{\sigma}\right)$$

It turns out that a slight change to the parameterization, due to Olsen (1978), makes this a better behaved function: instead of $\{\beta, \sigma\}$, change to $\{\gamma, h\} \equiv \{\beta / \sigma, 1 / \sigma\}$. The log likelihood is then

$$(15) \quad K + \log h - \frac{1}{2} (hy_i - X_i\gamma)^2 - \log \Phi(X_i\gamma - hTR)$$

This is what **LDV** does internally, so if you do a **TRACE** option, it will show coefficient estimates for that parameter set. The model is switched back to the standard parameterization (with recomputed covariance matrix) once estimation is finished.

If y is censored below at TR (we don't observe a true value of y if $y < TR$), then the log likelihood elements are (again with Olsen's parameterization)

$$(16) \quad \begin{cases} K + \log h - \frac{1}{2} (hy_i - X_i\gamma)^2 & \text{for observations which are not censored} \\ \log \Phi(hTR - X_i\gamma) & \text{for those that are} \end{cases}$$

This is often called the Tobit model. To use this with **LDV**, the censored values of y should be equal to their limit.

Using LDV

LDV allows you to estimate a model with either censoring or truncation, and limited either above, below or on both ends. You select which form of limit the dependent variable has with the options **TRUNCATE** or **CENSOR**. The choices for each are **LOWER**, **UPPER** or **BOTH**. (The default is **NEITHER**).

To set the truncation limit, use the option **UPPER** for the upper limit (if it exists) or **LOWER** for the lower limit. The limit can be a single value for all entries, or can vary across individuals. It does, however, have to be computable prior to estimation, that is, you can't have the limit depending upon estimated parameters. For instance, the following estimates a model with censoring at both ends with a lower limit of 0 and upper of 4. (In the application from which this is taken (from Greene's 6th edition), the dependent variable is forcibly being censored at 4 because the values above 4 were categories, rather than observable values.)

```
set ytop = %min(y,4)
ldv(censor=both,lower=0.0,upper=4.0) ytop
# constant z2 z3 z5 z7 z8
```

and, from the same application, this estimates the model truncated at zero, restricting it to the sample with non-zero values of the series `AFFAIR`.

```
ldv(truncate=lower,lower=0.0,smpl=affair) y
# constant z2 z3 z5 z7 z8
```

If some entries are censored, while others aren't, you can show this by providing a series for `UPPER` or `LOWER` with entries in that set to `%NA` (missing value) for those that aren't limited. Suppose, for instance, that you have a series `LIMIT` in the data set which is the upper limit for an entry if it's non-zero, and shows no limit if it's zero. You could estimate this with a sequence like:

```
set upper = %if(limit==0,%na,limit)
ldv(censor=upper,upper=upper) ...
```

Because the log likelihood is well-behaved (when reparameterized as shown), and the second derivatives are rather simple to compute, **LDV** always estimates by Newton-Raphson (page UG–115). You can use the standard set of options for computing robust covariance matrices, but keep in mind that the estimates themselves are unlikely to be consistent if the assumption of normality is wrong.

Like **DDV**, you can compute the “generalized residuals” with **LDV** by using the `GRESIDS` option. Note, however, that these are computed using the reparameterized model. The only difference between the generalized residuals based upon the two parameterizations is in a scale factor, which is usually of no consequence.

LDV also can do *interval estimation*, where the “dependent variable” really consists of a pair of values which bracket an otherwise unobservable amount. Use the option `INTERVAL`, and provide the bracketing values with the `UPPER` and `LOWER` options. Use an `%NA` for either one to show that it is unlimited; for instance, if an entry in the `UPPER` series is `%NA`, it means that the value was unbounded above. Note that you still need a dependent variable series, though its only real purpose is to show which entries should be used in estimation: any entry with a missing value in the dependent variable is skipped. The following, for instance, is from Verbeek (2008, example 7.2.4). The data were generated by starting with an initial bid (`BID1`). If an individual indicated an unwillingness to pay that much, a lower number was offered which also could be accepted or rejected. If the initial bid was accepted, a higher value was tried. The series `UPPER` and `LOWER` are created to show the upper and lower bounds created by this sequence.

```
set upper = %if(nn,bidl,%if(ny,bidl,%if(yn,bidh,%na)))
set lower = %if(nn,%na,%if(ny,bidl,%if(yn,bidl,bidh)))
ldv(upper=upper,lower=lower,interval,gresid=gr) bidl
# constant
```

Using MAXIMIZE

It's useful to see how to estimate a limited dependent variables model using **MAXIMIZE**, in case you get an application which doesn't fit the form of **LDV**. As was the case with the logit and probit models, this is simplified by creating a FRML which computes $X_i\beta$.

To estimate a regression truncated below, do something like the following (where the series LOWER is set equal to the truncation values):

```
nonlin(parmset=sparms) sigmasq
linreg y
# constant x1 x2
frml(lastreg,vector=b,parmset=bparms) zfrml
compute sigmasq=%seesq
frml truncate = (z=zfrml(t)) , $
    %logdensity(sigmasq,y-z)-%logcdf(sigmasq,z-lower)
maximize(method=bfgs,parmset=bparms+sparms) truncate
```

This estimates the model using the standard parameterization, starting from the least squares estimates.

The FRML for the log likelihood function for censored observations requires use of %IF to select the appropriate branch.

```
frml tobit = (z=zfrml(t)) , %if(y==lower,$
    %logcdf(sigmasq,z-lower) , $
    %logdensity(sigmasq,y-z))
```

For example, this would estimate the example from Greene (page UG–405) using **MAXIMIZE** censoring below (only) at 0:

```
linreg y
# constant z2 z3 z5 z7 z8
nonlin(parmset=sparms) sigmasq
frml(lastreg,vector=b,parmset=bparms) zfrml
compute sigmasq=%seesq
frml tobit = (z=zfrml(t)) , %if(y==0,$
    %logcdf(sigmasq,-z)) , $
    %logdensity(sigmasq,y-z))
maximize(method=bfgs,parmset=bparms+sparms) tobit
```

Sample Selection and Heckit Estimators

The model with censoring below at 0 is often called the *Tobit* model. In the original context, the dependent variable was expenditure on a car. If the individual didn't buy a car, this showed zero. One potential problem with the tobit model is that it combines two decisions (buy a car or not, and, if so, spend how much) into a single regression equation. An alternative way to model this is to have a separate model (a probit, presumably) for the first decision, and given a "yes" answer on the first decision, a regression to explain the amount spent.

The first step probit is straightforward. The second step, however, is likely subject to selection bias. If the underlying model for the probit is to choose to purchase if and only if

$$(17) \quad X_i\gamma + v_i > 0$$

and the model for the expenditure is

$$(18) \quad y_i = X_i\beta + u_i$$

(the use of the same explanatory variables is just for convenience), then, if there is a correlation between u and v , the estimates of β in (18) will be biased. Heckman's (1976) idea is to compute the bias in u :

$$(19) \quad E(u_i | X_i, X_i\gamma + v_i > 0)$$

and adjust the second regression to take that into account. If u and v are assumed to be joint normal and i.i.d. across individuals, then u can be written:

$$(20) \quad u_i = \lambda v_i + \xi_i$$

where v_i and ξ_i are independent normals. So

$$(21) \quad E(u_i | X_i, X_i\gamma + v_i > 0) = E(\lambda v_i + \xi_i | X_i, X_i\gamma + v_i > 0) = E(\lambda v_i | X_i, X_i\gamma + v_i > 0)$$

With v as a standard normal (the usual assumption for a probit), then it can be shown that

$$(22) \quad E(v | v > -z) = \frac{\phi(z)}{\Phi(z)}$$

where ϕ is the density of the standard normal and Φ is its cumulative density. ϕ/Φ is the reciprocal of Mills' ratio, which you can obtain using the `MILLS` option of `PRJ` after estimating the first step probit model. Since λ is unknown, adding the inverse Mills' ratio to the regressor set and allowing its coefficient to be estimated will give (under suitable conditions) a consistent estimator of β . This estimator is sometimes known as Tobit II or Heckit.

Chapter 12: Cross Section/Panel Data

The options on **PRJ** which are useful for doing these types of two-step estimators are **MILLS** (inverse Mills' ratio) and **DMILLS** (derivative of the inverse Mills' ratio). The above was all based upon a first stage probit, which conveniently is based upon a standard normal with bottom truncation at zero. The **MILLS** option, however, can do a broader range of calculations. In particular, it can handle a non-unit variance, and truncation either at the top or bottom. In general, the expected value for a truncated $N(0, \sigma^2)$ is given by

$$(23) \quad E(v|v > -z) = \frac{\sigma \phi(z/\sigma)}{\Phi(z/\sigma)}$$

When we're looking to compute the expected value of a residual from the projection in a first stage estimator whose distribution is truncated at the value T_i , the normalized z_i takes the following values:

$$(24) \quad z_i = \begin{cases} \text{Bottom truncation} & (X_i\hat{\gamma} - T_i)/\sigma \\ \text{Top truncation} & (T_i - X_i\hat{\gamma})/\sigma \end{cases}$$

The **MILLS** option computes the value of ϕ/Φ for these values. **PRJ** has three options for describing the truncation and normalization procedure.

scale=the value of σ [1]
upper=SERIES or FRML of upper truncation points [unused]
lower=SERIES or FRML of lower truncation points [series of zeros]

The truncation points can differ among observations. For instance, cutoffs may depend upon some demographic characteristics. However, truncation must either be top truncation for all observations or bottom truncation for all. Note that, by default, the calculation is precisely the one needed for the Tobit II estimator with its first stage probit.

As first step in "heckit" procedure, estimate a probit for the "INLF" variable, using the explanatory variables from the hours equation

```
ddv(dist=probit) inlf
# constant nwifeinc educ exper expersq age kidslt6 kidsge6
```

Compute the inverse Mills' ratios from this

```
prj(dist=probit,mills=lambda2)
```

Run OLS again, including LAMBDA2 as an explanatory variable

```
linreg(smpl=inlf) lwage
# educ exper expersq constant lambda2
```


Tobit Models: TOBIT.RPF example

TOBIT.RPF is from Verbeek (2008, example 7.4.3). It estimates the shares of alcohol (SHARE1) and tobacco (SHARE2) using both standard Tobit (censored at zero) and a two-step estimator with a separate probit estimators.

```
open data tobacco.asc
data(format=free,org=columns) 1 2724 bluecol whitecol flanders $
    walloon nkids nkids2 nadults lnx share2 $
    share1 nadlnx age1nx age d1 d2 w1 w2 lnx2 age2
```

Tobit I models

```
ldv(lower=0.0,censor=lower) share1
# constant age nadults nkids nkids2 lnx age1nx nadlnx
ldv(lower=0.0,censor=lower) share2
# constant age nadults nkids nkids2 lnx age1nx nadlnx
```

OLS regressions on positive observations only

```
linreg(smpl=share1>0) share1
# constant age nadults nkids nkids2 lnx age1nx nadlnx
linreg(smpl=share2>0) share2
# constant age nadults nkids nkids2 lnx age1nx nadlnx
```

Tobit II models, which require first step probits for non-zero consumption. While not strictly necessary, we remap the share values into 0–1 dummies. (DDV would work fine with just the zero-non-zero coding).

```
set choice1 = share1>0
set choice2 = share2>0

ddv(noprint) choice1
# constant age nadults nkids nkids2 lnx $
    age1nx nadlnx bluecol whitecol
prj(mills=lambda)
linreg(smpl=share1>0,title="Tobit II") share1
# constant age nadults nkids nkids2 lnx age1nx nadlnx lambda
*
ddv(noprint) choice2
# constant age nadults nkids nkids2 lnx $
    age1nx nadlnx bluecol whitecol
prj(mills=lambda)
linreg(smpl=share2>0,title="Tobit II") share2
# constant age nadults nkids nkids2 lnx age1nx nadlnx lambda
```

12.4 Hazard Models

Background

Hazard models are a generalization of simple models of “time to failure.” They are used extensively in biometrics to analyze survival times. In econometrics, they are used in modelling transitions from one state to another. Examples are unemployed to employed, working to retired, unmarried to married. A good reference on the subject is Lancaster (1990).

The basic assumption is that the probability of leaving the initial state to the final state over a very short period of time can be modeled as

$$(25) \quad P(\text{exiting in } [t, t+dt] \text{ given no exit before } t) = \theta(t, X_i(t)) dt$$

where $X_i(t)$ are the characteristics of individual i at time t . In the simplest case, θ is a constant, and the exit times follow an exponential distribution with mean $1/\theta$. Obviously, the more interesting cases depend upon the individual or length of time. θ is known as the *hazard* function, perhaps not the best choice of terms for economics examples, but in biometrics these models are most often used to analyze deaths and other less than pleasant occurrences.

Estimation

In full generality, it is almost impossible to estimate the parameters governing the hazard function. If the hazard function varies over time, either because it directly depends upon time, or (even more difficult) the characteristics of individuals change over time, the above would require integrating θ over t to get the probability of exiting at T or earlier.

A common simplification is to assume that the dependence upon t comes only through a function which is independent of the individual. This is called a *proportional hazard* model. For estimation purposes, it is usually assumed that $\theta(t, X_i(t))$ is

$$(26) \quad \theta_0(t) \exp(X_i\beta)$$

If the baseline hazard function $\theta_0(t)$ is simple enough, it's possible to estimate the parameters directly using **MAXIMIZE**. The density function for the exit time is

$$(27) \quad \theta_0(t) \exp(X_i\beta) \exp\left(-\exp(X_i\beta) \int_0^t \theta_0(s) ds\right)$$

If, for instance, θ_0 is a constant, the **FRML** to compute the log of this would be

```
frml logl = z=zfrml,$
      log(theta)+z-exp(z)*theta*exittime
```

where the sub formula **ZRFML** computes $X_i\beta$ and **EXITTIME** is the data variable which gives the actual exit time.

In many cases, however, it is easier to explain how exit times might vary across individuals than to explain the actual lengths of time to exit. Without a precise form for the baseline hazard, full maximum likelihood can't be used. However, there is an alternative method of attack called *partial likelihood*. The idea here is that, if we look at the time an individual exits, we have a valid model of the *relative* probabilities of exit for all individuals who have still not exited. For each possible setting of β , we can form, for each individual, the probability that she is the one who exits at her exit time rather than someone else. It turns out this has all the desirable properties of a likelihood.

The one tricky part about the partial likelihood is that the probability calculation for an individual requires a sum across a subsample, a subsample which changes with the individual. In the example, we handle this by defining for each individual a "Risk Set" vector which has 1's only for those data points where the exit time is at least as large as it is for the individual in question. We also need to include an initialization **FRML** on the **MAXIMIZE** which computes the hazard functions for all the data points. Throughout this, NOBS is the number of observations. It's a good idea to use a variable for this and not code the number in directly. This is from Greene (2012, example 19.8). It first does maximum likelihood assuming a two-parameter Weibull distribution for $\theta_0(t)$, then the proportional hazards estimator.

```

nonlin b0 b1 p
frml zfrml = b0+b1*prod
frml weibduration = (z=zfrml) , (w=-exp(z)*dur^p) , $
    z+w*log(p)+(p-1)*log(dur)
compute p=1.0
maximize(method=bfgs) weibduration

```

RISKSET is a vector of vectors. Each individual has a vector and each vector has a slot for each individual. For individual i, slot j will be 1 if j's exit time is greater than or equal to i's.

```

dec vect[vect] riskset(nobs)
dec vect hazards(nobs)
do i=1,nobs
    dim riskset(i) (nobs)
    ewise riskset(i) (j)=dur(j)>=dur(i)
end do i

```

The function HazardCalc gets computed at the start of each function evaluation to recalculate the relative hazard rates. These are put into the vector HAZARDS. The probability that an individual is the one to exit at her exit time is the ratio of her relative hazard to the sum of those hazards across the risk set for her exit time.

```

nonlin b1
compute b1=0.0
frml hazard = exp(b1*prod)
function HazardCalc
    ewise hazards(i)=hazard(i)
end

```

Chapter 12: Cross Section/Panel Data

```
frml logl = log(hazards(t))-log(%dot(riskset(t),hazards))
maximize(method=bfgs,start=HazardCalc()) logl
nonlin b0 b1 p
frml zfrml = b0+b1*prod
frml weibduration = (z=zfrml), (w=-exp(z)*dur^p), $
    z+w+log(p)+(p-1)*log(dur)
compute p=1.0
maximize(method=bfgs) weibduration
```

RISKSET is a vector of vectors. Each individual has a vector and each vector has a slot for each individual. For individual i, slot j will be 1 if j's exit time is greater than or equal to i's.

```
compute nob=62
dec vect[vect] riskset(nob)
dec vect hazards(nob)
do i=1,nob
    dim riskset(i) (nob)
    ewise riskset(i) (j)=dur(j)>=dur(i)
end do i
```

The formula INIT gets computed at the start of each function evaluation to recalculate the relative hazard rates. These are put into the vector HAZARDS. The probability that an individual is the one to exit at her exit time is the ratio of her relative hazard to the sum of those hazards across the risk set for her exit time.

```
nonlin b
frml hazard = exp(b*prod)
frml init = %do(i,1,nob,hazards(i)=hazard(i)),0.0
frml logl = log(hazards(t))-log(%dot(riskset(t),hazards))
compute b=0.0
maximize(trace,method=bfgs,start=init) logl
```

12.5 Panel and Grouped Data

The following pages provide the basic information you need to use panel data sets in RATS. Topics covered include data set organization, reading data into RATS, and statistical techniques.

Organization of Panel Data Sets

Panel data refers to data sets which include data for several individuals (or firms, countries, etc.) over some number of time periods. The data set itself will usually have one of the following structures:

- A. Each variable has a single long block of values which contains data for all individuals. This is typical for survey data. The data can be grouped by *individual* or by *time*. For example, the data in the column on the left are grouped by individual (all time periods for the first individual, followed by all time periods for the second individual, etc.), while the column on the right is grouped by time (time period 1 for all individuals, time period 2 for all individuals, etc.):

Grouped by Individual

Individual 1, 1997:1
Individual 1, 1998:1
Individual 1, 1999:1
Individual 2, 1997:1
Individual 2, 1998:1
Individual 2, 1999:1

Grouped by Time

Individual 1, 1997:1
Individual 2, 1997:1
Individual 1, 1998:1
Individual 2, 1998:1
Individual 1, 1999:1
Individual 2, 1999:1

We are considering here only situations where there are a fixed number of time periods per individual.

- B. Each individual has a separate time series for each variable. This is most common when data are assembled from different sources.
- C. Each variable has a single long block of values, with a separate series identifying which individual is represented by an observation.

Panel and Grouped Data

The ideal arrangement for panel data for RATS is type "A" data grouped by individual. RATS stores a panel data series as an $N \times T$ stream of data, where N is the number of individuals in the series, and T is the number of time periods. The first T entries of the series represent all the time periods for the first individual, the next T entries represent all the entries for individual two, and so on. *Panel data series always have the same number of entries per individual.* Missing values are used to pad ranges if the data aren't aligned naturally.

RATS can also handle the type "C" data as "grouped" data. Many of the same operations can be applied to grouped data, but only if the model you want to estimate doesn't require lags—it's the blocking by individual with aligned time periods in the panel scheme which allows that to be handled properly. If you need the time sequencing to be correct and your data are organized differently, you will need to convert it. The instruction **PFORM** can be used for most such conversions.

CALENDAR and ALLOCATE with Panel Data

To work with panel data sets, you need to use the appropriate **CALENDAR** and **ALLOCATE** instructions. The syntax of **CALENDAR** for correctly formatted panel data is

```
calendar (panelobs=periods, other options) parameter (optional)
```

The **PANELOBS** option gives the number of time periods per individual. The rest of the **CALENDAR** instruction is the same as for time series data: it describes the time series structure of the data within each cross-section unit. However, you don't *need* to set this other information to use the panel data features of RATS.

The syntax for an **ALLOCATE** for panel data is

```
allocate individuals//lastperiod
```

where *individuals* is the total number of individuals in the panel set. Technically, *lastperiod* is the number of time periods for the *last* individual, though it invariably is just the "*periods*" from **CALENDAR**.

After you have things set up, you can use the functions %PANELSIZE() (number of individuals) and %PANELOBS() (number of time periods) if you want to make a program flexible enough to handle different pane sizes.

Examples

```
calendar (panelobs=20)  
allocate 50//20
```

This is an undated panel data set, with 50 individuals, and 20 time periods (observations) per individual.

```
calendar (panelobs=48,q) 2006:1  
allocate 20//2017:12
```

This is a panel data set, with 20 individuals and 48 time periods per individual. Each cross-sectional unit has quarterly data, starting with January, 2006.

Referencing Series Entries

With panel series, you reference time period *n* of individual *m* as *m*//*n*. For example:

```
compute starting2 = income(2//1)
```

sets **STARTING2** to entry 1 for individual 2 of **INCOME**, and

```
declare vector firstper(20)  
ewise firstper(i)=panelseries(i//1960:1)
```

creates **FIRSTPER** as a **VECTOR** of the 1960:1 entries for the first 20 individuals. Note that *m* and *n* can be any integer value or an integer-valued expression. As shown above, *n* can also be a date if you specified a date scheme on your **CALENDAR**.

Reading Panel Data From a File

You bring panel data into RATS using the same procedure as time series data.

1. Use **CALENDAR** (**PANEL**=...) and **ALLOCATE** instructions.
2. Open a data file with **OPEN DATA**.
3. Read in the data with a **DATA** instruction.

RATS format is the only one which really “understands” panel data. With your data in RATS format, you can select a reduced set of entries from within each cross-section. For instance, if you have data from 1975, but only want to work with the data from 1990, you can set a **CALENDAR** appropriately:

```
calendar (panel=84,q) 1990:1  
allocate 13//2010:4
```

You can also “pad” out the cross-sectional units. If you have data only for 8 observations apiece, but use **PANEL**=10 on **CALENDAR**, **DATA** will insert missing values for entries 9 and 10 within each unit.

However, if you are reading the data from any other format, the organization of the data on the file must match the **CALENDAR** *exactly*. Each individual must have exactly the number of periods specified by the **PANELOBS** option. See “Forming a Panel Data Set” on page UG–418 for instructions on getting your data into this format.

Special Functions

RATS has two functions which can be very useful when working with panel data.

- **%PERIOD**(*t*) returns the time period corresponding to entry *t*. This can be very useful for setting time period dummies (see below).
- **%INDIV**(*t*) returns the number of the individual corresponding to entry *t*.

For example, given a panel set with 2 individuals, and five periods per individual, entry 6 is the first period for the second individual, so **%PERIOD**(6) is 1 and **%INDIV**(6) is 2.

```
set ltrend = %period(t)
```

creates **LTREND** as a trend which repeats within each cross-sectional unit. If you are using a “dated” **CALENDAR**, you can use the date functions (**%YEAR**, **%MONTH**, **%DAY**, **%WEEKDAY**) to determine the year, month, day, or day of the week of a given entry.

SET(NOPANEL)

In **SET** and **FRML**, a reference to an entry in a panel data series is treated as missing if it's in the zone for a different individual than the one for the current entry **T**. This is usually what you want, since it prevents lags of a series from crossing the boundary into another individual's data range. This becomes a problem, however, when you *want* the transformation to cross boundaries. Perhaps you're doing bootstrapping, and the population from which you're drawing is the full set of observations. Or perhaps you're trying to extract data from across individuals into a new series. Use the **NOPANEL** option on a **SET** instruction to turn off this handling. For instance, the following does a random draw from all 200 observations of the series **RESIDS**:

```
boot select 1//1 10//20
set(nopanel) udraw = resids(select(t))
```

Handling Lags

If you run a regression involving lags or leads, any observation which requires a lagged or led value not available within a cross-section is dropped from the regression. Similarly, if you use a lag or lead in a **SET** instruction, you will get a value of **NA** if the lag or lead goes outside the cross-section. For instance, in

```
set dx = x-x{1}
```

DX will have a missing value in the first entry in each cross-sectional unit.

If you need a lag series to have a value of zero when it goes out of the individual's range, you can do that with something like this:

```
dec vect[series] flags(n-2)
do i=n-1,2,-1
    set flags(i-1) = %if(%period(t)<=i,0.0,lcrmte{i})
end do i
```

This generates a set of lag series for lags 2 to $n-1$, each of which zeros out when the time period is less than or equal to the lag being created.

Creating Dummies

If you need separate dummies for each individual, you can use the **PANEL** instruction with the **DUMMIES** option. For instance

```
panel (dummies=dummies)
```

will create a `VECT [SERIES]` called `DUMMIES` where `DUMMIES (1)` is a dummy for individual 1, `DUMMIES (2)` for individual 2, etc. You can also use **SET** with the `%INDIV` function to do dummies like this. For instance

```
dec vector[series] dummyx(%panelsize())
do i=1,%panelsize()
    set dummyx(i) = (%indiv(t)==i)*xreg
end do i
```

This creates the `VECTOR [SERIES]` called `DUMMYX` with dummied-out copies of `XREG`.

Selecting Subsamples

If you want to limit an estimation to a consecutive range of entries, you can simply specify the endpoints using the *start* and *end* parameters as you would with non-panel data. For example, to limit a regression to a range of observations for individual two, you would do something like this:

```
linreg y 2//2011:1 2//2016:12
# constant x1 x2
```

using panel-format date references.

However, this approach will *not* work for selecting a subset of time periods from *each* individual. For example, if you wanted to include observations from all individuals, you would *not* want to do:

```
linreg y 2011:1 2016:12
# constant x1 x2
```

RATS would just interpret these dates as referring to individual one, and would exclude all remaining individuals from the estimation.

Instead, you need to create a dummy variable with non-zero values in the entries you want to include, and then use this series with the **SMPL** option. The `%PERIOD` and `%INDIV` functions, and the various logical operators, are very handy for this. For example, to include data from 2011:1 through 2016:12 for all individuals:

```
set panelsmpl = %period(t)>=2011:1.and.%period(t)<=2016:12
linreg(smpl=panelsmpl) y
# constant x1 x2
```

12.5.1 Forming a Panel Data Set: The Instruction PFORM

If your data are not already in the correct form for a RATS panel data set, you may be able to use the instruction **PFORM** to rearrange it. Our recommendation is that you run a program to transform the data and write it to a RATS format file. Use the RATS format file when you are actually analyzing the data.

PFORM can take several different forms depending upon the current organization of your dataset. Note that you use **PFORM** to create one series at a time. If you need to concatenate separate series to create a panel set, use

```
pform  newseries  
# list of individual series
```

for instance,

```
pform exrate  
# australia canada france germany japan netherlands uk us
```

concatenates eight series into a single panel. The constructed series will have data for each individual running from the earliest valid data point across all the input series to the final one, again across all the input series. If you need to construct several such series, make sure you don't run into a problem with different patterns of missing values causing the constructed series to not be properly aligned. You can use the **SMPL** option to enforce a particular sample for each series. If the raw data series are actually defining a time period rather than an individual, you can use the **INPUT=TIME** option to have them constructed properly. For instance,

```
pform(input=time) logr  
# logr70 logr71 logr72 logr73 logr74
```

creates **LOGR** as a series with five observations per individual.

If you have a dataset with a tag series for the individuals and another for the time period, use something like

```
pform(indiv=id,time=year) p_n  
# n
```

which creates **P_N** from an input series **N**, where each individual has data running from the earliest observed value of **YEAR** to the last observed value.

Finally, if you have an input series which needs to be repeated across individuals or across time, you can use **PFORM** with the options **REPEAT** and **INPUT=TIME** or **INPUT=INDIVIDUAL**. Before you do this, however, you need to set up a panel **CALENDAR** scheme, so **PFORM** knows what the target locations are. **INPUT=TIME** is used with the more common situation where the series is a time series which is the same for each individual:

```
pform(input=time,repeat) p_gdp  
# gdp
```

12.5.2 Panel Data Transformations: the Instruction PANEL

In addition to individual and time dummies, it is often necessary to transform data by subtracting off individual or time period means. This is done with the instruction **PANEL**. For the input series, **PANEL** creates a linear combination of the current entry (**ENTRY** weight), the mean of the series for an individual (**INDIV** weight), the mean across individuals for a given time period (**TIME** weight), the sums in each direction (**ISUM** and **TSUM**) and observation counts (**ICOUNT** and **TCOUNT**). For instance,

```
panel(entry=1.0,indiv=-1.0) series / dseries
panel(time=1.0) series / timemeans
```

The first of these creates **DSERIES** as the deviations from individual means of **SERIES**. The second creates **TIMEMEANS** as a series of means for the different time periods. This is copied across the individuals, so that, for instance, the first time period in each individual's block will be equal to the mean of the first time period across all individuals.

You can also input component variances for a GLS transformation and let **PANEL** do the work of transforming the data. The following, for instance, does a standard random effects GLS transformation of **TV** to **G_TV**, based upon individual effects with **VINDIV** as the variance of the individual component and **VRANDOM** as the variance of the purely random component:

```
panel(vrandom=vrandom,vindiv=vindiv) tv / g_tv
```

This also includes the option **GLS** which can be used to change the form of the GLS transformation. For instance, **GLS=BACKWARDS** added to the above will create **G_TV** using only "backwards" means of **TV**, that is the transformed series at time period t will be constructed using only data from time periods 1 to t —there are some statistical methods where forwards or backwards construction of the transformation is important.

PANEL also has an option to compute a separate sample variance for the data for each individual or for each time period. That series can be used as a **SPREAD** option in a **LINREG**. It should only be used if you have enough data per individual to give a reasonably sharp estimate of the variance.

```
linreg logc
# constant logpf lf logq f2 f3 f4 f5 f6
panel(spreads=firmvar) %resids
linreg(spread=firmvar) logc
# constant logpf lf logq f2 f3 f4 f5 f6
```

12.6 Statistical Methods for Panel Data

Analysis of Variance: the Instruction PSTATS

Does a single statistical model seem to unite the individuals (or time periods) in the data set, or not? The instruction **PSTATS** can be used to help answer that question. It performs an analysis of variance test for common means, across individuals, across time, or both. This can be applied to data series themselves, but is more typically applied to the residuals from a simple regression across the entire data set.

To do this, use **PSTATS** with the option **TEST**. The **EFFECTS** option allows you to choose a test for **INDIVIDUAL** effects, **TIME** effects, or **BOTH**. The following output, for instance, gives a marginal significance level on the test for individual effects of .3597. We would conclude that there really isn't compelling evidence that the individual means differ across individuals.

Analysis of Variance for Series RESIDS					
Source	Sum of Squares	Degrees	Mean Square	F-Statistic	Signif Level
INDIV	1.5912296947623	39	.0408007614042	1.080	0.35971320
ERROR	6.0418415088589	160	.0377615094304		
TOTAL	7.6330712036212	199			

PSTATS will also test a series for equality of variances. To do this, use the **SPREAD** option with the appropriate **EFFECTS** option. (You can't choose **BOTH** for this).

Linear Regressions

When you have a linear regression model, there are a number of techniques which can exploit the structure of a panel data set. See, for instance, Baltagi (2008) for details. Because a panel data set has two or more observations per individual in the sample, it offers possibilities to reduce errors which would occur if only a simple cross section were available. However, it is necessary to make some assumptions which tie the model together across individuals. Among the possibilities:

1. All coefficients are assumed to be constant across individuals (and time).
2. Some coefficients are assumed to be constant, while others differ.
3. The coefficients are assumed to differ, but are "similar," so that large differences are implausible.

In addition, there can be assumptions about a link among the error terms:

1. The error terms can be assumed to be independent across individuals (or across time), but correlated within an individual's record.
2. The error terms can be assumed to be correlated at a given point in time across individuals, but independent across time.

Even if an assumption is plausible for your data set, you need the proper amount of data to employ the technique. For instance, if you have only a few observations per individual, it will be difficult, if not impossible, to allow the coefficients to vary across individuals, since you don't have enough data for any individual to tack down the

coefficient estimates. Similarly, if you have many individuals, you may not be able to estimate freely a covariance matrix of error terms across individuals.

The most commonly employed techniques are the Fixed and Random effects estimators. Fixed effects allows intercepts to vary while keeping other coefficients fixed. Random effects is a related technique which works through assumptions on the error term. The next section is devoted to them. The Random Coefficients model (Swamy's method) is shown in example `SWAMY.RPF` (page UG-427). We'll describe the others more briefly here.

Regressions with Dummy Variables

If you want to allow coefficients other than the intercepts to vary (freely) across individuals, you can create individual dummies times regressors for each varying regressor. (If everything is fixed except the intercepts, you can use a fixed effects estimator.) If you have a very large number of individuals, this can become impractical, as the regression can be too big to be handled.

Part of the `PANEL.RPF` example (page UG-426) shows how to do a regression with dummy variables.

Heterogeneous Regressions

The instruction **`SWEEP`** with the option `GROUP=%INDIV(t)` can handle a variety of complicated operations based upon linear regressions in which the coefficients are allowed to vary (freely) across individuals. Among these are the calculations of means of the coefficient vectors (mean group estimators) and the extraction of residuals from “nuisance” variables.

```
sweep(group=%indiv(t),var=hetero)
# invest
# constant firmvalue cstock
```

does a “mean group” regression of `INVEST` on `CONSTANT`, `FIRMVALUE` and `CSTOCK`, producing in `%BETA` the average of the coefficients across individuals and in `%XX` the covariance matrix of the estimates allowing the variances to differ among individuals.

```
sweep(group=%indiv(t),series=tvar)
# dc lpc{1} lndi dp
# dy ddp constant
```

regresses each of the series `DC`, `LPC{1}`, `LNDI` and `DP` on `DY`, `DDP` and `CONSTANT`, with a separate regression for each individual, producing the `VECT[SERIES] TVAR` with the residuals from those regressions.

Autoregressive Errors: Using AR1

There are some specialized techniques which combine autoregressive errors with some other panel data methods. These aren't available in RATS. If you run an **AR1** instruction on a panel data set, you have two ways of handling the estimation of ρ :

- You can assume that it is fixed across all cross-sections (default treatment).
- You can assume that it is different in each (use option **DIFFERING**).

We don't recommend the use of **DIFFERING** unless you have many time periods per cross-section. The error introduced into the GLS procedure by using a large number of rather poorly estimated serial correlation coefficients can be very large.

If you use **DIFFERING**, **AR1** *does not* iterate to convergence and leaves the ρ 's out of the covariance matrix of coefficients.

Robust Errors

With time series data, you can't, in general, handle arbitrary and unknown patterns of serial correlation—the requirement for consistency is that the window width has to go to infinity slower than the number of data points, and to guarantee a positive definite covariance matrix, you need to use a lag window type (such as Newey-West) which further cuts the contribution of the longer lags. With panel data, however, you can rely on the “ N ” dimension to provide consistency. Assuming that “ T ” is relatively small, you can correct for arbitrary correlation patterns by using **ROBUSTERRO** with the option **LWINDOW=**PANEL. This computes the following as the center term in the “sandwich” estimator, the positive definite:

$$(28) \quad \sum_i \left(\sum_t X_{it} u_{it} \right) \left(\sum_t X_{it} u_{it} \right)'$$

For a similar calculation based upon grouping on some identification other than individual, use the **CLUSTER** option—that uses the same center term, but with “ i ” defined based upon the input clustering expression.

Seemingly Unrelated Regressions

Seemingly unrelated regression techniques (Section 2.7) are, in general, applied to panel data sets, though usually to “small N , large T ” data sets. As implemented in RATS using the instruction **SUR**, these require a well-formed panel of type “B.” You can, however, estimate using a type “A” panel set, using the instruction **PREGRESS** (Panel Regress) with the option **METHOD=**SUR. This estimates the model

$$(29) \quad y_{it} = X_{it}\beta + u_{it}$$

$$(30) \quad Eu_{it}u_{js} = \begin{cases} \sigma_{ij} & \text{if } t = s \\ 0 & \text{otherwise} \end{cases}$$

Note that the number of free parameters in the covariance matrix is $N(N+1)/2$. If you have relatively short time series, you may quickly exhaust your ability to estimate an unconstrained covariance matrix if you include many individuals. An alternative which can be applied to a data set like that if N is too large to make sur feasible is to estimate by least squares, but then compute Panel-Corrected Standard Errors. That can be done using the procedure **@REGPCSE**.

First Difference Regressions

If the model is (29) with the assumption that

$$(31) \quad u_{it} = \varepsilon_i + \eta_{it}$$

one approach for dealing with the individual effects (ε) is to first difference the data within each individual, since

$$(32) \quad u_{i,t} - u_{i,t-1} = \varepsilon_i + \eta_{i,t} - \varepsilon_i - \eta_{i,t-1} = \eta_{i,t} - \eta_{i,t-1}$$

While it's possible to do this manually (with **SET** or **DIFFERENCE**), you can also run this using **PREGRESS** with the option **METHOD=FD**.

Instrumental Variables

Instrumental variables estimators are important in panel data work because, in many cases, correcting for individual effects will create correlation problems between the transformed errors and the transformed regressors. For instance, if the explanatory variables include lagged dependent variables, neither fixed effects nor first differencing will provide consistent estimates for “large N , small T .” (For first differencing, the bias doesn't go away even with large T).

To do instrumental variables with panel data, you use the same instructions as you would for time series data: **INSTRUMENTS** to set the instrument set and **LINREG, AR1** or another instruction with the **INST** option. The main difference isn't with the instructions used, it's the process required to create the instrument set—in panel data sets, the instruments are often only weakly correlated with the regressors, so often large sets of them are used. See, for instance, the **ARELLANO.RPF** example (page UG-429).

Panel Unit Roots and Cointegration

It's extremely difficult with a single time series of limited length to tell the difference between the permanent response to a shock implied by a unit root, and a response with a half-life of 20 quarters (dominant root roughly .97). As in other cases, panel data can offer an alternative way to bring more data to bear on a questions. If we can't get a longer time span from one country, what about doing joint inference on multiple countries?

There are many panel unit root testing procedures. This is probably not surprising given how many testing procedures there are for single time series. And with panel data, there is the added complication that there are different choices for what's homogeneous and what's heterogeneous. If (as is typical) the null is that all individuals have a unit root, is the alternative that none of them do? That some might and some might not? Are the short-term dynamics different from one individual to the next? And so on.

The procedures available are `@LEVINLIN` for the Levin-Lin-Chu(2002) test, `@HTUNIT` for the Harris-Tzavalis(1999) test, `@IPSHIN` for the Im-Pesaran-Shin(2003) test, `@BREITUNG` for the Breitung(2000) test and `@HADRI` for the Hadri(2000) test. These are covered in great detail as part of the *Panel and Grouped Data* course.

With cointegration, you can test or estimate a model or both. Again, a major complication is deciding what is heterogeneous and what is homogeneous. For purposes of testing, allowing the cointegration vectors to vary across individuals is by far the simplest. The procedure `@PANCOINT` produces test statistics aggregated across individuals using several different underlying tests that would be applied to a single individual, such as Engle-Granger tests.

For estimation, it's also simpler to assume that the cointegrating vectors are heterogeneous. Two procedures are available for handling this, both based upon the work of Peter Pedroni. `@PANELEFM` does Fully Modified least squares (individual by individual) while `@PANELDOLS` does dynamic OLS. One thing to note, particularly with `@PANELDOLS`, is that because the estimation is done individual by individual, it is very easy to run out of usable data points. (DOLS uses lags and leads of every right hand side endogenous variable, which can make for a very large regression very quickly). Assuming a homogeneous cointegrating vector requires a more complicated approach since that has to coexist with almost certainly heterogeneous short-run dynamics.

Panel VAR's

We have had more than a few questions about panel VAR's and almost no one who has asked has had any idea what they actually meant. The seminal paper on panel VAR's is Holtz-Eakin, Newey and Rosen(1988), which has a (very) large N small T data set, with lag coefficients fixed across individuals and only intercepts varying. In practice, that's unlikely to be the desired model if you have a small N large T data set. For that, neither fully heterogeneous regressions (where there is little reason for even thinking of the data as a "panel") nor fully homogeneous regressions (which can be estimated by just running a standard VAR on the panel-stacked data) are likely to be interesting. Instead, it probably makes more sense to use some type of shrinkage or mean-grouped estimator. An example of that is provided as part of the *Panel and Grouped Data* course.

12.7 Fixed and Random Effects Estimators

Background

The basic regression model for a (balanced) panel data set is

$$(33) \quad y_{it} = X_{it}\beta + u_{it}, \quad i = 1, \dots, N; \quad t = 1, \dots, T$$

These two types of estimators are designed to handle the systematic tendency of u_{it} to be higher for some individuals than for others (individual effects) and possibly higher for some time periods than for others (time effects).

The fixed effects estimator does this by (in effect) using a separate intercept for each individual or time period. Since N is usually large, we actually implement it either by subtracting out individual and/or time means using the instruction **PANEL** and then doing **LINREG**, or by using the **PREGRESS** instruction.

The random effects estimator is based upon the following decomposition of u_{it} :

$$(34) \quad u_{it} = \varepsilon_i + \lambda_t + \eta_{it}$$

where ε is the individual effect, λ the time effect, and η the purely random effect. β is estimated by GLS using the structure imposed upon u_{it} by this assumption.

A Comparison

There are advantages and disadvantages to each treatment of the individual effects. A fixed effects model cannot estimate a coefficient on any time-invariant regressor, such as sex, schooling, etc., since the individual intercepts are free to take any value. By contrast, the individual effect in a random effects model is part of the error term, so it must be uncorrelated with the regressors. This means, for example, that a systematic tendency to see higher values for those with higher levels of schooling will be reflected in the coefficient on schooling.

On the flip side, because the random effects model treats the individual effect as part of the error term, it suffers from the possibility of bias due to a correlation between it and regressors (such as between unobservable talent and observable schooling).

See the discussion of these points in Hausman and Taylor (1981).

Implementation

The instruction **PREGRESS** (Panel Regress) estimates both of these. This is similar in syntax to **LINREG**, but you choose **METHOD=FIXED** or **METHOD=RANDOM** and which type of effects to allow: **EFFECTS=INDIV**, **TIME** or **BOTH**. For random effects, **PREGRESS** first runs a fixed effects regression and estimates the variances of the components from the fixed effects residuals. You can, however, override this and provide your own variance estimates using the **VINDIV**, **VTIME** and **VRANDOM** options.

Fixed and Random Effects Estimators: PANEL.RPF example

PANEL.RPF estimates a model using several different methods using **PREGRESS**. It also demonstrates how the equivalent estimators to fixed and random effects can be done using least squares with dummy variables and with panel data transformations.

```
cal(panelobs=20) 1935
all 10//1954:1
open data grunfeld.dat
data(format=prn,org=cols)
```

Do fixed and random effects. The intercept in the fixed effects estimator gets zeroed out as a time-invariant variable, but we include it to maintain the same form as the other regressions.

```
preg(method=fixed) invest
# constant firmvalue cstock
preg(method=random) invest
# constant firmvalue cstock
```

First difference regression

```
preg(method=fd) invest
# firmvalue cstock
```

SUR

```
preg(method=sur) invest
# constant firmvalue cstock
```

Do fixed effects as least squares with dummy variables

```
panel(dummies=idummies)
linreg invest
# firmvalue cstock idummies
```

Random effects done using transformed data. We first need estimates of the component variances. The simplest (though not most accurate) can be computed using **PSTATS** with the residuals from a linear regression. **PSTATS** does a Wallace-Hussain variance calculation treating the input information as data (rather than residuals), so it will give a somewhat different values for the component variances from the one you get from **PREGRESS** (even with **VCOMP=WH**) since the latter uses information about the regression being run.

One other difference between the filtered regression and **PREGRESS** with **METHOD=RANDOM** is that the latter (in effect) uses the computed random variance component in calculating the covariance matrix, while the filtered least squares estimator uses the standard OLS estimator on the filtered data, thus recomputing the residual variance based upon the results.

```

linreg invest
# constant firmvalue cstock
pstats(effects=indiv) %resids

set ones = 1.0
panel(gls=standard,effects=indiv,$
      vrandom=%vrandom,vindiv=%vindiv) invest / ifix
panel(gls=standard,effects=indiv,$
      vrandom=%vrandom,vindiv=%vindiv) firmvalue / ffix
panel(gls=standard,effects=indiv,$
      vrandom=%vrandom,vindiv=%vindiv) cstock / cfix
panel(gls=standard,effects=indiv,$
      vrandom=%vrandom,vindiv=%vindiv) ones / constfix
linreg(title="Random Effects using Transformed Data") ifix
# constfix ffix cfix

preg(method=random,vcomp=wh) invest
# constant firmvalue cstock

```

Random Coefficients Model: SWAMY.PRF example

The basic idea in the random coefficients model is that the cross-section coefficient vectors are “drawn” from a distribution with a common mean. This is sometimes known as Swamy’s (1970) model. An almost identical analysis from a purely Bayesian standpoint is given in Leamer (1978), pp 272-5.

The assumptions underlying the model are

$$(35) \quad \beta_i = \beta + u_i, \text{ with}$$

$$(36) \quad \text{var}(u_i) = \Delta$$

If Δ is small, the coefficients will be almost identical. If it were extremely large, the individual coefficient vectors would be, in effect, unconstrained. With a moderate value, the estimator suggests that they have *similar*, but not identical, values.

This requires three passes. The first collects information on the individual regressions. The second determines the precision of the overall estimate, and the final one takes a matrix weighted average of the individual estimates to get the grand estimate. The example file is `SWAMY.PRF`. This (and the related mean groups estimator) are also provided by the procedures `@SWAMY` and `@MEANGROUP`.

```

cal(panelobs=20) 1935
all 10//1954:1
open data grunfeld.dat
data(format=prn,org=cols)

```

The `@SWAMY` procedure can be used to do the analysis below.

```

@swamy invest
# constant firmvalue cstock

```

Chapter 12: Cross Section/Panel Data

```
compute nindiv=10
compute ncoeff=3

dec vect[symm] xxmats(nindiv)
dec vect[vect] betas(nindiv)
dec vect      sigmas(nindiv)
dec symm delta(ncoeff,ncoeff)
dec symm xxgls(ncoeff,ncoeff)
dec vect betagls(ncoeff)
dec integer k
```

First pass, run OLS regressions over each individual. Save %XX, %BETA and %SEESQ. Also accumulate the sum of the coefficients and their outer product.

```
compute delta=%const(0.0)
compute betagls=%const(0.0)
do i=1,nindiv
  linreg(noprint) invest i//1 i//20
  # constant firmvalue cstock
  compute xxmats(i)=%xx
  compute betas(i)=%beta
  compute sigmas(i)=%seesq
  ewise delta(j,k)=delta(j,k)+%beta(j)*%beta(k)
  compute betagls=betagls+%beta
end do i
```

Estimate the covariance matrix of beta's

```
ewise delta(j,k)=1.0/(nindiv-1)*$
  (delta(j,k)-(1.0/nindiv)*betagls(j)*betagls(k))
```

Second pass. Compute the GLS covariance matrix. While we're at it, replace XXMATS with the precision.

```
compute xxgls=%const(0.0)
do i=1,nindiv
  compute xxmats(i)=inv(delta+sigmas(i)*xxmats(i))
  compute xxgls=xxgls+xxmats(i)
end do i
compute xxgls=inv(xxgls)
```

Final pass. Compute the grand coefficient matrix

```
compute betagls=%const(0.0)
do i=1,nindiv
  compute betagls=betagls+xxgls*xxmats(i)*betas(i)
end do i
linreg(create,coeffs=betagls,covmat=xxgls,form=chisquared) invest
# constant firmvalue cstock
```

Arellano-Bond GMM Estimator: ARELLANO.RPF example

The example file `ARELLANO.RPF` is based upon Example 11.3 from Wooldridge (2010). It estimates an autoregression on panel data (seven years of data for 90 counties) for the log of the crime rate. Because of the lagged dependent variable, a fixed effects estimator will have substantial bias with T being this small, as will the first difference estimator. The model is estimated using the GMM estimator of Arellano and Bond (1991). This is an IV estimator on the first differenced data using all available instruments for all potential lags.

```
cal(panelobs=7) 1981
all 90//1987:1
open data cornwell.prn
data(org=columns,format=prn) / county year crmrte
```

Transform to log first differences

```
set lcrmrte = log(crmrte)
diff lcrmrte / clcrmrte
```

The regression equation is 1st difference of the log crime rate on its lag. First differencing eliminates the individual county effect, but almost certainly produces serially correlated residuals, and thus, OLS would give inconsistent estimates because of the lagged dependent variable. The first set of estimates does instrumental variables using the second and third lags of the log crime rate as instruments. (The first lag still has a correlation with the residual). This is just run over one observation per individual (the final one) to avoid having to correct the covariance matrix for serial correlation.

```
instruments constant lcrmrte{2 3}
set smpl = %period(t)==1987:1
linreg(instruments,smpl=smpl) clcrmrte
# constant clcrmrte{1}
```

Check 1st stage regression

```
set cllag = clcrmrte{1}
linreg(smpl=smpl) cllag
# constant lcrmrte{2 3}
```

The lags tend to be rather weak instruments, and using a standard instrument setup will severely restrict the number of data points or instruments which could be used. A different approach is to create a separate instrument for each lag and for each time period and use GMM to weight them. This is the Arellano-Bond estimator. The instruments need to be constructed. This shows how to do this, however, you could more simply use the procedure `@ABLags`:

```
@ABLags lcrmrte abivs
```

Chapter 12: Cross Section/Panel Data

```
compute m=7
dec vect[series] abivs((m-2)*(m-1)/2)
compute fill=1
do period=m,3,-1
  do lag=period-1,2,-1
    set abivs(fill) = %if(%period(t)==period,clcrmte{lag},0.0)
    compute fill=fill+1
  end do lag
end do period
```

The overidentification test is included in the regression output.

```
instrument constant abivs
linreg(title="Arellano-Bond",$
  instruments,optimalweights,lwindow=panel) clcrmte
# constant clcrmte{1}
```

13. Other Models and Techniques

This chapter includes several topics which don't fit into other chapters. It includes non-parametric regression techniques, linear and quadratic programming, and neural network models.

Non-Parametric Regression and Density Estimation

Linear and Quadratic Programming

Portfolio Optimization

Neural Networks

13.1 Non-Parametric Regression and Density Estimation

The Tools

DENSITY does kernel density estimation. It can estimate the density function and, for the differentiable kernels, the derivative of the density.

NPREG is for the non-parametric regression of one series on another. Its design is quite similar to **DENSITY** for the kernel-based regressions (Nadaraya-Watson). There's also an option for LoWeSS (Locally WEighted Scatterplot Smoother), which isn't used as much in econometrics.

Both **NPREG** and **DENSITY** have a `GRID=INPUT/AUTOMATIC` option. When you're using them just to get a graph of the shape, the `GRID=AUTOMATIC` usually works fine. Use `GRID=INPUT` when you need either to better control the grid, or, as in example `NPREG.RPF` (next page), you need to evaluate at existing data in order to use the output as part of a more involved calculation. The "grid" doesn't really have to be a grid—there's no computational reason for it to be organized in any particular way.

Both instructions have Wizards on the Statistics menu, which can be rather handy since the instructions themselves are somewhat complicated. Note in particular that each wizard includes a "Create Graph" checkbox, which adds a **SCATTER** instruction to display the density or the x-y regression

The graph instructions will generally have to be edited somewhat to add a header or footer, but this still can be a time-saver if you want the graphical output.

Bandwidths

The default bandwidth on either instruction is:

$(.79 IQR)N^{-1/5}$, where *IQR*=interquartile range, *N*=number of observations

This has certain optimality properties in larger samples (see the discussion in Pagan and Ullah, 1999), but you might find it to be too small in many applications. You can use the `SMOOTHING` option to adjust this up or down. The bandwidth is multiplied by the value given on that option, so `SMOOTHING=1.0` (the default) gives the default bandwidth, while `SMOOTHING=2.0` doubles the default. The bandwidth used can be obtained after the instruction from the variable `%EBW`.

Examples

By far the most common use of **DENSITY** is graphing densities from simulations. These are usually quite straightforward, as you can let **DENSITY** set up the grid as is done here (`GINTER` is the grid and `FINTER` the corresponding estimated density).

```
density(smoothing=1.5) inter 1 ndraws ginter finter
scatter(style=lines,footer="Posterior for Intercept")
# ginter finter
```


The example file `GARCHSEMIPARAM.RPF` estimates a GARCH model using a non-parametric estimate for the density function of the standardized residuals (rather than using a conventional Normal or t). This code segment first estimates a GARCH(1,1) model, then the density function for the standardized residuals, using a very fine input grid running over $[-6,6]$. The computed density is graphed against the standard Normal density.

```
garch(p=1,q=1,hseries=h,resids=u) / xjpn
compute gstart=%regstart(),gend=%regend()
set ustandard gstart gend = u/sqrt(h)
@gridseries(from=-6.0,to=6.0,size=.001,pts=gpts) xgrid
density(type=gauss,grid=input) ustandard gstart gend xgrid fgrid
set ngrid 1 gpts = %density(xgrid)
scatter(style=line,key=upleft,klabels=||"Empirical","N(0,1)"||) 2
# xgrid fgrid
# xgrid ngrid
```

The example file `NPREG.RPF` is taken from Pagan and Ullah (1999, p. 248). This does a correction for heteroscedasticity using a non-parametric function of population. The code segment here first estimates the model by least squares, then does conventional weighted least squares using the square of population as the scedastic function.

NPREG is then used to get a non-parametric fit of the squared residuals on population, creating `VPOPNP` as the fitted estimate. Since the actual data series `POP` was used as the *grid*, `VPOPNP` will also align with the data, so it can be used directly in **LINREG** as the `SPREAD` option.

```
linreg exptrav / resids
# constant income
set popsq = pop^2
linreg(spread=popsq) exptrav
set ressq = resids^2
npreg(grid=input,type=gaussian) ressq pop / pop vpopnp
linreg(spread=vpopnp,$
    title="Semiparametric Weighted Least Squares") exptrav
# constant income
```

Chapter 13: Other Techniques

Example program `ADAPTIVE.RPF` is also taken from Pagan and Ullah (1999, p. 250). It does an adaptive kernel estimator. For the regression equation

$$(1) \quad y_t = X_t\beta + u_t$$

if the density of u is the (unknown) f , the derivative of the log likelihood with respect to β is

$$(2) \quad -\sum X_t (f'(u_t)/f(u_t)) \equiv -\sum X_t \psi_t$$

The adaptive kernel estimator is a two-step estimator which starts with OLS and uses kernel estimates of

$$(3) \quad \psi_t = f'(u_t)/f(u_t)$$

Because we need to compute the density at the given data points, this uses `GRID=INPUT` with the `RESIDS` providing the evaluation points.

```
linreg(robusterrors) price / resids
# sqft yard pool lajolla baths firepl irreg sprink view lsqft $
lyard constant
density(type=gauss,derives=f1,grid=input) resids / resids fu
set psi = f1/fu
```

The second step is to compute the change in β that would push (2) towards zero. A “method of scoring” step would take this step as (minus) the information matrix times the gradient. Assuming that u and X are independent, the information matrix can be estimated as

$$(4) \quad (1/T) \left(\sum \psi_t^2 \right) \left(\sum X_t' X_t \right)$$

which can be computed using **MCOV** with the `NOZUDEP` option:

```
mcov(lastreg,matrix=ibxx,nozudep) / psi
```

If you're not willing to assume that, you could use a BHHH estimate of

$$(5) \quad \sum X_t' \psi_t^2 X_t$$

This is computable using **MCOV** with `ZUDEP`, and is needed at any rate to estimate the covariance matrix, as it's the center of the "sandwich". This also computes the gradient, which is T times the mean vector of the products of `PSI` with the regressors. **LINREG** with `CREATE` is then used to display the regression output with the adjusted beta and sandwich estimator of the covariance matrix.

```
mcov(lastreg,matrix=ib,meanvector=mv) / psi
compute d=mv*%nobs
linreg(create,lastreg,form=chisquared,$
title="Adaptive Kernel Estimator",$
coeffs=%beta-inv(ibxx)*d,covmat=inv(ibxx)*ib*inv(ibxx))
```

13.2 Linear and Quadratic Programming

The instruction **LQPROG** provides quick and efficient solutions to linear and quadratic programming problems.

Constraints

LQPROG minimizes either a linear or quadratic function in $\mathbf{x}$ subject to

$$(6) \quad \mathbf{A}_e \mathbf{x} = \mathbf{b}_e, \quad \mathbf{A}_l \mathbf{x} \leq \mathbf{b}_l, \quad \mathbf{A}_g \mathbf{x} \geq \mathbf{b}_g$$

$$(7) \quad \mathbf{x}_i \geq 0 \text{ for all } i=1,\dots,nneg$$

where, by design, $\mathbf{b}_e \geq 0$, $\mathbf{b}_l \geq 0$, $\mathbf{b}_g \geq 0$. A constraint with a naturally negative value of $\mathbf{b}$ needs to be sign-flipped.

The constraints are input using matrices $\mathbf{A}$ and $\mathbf{b}$ constructed as follows:

$$(8) \quad \mathbf{A} = \begin{bmatrix} \mathbf{A}_e \\ \mathbf{A}_l \\ \mathbf{A}_g \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \mathbf{b}_e \\ \mathbf{b}_l \\ \mathbf{b}_g \end{bmatrix}$$

In other words, equality constraints must be listed first, followed by those inequality constraints which are $\leq$ a non-negative constant, and finally those which are written as $\geq$ a non-negative constant. You input the constraints to **LQPROG** using the options *A=A matrix* and *B=B vector*. You have to indicate the number of equality constraints with the *EQUAL=number of equalities*, and the number of $\geq$ constraints with *GE=number of non-negativity constraints*. The default for each of those options is 0, so all the input constraints are $\leq$ by default.

For linear programming, all the $\mathbf{x}$'s are constrained to be non-negative, that is, *nneg* in (7) is the number of variables. By contrast, a quadratic program can handle unbounded arguments since a quadratic form (with positive definite matrix) is unbounded above at large values in all directions. You have to arrange the elements of $\mathbf{x}$ so the ones which are forced non-negative come first. The option *NNEG=number of non-negative x's* can be used if there are some that are constrained to be non-negative and some unbounded. By default, *all* are constrained to be non-negative.

Chapter 13: Other Techniques

Linear Programming

LQPROG can solve linear programming problems of the form:

$$(9) \quad \text{minimize} \quad \mathbf{c}'\mathbf{x}$$

subject to the constraints described on the previous page. You input the cost vector with option `C=C vector`.

In this example, a firm is trying to allocate its advertising budget between magazine and television. Television costs more, but reaches a larger audience. The magazine readership, while smaller, has superior demographics. They want to allocate their budget in the most cost-effective manner given goals of meeting or exceeding total exposures and exposures for subgroups.

$$(10) \quad \begin{array}{ll} \text{minimize} & 40m + 200t \\ \text{subject to} & 4m + 40t \geq 160 \quad \text{total exposures} \\ & 3m + 10t \geq 60 \quad \text{high income} \\ & 8m + 10t \geq 80 \quad \text{age group} \\ & m, t \geq 0 \end{array}$$

The input matrices for this problem are

$$(11) \quad \mathbf{A} = [\mathbf{A}_g] = \begin{bmatrix} 4 & 40 \\ 3 & 10 \\ 8 & 10 \end{bmatrix}, \quad \mathbf{b} = [\mathbf{b}_g] = \begin{bmatrix} 160 \\ 60 \\ 80 \end{bmatrix}, \quad \mathbf{c} = [40 \quad 200]$$

The input constraints are all $\geq$, so we need the output `GE=3`. We can solve the problem with the following program:

```
dec rect a
dec vect b
compute a=||4.0,40.0|3.0,10.0|8.0,10.0||
compute b=||160.0,60.0,80.0||

lqprog (ge=3,c=||40.0,200.0||,a=a,b=b) x
```

The output is

```
In LPROG, minimum = 1000.00000000
Solution x =
10.000000000 3.000000000

Lagrange Multipliers
2.500000000 10.000000000 0.000000000
```

Quadratic Programming

LQPROG solves quadratic programming problems of the form:

$$(12) \text{ minimize } \frac{1}{2} \mathbf{x}' \mathbf{Q} \mathbf{x} + \mathbf{c}' \mathbf{x}$$

subject to the constraints described earlier. As noted there, you can have components of $\mathbf{x}$ which are allowed to be negative, but you must arrange the input matrices so that the ones subject to the non-negativity constraint come first. The Hessian matrix of the objective function, $\mathbf{Q}$, is an $nvar \times nvar$ symmetric matrix. The objective function is input to **LQPROG** using the options $Q=Q \text{ matrix}$ and $C=C \text{ vector}$.

For quadratic problems, **LQPROG** uses the active set method with the conjugate gradient method (see Luenberger, 1989). An initial feasible solution is obtained by employing linear programming with artificial variables.

Note that you might have some work to do in transforming a problem to the form (12). For instance, suppose that your objective function is

$$(13) \quad x_1^2 + x_2^2 + x_3^2 + x_1 x_3 - 10x_1 - 4x_3$$

To convert a function like this which is written out as a second order polynomial in $x_1, x_2, \dots, x_n$, call the second order terms $S_{ij}x_i x_j$. Then the elements for $\mathbf{Q}$ are:

$$(14) \quad \mathbf{Q}(i,j) = \begin{cases} 2S_{ij} & \text{for } i = j \quad (\text{diagonal elements}) \\ S_{ij} & \text{for } i \neq j \quad (\text{off-diagonal elements}) \end{cases}$$

The diagonal elements of $\mathbf{Q}$ are twice the quadratic coefficients on the “squared” terms, so for instance $\mathbf{Q}(1,1)$ should be twice the coefficient on x_1^2 . The off-diagonal elements of $\mathbf{Q}$ are set equal to coefficients on the corresponding quadratic term: $\mathbf{Q}(1,2)$ would be the coefficient on $x_1 x_2$. For (13), the matrices in the objective function are:

$$(15) \quad \mathbf{Q} = \begin{bmatrix} 2 & 0 & 1 \\ 0 & 2 & 0 \\ 1 & 0 & 2 \end{bmatrix}, \quad \mathbf{c}' = [-10, 0, -4]$$

Chapter 13: Other Techniques

Examples

The following (from example `QPROG.RPF`) estimates a regression subject to non-negativity constraints and an adding up constraint. In matrix form, the sum of squared residuals can be written

$$(16) \quad (\mathbf{y} - \mathbf{X}\beta)' (\mathbf{y} - \mathbf{X}\beta) = \beta' \mathbf{X}' \mathbf{X} \beta - 2\mathbf{y}' \mathbf{X} \beta + \mathbf{y}' \mathbf{y}$$

The last term doesn't involve β and so can be ignored in finding the optimal coefficients. If we multiply this by 1/2 we can read off $\mathbf{Q} = \mathbf{X}' \mathbf{X}$ and $\mathbf{c} = -\mathbf{X}' \mathbf{y}$. For illustration, we will add to this the constraint that $\mathbf{1} \bullet \beta \leq 1$. $\mathbf{X}' \mathbf{X}$ and $\mathbf{X}' \mathbf{y}$ can be obtained as submatrices of the cross-moment matrix of the regressors and $\mathbf{y}$.

```
compute [symmetric] q = %xsubmat(%cmom,1,13,1,13)
compute [vector]    c = -1.0*%xsubmat(%cmom,14,14,1,13)

compute a=%fill(1,13,1.0)
lqprog(c=c,q=q,a=a,b=1.0) x
```

`LQPROG` is the ideal tool for solving portfolio optimization problems. If you need to solve these subject to a restriction against taking short positions, there is no simple alternative.

This is from example file `PORTFOLIO.RPF`. With the covariance matrix of the returns in `OMEGA` and the means in `MU`, the following uses `LQPROG` to compute the minimum variable portfolio, and uses the weights to compute the mean. If you want to allow short positions, add the `NNEG=0` option to `LQPROG`.

```
compute [rect] ones=%fill(1,n,1.0)
lqprog(q=omega,a=ones,b=1.0,equalities=1) x
compute minr=%dot(x,mu)
```

This part finds the minimum variance portfolio with the different expected returns (across the grid in `ERETS`). Note—the problem being solved is to $\min (1/2)\mathbf{x}'\mathbf{Q}\mathbf{x}$ subject to the constraints. To get the variance itself, we need to multiply the optimized value by 2. (What's actually saved and later graphed is the standard deviation). The `LQPROG` now has two equality constraints: weights sum to one, and the weights times means sum to the target return. The `~~` operator is used to "glue" together the constraint matrices.

```
set erets 1 101 = minr+(maxr-minr)*.01*(t-1)
set srets 1 101 = 0.0
do t=1,101
    lqprog(noprint,q=omega,a=ones~~tr(mu),b=1.0~~erets(t),equal=2)
    compute srets(t)=sqrt(2.0*%funcval)
end do t
```

13.3 Neural Networks

Artificial neural network models provide a powerful alternative to standard regression techniques for producing time-series and cross-sectional models. Neural networks are particularly useful for handling complex, non-linear univariate and multivariate relationships that would be difficult to fit using other techniques.

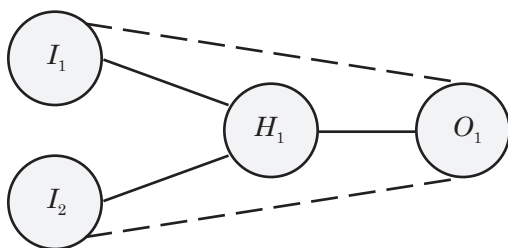
Despite the fact that neural networks are simply a class of flexible non-linear functional forms, there has arisen a whole different set of terminology to describe their structure and the fitting process. While we present the standard neural network terms, we try also to provide a translation into a standard regression framework.

Overview of Neural Networks

A neural network consists of an *input layer* containing one or more input *nodes* (in effect, the explanatory variables in a regression model), and an *output layer* of one or more output nodes (analogous to the dependent variable(s)). They normally also include a *hidden layer*, which lies between the input and output layers and consists of one or more hidden nodes.

Each input node is connected to each hidden node, and, in turn, each hidden node is connected to each output node. The model may also include direct connections between each input node and each output node. These can be used in addition to, or instead of, the hidden layer connections.

The diagram below represents a simple neural network with two input nodes (I_1 and I_2), one hidden node (H_1), and one output node (O_1). The solid lines represent the connections between the input nodes and the hidden node, and between the hidden node and the output node. The dashed lines represent direct connections between the input nodes and the output node.



The hidden nodes and output nodes each have sets of weighting values associated with them. These serve the same purpose as the coefficients of a regression model. Given a set of input values, the weights determine the output(s) of the network.

In particular, each hidden node has a *bias weight* (intercept, in effect), plus separate *input weights* for each input node attached to it. In the example above, H_1 would have three weights: a bias weight (which we'll call h_{10}), the input weight associated with input I_1 (h_{11}), and the input weight for I_2 (h_{12}). Given the input values I_1 and I_2 , the value of this hidden node would be given by:

$$u_1 = h_{10} + I_1 h_{11} + I_2 h_{12}$$

The actual output of the hidden node is determined by applying a “squashing” function to the value u , which scales the output to range between zero and one.

Output nodes have a similar set of weights. For the example above, we would have o_{10} (the bias weight on output node 1), o_{11} (the weight of output node 1 for hidden node 1), d_{11} (weight of output node 1 on the connection with input node 1), and d_{12} (weight of output node 1 on the connection with input node 2). If we use a logistic squashing function for the hidden node, the output of this model would be:

$$s_1 = 1 / (1 + e^{-u_1})$$

$$O_1 = o_{01} + o_{11} s_1 + d_{11} I_1 + d_{12} I_2$$

Fitting a neural network model involves “training” the model by supplying sets of known input and output values, and allowing the neural network algorithm to adjust the hidden node and output node weights until the output produced by the network matches the actual output in the training sample to the desired degree of accuracy.

Once trained, the model can then be used to generate new output data (fitted values or forecasts) from other sets of inputs. Assuming the fit is good, and that the relationships represented by the sample input and output data generalize to other samples, the model can produce good predictions.

However, care must be taken to avoid “overfitting” the model. This occurs when a network is trained to such close tolerances that it ends up modeling the “noise” in the training sample, not just the “signal.” Such a network will generally do a poor job of predicting appropriate output values for other sets of inputs. See below for details.

The NNLEARN and NNTEST instructions

The **NNLEARN** instruction is used to define a new neural network model, and to do additional training on an existing model. The **NNTEST** instruction is used to generate output (that is, fitted values) from a set of inputs, using a network model estimated using **NNLEARN**. For simple models, you may only need to use **NNLEARN** once to fit the network. For more complex models, you will often use **NNLEARN** and **NNTEST** in an iterative process, first training the model using **NNLEARN**, using **NNTEST** to generate output for comparison with training or validation samples, doing additional training as needed with **NNLEARN**, and so on.

SERIES variables are used for both the inputs and outputs on **NNLEARN** and **NNTEST**. This puts all of RATS’ data handling and transformation capabilities at your disposal in preparing data for neural network modeling, and for handling output. For example, you can: use the standard sample-range capabilities to train and test models using different samples; use standard lag or lead notation on the inputs for constructing time-series models; and graph fitted values against actual output.

The Backpropagation Algorithm

NNLEARN uses backpropagation techniques with an adaptive learning rate algorithm to train the model to a user-specified level of convergence. Translation: this is similar to steepest descent except that the derivatives for each weight are adjusted separately based upon the history of recent iterations. If you have more than one hidden node, the basic model isn't identified in the standard statistical sense. Starting from randomized initial conditions, backpropagation does a lot of searching back and forth to find features for each of the nodes to pick up. This takes a lot of calculation, but allows the model to fit quite complex surfaces.

Because the entire process of fitting a neural network is so different from standard hill-climbing procedures, the options for controlling the process and determining convergence are different as well. For instance, there is an **ITERATIONS** option, but the required limit on this in most applications is quite large. The 100 or so that are usually the most required for a hill-climbing procedure will be far too small. **NNLEARN** does less calculation per "iteration" but also accomplishes less in terms of improving the fit (and may very well accomplish nothing). You might need tens or even hundreds of thousands of iterations to properly train a complex network.

Using NNLEARN

To create a neural net model, run the **NNLEARN** instruction with a set of input and output data, using the **SAVE** option to save the generated weights in a *memory vector*.

To do additional training on the same model, just do another **NNLEARN** instruction using the same memory vector on the **SAVE** option. You can either train on the same data set (perhaps setting a tighter convergence criterion or simply allowing more iterations if the desired criterion hasn't been met), or you can train the model using other training samples (different entry ranges of the same input and output series, or entirely different input/output series). **NNLEARN** will use the existing values stored in the memory vector as starting values for the estimation. The new weights computed by **NNLEARN** will be saved back into the memory vector, replacing the previous values.

You may want to copy the contents of the memory vector to another array after each training step (with an instruction of the form **COMPUTE array = memory vector**). This allows you to go back to an earlier model if you find that subsequent training has resulted in overfitting. It also makes it easier to compare the outputs generated by the model at various stages of training.

You can also save memory vectors to disk for use in a later session. Use **OPEN COPY** and **WRITE (UNIT=COPY)** to write the vector to disk, and **DECLARE VECTOR, OPEN DATA** and **READ (VARYING)** to read it back into memory.

Tips on Fitting Neural Network Models

As noted above, a potential pitfall of neural network models is that they are prone to “overfitting” the data. The goal in creating a neural network is to model accurately the underlying structural relationship between your input data and output data. If you use too many hidden nodes, or allow the model to train too long (if you use too tight a convergence criterion), your neural network may overfit the data, meaning that in addition to fitting the underlying signal, the network also models the noise in the training sample. Although an overfitted model will do an excellent job of modeling a particular training sample, it will often do a very poor job of modelling the general behavior of the process.

There are several ways to avoid overfitting. One is to make sure that you use only the minimum required number of hidden nodes. However, finding the optimal number of hidden nodes is often very difficult. If you end up using too few hidden nodes, the network will be unable to produce a good fit.

The other is to stop the training process before overfitting occurs. We would recommend that you use this method, because it is usually much easier to stop the training at the appropriate point than to try and determine the optimal number of hidden nodes. Also, this approach allows you to err on the side of having too many hidden nodes, rather than too few.

As suggested above, this is an iterative process, with the following basic steps:

- 1) Create and train the model using **NNLEARN**, with a fairly loose convergence criterion, and/or a relatively low limit on the number of iterations. Be sure to use the **SAVE** option to save the memory vector
- 2) Use a **COMPUTE** instruction to copy the current values of the memory vector to another array. Use a different name each time you do this step. If you find that you have overtrained the model, just go back to the memory vector saved previously and either use it as is, or do additional training (but to a looser criterion than the one that produced the overfitted model), or using a different set of training data.
- 3) Generate fitted values from the model using **NNTEST** with the estimated memory vector. Use graphs or **PRINT** instructions to compare the fitted and actual values. If you have a “validation” sample (a set of input and output values excluded from the training sample to be used to test the quality of the fit), use the validation sample inputs, and compare the resulting output to the validation sample outputs (by looking at the sum of squared errors, graph the validation and fitted values, etc.).
- 4) Repeat these steps until the desired fit is achieved. You can do additional training using the same set of sample inputs and outputs with a tighter convergence criterion (or higher iteration limit), or you can use additional training samples. Be sure to use the same memory vector on the **SAVE** option each time so the model starts up where it left off. However, by saving each set of weights into a

different array in step (2), you'll be able to go back to previous states if you find that the model eventually trains to the point of being overfit.

You are looking for the level of training that produces the minimum error when comparing network output to a separate validation sample. As the model fit improves with training, the errors should decrease steadily. At some point, however, the errors may begin to rise again, indicating that the model is probably starting to overfit the training sample (and thus providing a poorer fit for the validation sample).

We recommend that you always use the `TRACE` option to track the progress of the estimation. Some models will require tens of thousands or hundreds of thousands of epochs to produce a good fit. You may also find that **NNLEARN** sometimes reaches “plateaus” where the mean square error doesn't change much for several thousand iterations, and then makes a significant improvement.

Also, be aware that the initial values used for the internal weights of a new network are determined randomly (if you aren't using `SAVE` to supply an existing model and weights), so results will differ slightly from run to run. You can use the **SEED** instruction to seed the random number generator if you want the results of a program to be exactly reproducible.

Finally, you may occasionally run into situations where **NNLEARN** has trouble getting started, perhaps doing many thousands of epochs without making any significant progress. In such cases, you might want to interrupt and re-start the **NNLEARN** instruction in the hopes that a different set of initial values will perform better. However, these cases are fairly rare. In general, you simply need to be patient.

Convergence

In order to construct a useful neural network model, you will need to train the model sufficiently so that it models accurately the underlying behavior of the data, but not so tightly that it “overfits” the data used to train the model. That is, you want to model the “signal” present in your data, but not the noise.

RATS provides three options for controlling the convergence of the model. In most cases, you will use these options in an iterative process that involves invoking **NNLEARN** several times for the same model. The options available are:

```
iters=iteration limit [no limit]
cvcrit=convergence criterion [.00001]
rsquared=minimum R-squared level
```

The **CVCRT** and **RSQUARED** options are mutually exclusive—they provide two ways of specifying the convergence criterion for the learning process. Both can produce equivalent fits, they simply offer different ways of thinking about the criterion. The default setting is **CVCRT**=.00001 (if you use both, RATS will take the **CVCRT** setting).

Chapter 13: Other Techniques

If you use the **CVCRT** option, RATS will train the model until the mean square error (the mean of the squared error between the output series and the current output values of the network) is less than the **CVCRT** value.

If you use the **RSQUARED** option, RATS will train the model until the mean square error is less than $(1 - R^2)\sigma^2$, where R^2 is the value specified in the **RSQUARED** option, and σ^2 is the smallest of the output series variances.

The main disadvantage of **CVCRT** is that it is dependent on the scale of the variables in the model. For example, suppose a **CVCRT** of .0001 produces a good fit for a particular model. If you took the same model, but multiplied the output series by a factor of 10000, this **CVCRT** setting would probably be much too tight.

The **RSQUARED** option is particularly handy for problems where you have a reasonable idea of what kind of “goodness of fit” you can expect from the model. Perhaps more importantly, the **RSQUARED** criteria is less dependent on the scale of the output because it is scaled by the output variance.

By default, **NNLEARN** will iterate until it satisfies the criteria set with **CVCRT** or **RSQUARED**. You can use **ITERS** to place an upper limit on the number of iterations that **NNLEARN** will perform—it will stop iterating after the specified number of iterations, even if the criteria has not yet been met.

Padding the Output Range

The individual inputs are all rescaled internally to a range of [0,1]. In most cases, this will ensure that the coefficients are all within a few orders of magnitude of 1, which makes it easier to fit by backpropagation. Just as with any type of model that is fit to data, if you try to forecast values using explanatory variables that are far removed from those used in fitting (training) it, the results may be unreliable.

A more serious problem with neural nets, though, comes from the restriction on the range of the output variables. The rescaling of the inputs is done for convenience: since each has a freely estimated multiplicative coefficient wherever it appears, any scale choice can be “undone” by doing the opposite rescaling on the coefficients. However, the rescaling of the outputs is mandatory because of the use of the “squashing” function. If the output values are also rescaled to [0,1], then the network’s outputs will be constrained to the range from the training sample, since the squashing function can’t produce a value larger than 1. This will be a problem if you’re attempting to forecast a trending variable. To avoid this problem, you can use the option **PAD=fraction to pad**. This provides a value between 0 and 1 which indicates the fraction of “padding” to include when rescaling the output variables.

If, for instance, you choose **PAD=.2**, the smallest output value in the training sample will be mapped to .1 while the largest will be mapped to .9. If the original range of the data were from 7.2 to 8, this would allow the network to produce forecasts up to 8.1 and down to 7.1.

Examples

This example fits a neural network to the function:

$$y_t = \sin(20x_t) \text{ where } x_t = .01, .02, \dots, 1.00$$

You can experiment with this example to see the effects of changing the number of hidden nodes and/or the convergence criterion. We've used the CVCRIT option in this example—if you use RSQUARED, try a setting of about 0.8:

```
all 100
set input = t/100.
set sinewave = sin(input*20)
nnlearn(hidden=6,save=memvec,trace,cvcrit=.01,ymax=1.0,ymin=-1.0)
# input
# sinewave
nnctest / memvec
# input
# output
graph(key=below) 2
# sinewave
# output
```

This is an example from Tsay (2005, pp. 180–181). It fits a model with two hidden nodes and direct connection from inputs to the returns on IBM stock, using three lags as inputs. The sample through 1997:12 is used for training, while the sample from 1998 on is forecast.

```
nnlearn(rsquared=.10,itters=100000,hidden=2,direct,save=nnmodel) *
1997:12
# ibmln{1 2 3}
# ibmln

nnctest 1998:1 1999:12 nnmodel
# ibmln{1 2 3}
# nnfore
@uforeerrors ibmln nnfore
```

Neural Network: NEURAL.RPF example

Example `NEURAL.RPF` fits a neural net to a binary choice model. Like a probit model (also estimated here), the neural net attempts to explain the `YESVM` data given the characteristics of the individuals. Aside from a different functional form, the neural net model also differs by using the sum of squared errors rather than the likelihood as a criterion function.

Linear probability model with "fitted" values. Because the LPM doesn't constrain the fitted values to the $[0,1]$ range, some of them may be (and are) outside that.

```
linreg yesvm
# constant public1_2 public3_4 public5 private years teacher $
  loginc logproptax
prj lpmfitted
```

Probit model. Compute the fitted probabilities.

```
ddv(dist=probit) yesvm
# constant public1_2 public3_4 public5 private years teacher $
  loginc logproptax
prj (distr=probit,cdf=prfitted)
```

Neural network. We use two hidden nodes and one direct. (Two hidden nodes alone can't cover the space of values well enough). Note that the `CONSTANT` isn't included in the explanatory variables, since it's automatically included.

```
nnlearn(hidden=2,direct=1,itors=10000,save=nnmeth)
# public1_2 public3_4 public5 private years teacher $
  loginc logproptax
# yesvm
```

Compute the forecast values from the network.

```
nntest / nnmeth
# public1_2 public3_4 public5 private years teacher $
  loginc logproptax
# testvm
```

Compute the number of correct predictions for the various models and display them in a `REPORT`.

```
sstat(smpl=yesvm==0) / 1>>nos testvm<.5>>nnnos $
  lpmfitted<.5>>lpmnos prfitted<.5>>prbnos
sstat(smpl=yesvm==1) / 1>>yes testvm>.5>>nyyes $
  lpmfitted>.5>>lpmyes prfitted>.5>>prbyes
*
report(action=define,$
  hlabels=||"Vote","Actual","Neural Net","LPM","Probit"||)
report(atcol=1) "No" nos nnnos lpmnos prbnos
report(atcol=1) "Yes" yes nnyes lpmyes prbyes
report(action=show)
```

1. Spectral Analysis

RATS is one of the few econometric software programs which includes spectral analysis. We consider it to be an important tool in time-series analysis.

With RATS, you do spectral analysis using a sequence of primitive instructions such as **FFT** for Fourier transform and **WINDOW** for spectral window smoothing. This permits a great deal of flexibility, and ensures that you know exactly what is happening. On the other hand, the proper use of the instructions demands a somewhat higher level of sophistication on your part than from some other programs.

However, we have put together procedures which handle many of the key applications. These operate much like standard RATS instructions. You can rely upon them if you are unsure of your skills.

While there are some uses for direct spectral and cross-spectral analysis, the most important use of spectral methods is as a computational device. Many techniques are actually applications of the frequency domain filtering (Section 14.8).

Complex Data

Fourier Transforms

Spectra and Cross Spectra

Filtering

Forecasting

Fractional Integration

14.1 Complex-Valued Expressions

Mixing Modes

Complex is a more general type than real or integer. If needed, RATS will automatically convert an integer or a real to complex (with zero imaginary part). The explicit type conversion functions are:

<code>%real(z)</code>	returns the (real-valued) real part of z
<code>%imag(z)</code>	returns the (real-valued) imaginary part of z
<code>%cmplx(x1,x2)</code>	creates the complex $x1 + ix2$ from the real numbers/expressions $x1$ and $x2$.

Using Operators with Complex Values

The following operators can be used in complex-value calculations:

- $+$, $-$ and $/$ function as expected.
- $*$ multiplies the first number by the *conjugate* of the second number, that is, $A * B = A\bar{B}$.
- $^$ or $**$ function as expected. The only operation *not* supported is raising a negative real number (with zero imaginary part) to a power.

Relational Operators

You can use the $==$ and $<>$ operators to test equality or inequality of two complex numbers. You cannot use any of the other relational operators ($>$, $<$, $>=$, $<=$).

Predefined Variables

`%pi` is the constant π (REAL variable)

Functions

The functions of particular interest are (x 's are real, z 's are complex, t 's are integer, N is the number of frequency ordinates):

<code>sin(x), cos(x), tan(x)</code>	Sine, cosine, tangent (real-valued)
<code>%asin(x), %acos(x), %atan(x)</code>	Inverse sine, cosine, tangent
<code>%cmplx(x1,x2)</code>	Complex number $x1 + ix2$
<code>%real(z), %imag(z)</code>	Real and imaginary parts (real-valued)
<code>%conjg(z), %csqrt(z)</code>	Complex conjugate and complex square root
<code>%cabs(z), %arg(z)</code>	Absolute value and argument. z can be decomposed as $\%CABS(z) * EXP(i\%ARG(z))$
<code>%clog(z), %cexp(z)</code>	Complex natural logarithm and antilogarithm
<code>%unit(x), %unit2(t1,t2)</code>	Unit circle values e^{ix} and $e^{i2\pi(t_1-1)/t_2}$
<code>%zlag(t1,t2)</code>	Unit circle value $e^{i2\pi t_2(t_1-1)/N}$

14.2 Complex Series

Complex Series vs Real Series

Complex series and real series are separate types. You can't substitute one where RATS expects the other. While there are many similarities in structure between the two types, there is *little* similarity in their use. This section describes the differences.

The data type name of a complex series is `SERIES [COMPLEX]`, or `CSERIES` for short.

Creating Series

As with real series, you can create complex series by name when needed. However, it usually is more convenient to set up the full block of series needed (using **FREQUENCY**) and to refer to the series by numbers rather than by names. There are several reasons for this:

- Most frequency domain procedures are short and straightforward and are adapted easily for use with different data sets. Referring to series by number simplifies this process.
- A single series usually is subjected to several transformations in succession. In most instances, you can keep transforming the series onto itself. Giving a single name to the series would be misleading.

These points will become more obvious as you look at the examples.

CSET Used Less Often

You will note a similarity between the frequency domain instructions and the time domain instructions. However, a different set of instructions is important. While there is an instruction **CSET** (the complex analog of **SET**), specialized transformation instructions, such as **CMULTIPLY** and **CLN** (complex log) are used much more often.

FREQUENCY Instruction

The **FREQUENCY** instruction sets up a block of complex series of a specific length. Each use of **FREQUENCY** eliminates the previous block of complex series. This makes it possible (and desirable) to write self-contained procedures for your analysis, as each procedure can have its own **FREQUENCY** instruction to meet its needs.

FREQUENCY also defines a reserved variable `%Z`, which is a `RECTANGULAR [COMPLEX]` array with dimensions `length x cseries` that holds the complex series set up by **FREQUENCY**. You can access entry *m* of complex series *n* as `%Z (m, n)`.

No “CSMPL” Instruction

There is no “CSMPL” instruction corresponding to **SMPL**. Since most series are defined over the full **FREQUENCY** range and most transformations use all entries, the use of the defined range as the default is almost always adequate.

14.3 Complex-Valued Matrices and FRML's

Data Types

The two main types of matrices of complex numbers are the `VECTOR[COMPLEX]` (or `CVECTOR` for short) for a one-dimensional array and `RECT[COMPLEX]` (`CRECT`) for a general two-dimensional array. There's also a `SYMMETRIC[COMPLEX]`, but symmetric arrays aren't as useful as the real-valued version. (The true analogue of the symmetric real array is a conjugate symmetric or self-adjoint matrix).

You declare these with instructions like **DECLARE**, **LOCAL**, **TYPE** and **OPTION** just as you would real-valued arrays. And you can dimension them with **DIMENSION**, or on **DECLARE** or **LOCAL**.

You can define series of complex matrices: `SERIES[CRECT]`, for instance, is a series of complex 2-dimensional arrays. And you can define FRML's which evaluate to complex numbers or complex matrices with the types `FRML[COMPLEX]` (formula evaluating to complex numbers), `FRML[CVECTOR]` and `FRML[CRECT]`.

Operators and Functions

When applied to complex *matrices*, the operators `+`, `-` and `*` have the expected meanings. Note that `*` is just a standard matrix multiply. You can use `*` to multiply complex matrices by real-valued matrices or real or complex scalars as well.

There's a much more limited set of functions for complex matrices:

<code>%cxinv(Z)</code>	Matrix inverse
<code>%cxadj(Z)</code>	Adjoint (conjugate transpose)
<code>%cxsvd(Z)</code>	Singular value decomposition
<code>%cxeigdecomp(Z)</code>	Eigen decomposition

`%CXSVD` and `%CXEIGDECOMP` each create a `VECT[CRECT]` with the components of the decomposition.

Most calculations with complex-valued arrays are done with **COMPUTE** or with **GSET**. The latter is used when working with series of complex arrays, when you have one for each frequency.

Complex Eigenvalues/vectors of Real-valued Matrices

The instructions **EIGEN** (for eigen decomposition) and **QZ** (for generalized Schur decomposition) can both create complex vectors for their eigenvalues. For both instructions, you use the option `CVALUES=VECTOR[COMPLEX] for eigenvalues`. With **EIGEN**, you can also get the complex matrix for the (column) eigenvectors with the `CVECTORS=RECT[COMPLEX] of eigenvectors`.

14.4 Fourier Transforms

Description

The Finite Fourier transform of the series $\{X(t), t = 1, \dots, T\}$ is

$$\tilde{X}(2\pi j/T) = \sum_{t=1}^T X(t) \exp(-2\pi i j(t-1)/T)$$

The frequencies run from 0 to $2\pi(T-1)/T$ by increments of $2\pi/T$.

Note that this is sometimes defined using a $1/\sqrt{T}$ multiplier (and $1/\sqrt{T}$ also on the inverse transform). We have found that the formula above (with a $1/T$ multiplier on the inverse) is more convenient as a different scale than $1/\sqrt{T}$ ends up being needed. The instruction **FFT** computes the Fourier transform and **IFT** computes its inverse.

RATS uses a Fast Fourier transform algorithm which can transform series of any length, although it is much faster for lengths which are products of powers of two, three and five.

RATS also uses a general complex FT. You can apply it to any series, not just real-valued series.

Frequency and Entry Mappings

Different ways of mapping entries to frequencies and back are used in spectral analysis packages. We have chosen the above because our primary interest is in (one-sided) time series data. It means, however, that if you are interested in two-sided sequences (such as the covariogram, or a two-sided distributed lag), you have to do some rearranging.

The two-sided finite sequence

$$\{x_{-N}, x_{-N+1}, \dots, x_{-1}, x_0, x_1, \dots, x_N\}$$

is represented in this mapping as the one-sided series

$$\{x_0, x_1, \dots, x_N, x_{-N}, x_{-N+1}, \dots, x_{-1}\}$$

This is true whether you are inputting the sequence to **FFT** or getting output from **IFT**. For instance, if you compute a covariogram by inverse transforming the periodogram, lag 0 (the variance) will be in entry 1, lag 1 in entry 2, etc., and lag -1 (which is identical to lag 1) will be in the last entry, lag -2 in the second to last, etc. The procedure **@CSERIESSYMM** can be used to properly symmetrize a series. For instance, this sets up a triangular window by defining the front end, then symmetrizing:

```
cset 2 1 320 = %if(t<=k,1-(t-1)/k,0.0)
@cseriessymm 2
```

14.5 Preparation and Padding

Preparation of Data

Most data used in spectral analysis are prepared in the time domain and then copied to the frequency domain using the instruction **RTOC** (Real TO Complex). Frequency domain techniques are among the fussiest of all time series techniques concerning data preparation. Applying **FFT** and **IFT** in succession will reproduce *any* series, no matter how non-stationary, but almost any real work done in the frequency domain will require some calculations which are sensitive to improper differencing or de-trending.

There are two technical reasons for this:

- Finite Fourier methods treat the data (in effect) as being periodic with period T , where T is the number of data points. Thus, the point “before” entry 1 is entry T . Clearly, if a series has a trend, it can’t be reasonably represented in such a fashion.
- The statistical properties of spectral estimators break down when the series does not have a well-behaved spectral density.

Thus, whenever the method you are using requires you to take the spectrum of a series, you should make sure:

- the series is properly differenced (usually with **DIFFERENCE** or **SET**) or detrended.
- the series has a mean of zero, or close to it.

Padding

The *length* on the **FREQUENCY** instruction is usually chosen to be a convenient number of frequencies for Fourier analysis. The extra length beyond the actual number of data points is the *padding*. These extra entries are set to zero when you transfer data using **RTOC**. There are two considerations in choosing *length*:

- The exact seasonal frequencies are included if the number of frequencies is a multiple of the number of periods per year. For example, with quarterly data, make sure you use a number divisible by 4, with monthly, divisible by 12, with business day, divisible by 5.
- The Fourier transforms (instructions **FFT**, **IFT** and **TRFUNC**) are much faster for series lengths which have small prime factors (2,3 and 5) than for lengths which are products of large primes. For instance, applying **FFT** to length $1024 = 2^{10}$ is roughly 40 times faster than applying it to $1023 = 3 \times 11 \times 31$. Given the speed of modern computers, this is only a minor consideration if you are just applying the Fourier transform a few times. However, if the Fourier transform is part of a function evaluation for non-linear estimation, a timing gap like that can be quite significant.

You can use the function `%FREQSIZE(n)` to get a recommended size for n actual data points. It returns the smallest integer of the form $2^m S$ which is greater than or equal to n . (S is the number of periods per year).

For procedures that require filtering in the frequency domain (Section 14.8), the series must be padded, preferably to at least double length. To get a recommended length in that case, use `2*%FREQSIZE(n)`.

Missing values in a data set are replaced by zeros when they are copied to the frequency domain. Thus, you don't have to concern yourself about undefined elements at the ends of your series, since they will just become part of the padding.

RATS provides the instruction **TAPER** for applying a taper to a data set prior to Fourier transforming it. A taper scales the ends of the actual data (not the padded series) with a graduated series so that it merges smoothly with the zeros in the padding.

Example

```
linreg logx 1977:1 2018:7 resids
# constant trend
frequency 1 768                               Pad 497 monthly to 768=(12)(64)
rtoc
# resids
#      1
```

14.6 Getting Output

Tools

Use **CPRINT** to print complex series. You can also use **DISPLAY** or **WRITE** to print individual complex numbers or expressions. Other types of output (such as graphics) will require you to transfer data to real series (see below).

Reducing Ordinates

CPRINT can produce a vast amount of output when you have a long series or have padded it substantially. Since spectral densities and related functions are usually quite smooth, you lose little by printing just a subset of the frequencies. You can use the **INTERVAL** option on **CPRINT** to pick entries at a regular interval.

Also, because a spectral density is symmetric about 0 and a cross-spectral density is conjugate-symmetric, you really only need to look at frequencies 0 to π . In RATS, these are entries 1 to $(T/2) + 1$. If you cut the frequencies down to half in this way, use the option **LENGTH=T** on **CPRINT** so the entries will be labeled properly.

This cuts 288 ordinates by printing just half (to 144) then picking every 4th one.

```
cprint(lc=freq,length=288,interval=4) 1 144 3
```

Using GRAPH and SCATTER for Complex Series

Neither **GRAPH** nor **SCATTER** is designed to handle complex series directly, but you can use them to graph a real series which has been sent from the frequency domain using **CTOR**.

You can use either instruction to graph a spectral function. Each has minor advantages and disadvantages—we will demonstrate both.

GRAPH is good for quick views. The only drawback for this is that **GRAPH** will want to label the horizontal axis as if you were graphing a time series and not a function of frequencies. The option **NOTICKS** is handy to suppress the misleading labeling.

If you graph a spectral density estimate, the values will usually have a huge range. The peak values can easily be many orders of magnitude larger than most. Remember that this is a decomposition of the *variance*, so size differences get squared. In most cases it is more useful to graph the log of the spectrum rather than the spectrum. You can do this either by taking the log first, or by using the **LOG** option on **GRAPH**, such as **LOG=10**. For **SCATTER**, the analogous option is **VLOG=10**.

SCATTER is better for “production” graphs. It can do a better job of handling the “frequency” axis, though that comes at the cost of some additional instructions to create a series to represent the frequencies, of the form:

```
set frequencies = (t-1.0)/N
```

where “N” depends upon how much of the frequency range you’re graphing. The scaling doesn’t affect the appearance of the graph itself, just the labeling on the x axis.

Suppose you are graphing half the frequencies (thus 0 to π). If you use (T-1.0) divided by the number of frequencies, the x axis will run from 0 to 1.0 and is thus showing the fractions of π . A slight change to

```
set frequencies = 6.0*(t-1.0)/N
```

would map the x-axis to 0.0 to 6.0. The integer x-values would be multiples of $\pi/6$, which are the harmonics for monthly data. The ultimate in fancy labeling uses the XLABELS option. This allows you to specify full strings which are equally spaced across the x axis. Using, for instance,

```
XLABELS=| | "0", "p/6", "p/3", "p/2", "2p/3", "5p/6", "p" | |
```

will, when combined with the instruction

```
grparm(font="symbol") axislabels 12
```

give you labels of 0, $\pi/6$, $\pi/3$, etc.

The following example is taken from the SPECTRUM.SRC procedure file. HALF is half the number of frequencies. The spectral density is plotted on a log scale.

```
ctor 1 half  
# 2  
# spect  
set frequencies 1 half = (t-1.0)/half  
scatter(style=lines,header=header,$  
  hlabel="Fractions of Pi",vlog=10.0) 1  
# frequencies spect 1 half
```

14.7 Computing Spectra and Cross Spectra

Periodograms

The periodogram of a series is its squared Fourier transform. The cross-periodogram of two series is the product of one FT by the other's conjugate. You compute both of these using **FFT** followed by **CMULTIPLY**.

```
fft 1
fft 2
cmultiply(scale=1./(2*pi*points)) 1 1 / 3
cmultiply(scale=1./(2*pi*points)) 1 2 / 4
```

The correct scale factor is $1/(2\pi N)$ where N is the number of actual data points. We represent N above as **POINTS**. This might be a number or a variable previously computed.

The raw periodogram is used directly in the cumulated periodogram test (see **CACCUMULATE**), and can be inverse transformed to get the covariogram (see description of lag windows on the next page). However, usually it is smoothed to get a consistent estimate of the spectral density or cross-spectral density.

Smoothing

While the periodogram gives an *unbiased* estimate of the spectrum, it is *inconsistent*, as the variance of the estimators does not go to zero as the number of data points grows.

RATS provides the method of spectral window smoothing for producing consistent estimates. The alternative procedure is the lag window or weighted covariance method, described on the next page.

The instruction **WINDOW** carries out the smoothing process. This estimates the spectral (cross-spectral) density at an ordinate by taking a weighted average of the periodogram at neighboring ordinates. For a continuous spectral density, there are two requirements for this to produce consistency:

- The window width must go to infinity as the number of data points increases (ensuring that the variance goes to zero). In a practical sense, this means that a window which is too narrow will produce imprecise estimates.
- The window width must increase at a rate slower than the increase in the number of data points (ensuring that the bias goes to zero). Translation: a very wide window will flatten the peaks and troughs too much.

Continuing from above, the following smooths 3 and 4 with a flat window of width 13, producing series 5 and 6.

```
window(width=13) 3 / 5
window(width=13) 4 / 6
```


Tapers

The smoothing method, particularly with the default flat window, can produce some spurious ripples in the estimate due to “data window leakage.” The data set is, theoretically, just a piece of an infinite data set, and we cut it off sharply at both ends. Data window leakage is the result of Fourier transforming this sharp-edged window.

A taper reduces this by scaling the ends of the data so they merge smoothly with the zeros on either side. Apply the RATS instruction **TAPER** to the *unpadded* part of the series prior to the **FFT** instruction. Tapering has little effect when you use **WINDOW** with **TYPE=TENT**. When you apply a taper, you need to change the scale factor on **CMULTIPLY** to $1.0 / (2 * \%PI * \%SCALETAP)$, where **%SCALETAP** is computed by **TAPER**.

Lag Windows

Lag window or weighted covariance estimators were the favored method for computing spectral densities before the Fast Fourier Transform was developed. They are described in Koopmans (1974) on pages 325-6. A simple form of this is used in modern econometrics in the Newey-West covariance estimator—the triangular weighting scheme used in Newey-West is known as the Bartlett window in spectral analysis.

The periodogram and the covariogram are inverses of each other under Fourier transformation. The periodogram gives inconsistent estimates of the spectrum because it weights equally (in Fourier transformation) the covariances with a small number of lags, which are estimated with almost a full set of data, and those at high lags, which are computed with almost none. In lag window estimation, we drop a window over the covariogram which gives zero weight to the high lags. The requirement for consistency here is that the number of covariances given non-zero weight goes to infinity at a rate slower than the increase of the number of data points.

If you want to employ the lag window method, the simplest way to compute the covariogram is to inverse transform the periodogram. You must pad to at least double the number of data points. The “windowing” process is done by setting up a weight series with the desired form. Make sure to symmetrize it by making the end terms match up with the starting ones. This does a Bartlett window of 13 lags applied to a data set with 130 data points (padded to 320). Series 2 will be the estimate of the spectral density. We sneak the $1/2\pi$ factor in when we multiply 1 by 2.

```
compute k=13.0
frequency 2 320
rtoc
# x
# 1
fft 1
cmult(scale=1./130) 1 1
ift 1
cset 2 = %if(t<=k,1-(t-1)/k,0.0)
@cseriesymm 2
cset 2 = (1.0/(2*%pi))*%z(t,1)*%z(t,2)
fft 2
```

These three lines transform series 1 into the covariogram of the input series.

Chapter 14: Spectral Analysis

Cross-Spectral Statistics

The examples at the start of the section constructed an estimate of the cross-spectral density. Usually, we are interested not in the density itself (which is complex-valued) but statistics derived from it:

Phase lead	the fraction of a cycle by which one series leads (lags) the other at each frequency.
Coherence (squared)	the proportion of the variance of either series which can be explained (linearly) by the other, again frequency by frequency.

RATS provides the instruction **POLAR** for computing these statistics. It does a polar decomposition of a series. The phase can be obtained directly from the cross-spectral density, but the coherence requires that we normalize the density by the product of the square roots of the spectral densities of the series. Continuing the example:

```
cmultiply(scale=1./(2*pi*points)) 2 2 / 7          need spectrum of 2
window(width=13) 7 / 8
cset 7 = %z(t,7)/%csqrt(%z(t,6)*%z(t,8))
polar(periods) 7 / 8 9
```

Series 8 is the coherence, 9 is the phase lead of series 1 over 2 (since we **CMULTIPLY**ed in that order). The **PERIODS** option converts the phase lead from radians to periods. Note that the phase lead is almost meaningless (and very poorly estimated) when the coherence is small.

Procedures

The procedures **@SPECTRUM** and **@CROSSPEC** compute and (optionally) graph a spectral density, and coherence and phase, respectively.

```
@crosspec ( options )  series1 series2 start end
@spectrum ( options )  series start end
```

The series input to the procedures are REAL series. Most of the options are common to both:

```
ordinates=number of ordinates [depends upon length,seasonal]
window=[flat]/tent/quadratic
width=window width [depends upon ordinates]
taper=trapezoidal/cosine/[none] (@spectrum only)
wtaper=taper width as a fraction of length [.25] (@spectrum only)
[graph]/nograph
header=header for high-resolution graph
footer=footer for high-resolution graph
```

The following segment is taken from the example file `SPECTRUM.RPF`. The first of these computes the periodogram (unsmoothed) estimate, the second one uses a custom smoothing window of width 7—that requires 4 (relative) weight values for the center and three terms out.

```
@spectrum(footer="Periodogram of Wolfer Sunspot Numbers",$
  periodogram,nologscale) cspots
@spectrum(footer="Spectral Estimate of Wolfer Sunspot Numbers",$
  weights=| | 3.0,3.0,2.0,1.0 | | ,nologscale,spectrum=smoothed,$
  ordinates=128) cspots
```

`@ARMA Spectrum` produces a graph of the spectral density for an input ARMA model where the model is in the form of an `EQUATION`. This could be an equation defined by `BOXJENK` or `LINREG`, or it could be a test equation created using `EQUATION`:

```
@ARMA Spectrum ( options )      equation
```

with the options

```
ordinates=# of frequency ordinates used [512]
header=Header for high-resolution graph
footer=Footer for high-resolution graph
```

The `@SSMSpectrum` procedure returns the estimated multivariate spectrum from a state space model (Chapter 10) given by the `A` and `SW` options. It returns in its argument a series of complex matrices.

```
@SSMSpectrum ( options )      spectrum
```

where *spectrum* will be a `SERIES[CMATRIX]` created by the procedure with the estimated spectral density. 0 frequency will be in entry 1; in general, frequency $2\pi(t-1)/\text{ORDINATES}$ will be in entry t . In order to avoid problems with non-stationary models, the 0 frequency isn't calculated. The options are

```
a=transition matrix for the state space model
f=loadings from shocks to the states [identity]
sw=covariance matrix of the shocks in the state equation
```

These are the same options as would be used in a `DLM` instruction.

```
ordinates=# of frequency ordinates used for calculations [512] .
```

This will return values only out to frequency π , thus to entries $\text{ORDINATES}/2 + 1$.

```
components=VECT[INT] of components
```

Use this to indicate which state components (numbered in 1,...,# of states) that you actually want. For instance, `COMPONENTS=| | 1 | |` will return the estimate just for the first component.

14.8 Frequency Domain Filtering

Uses of Filtering

Many applications of spectral analysis use some form of frequency domain filtering:

- Fractional differencing and integration (Section 14.10) uses a filter which is exactly represented in the frequency domain, but only can be approximated in the time domain.
- Maximum likelihood in the frequency domain (`FRACTINT.RPF`, Section 14.10) relies upon frequency domain filtering to estimate time domain models.
- Sims' seasonal adjustment procedure uses a "filter" which can only be represented in the frequency domain. See the example file `FREQDESEASON.RPF`.
- Hannan's efficient estimator (`HANNAN.RPF`) does GLS for arbitrary (stationary) serial correlation structures by computing in the frequency domain a filter to whiten the regression residuals.
- The spectral forecasting procedure (Section 14.9) uses a transfer function computed in the frequency domain.

Technical Information

Filtering is a *convolution* operation, which can be represented easily in the frequency domain. If

$$(1) \quad y_t = \sum_{s=-\infty}^{\infty} a_s x_{t-s}, \text{ then}$$

$$(2) \quad \tilde{y}(\omega) = \tilde{a}(\omega) \tilde{x}(\omega)$$

that is, the complicated convolution operation (1) is converted into simple multiplication in the frequency domain. The filtering procedure is thus:

1. Apply **FFT** to x to get $\tilde{x}$.
2. Compute $\tilde{a}$. You can use the instruction **TRFUNC** if it is a standard finite filter, **CMASK** for a filter to shut out frequency bands, and **CSET** or some other combination of instructions for other filter types.
3. Multiply the two Fourier transforms. Remember that RATS conjugates the second item in a **CMULTIPLY** or ***** operation.
4. Inverse transform the product using **IFT**.

Frequency domain filtering is most useful when the filter a is theoretically infinite but has a relatively simple Fourier transform. If a has a small number of coefficients, direct calculation with **FILTER** or **SET** is quite a bit faster.

Wraparound Effect

Consider the output obtained by passing a series $y(t)$ through the filter $A(L)$. If you do this in the time domain using **FILTER**, there are entries on one or both ends which cannot be computed for lack of lags or leads of y . If you filter in the frequency domain, the effect is as if three copies of y were laid end to end. The filter A is applied to this new series, with the middle section extracted as the output.

$$\left| 1, 2, \dots, N-1, N \mid \underbrace{1, 2, \dots, N-1, N}_{A(L)y(N)} \mid 1, 2, \dots, N-1, N \right|$$

If we look at the entries which we cannot compute with **FILTER**, we see that they get computed in the frequency domain, but use data from both ends of the series. For instance, a filter which lags back to the non-existent entry 0 will take entry N in its place. This *wraparound effect* hits only those entries which we could not compute anyway. The other entries are unaffected.

However, the time domain filter equivalent to most important frequency domain filters is an *infinite* lag polynomial. Thus all entries will be affected by wraparound to some extent. To reduce the seriousness of this, the series should:

- have mean zero
- have no trend
- be well-padded

That way, the wrapping will be picked up by the zero padding at the ends of the data. With a mean zero trendless series, these zeros will not create the magnitude of bias that would be created if, for instance, there were a decided trend in the data. (In some cases, a fixed non-zero mean can be used with the data padded with that value).

Our recommendation is that you pad to at least double the length of the data.

Frequency domain filtering is usually much more efficient than time domain filtering if you need the entire transformed series (not just a single) entry, and the filter is more than a few terms in the time domain. A particularly important example of this is fractional differencing, which can be done in RATS using the **DIFF** instruction with the **FRACTION** option. The time domain lag polynomial for this is (theoretically) infinite. Most applications which do the calculations in the time domain truncate the filter at some large length (500 or 1000 terms). The calculation done in the frequency domain is many times faster and is more accurate.

14.9 Forecasting

Technical Information

Spectral methods can be used to forecast univariate time series. The **@SPECFORE** procedure implements the algorithm described here, which follows a suggestion made by John Geweke. The basis of the method is the moving average representation:

$$(3) \quad X_t = c(L)\varepsilon_t, \text{ where } c(0) = 1 \text{ and } \varepsilon \text{ is fundamental for } X$$

Box-Jenkins techniques attempt to represent c as a rational function: the numerator is the moving average part, the denominator is the autoregressive. Spectral methods allow us to compute an estimate of the Fourier transform of c , which can be used to compute forecasts. Their advantage over Box-Jenkins is that the same “model” applies to all data series, so the forecasting process is automatic. That, unfortunately, is one of the disadvantages as well—see the cautions below.

The following derivation can be found in Koopmans (1974), p. 235-237. Write the spectral density of X as the z -transform

$$(4) \quad f_X(z) = c(z)c(z^{-1})\sigma^2$$

Under reasonable regularity conditions on c , $\log(f_X)$ has a Laurent expansion:

$$(5) \quad \log f_X(z) = d(z) + d(z^{-1}) + d_0$$

where d is a one-sided polynomial in positive powers of z . If we take the $\exp$ of both sides of (5), and combine it with (4), we get:

$$(6) \quad c(z)c(z^{-1})\sigma^2 = \exp(d(z))\exp(d(z^{-1}))\exp(d_0)$$

To get the Fourier transform of c we need to

1. compute the log spectral density of X
2. mask the negative and zero frequencies to get the Fourier transform of d
3. take its exp, frequency by frequency (again, omitting the negative and zero frequencies)

We now have a “non-parametric” estimate of the moving average representation of the series. Once we have an estimate of the FT of c , we divide the FT of X by the FT of c to get the FT of ε process, set ε to zero outside of the range of actual data, then filter ε by c (in the frequency domain) to get the forecasts.

Note that the formulas (5) and (6) apply only to univariate processes—there’s no straightforward extension of this to multivariate processes.

Cautions

As with other frequency domain filtering techniques, the series must be well-padded. Also, please note the following:

- Data series which can be adequately represented by a Box-Jenkins model with few parameters will usually be better forecast using that technique, as the model is more sharply estimated.
- Spectral methods require more data (100+ data points to be safe) than other techniques, since they do not use it as efficiently.
- The series you run through the model *must* be stationary: proper differencing is vital.

Code

Compute the log spectral density. Scale factors will wash out.

```
fft 1
cmult 1 1 / 2
window 2 / 3
cln 3
```

Back to sequence-space to shut out the negative powers in the Laurent expansion. FT back and exponentiate to get FT of $\exp(D(Z))$.

```
ift 3
cset 3 = %z(t,3)*(t<=ordinate/2)
fft 3
cexp 3
```

IFT to get the sequence c. Normalize so that $C(0)=1$

```
ift 3
cset(scratch) 3 = %z(t,3)/%z(1,3)
```

Filter the input series by $1/C(L)$ to get residuals. Mask it outside the data range.

```
fft 3
cset 1 = %z(t,1)/%z(t,3)
ift 1 / 2
cset 2 = %z(t,2)*(t>=start1.and.t<=end1)
```

Filter the residuals E by $C(L)$ in the frequency domain

```
fft 2 / 1
cset 1 = %z(t,1)*%conjg(%z(t,3))
ift 1
```

14.10 Fractional Differencing/Integration

Fractional integration (or fractional differencing) was devised independently by Granger and Joyeux (1980) and Hosking (1981) to provide a way to model a process with “long-memory”—defined as having autocorrelations which damp off at a slow enough rate that they sum to infinity. Standard ARIMA models can only approximate this type of behavior, as a non-integrated ARMA model has autocorrelations that are exponentially damped, and are thus summable, and any integrated model has autocorrelations which don’t damp at all.

Fractional integration is modelled with the help of the specialized lag function $(1 - L)^d$ where d is non-integer. This can be expanded in the time domain using a generalized binomial expansion. In fact, in RATS, the coefficient on lag k can be obtained as `%BINOMIAL(d, k) * (-1)^k`—this gives rise to an infinite one-sided lag polynomial.

There has been only a modest amount of empirical work done in econometrics using fractional integration, particularly when compared with the amount done using unit root statistics. Part of this is clearly due to the fact that a first difference is easy to compute and has a relatively simple economic interpretation, while a fractional difference can’t even be computed exactly in the time domain—the filter always has to be truncated. It is also true that fractional integration is another entrant in the set of models proposed to explain long term behavior of data sets which in many cases simply aren’t long enough to separate one clearly from the others.

Fractional integration is a natural candidate for attack in the frequency domain since the lag function $(1 - L)^d$ can be computed easily there. For instance, Geweke and Porter-Hudak (1983) proposed a method for estimating d which involves analyzing the behavior of the periodogram of the data for low frequencies. The procedure `@GPH` implements this. A more recent refinement of this by Andrews and Guggenberger (2003) is available as `@AGFRACTD`.

If you just need to fractionally difference (or integrate) a series with a *known* value of d , you can use the **DIFFERENCE** instruction with the option `FRACTION` (you might also need the `DIFFERENCE` option as well). The frequency domain computations described here are used by **DIFFERENCE**. Because this is an infinite lag polynomial, the calculation is subject to the wraparound effect described on page UG–461. **DIFFERENCE** has a `PADDING` option if you need to pad out with a non-zero value which is sometimes appropriate. For instance, Baillie, Bollerslev and Mikkelsen (1996) estimate a FIGARCH model, which requires calculation of a fractionally differenced *squared* residual. Since the fractional difference is applied to a squared series, a zero value for padding would be inappropriate. Instead, it uses an unconditional estimate of the squared residual:

```
diff(fraction=d,pad=uupresample) uu / uufilter
```

As a general rule, if you run across a paper which uses fractional differencing or integration as part of the calculation (that is, doing something more than just estimating d) and it does not provide *any* details for how the infinite filter was handled, you should probably look for a better, more carefully done, example.

FRACTINT.RPF

The example `FRACTINT.RPF` estimates an ARFIMA model (AutoRegressive, Fractionally Integrated, Moving Average) model using frequency domain techniques. This is done using the instruction **MAXIMIZE** applied to a criterion function which is defined frequency by frequency. This function, which can be used more broadly than just for ARFIMA models, looks similar to those used in time domain likelihoods. However, where you would usually see a squared residual divided by a variance, it uses the ratio between the periodogram of the data and the spectral density implied by the model and its parameters. Fox and Taquq (1986) work through the theory of this method of estimation. It's possible to concentrate out the innovation variance, leaving a simpler function than the one shown here, but we've found that leaving the function in its original form works better.

An ARFIMA model takes the form

$$(7) \quad (1-L)^d A(L)y_t = B(L)u_t$$

where A and B are standard autoregressive and moving average polynomials. Solving this out gives

$$(8) \quad y_t = (B(L)/A(L))(1-L)^{-d} u_t$$

The square of the transfer function of the lag polynomial on the right is the spectral density function that we want (except for scale by the variance of u). The easiest way to compute this is to use a complex-valued **FRML** to give us the transfer function itself. This makes it simpler to change from one model to another, since the model is entirely described by the transfer function.

If you use this method of estimation, there are two things you must be sure to do:

1. Run the **MAXIMIZE** instruction across the full range of frequency domain ordinates. This is, almost certainly, longer than your actual data set.
2. Include the extra factor of actual observations divided by the number of frequency ordinates. Because of the procedure's reliance on frequency domain filtering, it only works acceptably if the data are padded. This factor corrects for the effect of padding.

As described above, this starts out with a complex-valued **FRML**, which computes the transfer function. The chosen model is an ARFIMA(1,d,1) which has four free parameters: the autoregressive parameter (A), the moving average parameter (B), the fractional difference parameter (D) and the innovation variance ($IVAR$). In the **FRML**, the `%CONJG` function on the final term is necessary in the next function since the `*` operator conjugates the right operand, and we want a straight multiply.

```
declare frml[complex] transfer
nonlin a b d ivar
frml transfer = (1+b*%zlag(t,1))/$(
    ((1-a*%zlag(t,1))*%conjg((1-%zlag(t,1))^d))
```

Chapter 14: Spectral Analysis

This next runs an AR(1) solely to get a reasonable guess for the variance. None of the other parameters are scale-sensitive, so they're given basic guesses.

```
linreg(noprint) usasur
# constant usasur{1}
compute a=.9, b=0.0, d=.5, ivar=%seesq
```

Following the standard practice for frequency-domain filter, this uses double the recommended number of ordinates,

```
compute nobs = %allocend()
compute nords = 2*%freqsize(nobs)
freq 3 nords
```

For the so-called Whittle likelihood, the (unsmoothed) periodogram provides sufficient statistics for the data. We need a real-valued copy of that since **MAXIMIZE** eventually needs a real-valued log likelihood.

```
rtoc
# unemp0
# 1
fft 1
cmult(scale=1.0/nobs) 1 1 / 2
set periodogram 1 nords = %real(%z(t,2))
```

The log likelihood for the sample is computed by summing log likelihood elements for each ordinate. Given a particular set of parameters, the theoretical spectral density is the variance times the square of transfer function. The observed “data” value is the periodogram. The %LOGDENSITYCV function uses these to compute the log likelihood of the ordinate. Because there are NORDS ordinates, but only NOBS observations, the number of observations parameter in the %LOGDENSITYCV is set to the fraction that corrects for that. **MAXIMIZE** then can estimate the model as with any other log likelihood, except that it needs to be summed across the number of ordinates, rather than the number of observations.

```
frml likely = trlamba=transfer, %logdensitycv($
    ivar*%cabs(trlamba)^2,periodogram,(float(nobs)/nords))
maximize(pmethod=simplex,piters=5,method=bfgs) likely 1 nords
```

Dividing the Fourier Transform of the series by the transfer function gives the Fourier Transform of the residuals. Inverse transform and send the relevant part back to the time domain. The first few residuals in this as they are likely to be very large with a correspondingly large variance, so they are skipped from the residual autocorrelation analysis:

```
cset 3 = %z(t,1)/transfer(t)
ift 3
ctor 1 nobs
# 3
# resids
corr(qstats,span=24) resids 3 *
```

15. Programming Tools

While not the longest chapter in the book, this chapter may be the most important. It's the programming tools described here that give RATS its tremendous power. Even if you don't use these yourself, you are an indirect consumer every time you use a procedure written by someone in the RATS community.

RATS provides an environment in which a user can, in effect, design his or her own instructions to do statistics in the manner desired. With the more advanced techniques, it's possible to create a fully menu-driven application.

Program Control and Compile Mode

Loops and Conditional Statements

Procedures and Functions

Repetitive Analyses

Interactive Procedures

Handling Output

15.1 Program Control and Compile Mode

Loops, Conditional Blocks, and Procedures

RATS offers several tools for program control. You can

- execute instructions only if a certain condition holds (or fails to hold) using **IF** and **ELSE**.
- loop over a range of values or a list with **DO** or **DOFOR**.
- execute a loop under the control of a condition using **WHILE** or **UNTIL**.
- write entire subprograms with **PROCEDURE** and **FUNCTION**.
- use the { and } symbols (or the equivalent keywords **BEGIN** and **END**) to define a “block” of instructions. Usually, these are used to execute a block of several instructions following a conditional statement when that conditional statement is itself part of a compiled section (such as a loop or procedure).

Procedures offer so much power, and are so useful, that we have put a discussion of their special properties in a separate section (15.2).

Compile Mode

Ordinarily, RATS executes instructions in order, processing one instruction completely before going on to the next. However, loops, conditional blocks, procedures (and the { and } symbols) put RATS into *compile mode*: RATS reads in and interprets an entire block of instructions. Rather than executing them immediately, it converts them into an internal code and stores this code. When RATS reaches an **END** instruction, signalling the end of the structure, it begins executing the compiled code (except for a **PROCEDURE** or **FUNCTION**, which are executed only on command).

Loops

RATS offers five different looping instructions. Two of these (**DO** and **DOFOR**) were first introduced in Section 1.8 in the *Introduction*.

DO	is a standard loop instruction which loops over an integer index. It is like a FORTRAN DO or a BASIC FOR.
DOFOR	is less standard but often very useful. It loops over a list of items: series, matrices, numbers, etc.
WHILE	loops as long as an expression is true.
UNTIL	loops <i>until</i> an expression is true. It always executes the enclosed instructions at least once—by contrast, WHILE can fail to loop a single time.
LOOP	loops unconditionally. You need to use a BREAK instruction to get out of the loop.

With any of the loops, you can use **NEXT** to skip directly to the next pass or **BREAK** to break out of the loop entirely. Note that, if you have nested loops a **NEXT** or **BREAK** applies only to the loop in which the instruction is located. **DO**, **DOFOR** and **LOOP** have a somewhat different setup from **WHILE** and **UNTIL**. They all loop over the instructions down to a matching **END**. Some examples follow.

```
do rend=1999:1,2017:12
  linreg y 1970:1 rend
  # constant y{1}
end do
```

This does regressions over a changing range of entries. All the regressions have the same starting date (1970:1), while the ending periods range from 1999:1 through 2017:12. This takes advantage of the fact that dates in RATS are handled as integer entry numbers.

```
dofor i = realgnp to realbfi
  set(scratch) i = log(i{0}/i{1})
end dofor
```

This does a percent change calculation for series from REALGNP to REALBFI. We use `I{0}` rather than just `I` because `I` (an `INTEGER` variable) is perfectly legal by itself in an expression. Including the lag notation tells RATS to use interpret `I` as a series.

WHILE (and **UNTIL**) loop only over the next instruction or block of instructions. When you want the conditional loop to include more than one instruction, you must enclose them within braces `{` and `}`. This will update both `COUNT` and `TOTAL` as long as `TOTAL` is less than 100.

```
{
  if docount==1 {
    compute total=0.0, count=0
    while total<100.0 {
      compute count=count+1
      compute total=total+series(count)
    }
  }
}
```

IF, **WHILE** and **UNTIL** statements need to be part of a larger compiled section, which is why everything above is enclosed in an outer set of `{` and `}`. It never hurts to use extra enclosing braces if you're not sure how instructions will be grouped.

It's a very good idea to indent lines to show the flow of execution as is done here. The menu operations *Edit—Indent Lines* and *Edit—Prettify* are the easiest way to do this. The default level of indenting is 3 spaces, which is what we used in all the RATS examples.

15.2 Procedures and Functions

What Are They?

The most powerful compiler structures are **PROCEDURES** and **FUNCTIONS**. In effect, procedures allow you to create your own “instructions” from a sequence of other RATS commands, while functions define callable functions similar to built-in functions like `INV` and `%DENSITY`.

We include hundreds of procedures with RATS, and updates of those, plus other new ones are available for downloading from our web site and other sources (*Help—Update Procedures* can help maintain your collection of procedures.) These allow you to perform many complex tasks without having to code them yourselves. In the following pages, we’ll tell you how to make use of these existing procedures. Then, we’ll discuss the various elements of RATS procedures and guide you through the process of writing a complete procedure so you can get a feel for how it’s done. We’ll focus mainly on procedures—functions are similar but can’t use all the features of a procedure. We’ll often use the word *subprogram* when discussing information that applies to both procedures and functions.

First, a look at the basic elements of a procedure: A procedure begins with a **PROCEDURE** statement and ends with a matching **END**. A procedure may have some, all, or none of the following special features:

- *Parameters*, the names of which are listed following the procedure name on the **PROCEDURE** statement. Use these to pass information to the procedure. Their types are set using the statement **TYPE**.
- *Local variables*, recognized only within the procedure. They are set up by the statement **LOCAL**.
- *Options*, which operate much like those for a standard RATS instruction and are set up using the statement **OPTION**.
- *Supplementary cards*, which are defined with the instruction **ENTER**.

A function begins with **FUNCTION**, ends with a matching **END**, and can include parameters and local variables, but not options and supplementary cards.

Comparison With Other Structures

Unlike other types of compiled structures, these are not executed automatically when the processing of its code is finished. Instead, RATS saves the generated code internally and only executes it when it receives an **EXECUTE** instruction (or the `@procedure` shortcut) for a procedure, or sees *function name(arguments)* for a function. They can be invoked any number of times, again differing from the other structures, which execute only once.

Because a subprogram is not a loop, you cannot use **BREAK** to leave it before the end. Use **RETURN** instead. Also, if you have calls to a subprogram nested inside another subprogram, you must compile the code for the *inner* subprogram first, so that RATS will recognize the procedure or function call when you compile the main routine.

15.2.1 Using Existing Procedures

Storing Procedures in External Files

Procedures and functions do not need to be in the same file as the program which uses them. There are several advantages to keeping them on separate files:

- If you have a subprogram which can be used by more than one program, you only need to maintain one copy of the subprogram.
- Keeping subprograms on separate files means that your main program files will be shorter and easier to edit and debug.
- You can call in a subprogram even when you're working interactively.

Before you can execute a subprogram stored on a separate file, RATS needs to execute (or “compile”) the instructions on that file. In many cases, RATS can locate and execute the appropriate file automatically. Otherwise, you can use **SOURCE** to tell RATS which file to execute. See below for details.

Look Up the Syntax!

Descriptions of the most important procedures are included in the RATS Help and most also have a separate page on the RATS forum (<https://estima.com/forum>). These describe the syntax in a form similar to that used for instructions. For those that are newly-written (or less heavily used) and thus not covered in those sources, the first thing you should do is to open the file and read through the comments at the top. These are ordinary text files, so you can open them by doing *File—Open* in RATS, or any other text-editor or word-processor. All of the procedure files that we have written include comments describing the procedure syntax, options available, and, usually, technical details and references as well.

If you create a procedure yourself that you wish to share with others, make sure you include similar information, so that the future users will know how to use it, and will know whom to thank.

Compiling the Procedure

To use a procedure or function that is stored on an external file, RATS will first need to execute the commands on the file. This “compiles” the procedure or function code. Procedures can then be executed using the **EXECUTE** command or (more commonly) *@procedure*. For functions, you simply use the function name (and any parameters) in an expression.

For procedures, the easiest approach is to let RATS locate and compile the procedure file automatically, by simply executing the procedure. If the procedure has not already been compiled, RATS will search for a filename matching the name of the procedure, with an extension of *.SRC*. For example, if you do **@DFUNIT**, RATS will search for a file called *DFUNIT.SRC*. It will start searching in the “Procedure Directories” specified in the *File—Preferences* dialog. If necessary, RATS will then search the current working directory and the directory containing the RATS executable file. If it still

Chapter 15: Advanced Programming Tools

fails to find a matching file, you will see a dialog box prompting you to locate the file.

If you prefer, you can use an explicit **SOURCE** instruction to read in and execute the commands on an external file. This is required for functions (RATS won't search for those), for procedure files where the procedure name does not match the file name, for procedures located in a directory not included in the search paths described above, or for a custom variation of a standard procedure.

Suppose you are writing a program that will require the **@DISAGGREGATE** procedure to interpolate low-frequency data to a higher frequency. If you have a custom version of this saved on a directory called `c:\users\myname\documents\my rats\`, you can either put the path to your custom procedures above the standard one in the *Procedure Directories* on your *Preferences* (which might make sense if you have quite a few of these) or use an explicit **SOURCE** as shown below. Note the “..” around the name, which is required if there are any embedded spaces and never hurt.

```
source "c:\users\myname\documents\my rats\disaggregate.src"
calendar(m) 1990
open data quarterly.rat
data(format=rats) 1990:1 2017:12 x1 x2
@disaggregate(model=loglin,tsmodel=rw1) x1 / xm
# constant x2
```

Procedure Libraries

Procedure libraries are collections of procedures and functions which are brought in at the start of the program, and also whenever the program is reinitialized, using, for instance, *File—Clear Memory*. This makes sure the procedures included in the library are always available. There are several ways to specify a procedure file:

1. The RATS instruction **ENV PROC**=*name of procedure library*
2. The **/PROC**=*name of procedure library* option on the command line.
3. Through setting the procedure library in the preferences.

You can form the library either by copying the full text of the desired procedures to a single file, or by setting up a file with a group of **SOURCE** instructions for the procedures you want.

15.2.2 Writing Your Own Procedures and Functions

Instruction Order

The instructions that make up the subprogram must be in the following order:

```
PROCEDURE or FUNCTION statement
  TYPE and OPTION statements
  other RATS instructions
END
```

The first line provides the subprogram name and lists the names of the formal parameters. **TYPE** sets the variable types of the parameters and the function, while **OPTION** instructions define any options for a procedure. **DECLARE** and **LOCAL** statements may be scattered among the other instructions, but it's usually a good idea to keep them together before the executable instructions.

Learning How to Write Procedures

Start small!! If the first procedure you try to write uses 100 RATS instructions to do the latest wrinkle in statistical theory, you are likely to be working at it for a *very* long time. Begin with a more modest (or even trivial) operation and work up from there. We would suggest that you start with parameters only, then add local variables and finally options (if they make sense for your application). Get it to work correctly with proper input before putting in checks for garbage input.

Learn from our procedures!! We have provided many procedures which use the most advanced techniques that RATS has to offer.

Parameters

Parameters allow you to pass information stored in variables (data series, scalar variables, etc.) from the main program to a subprogram. You list the parameters on the **PROCEDURE** or **FUNCTION** statement itself following the subprogram name, with the parameters separated by blanks. We refer to these as *formal* parameters. You use **TYPE** statements to indicate what type of information they provide. You pass information to these formal parameters by specifying a matching set of *actual* parameters on the **EXECUTE** instruction or function reference. Within the subprogram, you manipulate the data using the formal parameter names. If you pass a parameter by address (see next segment), you can either use or set it within the procedure. If, however, you pass a parameter by value, you may only *use* it within the procedure.

By default, formal parameters of a **PROCEDURE** are all type **INTEGER**, passed by value, and those of a **FUNCTION** are **REAL**, again passed by value. The **FUNCTION** name itself is **REAL** by default. You will need to use one or more **TYPE** statements if you want them to take different types.

You should keep the number of parameters to a minimum. Options (page UG–475) tend to be much easier to use. Parameters have to be done in a particular order, so you should avoid anything beyond the typical *series start end result*.

Chapter 15: Advanced Programming Tools

Passing Parameters: By Value vs. By Address

Parameters can be passed to a procedure in one of two ways: by address, or by value.

- Passing by address means that the actual memory address of a variable is passed to the procedure. If you change the value of the formal parameter inside the procedure, you will change the value of the actual parameter.
- Passing by value means that the only information made available to the procedure is the value of the actual parameter. The procedure is not allowed to manipulate the value of the formal (or actual) parameter. In fact, you will get a syntax error if you try to reset the value.

You should pass a parameter argument by address *only* if it is being *set* by the procedure. Anything that is only an *input* to the procedure should be passed by value.

You distinguish between the two by prefixing the parameter name with `*` on a **TYPE** statement if you want it called by address. Without the `*`, it is called by value.

```
proc forecast depvar start end forecast
type series depvar *forecast
type integer start end
```

Parameters `DEPVAR` and `FORECAST` are both type `SERIES`. `DEPVAR`, however, is called by value, and `FORECAST` by address. `START` and `END` are `INTEGERS`, passed by value. So, `DEPVAR`, `START`, and `END` will be inputs to the procedure and `FORECAST` will be the output that is computed and returned to the main program.

```
function sum num1 num2
type integer sum num1 num2
compute sum = num1+num2
end
*
compute a=5, b=10
disp a "+" b "=" sum(a,b)
```

This computes the sum of two integer values, and returns the result as an integer. `NUM1` and `NUM2` are passed by value, so the function can use these variables, but it cannot *change* their value. The function name itself is always considered to be passed by address (there wouldn't be much point otherwise), so you don't need a `*` on it.

It's a good idea to include a **TYPE** for every parameter. If you don't, the default for a **PROCEDURE** parameter is `INTEGER` and for a **FUNCTION** is `REAL`. (This reflects the most common uses for each).

Local Variables

The variables in a RATS program are *global* unless you state otherwise—that is, you can use them within any part of the program, *including* procedures. There can be only one global variable with any one name.

A *local* variable (defined by **LOCAL**) can only be used in a subprogram, and *only within the subprogram which defines it*. RATS keeps global and local variables completely separate, so a local variable can have the same name as a global one. If a name conflict arises, the local variable takes precedence.

If you plan to use a subprogram in many different programs, you should write it using only parameters, local variables, and variables defined internally by RATS. That way there can be no possibility of conflict with the variable names in the main program. If you want some variables computed within a procedure to be accessible to the user later (the way that %RSS and %BETA are for an instruction like **LINREG**), it is a good idea to use a special prefix (such as %) on the variable names, to avoid the possibility of conflict with the user's own variable names, or with RATS reserved names in future releases.

LOCAL takes the same form as **DECLARE**.

```
procedure bayestst series start end
type series      series
type integer     start end
local series     sdiff strend
local integer    start1 end1
local real       teststat margalph alphatest
```

The procedure **@BAYESTST** has several local variables. Note particularly the use of local series. If you use a series locally, it is very important to declare it as such.

Options

The **OPTION** statement defines options for the procedure. (They can't be used with a function). When you call the procedure, you can choose options in a field immediately after the procedure name, much like options on regular RATS instructions. You can also define default values or settings for procedure options. Like parameters, options can be passed by value or by address. To have it passed by address, prefix the formal parameter name with *****.

See the description for **OPTION** for details—we just give some quick information here. For the three types (switch, choice and value), use

option	switch	<i>option name</i>	<i>Default</i>	(1 for ON, 0 for OFF)
option	choice	<i>option name</i>	<i>Default number</i>	<i>List of Choices</i>
option	<i>data type</i>	<i>option name</i>	<i>Default value</i>	

- Option names are local to the procedure.
- You must use the full name for *option name* within the procedure.

Chapter 15: Advanced Programming Tools

- When you invoke the procedure using **EXECUTE** or *@procedure*, only the first three characters of the option name are significant (just like standard RATS options). Make sure you do not use conflicting option names. For instance, **RESIDUALS** and **RESTRICT** both abbreviate to **RES**. RATS will give an error if you use two options with conflicting names.
- *Do not use a name which begins with the letters NO*. This is reserved for turning off switch options.

Passing Information to RATS Instructions

RATS allows you to pass information to instruction options in a more flexible fashion than was described in the Introduction. With all types of options (including switch and choice), you can use the syntax

```
instruction(option=expression or variable)

procedure nonsense
option switch print 1
option symmetric *vcvmat
vcv(print=print,matrix=vcvmat)
end
```

In this example, the **PRINT** switch is on by default, so the **PRINT** option on **VCV** will also be on by default. If, however, you use the option **NOPRINT** when you execute **NONSENSE**, the **PRINT** variable will be “off” (value 0) which will turn off the **PRINT** option on **VCV**.

The **VCVMAT** option has no default. If you don’t use the option when you execute the procedure, the **MATRIX** option on **VCV** will behave as if you didn’t use it at all.

Supplementary Cards

RATS allows you some flexibility in dealing with supplementary cards for RATS instructions within the procedure. You can either:

- code them immediately after the instruction inside the procedure which requires them (if you know what the values on the card will be), or
- put them after the **EXECUTE** instruction which invokes the procedure. If an instruction with an omitted supplementary card is within a loop, **EXECUTE** must be followed by one supplementary card for each pass through the loop.

The latter method allows you to act as if the supplementary cards are for the **PROCEDURE** rather than for instruction within it which requires them. When a single instruction requires several supplementary cards, you must treat them all in the same fashion—you cannot include three in the procedure and leave out the fourth.

If you need a list of regressors, the best way to get them is by using an **EQUATION** instruction applied to a local equation. For instance:

```
local equation xvar
equation xvar *
```

This will define the equation using the regressor list that follows the line executing the procedure. This was taken from the **@DISAGGREGATE** procedure, so

```
@disaggregate(type=linear) loggdp
# constant logip logsales
```

will define XVAR (within the procedure) as an equation with **CONSTANT**, **LOGIP** and **LOGSALES** as its explanatory variables. You can then use the **%EQNxxxx** functions to work with this. For instance, **%EQNSIZE(XVAR)** will be the number of variables in the list, **%EQNTABLE(XVAR)** the table of series/lag combinations and **%EQNXVECTOR(XVAR,T)** will be the vector `|| 1, LOGIP(T), LOGSALES(T) ||`

If you need some other type of list of series (where lag notation isn't permitted), bring it in as an **VECTOR[INTEGER]** array using **ENTER(VARYING)** and use that array when you need the list. For instance, this is taken from the **@JOHMLE** procedure:

```
local vect[int] v
enter(varying) v
```

You can use **%SIZE(V)** to see how many elements there are in that.

The %DEFINED Function

You can use the function **%DEFINED(option/parameter name)** to determine whether an option or parameter was given a value when the procedure was executed. It returns 1 if it was defined and 0 otherwise. If an option has a default value, it will *always* be defined, since it will, at minimum, get the default value. We use **%DEFINED** extensively in our procedures to ensure the user executed the procedure properly.

The INQUIRE Instruction

INQUIRE is very handy when you want to write a procedure which (perhaps optionally) determines the range itself. We use it in most of our procedures.

```
procedure stockwat series start end
type series series
type integer start end
option integer arcorr 1
local integer start1 startm endm
inquire(series=series) startm<<start endm<<end
inquire(series=series) start1
if .not.%defined(start)
    compute startm=start1+arcorr+3
```

The first **INQUIRE** sets **STARTM** to the value of the parameter **START** if the user provides one, otherwise it will be the first defined entry of **SERIES**. **ENDM** is similar. The second **INQUIRE** just determines the first defined entry of **SERIES**. If there was no value for **START**, the **COMPUTE** sets **STARTM** to begin **ARCCORR+3** entries into **SERIES**.

15.2.3 Putting It All Together: An Example

We will now develop a **PROCEDURE** for implementing a General LM test for serial correlation from page UG–83. We will start with a very simple procedure, concentrating upon getting the calculations correct, and then refine it. We need to input the following information to the procedure:

- The dependent variable
- The regressors
- The number of lags to test

To make this look as much like a RATS instruction as possible, the dependent variable should be a parameter, the regressors should come from a supplementary card, and the number of lags should be an option.

The only tricky part about our first attempt is handling the list of regressors. We need it in two places: the initial regression and the auxiliary regression. We pull it in with the **EQUATION** instruction, then extract that list with the **%RLFROMEQN** function. The regressor list functions (page UG–482) come in very handy in writing procedures.

```
procedure sctest depvar
  type series depvar

  option integer lags 1
  local equation regeqn
  local vect[integer] reglist

  equation regeqn depvar
  linreg(equation=regeqn)
  compute reglist=%rlfromeqn(regeqn)

  linreg(noprint,dfc=%nreg) %resids
  # reglist %resids{1 to lags}
  disp "LM statistic" %trsq "Signif Level" %chisqr(%trsq,lags)
end
```

We should now check against the direct code from page UG–83 to make sure the calculations are correct. After we're sure we're doing the calculations correctly, we can make refinements. We will make four changes:

- We will improve the output. This is a straightforward change to the **DISPLAY** instruction at the end of the procedure.
- We will add a **PRINT** option, to allow the user to suppress (with **NOPRINT**) the output from the first regression.
- We will have **SCTEST** define the standard variables **%CDSTAT** and **%SIGNIF**.
- We will allow the user to specify a *start* and *end* range.

The last is the only difficult part of this. The range is easy to handle if we either require the user to give an explicit range, or if we *always* use a default range. It gets complicated when we let the user do either. This is where **INQUIRE** comes in. We can determine the range of the initial regression by using an **INQUIRE** instruction with the **EQUATION** option. If the user passes values to the **START** or **END** parameters, they override the values from **INQUIRE**. Once we have values for the range on the first regression, the auxiliary regression just starts **LAGS** entries farther up.

```
procedure sctest depvar start end
  type series depvar
  type integer start end

  local vect[integer] reglist
  local integer start1 endl
  local equation regeqn

  option integer lags 1
  option switch print 1

  equation regeqn depvar
  compute reglist=%rlfromeqn(regeqn)
  inquire(equation=regeqn) start1<<start endl<<endl
  linreg(print=print,equation=regeqn) depvar start1 endl

  linreg(noprint,dfc=%nreg) %resids start1+lags endl
  # reglist %resids{1 to lags}
  compute %cdstat=%trsqr , %signif=%chisqr(%trsqr,lags)
  disp "LM test for Serial Correlation of Order " lags
  disp "Test Statistic " %trsqr "Signif Level " #.##### %signif
end
```

Suppose we want to use the procedure to regress *Y* on three regressors (a constant, *x1* and *x2*) and test the residuals for serial correlation with a lag length of 4. We can do that with the following instruction:

```
@sctest(lags=4) y
# constant x1 x2
```

An alternative handling for the range that would work here is to run the regression using **START** and **END**, then use **%REGSTART()** and **%REGENGEND()** to recover the range used. That allows **LINREG**'s standard handling of the range parameters to do the work.

```
linreg(print=print,equation=regeqn) depvar start end
compute start1=%regstart(),endl=%regend()
```

Chapter 15: Advanced Programming Tools

Checking Local Versus Global Declarations

When writing and testing a new procedure, you may find the instruction **DEBUG (SYMBOL)** helpful. If you do **DEBUG (SYMBOL)** and then compile your procedure code, RATS will report any variables that are created as global variables, or that are declared as local but aren't actually used. For example:

```
debug (symbol)
source kfilter.src
```

produces output like this:

```
KFILTER, line 53                Using TIME as INTEGER
```

indicating that `TIME` is being defined as a global integer, on line 53 of the procedure.

The FIXED Statement—Creating Lookup Tables

FIXED can be used in a subprogram to set up a lookup table for (for instance) critical values. It creates an array (or array of arrays) whose dimensions and content are known constants. You can't use expressions in the dimensions or the values.

FIXED can set up only one array per statement, which is followed by its values. It can be placed anywhere within the subprogram, though it generally makes sense to put it near the top, typically between the other declaration statements and the first executable instructions. The array that it creates is local to the subprogram. The following creates a lookup table with three branches, each of which is 2×6 . You would access a value of this with `CL(branch needed) (row needed, column needed)`.

```
fixed vect[rect] cl(3)(2,6)
-1.6156  -0.1810   0.0000   0.0000   0.0000   0.0000
-1.9393  -0.3980   0.0000   0.0000   0.0000   0.0000

-1.9480  -17.8390  104.0860   0.8020   5.5580  -18.3320
-1.6240  -19.8880  155.2310   0.7090   5.4800  -16.0550

-2.8380  -20.3280  124.1910   1.2670  10.5300  -24.6000
-2.5500  -20.1660  155.2150   1.1330   9.8080  -20.3130
```

Instruction Locations—Debugging Compiled Code

When RATS processes a compiled structure, it prefixes the actual code on each line by a pair of numbers in parentheses, which can be very helpful in debugging problems with compiled sections. (Note: these numbers normally aren't displayed when running in interactive mode, but you can see them by using a **SOURCE (ECHO)** command to execute your program file.)

The first number is the structure level *at the end of the previous line*. The second number is the location of the instruction within the compiled section. Consider the following example, taken from the **@DFUNIT** procedure:


```
(01.0155) inquire(series=series) startl<<start endl<<endl
(01.0207) *
(01.0207) set sdiff startl+1 endl = series-series{1}
(01.0271) set strend startl+1 endl = t
```

If an error occurs during execution of the compiled section, the error message will include a line indicating the location of the error. For example, if you were to try to apply **@DFUNIT** to a series that contained no data, you would get the message:

```
## SR10. Missing Values And/Or SMPL Options Leave No Usable Data Points
The Error Occurred At Line 17, Location 0206 of DFUNIT
```

The first line in the traceback says the occurred at location 0206. That tells us the error was generated by the **INQUIRE** instruction, which occupies memory locations 155 through 206. The second traceback line gives the same information, but tells what line in the procedure caused it. *Note that this is the line number from the start of the actual procedure or loop.* Most procedure files have a header of comments at their top, but the line count used here doesn't start until the **PROCEDURE** statement.

The structure level is helpful in locating block nesting errors. Consider this code from later in the procedure:

```
(02.1417) if lags {
(03.1432)   summarize(noprint)
(03.1437)   # sdiff{1 to lags}
(03.1493)   compute fiddle=1.0/(1.0-%sumlc)
(03.1524) }
(02.1524) else
```

Suppose that we forgot the **}** before the **ELSE**. The **ELSE** location would be prefixed with 03, instead of 02 which is what we would expect (remember that the number prefixing a line is the level at the *end* of the previous line).

The **SOURCE** instruction includes an **ECHO/NOECHO** option, which controls whether or not the instructions being executed are displayed in the output window or file. When trying to debug an error, or when using a procedure for the first time, you may find it handy to use the **ECHO** option to see the instructions echoed to the output window.

15.2.4 Regressor List Functions

RATS provides several functions for working with regressor lists, and these can be helpful in writing procedures, particularly when you need to use lags. A regressor list is represented by a `VECTOR[INTEGER]` which codes information about the series and lags. Use these functions to manipulate the list; don't try to do it yourself.

<code>%rladdone(R,S)</code>	Add a regressor to a regressor list
<code>%rladdlag(R,S,L)</code>	Add a lagged regressor to a regressor list
<code>%rladdlaglist(R,S,L)</code>	Add a list of lags to a regressor list
<code>%rlconcat(R1,R1)</code>	Concatenate two regressor lists
<code>%rlcount(R)</code>	Return the number of regressors in a list
<code>%rlempty()</code>	Creates an empty regressor list
<code>%rlfromtable(Table)</code>	Convert a regression table to a regressor list
<code>%rlverify(R)</code>	Verify a regressor list

In earlier versions of RATS, procedures often used **ENTER (VARYING)** instructions to build up regressor lists. The regressor list functions are somewhat cleaner and they can handle lag lists more flexibly. The following builds up a list of the dependent variables and regressors for a cointegration procedure. The original dependent variables are in the `VECT[SERIES] V`, while the differenced versions are in `DV`. The list needs to include:

1. The current differenced variables (added with `%RLADDONE`)
2. The first lag of the original variables (added with `%RLADDLAG`)
3. $L-1$ lags of each differenced variable (added with `%RLADDLAGLIST`)
4. The deterministic variables, if needed (`%RLADDONE`)

First, we define an empty regression list called `REGLIST`. We then use other regressor list functions to add additional terms to the existing list and save the result back into `REGLIST`.

```
compute reglist = %rlempty()
do i=1,numvar
  compute reglist=%rladdone(reglist,dv(i))
end do i
do i=1,numvar
  compute reglist=%rladdlag(reglist,v(i),1)
end do i
if lags>1
  do i=1,numvar
    compute reglist=%rladdlaglist(reglist,dv(i),$
      %seq(1,lags-1))
  end do i
if determ>=2
  compute reglist=%rladdone(reglist,constant)
if determ>=3
  compute reglist=%rladdone(reglist,trend)
```

15.3 Repetitive Analyses: Cutting Your Workload

Background

Suppose that you need to do an analysis of 200 stock prices, or 25 countries, or some similarly gruesome task. If you're like us, you want to get it accomplished with as little work as possible. The brute force method—cloning code for each series or country and hand editing it—is a time-consuming, and worse, very inflexible way to handle this.

Fortunately, RATS provides a kit of tools for dealing with these situations. The more structured the analysis, the easier it is to write a short program.

Organizing Your Data

The first step is to *save your data in one of the formats where data are read by name*—such as RATS format or a spreadsheet. RATS format is better, because you are likely to need to dip into the file many times. We will be changing the name (label) of a series or set of series with each pass, and re-executing **DATA**, so we can keep refilling the same memory space with new data.

15.3.1 Numbered Series and Arrays of Series

The other key step is to use numbered data series or arrays of series. (See page UG–23 for more details). For numbered series, you need to set aside a block of series either with **ALLOCATE** (using the *series* parameter), with **SCRATCH**, or with **DECLARE VECT[SERIES]**. For instance,

```
allocate 5 2018:2
```

creates series 1 to 5. We can reference them as 1, 2, 3, 4 and 5 or with variables such as DO loop indexes which take those values. Note that to use these series we must refer to them explicitly with numbers, integer variables, or names assigned to them with the instruction **EQV**.

However, VECTOR of SERIES are generally easier and more foolproof than series numbers. For example, instead of the above, do:

```
allocate 2018:2
dec vector[series] vs(5)
```

You can refer to specific series using the subscript on VS (and you can use a loop index for that subscript to easily loop over the set of series). You can also use the name of the vector by itself in place of a list of series when you want an operation (such as **PRINT** or **COPY**) to apply to all series in the VECTOR. For example:

```
print / vs(1)           Prints the series stored in element 1 of VS
print / vs              Prints all 5 series stored in VS
```

Using LABEL Variables and the %S Function

LABEL variables and VECTORS of LABELS play an important role in these examples. The **LABELS** instruction resets the label of a series, which, as mentioned above, is what the **DATA** instruction uses to locate data on a RATS format or spreadsheet file.

“Labels” are strings of up to sixteen characters. You can concatenate them using either the explicit function %CONCAT (a,b), or simply by using a+b. The a+b notation also works if b is an integer or integer-valued expression. The function %S (label) can be used to refer to (or create) a series with the given label. Examples of the use of labels:

```
declare label lab lab1 lab2
compute lab="abcde"
compute lab1=lab+lab           lab1="abcdeabcde"
compute lab3=lab+3             lab3="abcde3"
set %s(lab3) = %s(lab)^3       Series ABCDE3 = ABCDE^3
```

Examples

VECTORS of LABELS are particularly useful for reading in or storing lists of variable or file names. For example, suppose you have a text file containing a list of (an unknown number of) series names, and you want to read those series in from a data file. You could do something like:

```
declare vector[label] names
open data namelist.txt
read(varying) names
compute num = %size(names)
declare vector[series] vars(num)
labels vars
# names
open data data.xls
data(format=xls,org=cols) / vars
```

This pulls in all the variable names from NAMELIST.TXT, determines how many names were read in, creates a VECTOR of SERIES of that size, and assigns the labels to the series in the vectors. It then reads in those series from a spreadsheet (RATS looks for series on the file whose names match the labels of the VARS series).

This next example computes beta values for a group of stocks. First, consider computing a beta value for a single security, which requires regressing the return for the security on the market return. Let's let MARKET be the market return and series 1 be the return on the single security. We just regress series 1 on MARKET and report the result:

```
linreg(noprint) 1
# constant market
display(unit=copy) %1(1) @20 ##.##### %beta(2)
```

Note that %L(1) is the label of series 1, %BETA(2) is the second coefficient. We use NOPRINT on **LINREG** and instead list the important information to the COPY unit.

Now all we need to do is to write a loop around this which resets the labels with each pass. Here we will handle the security labels by reading them from a file. The file consists of the symbols separated by blanks. (This is example BETAS.RPF).

```
cal(m) 1986
allocate 1 1996:12
open data ticker.lst
declare vector[label] tickers
read(varying) tickers
compute ntickers=%rows(tickers)
open data returns.rat
open copy betas.lst
data(format=rats) / market
do i=1,ntickers
    labels 1
    # tickers(i)
    clear 1
    data(format=rats) / 1
    linreg(noprint) 1
    # constant market
    display(unit=copy) tickers(i) @20 ##.##### %beta(2)
end do i
```

Create 1 numbered series
Open file with symbol list
Declare label vector
Read ticker labels into vector

Open RATS file with data
Open file for storing computed betas
Read market return

Relabel series 1 using Tickers vector

Reset all entries to N/A
Read data for series 1

This next program does an overlay graph of unemployment and inflation for seven countries, using a VECTOR of SERIES. The labels here are generated by appending SUR (Standardized Unemployment Rate) and DEFLS suffixes to three letter country prefixes. The **INQUIRE** instruction determines the maximum range over which both inflation and unemployment are defined. This is example INFLUNEM.RPF.

```
open data oecddata.rat
cal(q) 1960
all 1998:4
declare label country
declare vector[series] vars(2)
dofor country = "can" "deu" "fra" "gbr" "ita" "jpn" "usa"
    labels vars(1) vars(2)
    # country+"sur" country+"defls"
    clear vars
    data(format=rats) / vars
    set inflation = log(vars(2){0}/vars(2){1})
    inquire(regressorlist) n1 n2
    # vars(1) inflation
    graph(header="Inflation + Unemployment for "+country,$
        overlay=line) 2
    # vars(1) n1 n2
    # inflation n1 n2
end dofor
```

15.3.2 HASH Aggregator

A **HASH** is similar to a **VECTOR**, except that the elements are indexed by strings (called “keys”) rather than by numbers. It can’t be used directly in calculations the way a **VECTOR** can, but is instead designed to make it simpler to organize information. You can create a **HASH** of any data type. The one function which applies to a **HASH** is **%KEYS**, which returns a **VECTOR[STRINGS]** with the keys. The keys aren’t case-sensitive, so you can use “abc” or “ABC” or “AbC” to reference the same element.

If you **DISPLAY** a **HASH**, it will show the combination of key and value as in the following:

```
"def" =>      13.50000
"ghi" =>      11.30000
"jlk" =>      15.30000
```

As with other aggregate types, the data type for a **HASH** is written as **HASH[basetype]**, such as **HASH[REAL]** or **HASH[VECT[VECT]]**. It’s important to note that a new element is added to the **HASH** any time you use a new string, even if it’s by mistake. For instance,

```
dec hash[series] rseries
set rseries("defined") = %ran(1.0)
stats rseries("notdefined")
```

will add a second (empty) series named “notdefined” to the **HASH** and give a warning about having no data on the **STATISTICS** instruction.

An example is the following:

```
open data "g7rxrate.xls"
calendar(a) 1970:1
data(format=xls,org=columns) 1970:01 2003:01 canrxrate frarxrate
  gbrxxrate gerrxrate itarxrate jpnrxrate
*
dec hash[integer] lags
dofor s = canrxrate to jpnrxrate
  set lx = log(s{0}/s{1})
  @adfautoselect(maxlags=4,print,crit=aic) lx
  compute lags(%l(s))=%%autop
end dofor s
```

This runs a set of **@ADFAUTOSELECT** procedures on data, and saves the number of chosen lags into a **HASH[INTEGER]**. Now **LAGS("CANRXRATE")** will be the number of lags chosen when **S** is **CANRXRATE** and **LAGS("ITARXRATE")** will be the number chosen when **S** is **ITARXRATE**. Note that you have to use “...” around literal strings in referencing an element.

15.3.3 LIST Aggregator

A `LIST` is similar to a `VECTOR`, except that its length is easily adjustable. Its main value is for organizing information where the number of elements isn't known in advance.

As with other aggregate types, the data type for a `LIST` is written as `LIST[basetype]`, such as `LIST[INTEGER]` or `LIST[STRING]`. The `basetype` can only be `INTEGER`, `REAL`, `COMPLEX`, `LABEL` or `STRING`.

The `+` and `-` operators work differently for a `LIST` than for a `VECTOR`:

<code>list+basetype element</code>	appends a single element to the list
<code>list+LIST of the basetype</code>	appends the contents of the second list
<code>list+VECTOR of the basetype</code>	appends the contents of the <code>VECTOR</code>
<code>list-basetype element</code>	removes the element from the list (if it exists)
<code>list-LIST of the basetype</code>	removes any elements from the second list from the first
<code>list-VECTOR of the basetype</code>	removes any elements from the <code>VECTOR</code> from the list

For instance

```
dec list[integer] mylist
compute mylist=mylist+%seq(1,5)
compute mylist=mylist+20
compute mylist=mylist-||3,5||
```

leaves `mylist` with elements 1,2,4,20 (it was 1,2,3,4,5,20 before the 3,5 were removed).

LISTEXAMPLE.RPF Example

LISTEXAMPLE.RPF does a sequence of regressions, removing from the sample each time just the most extreme outlier, so long as there is a data point with a standardized residual greater than 2.5 in absolute value. The LIST[INTEGER] called OUTLIERS is used to keep track of the entries as they are removed.

```
set working = 1.0
dec list[integer] outliers
loop
  linreg(noprint,smpl=working) logy / resids
  # constant logk logl
  prj(xvx=px)
  set stdresids = abs(resids/sqrt(%seesq*(1-px)))
  ext(noprint) stdresids
  if %maximum>=2.5 {
    compute outliers=outliers+%maxent
    compute working(%maxent)=0
  }
  else
    break
end loop
*
do i=1,%size(outliers)
  disp "Outlier at" outliers(i)
end do i
```


15.4 Interactive Procedures

RATS provides a variety of tools for writing highly interactive programs and procedures. You can construct programs which allow the end-user to control the program using pull-down menus, view messages and progress bars via pop-up dialog boxes, select series from drop-down lists, and more.

Tools

RATS offers several ways to control an *interactive procedure*: a set of steps controlled by a user through dialogs and prompts. These range from using simple prompts and informational dialogs to simplify routine or oft-repeated tasks, to implementing complex menu-driven procedures like the CATS cointegration analysis package.

- The **SOURCE** instruction can be extremely helpful when you are designing procedures where the user picks one of a set of (RATS) programs to run. Put the programs on separate files and use separate **SOURCE** instructions for each.
- **MESSAGEBOX** and **INFOBOX** display information in pop-up dialog boxes. **MESSAGEBOX** is used for messages which require a user-response (such as Yes, No, or Cancel), while **INFOBOX** is used to display informational messages (which can be updated) and graphical “progress” bars.
- **USERMENU** defines a pull-down menu, which can be used to control the program. The CATS cointegration analysis procedure (available separately from Estima) is the most sophisticated implementation of **USERMENUS**.
- **SELECT** displays a dialog box that allows the user to select from a list of series, strings, or integers.
- **QUERY** obtains information (as a line of text) from the user. The input text is then interpreted.
- **MEDIT** displays the contents of an array in a spreadsheet style window, which makes it easy to view, enter, or edit the values in the array.
- **MENU** and **CHOICE** display a dialog prompting the user to choose one action from a single list of choices.
- **DIALOG** allows you to create custom user-defined dialog boxes. It provides all the capabilities of **MENU/CHOICE**, **SELECT**, and **QUERY** and a lot more.

Using MESSAGEBOX

MESSAGEBOX displays one of a limited set of standard dialog boxes to give information to the user or request answers to simple “Yes/No” type questions.

```
messagebox(style=alert) "Matrix is Singular"  
messagebox(style=yesno,status=ok) "Save to File?"
```

The first of these is an “alert,” which simply displays the message and waits for the user to dismiss the dialog before continuing. The second includes “Yes” and “No” buttons. The variable OK will be 1 if the user clicked “Yes” and 0 if “No.”

Using USERMENU and SELECT

The example below uses the **USERMENU** and **SELECT** instructions to create a menu-driven program for specifying and estimating a least squares regression model. The **USERMENU** instruction creates the menu and controls program flow, while the **SELECT** command allows the user to select dependent and independent variables from all series currently available.

Before the user can run a regression, she must select a dependent variable and a set of regressors. Thus, we disable the “Run Regression” item until those two steps have been done. We use check marks to indicate which of the first steps the user has done. A version of this program is included with RATS, in the file `OLSMENU.RPF`.

Note, by the way, that we started with a *much* more modest version of this (without the check marks and item disabling and enabling), working up to the final version.

The first few lines of this program are specific to the data set being used. They define the frequency, starting and ending date, read in the data, and do the necessary transformations. To use the OLS menu program with your own data, simply modify or add to these lines as needed for your data set

```
calendar(a) 1922
allocate 1941:1
open data fooddata.dat
data(format=free,org=obs) / $
    foodcons prretail dispinc foodprod prfarm
set trend = t
set avgprices = (prfarm+prfarm{1})/2.0
set logconsump = log(foodcons)
set logincome = log(dispinc)
set logprice = log(prretail)
```

The following code is the main part of the OLSMENU program. After executing these lines, a menu called “OLS” is added to the menu bar. The user must select a dependent variable and a list of independent variables before they can choose the “Run Regression” operation, which does the OLS regression. The user can change the variables and re-run the regression as desired. Select “Done” to remove the menu from the menu bar and return control to “ready” mode.

```
compute depvar=0,regressors=0
compute dep_item = 1, ind_item = 2, run_item = 3, done_item = 4

usermenu(action=define,title="OLS") $
    1>>"Select Dependent Variable" $
    2>>"Select Regressors" $
    3>>"Run Regression" $
    4>>"Done"
```

Disable the 'Run Regression' item:

```
usermenu(action=modify,enable=no) run_item
```

Start a Loop and activate the menu

```
loop
```

```
usermenu
```

```
if %menuchoice==1 {
```

With one parameter, SELECT allows only one choice, returning it here in the variable DEPVAR.

```
select(series,status=ok) depvar
```

If user made a selection, check the DEP_ITEM:

```
if ok {
```

```
    usermenu(action=modify,check=yes) dep_item
```

If we have both a dependent variable and regressors enable the 'Run' item:

```
    if depvar>0.and.regressors>0
```

```
        usermenu(action=modify,enable=yes) run_item
```

```
    }
```

```
}
```

```
if %menuchoice==2 {
```

With two parameters, SELECT allows multiple selections, returning the number of choices in REGRESSORS and the list of choices in REGLIST.

```
select(series,status=ok) regressors reglist
```

```
if ok {
```

```
    usermenu(action=modify,check=yes) ind_item
```

```
    if depvar>0.and.regressors>0
```

```
        usermenu(action=modify,enable=yes) run_item
```

```
    }
```

```
}
```

```
if %menuchoice==3 {
```

```
    linreg depvar
```

```
    # constant reglist
```

```
}
```

```
if %menuchoice==4
```

```
    break
```

```
end loop
```

```
usermenu(action=remove)
```

This will allow the user to run regressions with various combinations of regressors and dependent variables until they select the "Done" item from the OLS menu.

Using QUERY and MEDIT

QUERY and **MEDIT** are useful for obtaining numerical or character information from a user. **QUERY** is used to get a limited amount of information, usually a single number or string. **MEDIT** gets an entire array. **MEDIT** is also useful for displaying a set of results in matrix form. See page UG–497.

QUERY pops up a dialog box with a prompt for the user. This is a “modal” dialog, so the program will wait until the user is done with it before continuing. **QUERY** can be set to accept real values, integers or strings. You need to make sure that any variable you want filled in has the type that you want. If you need an integer, **DECLARE** it as an **INTEGER**. With the **VERIFY** option, you can check that the user has entered valid information, and prevent him from exiting the dialog until the input is corrected, or he cancels. You should *always* include a **STATUS** variable, and check it to see if the user has cancelled.

MEDIT pops up a “spreadsheet” with cells for a (real-valued) matrix. If you are using it for input, you should use the options **EDIT** and **MODAL** to ensure that the user fills in the matrix before the program continues. The matrix should be **DECLARE**’d and **DIMENSION**’ed first. You *can’t* request an array with variable dimensions—they must be known in advance. It’s usually a good idea to zero out the array first.

The following takes the most recent regression and requests information to do a set of linear restrictions on it. (Similar to what the *Regression Tests* wizard does). The **QUERY** asks for the number of restrictions. The user then fills in the restrictions in rows of a matrix using **MEDIT**. The **HLABELS** option on the **MEDIT** is set up to label each column with the regressor label. The final column is labeled with =.

```
procedure TestRestrict
  local integer nrestr i
  local vect[strings] hlabels rlabels
  local rect r capr
  local vect lowr
  local integer i
  *
  * Ask for the number of restrictions, make sure the
  * value is positive and no bigger than %NREG.
  *
  query(prompt="How Many Restrictions?",status=ok,$
    verify=nrestr>0.and.nrestr<= %nreg,errmessage="Illegal Value") $
    nrestr
  if .not.ok
    return

  dim r(nrestr,%nreg+1)
  compute r = %const(0.0)
  dim hlabels(%nreg+1)
  compute rlabels = %eqnreglabels(0)
```

```
do i=1,%nreg+1
  if i<=%nreg
    compute hlabels(i)=rlabels(i)
  else
    compute hlabels(i)=" "
  end do i
medit(modal,edit,hlabels=hlabels) r
compute capr=%xsubmat(r,1,nrestr,1,%nreg)
compute lowr=%xcol(r,%nreg+1)
mrestrict nrestr capr lowr
end test
```

Using MENU and CHOICE

MENU was originally designed to handle program flow under user control, but **USERMENU** now does that in a more flexible way. However, **MENU** still has its uses. It displays a dialog box with radio buttons for selecting one and only one of the choices provided by the **CHOICE** blocks. The string on the **MENU** instruction is the overall prompt for the choices. Each **CHOICE** instruction is followed by the instruction or block of instructions to be executed if that is chosen. If there is more than one instruction controlled by a **CHOICE**, enclose the group within { }.

Combining **MENU** with **SOURCE** is an easy way of controlling a set of steps in a procedure. For example:

```
loop
  menu "What now?"
  choice "Identification"
    source identify.src
  choice "Estimation"
    source estimate.src
  choice "Diagnostics"
    source diagnost.src
  choice "Forecast"
    source forecast.src
  choice "end"
    halt
  end menu
end loop
```

will keep executing the menu until **END** is chosen, with each other choice running the instructions on a source file.

Using DBOX

The **QUERY** command discussed earlier is useful for getting a single piece of user input via a simple dialog box. The **DBOX** command is a much more powerful instruction for creating custom dialog boxes. It can generate dialogs with multiple fields, and supports many different types of dialog box elements, including simple text labels (used to identify fields or present information), text fields to input numbers or strings, check boxes to turn settings on or off, radio buttons to select from a list of choices, and several kinds of “list boxes” allowing the user to select one or many items from lists of series or other information.

To create a dialog box, you need at least three **DBOX** instructions. The first command uses the option **ACTION=DEFINE** to begin the definition of the dialog box:

```
dbox(action=define)
```

This is followed by one or more additional **DBOX** commands with the (default) **ACTION=MODIFY** option to add the desired fields to the dialog. For example, this adds a checkbox field titled “Graph?”. The returned value (1 for on, 0 for off) will be stored in **GRAPHYESNO**:

```
dbox(action=modify, checkbox="Graph?") graphyesno
```

Finally, use the **ACTION=RUN** option to display the dialog box:

```
dbox(action=run)
```

Below is a simple example, taken from the **@RegHeteroTests** procedure.

```
procedure RegHeteroTests  
local integer lm exlm  
local vect[integer] selreg  
local series usq logusq  
local rect[integer] fulltable seltable  
local integer nselect i j
```

Initiate the dialog box, and supply a title:

```
dbox(action=define,title="Regression-Based Tests")
```

*Add a static text field to the dialog (note that we omit the **ACTION** option because **ACTION=MODIFY** is the default).*

```
dbox(stext="Which Tests Do You Want?")
```

*Add two check boxes, one for each of the two available tests. The user can turn on either or both boxes. The settings of the check boxes (1 for on, 0 for off) will be stored in **LM** and **EXLM**, respectively.*

```
dbox(checkbox="u^2 form") lm  
dbox(checkbox="log u^2 form") exlm
```

Now add a title for the list of regressors field. The SPECIAL option puts an extra space between the previous field and this new text field.

```
dbbox(stext="Select Regressors",special=gap)
```

Display a scrolling list of the regressors from the most recent regression. The user can select one or more of these.

```
dbbox(scroll,regressors) selreg
```

Activate the dialog box. If the user selects "Cancel" in the dialog box, the status check routine will exit from the procedure.

```
dbbox(action=run,status=ok)  
if .not.ok  
  return
```

The remaining commands use the information collected from the dialog box to perform the selected tests.

```
* How many regressors were selected?
```

```
compute nselect=%rows(selreg)
```

```
* Pull them out from the regressor table
```

```
compute fulltable=%eqntable(0)
```

```
dim seltable(2,nselect)
```

```
ewise seltable(i,j)=fulltable(i,selreg(j))
```

```
* Create the transformed residuals series
```

```
set usq = %resids^2
```

```
set logusq = log(usq)
```

```
if lm {
```

```
  linreg(noprint) usq
```

```
  # %rlfromtable(seltable)
```

```
  cdf(title="LM Test on U^2") chisqr %trsquared nselect-1
```

```
}
```

```
if exlm {
```

```
  linreg(noprint) logusq
```

```
  # %rlfromtable(seltable)
```

```
  cdf(title="LM Test on log(u^2)") chisqr %trsquared nselect-1
```

```
}
```

```
end RegHeteroTests
```

See the description of **DBOX** for complete details on this instruction.

15.5 Handling Output

Using REPORT

REPORT is a very useful tool for building up tabled output. It's covered in Section 1.9. When you create a report inside a procedure, it's a good idea to declare it as a **LOCAL REPORT**. This avoids conflicts with reports that the user might be generating. If you do that, you need **USE** options on all the **REPORT** instructions. For instance, this is (part of) `CVSTABTEST.SRC`:

```
local report sreport
report (use=sreport, action=define, title=1title)
report (use=sreport, atrow=1, atcol=1, span) 1title
report (use=sreport, atrow=2, atcol=1) "Fluctuation Statistic" sstat
```

Using OPEN(WINDOW)

The **OPEN** instruction with the option **WINDOW** creates a new text window into which you can direct output. With the **CHANGE** instruction, you can direct *all* subsequent output to this window, or, with **UNIT** options on instructions like **DISPLAY** and **COPY**, you can put selected information into it. For example:

```
open(window) tempout "Second Window"
change output tempout
display "This will go to the tempout window"
change output screen
display "This will go to the main output window"
display(unit=tempout) "This will also go to tempout"
```

The **OPEN** instruction creates a window whose unit name is `TEMPOUT`. The window title (which is what appears on the title bar and in the *Window* menu) is “Second Window.” The first **CHANGE OUTPUT** switches output to that window; the second switches it back to the standard output window (whose unit name is always `SCREEN`).

Graph Window Titles

By default, **RATS** simply labels new graph windows by number, for instance, “Graph.01” and “Graph.02”. The **WINDOW** options on **SPGRAPH**, **GRAPH**, **SCATTER**, **GBOX** and **GCONTOUR** allow you to assign your own titles to the graph windows. These titles appear on the title bar of the window, and in the *Window* menu.

For procedures which produce several graphs, assigning meaningful window titles to each graph can be very helpful to the user, particularly for selecting a graph from the *Window* menu.

```
graph(window="Kalman filter/smooth") 3
# lgnp
# hpsmooth
# hpfilter
```


Spreadsheet Windows

Some instructions (**PRINT** and most of the forecasting instructions) include a **WINDOW** option, which allows you to direct their output to a spreadsheet style window, rather than a standard text window. This will give you a separate window on the screen for each of these operations. While these windows are “read-only,” the information in them can easily be transferred into spreadsheet programs or word processors by using *File—Export...* or using the *Cut* and *Paste* operations.

MEDIT

We looked at **MEDIT** (Matrix EDIT) in “Using QUERY and MEDIT” on page UG–492 as an instruction for getting information *into* RATS. It is also handy for displaying information *from* RATS. If you want to display the contents of a matrix, use **MEDIT** with the options **NOEDIT** and **NOMODAL**. With the **NOMODAL** option, **MEDIT** simply displays the information and continues. To help the user find the desired information, **MEDIT** has a **WINDOW** option for labelling the window on the title bar and in the Window menu. From the **MEDIT** window, the user can export the data, or cut and paste it into another program.

MEDIT is most useful when you need to display a single array. **REPORT** is more flexible when the data are generated across several instructions.

Using INFOBOX

INFOBOX allows you to keep the user informed about the progress of a time-consuming operation. While it can be used to post a simple message, its main purpose is to display a progress bar (and a guess at the time to completion). Note, by the way, that **INFOBOX** behaves quite differently from **MESSAGEBOX**. An **INFOBOX** displays a window on the screen which requires no response from the user—it removes itself from the screen under the control of your program. A **MESSAGEBOX**, on the other hand, stops the execution of the program until the user answers the question it asks, or, in the case of an alert box, acknowledges having seen it by clicking OK.

The following displays a progress bar for an operation which takes **NDRAWS** draws.

```
infobox(action=define,progress,lower=1,upper=ndraws) $
  "Gibbs Sampler"
do draw=1,ndraws
  ( instructions which actually do things)
  infobox(current=draw)
end do draw
infobox(action=remove)
```


16. Simulations, Bootstrapping, and Monte Carlo Techniques

RATS can handle easily a wide variety of Monte Carlo experiments and simulations. With its many programming features plus a wide variety of probability functions, it has the needed flexibility, since each application of these techniques tends to have its own special characteristics. Simulation-based estimators have become standard practice in the profession, and it's important to be comfortable with them in doing empirical work.

The emphasis here is on demonstrating the tools available, along with several fully worked examples which can provide a guide for your own work. Our textbook example folder includes Koop's (2003) *Bayesian Econometrics* which add quite a few more examples.

If you need more, we would strongly recommend the RATS *Bayesian Econometrics* course, which goes into much greater technical detail. Quite a bit of the content of the *Vector Autoregressions*, *Switching Models and Structural Breaks* and *State-Space and DSGE* courses also use techniques introduced here. For more information on those, check out

https://estima.com/courses_completed.shtml

Simulations

Rejection Method

Monte Carlo Integration

Importance Sampling

Gibbs Sampling

Metropolis-Hastings

Bootstrapping

Randomization

16.1 Bookkeeping

Overview

RATS provides several instructions, including **SIMULATE**, **FORECAST**, and **BOOT**, and a number of built-in functions, such as **%RAN** and **%UNIFORM**, which make it possible to implement a wide variety of Monte Carlo experiments and simulations. We will discuss these special instructions and functions in detail later on. First, let's look at the basic bookkeeping techniques that will be a part of almost any bootstrapping or simulation task.

Bookkeeping

Usually, it is a fairly simple task to set up for a single draw from your model. However, there is very little call for the results of a single draw—the typical goal is an estimate of the probability of an event, or a moment of a distribution. To compute these, you need to loop over the code performing the simulations and do some bookkeeping with the results. RATS provides the following tools to accomplish this:

- **DO** for looping over a block of instructions. RATS provides several other looping functions (**DOFOR**, **LOOP**, **WHILE**, and **UNTIL**), but **DO** is the one most often used for simulations.
- **COMPUTE** for updating counters and totals.
- **EWISE** for general record keeping with matrices.
- **SET** and **GSET** for updating counters and totals with series.

The examples below use techniques described in the next section for generating the random draws.

Using PARMSETS

Even if you're not actually doing non-linear estimation which requires a **PARMSET**, you can benefit from using them to organize your statistics. For instance, suppose you're running a loop and you need to keep track of **M1**, **M2**, **M3** and **M4**. If you do

```
nonlin(parmset=simstats) m1 m2 m3 m4
```

then the function **%PARMSPEEK(SIMSTATS)** will return a 4-vector with the current values of the four variables. You can thus do the calculations using statistics on the **VECTOR** rather than having to program up bookkeeping on the separate variables. When you're done, you can also use **%PARMSLABELS(SIMSTATS)** to get the labels for output.

Note that while, in this case, these are all scalars, they could be a mix of scalars and matrices, just like any other **PARMSET**.

Progress Indicators

Before you do 10,000 replications on a model, it's a good idea to start with a more modest number and make sure that the program is doing what you want. Write your code using a variable like `NDRAWS` to represent the number of desired draws so you can change it easily. When you're ready for a full run, keep in mind that these methods are called, with good reason, *computationally intensive*. Even with a fast computer, it can sometimes take minutes, or even hours, to run enough replications to get results at the desired level of accuracy. (In general, accuracy improves with the square root of the number of draws.) We use the **INFOBOX** instruction in these situations to give an idea of how things are going.

```
infobox(action=define,progress,lower=1,upper=ndraws) "Progress"
do draw=1,ndraws
  .... instructions ...
  infobox(current=draw)
end do draw
infobox(action=remove)
...
```

The *Help—Pseudo-Code Generator—Monte Carlo/Gibbs Loop* wizard can be used to set up this type of loop so all you have to do is fill in the details of the calculation itself.

Computing a Probability

To compute a probability, you need to keep track of the number of times an event occurred, and divide by the number of draws that you made. You can use an `INTEGER` variable, or entries of a series or array, to hold this information. For instance, the following uses the **SIMULATE** instruction to generate simulations of a GDP model, and uses the simulated data to compute the probability distribution for the onset of the first recession (defined as two consecutive quarters of decline in GDP). The series `HIST` counts the number of times this is reached at a particular period.

```
smp1 2018:3 2020:4
set hist = 0.0 Set counters to 0
do draw=1,ndraws
  simulate(model=gdpmodel)
  do date=2018:3,2020:4
    if gdp(date)<gdp(date-1) .and. gdp(date-1)<gdp(date-2) {
      compute hist(date) = hist(date)+1
      break
    }
  end do date
end do draw
set hist = hist/ndraws
```

Computing Moments

You compute these using the same formulas you would for any other random sample. To estimate the mean, sum the generated values and divide by the number of draws. To estimate a variance, add up the squares, divide by the number of draws and subtract the squared estimate of the mean. For example, the following computes the expected value of the maximum and minimum values achieved for a log Normal diffusion approximated by simulation over a grid. In this, MU is the expected return per year, SIGMA the volatility, PERIOD the fractional part of a year being analyzed and BASEPRICE the original price. GRID is the number of equally-spaced time periods used in the simulation.

```
compute hdrift = (mu-.5*sigma^2)*(period/grid)
compute hsigma = sigma*sqrt(period/grid)

set price 1 grid = 0.0
compute totalmax=0.0,totalmin=0.0
do draw=1,ndraws
  compute pcond=baseprice
  do i=1,grid
    compute pcond = price(i) = pcond*exp(hdrift+%ran(hsigma))
  end do i
  compute totalmax = totalmax+%maxvalue(price),$
    totalmin = totalmin+%minvalue(price)
end do draw
disp "Maximum" totalmax/ndraws "Minimum" totalmin/ndraws
```

The same thing using PARMSETS (page UG–500) for bookkeeping (which doesn't make much sense for just two variables, but illustrates the idea):

```
nonlin(parmset=minmax) minv maxv
compute total=%zeros(%size(%parmspeek(minmax)),1)
do draw=1,ndraws
  compute pcond=baseprice
  do i=1,grid
    compute pcond = price(i) = pcond*exp(hdrift+%ran(hsigma))
  end do i
  compute maxv=%maxvalue(price),minv=%minvalue(price)
  compute total=total+%parmspeek(minmax)
end do draw
compute total=total/ndraws
report(action=define)
report(atrow=1,atcol=1,fillby=cols) %parmslabels(minmax)
report(atrow=1,atcol=2,fillby=cols) total
report(action=show)
```

Computing Fractiles (Quantiles)

Fractiles (quantiles) are needed when you are computing critical values for a test. They are also useful when there is some concern that a distribution is asymmetric—if so, standard error bands computed using the mean and variance may give a misleading picture of the distribution. Unfortunately, there are no sufficient statistics for the quantiles of a distribution—you can't summarize the data by keeping a running sum as was done in the two examples above. To compute these, you have to save the entire set of draws and do the calculations later. Set up a vector, or possibly a matrix of vectors if you're keeping track of more than one random variable, and dimension the vectors to the number of draws that you're making. The instruction **STATISTICS** with the **FRACTILES** option computes a standard collection of quantiles for a data series. You can also use the `%FRACTILES(array, fractiles)` function, which takes as parameters an array of data and a vector of desired quantiles, and returns those quantile points in the data set. Just as a warning: if you do a *very* large number of draws (100,000 and up), computing the sample quantiles can actually take more time than computing the statistics in the first place.

This example produces simulations of **TBILLS**. `%FRACTILES` is used to find the 25th and 75th percentiles.

```
dec vect sims(ndraws)
do draw=1,ndraws
    simulate(model=ratemodel)
    compute sims(draw) = tbills(2018:12)
end do draw
compute ratefract = %fractiles(sims,||.25,.75||)
disp "25-75%iles of T-Bills" ###.### ratefract(1) "to" ratefract(2)
```

Computing a Density Function

Just as with quantiles, you need to save *all* the simulated values. Here, they must be saved in a data series, which you process using the instruction **DENSITY**. For instance, the following computes an estimated density function for the autoregressive coefficient when the data actually follow a random walk. The coefficient estimates go into the series **BETAS**. The density is estimated, then graphed. The default bandwidth for **DENSITY** has certain general optimality properties, but we've found that it tends to be too narrow for densities generated from Monte Carlo, so it helps to use the **SMOOTHING** option on **DENSITY** to slightly increase the bandwidth above the default.

```
set betas 1 ndraws = 0.0
do i=1,ndraws
    set(first=0.0) y = y{1}+%ran(1.0)
    linreg(noprint) y
    # y{1}
    compute betas(i) = %beta(1)
end do i
density(smoothing=1.50) betas / xb fb
scatter(style=lines)
# xb fb
```

Computing a Covariance Matrix of Coefficients

A covariance matrix of coefficients is a special case of a moment estimator. There are, however, some special tools available in RATS to help you with this. The following set up can be used with no modifications other than replacing the code for generating draws.

```
compute [vector] betamean = %zeros(number of regressors,1)
compute [vector] betacmom = %zeros(number,number)
do i=1,ndraws
```

```
    ... code to compute draw for coefficients %BETA ...
```

```
        compute betamean = betamean+%beta
        compute betacmom = betacmom+%outerxx(%beta)
end do i
compute betamean = betamean/ndraws
compute betacmom = betacmom/ndraws-%outerxx(betamean)
```

You can print the regression with the new coefficient vector and covariance matrix by using **LINREG** with the **CREATE**, **COEFFS**, and **COVMAT** options. You must also use the option **FORM=CHISQUARED**, because **BETACMOM** is a direct estimate of the covariance matrix:

```
linreg(create,covmat=betacmom,coeffs=betamean,form=chisq)    . . . . .
```

Weighted Draws

The examples examined so far assumed that all draws receive equal weight. There are techniques described later in the chapter which require placing unequal weights on the draws. The adjustment required for this is fairly simple when you're computing means and other moments: just sum the weight of a draw times the value obtained on that draw, and, at the end, divide by the sum of the weights.

It's not so simple with the quantiles and the density function. In addition to saving all the simulated values (do *not* multiply by the weights), you need to save the weights for each draw. On **DENSITY**, include the option **WEIGHTS=series of weights**. This will do the appropriate weighting of observations. To compute quantiles, use the **%WFRACTILES** function instead of **%FRACTILES**. This adds a third parameter which is the vector or series of weights. Note that the computation of quantiles for weighted draws can be quite time-consuming. If you do many draws, it may even take longer than generating the draws in the first place.

```
density(weights=weights,smpl=weights>0.00,smoothing=1.5) persist $
    1 draws xx dx
scatter(style=line,$
    header="Density of Persistence Measure (alpha+beta)")
# xx dx
```


16.2 Bootstrapping and Simulation Tools

In addition to the standard bookkeeping routines, most simulation and bootstrapping programs will make use of one or more of the specialized instructions and functions described below. These are used to generate the random draws and simulations that will be used to compute the moment statistics, probabilities, etc.

SIMULATE, FORECAST and UFORECAST

The instruction **SIMULATE** forecasts an equation or system of equations, automatically drawing shocks from a Normal distribution. It can handle linear equations or non-linear formulas. The syntax for **SIMULATE** is very similar to **FORECAST**, except there are extra options for inputting the covariance matrix for the Normal draws (*CV=covariance matrix* or *FACTOR=factor of covariance matrix*).

You can do random Normal simulations for a *single* equation using **UFORECAST** with the **SIMULATE** option. The variance for the draws for **UFORECAST** is the one saved with the equation or regression being forecast.

You can also generate data using a **FORECAST** instruction, by including an **INPUT**, **SHOCKS**, **PATHS**, or **MATRIX** option. While **SIMULATE** can only draw from a single multivariate Normal distribution, **FORECAST** with the **MATRIX** or **PATHS** options will let you feed any set of shocks into your forecasts.

Virtually every simulation or bootstrapping project requires a loop of some kind (usually a **DO** loop) and various bookkeeping instructions, as was described in the previous section. Most of the bookkeeping is handled using **COMPUTE**, **SET**, **GSET** and **EWISE** instructions. The following is a stylized example of the use of **SIMULATE**. This has a five equation model, with three structural equations (formulas **X1EQ**, **X2EQ** and **X3EQ**) and two identities (**X4ID** and **X5ID**).

```
vcv(matrix=v)
# resids1 resids2 resids3
group(vcv=v) model5 x1eq>>fx1 x2eq>>fx2 x3eq>>fx3 $
    x4id>>fx4 x5id>>fx5
smpl 2011:1 2011:12
do draw=1,ndraws
    simulate(model=model5)
    ... bookkeeping ...
end do draw
```

Built-In Functions for Generating Random Draws

RATS provides the following functions drawing from univariate real distributions:

<code>%ran(x)</code>	Returns a draw from a $\text{Normal}(0, x^2)$
<code>%uniform(x1, x2)</code>	Returns a draw from a $\text{Uniform}(x1, x2)$
<code>%ranbeta(a, b)</code>	Random beta with shape parameters a and b .
<code>%ranchisqr(df)</code>	Random chi-squared with df degrees of freedom
<code>%rangamma(r)</code>	Random gammas with shape parameter r
<code>%rant(df)</code>	Random Student- t with df degrees of freedom.
<code>%ranlognormal(mu, sigma)</code>	Random log normal with given mean and standard deviation.
<code>%rantruncate(m, s, l, u)</code>	Random $\text{Normal}(m, s^2)$ truncated to the interval l to u .

You can use **SET** to fill elements of a series using random draws. For example:

```
set u = %ran(sqrt(%seesq))
```

creates U as a series of random Normals with variance `%SEESQ`.

You can use **COMPUTE** for scalars and arrays. For arrays, you generally need to declare and dimension the array before using **COMPUTE**. However, there is a special function `%RANMAT` for the most common operation of this type—it both dimensions a matrix and fills it with standard Normals.

```
compute e = %ran(1.0)
declare symmetric s(4,4)
compute s = %rangamma(nu)
compute r = %ranmat(5,5)
```

E will be a single $N(0,1)$ draw, S will be a 4×4 **SYMMETRIC** filled with random gammas, and R will be a 5×5 **RECTANGULAR** filled with $N(0,1)$ draws.

Inverse Method

If a random variable has an invertible cumulative distribution function, you can draw variates easily by just applying the inverse to a uniform $(0,1)$ draw. For instance, the logistic has distribution function

$$1/(1 + \exp(-x))$$

If y is a $U(0,1)$ draw, then solve $1/(1 + \exp(-x))$ to get $x = \log(y/(1 - y))$. Since the distribution is symmetric, you can also get logistic draws with the slightly simpler $x = \log((1/y) - 1)$.

The following code generates a vector of 100 draws from a standard logistic:

```
dec vect draws(100)
ewise draws(i) = log(1/%uniform(0,1)-1)
```

Do *not* do `LOG(%UNIFORM(0,1)/(1-%UNIFORM(0,1)))`. Each use of `%UNIFORM` will give a different value, when we want the same one in both places. The original inverse formula (with the y 's occurring in two places) can be done either in two steps:

```
ewise draws(i) = %uniform(0,1)  
ewise draws(i) = log(draws(i)/(1-draws(i)))
```

or in one by using a comma to separate the two parts of the calculation:

```
ewise draws(i) = y=%uniform(0,1), log(y/(1-y))
```

Another example of the use of the inverse method is the Cauchy, which can be generated using

```
ewise draws(i) = tan(%pi/2.0*%uniform(-1.0,1.0))
```

Direct Method

Many random variables can be obtained by some function of Normal, uniform, chi-squared, gamma and beta random variables. For instance, another way to get draws from a Cauchy is to take the ratio of two standard Normals:

```
ewise draws(i) = %ran(1.0)/%ran(1.0)
```

The inverse method is slightly more efficient in this case, although most efficient is to use the fact that a Cauchy is a t with 1 degree of freedom and use `%rant(1)`.

Multivariate Normal

To draw a multivariate Normal with mean vector $\mathbf{X}$ and covariance matrix Σ , it is necessary first to get a factor of the Σ matrix: a matrix $\mathbf{F}$ such that $\mathbf{F}\mathbf{F}' = \Sigma$. Any such factor will do; the simplest one to obtain with RATS is the Cholesky factor computed using the RATS function `%CHOL` or `%DECOMP`. The (mean zero) multivariate Normal can then be drawn using the function `%RANMVNORMAL(F)`.

The following draws a single vector (`XDRAW`) from this distribution. If, as is typical, the actual draws would be inside a loop, the first two lines can be outside it:

```
dec rect f  
compute f = %chol(sigma)  
compute xdraw = x+%ranmvnormal(f)
```

Convenience Functions for Multivariate Normals

There are some specialized functions for drawing multivariate Normals whose means and covariance matrices are derived from standard calculations. The most important of these is `%RANMVPOST(H1,B1,H2,B2)`, which takes a pair of mean vectors ($\mathbf{B}$) and precision matrices ($\mathbf{H}$) and draws from the multivariate Normal posterior derived by combining them. By using this, you can avoid calculating the posterior mean and variance yourself.

Multivariate Student-*t*

To get a draw from a multivariate Student-*t*, use the function `%RANMVT (F, degrees)`. As with `%RANMVNORMAL`, the *F* gives a factor of the covariance matrix Σ , though here this is the covariance matrix of the underlying Normal. A draw from that multivariate Normal is divided through by the square root of a single draw from a chi-squared distribution with *degrees* degrees of freedom to give a draw for a multivariate *t*.

For example:

```
compute betau = xbase + %ranmvt(fxx,nuxx)
```

will draw a multivariate *t* with mean *XBASE*, degrees of freedom *NUXX*, and factor matrix *FXX*.

Wishart Distribution

A Wishart distribution is a multivariate generalization of a gamma and is used in modelling a covariance matrix. It specifies a density function for symmetric $N \times N$ matrices. The `%RANWISHART (N, x)` function in RATS takes two parameters: the first is the dimension of the output matrix (*N*) and the second is the degrees of freedom (*x*). It produces an $N \times N$ matrix whose expected value is *x* times an $N \times N$ identity matrix. More commonly, however, the “scale matrix” isn’t the identity, but a covariance matrix estimated from the sample. As is the case with the multivariate Normal, you start with a factor of this scale matrix. The two functions available for more general Wishart draws are `%RANWISHARTE (F, x)` and `%RANWISHARTI (F, x)`. The first draws a Wishart, the second an inverse Wishart. The latter is more commonly used, as it arises in standard Bayesian methods applied to multivariate regressions. The kernel of the density function from which `%RANWISHARTI` draws Σ is

$$|\Sigma|^{(-x-k-1)/2} \exp\left(-\frac{1}{2} \text{tr}\left((\mathbf{F}\mathbf{F}')^{-1} \Sigma^{-1}\right)\right)$$

where *k* is the dimension of Σ . For instance, if the posterior for Σ is

$$|\Sigma|^{-(T-p-k-1)/2} \exp\left(-\frac{1}{2} \text{tr}\left((T\hat{\Sigma})\Sigma^{-1}\right)\right), \text{ by inspection, } x = T - p, \text{ and } \mathbf{F}\mathbf{F}' = (T\hat{\Sigma})^{-1}.$$

If *T* is the number of observations from the most recent regression, and $\hat{\Sigma}$ is `%SIGMA`, the following will draw a matrix *W* from the posterior:

```
compute f = %decomp(inv(%nobs*%sigma))  
compute w = %ranwisharti(f,%nobs-p)
```

Random Directions (%RANSPPHERE)

The function `%RANSPPHERE(N)` draws a random “direction” in N space. It draws uniformly from the unit sphere in N dimensions, that is, from $\{x \in \mathbb{R}^N : \|x\| = 1\}$.

For example, the following draws a random point on the sphere in `NVAR` dimensions.

```
compute v1 = %ransphere(nvar)
```

Dirichlet Distribution

The Dirichlet distribution is a generalization of the beta to more than two choices; it’s used to model probabilities when there are three or more alternatives.

`%RANDIRICHLET(shapes)` is used to draw from this. `SHAPES` is an N -vector of positive numbers, and this draws an N -vector whose elements sum to one. The mean of each component is the ratio of its shape to the sum of all the shape parameters.

%RANLOGKERNEL()

In some cases, it’s necessary to know the density function at the random draw. If you arrange your calculation so that there is a single function call to one of the random functions, you can fetch the log of the kernel of the density for that draw using the function `%RANLOGKERNEL()`. In other words, this will work if you do `%RANMVT` to draw a random multivariate Normal, but won’t if you do it yourself by dividing a random Normal by the square root of a random gamma—in that case, `%RANLOGKERNEL()` will just give you the value for the last of the component draws that you did.

Note that, unlike all the functions which return the densities or log densities themselves, `%RANLOGKERNEL()` will leave out integrating constants that don’t involve any of the parameters—they aren’t needed to compare the relative probabilities of two draws.

Monte Carlo Test Simulation: MONTEARCH.RPF example

MONTEARCH.RPF does 10000 repetitions of a test for ARCH (Engle, 1982). The model simulated is

$$(1) \quad y_t = \beta_0 + \beta_1 X_t + u_t$$

$$(2) \quad u_t \sim N\left[0, \sigma^2 \left(1 + \alpha u_{t-1}^2\right)\right]$$

Because the (conditional) variance of u is not constant, the draws for the model must be generated using the non-linear systems solution procedure. **SIMULATE** only draws from a fixed distribution, so we have to add a third equation to the model which computes a standard Normal variate V . FRMLS for u and y are then “identities.”

You might think that you could simplify this three-equation setup by using a %RAN function directly in the UDEF FRML. This, however, will draw a new random number at every pass through the solver, so the model will never converge. **SIMULATE** draws only a single set of values and solves the model given those.

This program generates the process over the period 2 through 150, starting with $U(1) = 0$. Since this starts “cold” from an arbitrary value, we use only the last 100 observations to compute the test statistics. The 50 discarded observations are known as the *burn-in*, a phrase which will also be used later when we need to discard draws.

The test-statistics (TR^2) are stored in the series TESTSTAT, which runs from 1 to NDRAWS. At the end, a **STATISTICS** instruction computes its quantiles.

```
compute ndraws=10000,useobs=100,endobs=50+useobs
all endobs
set x 1 endobs = t

compute sigma=1.0,alpha =.20
frml(variance=sigma^2) vdef v = 0.0
frml(identity) udef u = v*sqrt(1+alpha*u{1}^2)
frml(identity) ydef y = 2.0 + 3.0*x + u
group archmod vdef udef ydef>>y

set u 1 1 = 0.0
set teststat 1 ndraws = 0.0
do draw=1,ndraws
    simulate(model=archmod) 2 endobs-1 2
    linreg(noprint) y 51 endobs resid
    # constant x
    set usquared 51 endobs = resid^2
    linreg(noprint) usquared 52 endobs
    # constant usquared{1}
    compute teststat(draw) = %trsquared
end do draw
stats(fractiles,nomoments) teststat
```

16.3 Rejection Method

Sometimes, distributions can't be derived as simple functions of the elementary distributions such as normal, gamma and beta. For instance, posterior densities can be the product of two distributions that can't easily be combined. The Rejection method is a tool which can often be used in univariate cases to generate draws. The Rejection method (or Acceptance–Rejection or Acceptance, all three terms are used) is used within RATS to generate draws from the normal and gamma distributions.

Suppose that we need draws from a $N(\mu, \sigma^2)$ truncated to the interval $[a, b]$. (Note: this is for illustration. RATS has a %RANTRUNCATE function to do this in practice). An obvious way to do this is to draw $N(\mu, \sigma^2)$ deviates, then reject any that fall outside of $[a, b]$. The accepted draws would have the desired distribution, but this process will be inefficient if the probability of the Normal draw falling in $[a, b]$ is fairly low.

An alternative is to draw a random number x from $U(a, b)$. Compute the ratio a between $f_N(x | \mu, \sigma^2)$ and the maximum that $f_N(\bullet | \mu, \sigma^2)$ achieves on $[a, b]$ and accept x with probability a , otherwise draw again. This is most easily done with the %RANFLIP(p) function, which returns 1 if a random $U(0,1)$ is less than p and 0 otherwise. This process trims away the uniform draws to match the shape of the Normal. This will be more efficient if the density function is fairly constant on $[a, b]$, so that the comparison ratio is close to one and almost all draws are accepted.

A stylized procedure for doing the rejection method is

```
loop
  compute x = draw from the proposal density
  compute a = acceptance function
  if %ranflip(a)
    break
end loop
```

The value of x when the loop breaks is the draw. If you know that you have set this up correctly, and you know that the probability of accepting a draw is high, then you can adopt this code as is. However, this will loop *until* a draw is accepted, and, if you make a bad choice for the proposal density, you might end up with an acceptance function which produces values very close to zero, so this could loop effectively forever. A “fail-safed” alternative is

```
do try=1, maximum_tries
  compute x = draw from the proposal density
  compute a = acceptance function
  if %ranflip(a)
    break
end do try
```

This will quit the loop if it fails to accept a draw in MAXIMUM_TRIES. How that limit should be set will depend upon the situation. If you set it to 100 and find that you're routinely hitting the limit, there is probably a better way to generate the draws.

To apply the rejection method to the density f , you need to be able to write

$$(3) \quad f(x) \propto \alpha(x)g(x)$$

where the proposal density $g(x)$ is one from which we can conveniently draw and

$$(4) \quad 0 \leq \alpha(x) \leq 1$$

In our two examples for drawing from the truncated Normal, these are (in order):

$$(5) \quad g = N(\mu, \sigma^2), \alpha = I_{[a,b]}$$

$$(6) \quad g = U(a, b), \alpha = f_N(x | \mu, \sigma^2) / \max_{[a,b]} f_N(\bullet | \mu, \sigma^2)$$

Any density for which $f(x)/g(x)$ is bounded will (theoretically) work as a proposal. However, as a general rule, you want to choose a distribution which has tails as thick as (or thicker) than the target density. For instance, if you use a Cauchy for g , you will get a small percentage of very extreme draws. If f is a thin-tailed distribution, f/g will be very small out there, and the extreme draws will be rejected. In other words, the rejection procedure thins out the Cauchy tails by rejecting most tail draws. If, however, f/g is quite large in the tails, the only way we can “thicken” the tails is by accepting the tail draws and rejecting most of the draws near the mode.

So long as $f(x)/g(x)$ is bounded, we can always choose

$$(7) \quad \alpha(x) = (f(x)/g(x)) / \max(f(\bullet)/g(\bullet))$$

(where the max is taken over the support of f) and that will give the most efficient rejection system for that particular choice of g . However, the rejection method is often applied in situations where the distribution changes slightly from one draw to the next. It's quite possible that you would spend more time computing the maximum in order to achieve “efficient” draws than it would take to use a simpler but less efficient choice of $\alpha(x)$. For instance, if we look at (6) above, we could also use

$$(8) \quad g = U(a, b), \alpha = \exp(-(x - \mu)^2 / 2\sigma^2)$$

If $\alpha \in [a, b]$, this will be exactly the same as (6). If it isn't, then this will reject more draws than (6), since $\alpha(x)$ is strictly less than one for all the x 's that will be generated by the proposal density. Whether the extra work of finding the normalizing constant in (7) is worth it will depend upon the situation. Here, the added cost isn't high because the maximum will be at one of $\{a, b\}$. If, however, f and g are non-trivial density functions, the maximum might very well only be found by some iterative root-finding technique like Newton's method. Since finding the normalizing constant needs to be done just once per set of draws, the extra work will generally pay off when you need many draws from the same distribution, but not if you need just one.

Neither of these rejection systems is actually used by %RANTRUNCATE. Because %RANTRUNCATE can handle intervals that are unbounded, the second method wouldn't apply, and the first can be quite inefficient. Instead, it uses a different rejection system, using a truncated logistic as the proposal density.

16.4 Standard Posteriors

Standard Normal Linear Model—Coefficients

The natural prior for the coefficients in the model

$$(9) \quad \mathbf{y} = \mathbf{X}\beta + \mathbf{u}, \mathbf{u} | \mathbf{X} \sim N(0, h^{-1}\mathbf{I})$$

where h is the precision (reciprocal of variance) takes the form

$$(10) \quad \beta \sim N(\bar{\beta}, \bar{\mathbf{H}}^\dagger)$$

For many reasons, these priors are usually stated in terms of their precision ($\bar{\mathbf{H}}$) rather than the variance. $\bar{\mathbf{H}}^\dagger$ is the generalized inverse of $\bar{\mathbf{H}}$, which allows the prior to be uninformative (have “infinite” variance) in some dimensions or directions. The data evidence from (9) is summarized as

$$(11) \quad \beta \sim N(\hat{\beta}, h^{-1}(\mathbf{X}'\mathbf{X})^{-1}), \text{ where the precision is } \hat{\mathbf{H}} = h\mathbf{X}'\mathbf{X}$$

The posterior from combining (10) and (11) is

$$(12) \quad \beta \sim N\left(\left(\bar{\mathbf{H}} + \hat{\mathbf{H}}\right)^{-1} \left(\hat{\mathbf{H}}\hat{\beta} + \bar{\mathbf{H}}\bar{\beta}\right), \left(\bar{\mathbf{H}} + \hat{\mathbf{H}}\right)^{-1}\right)$$

RATS offers three convenience functions for generating draws from the distribution in (12): %RANMVPOST, %RANMVPOSTHB and %RANMVPOSTCMOM.

%RANMVPOST(H1, B1, H2, B2) takes the precision matrices (H1 and H2) and their corresponding means (B1 and B2) and generates a draw. %RANMVPOSTHB(H, HB) takes the combined precision matrix, and the combined $\mathbf{H}\beta$ matrix (which is often easier to compute than β itself). %RANMVPOSTCMOM(CMOM, h, H2, B2) assumes that CMOM is the cross moment matrix of $\mathbf{X}, \mathbf{y}$ ($\mathbf{X}$ variables first), h is the h (reciprocal of residual variance) from (9) and H2 and B2 are the prior precision and mean from (10).

Standard Normal Linear Model—Variance

The natural prior for the precision of $\mathbf{u}$ is

$$(13) \quad \nu s^2 h \sim \chi_\nu^2$$

(Note: s^2 is a scale parameter for the prior). The data evidence from (9) for h is summarized as

$$(14) \quad \left((\mathbf{y} - \mathbf{X}\beta)' (\mathbf{y} - \mathbf{X}\beta) \right) h \sim \chi_T^2$$

(13) and (14) combine to produce the posterior

$$(15) \quad \left((\mathbf{y} - \mathbf{X}\beta)' (\mathbf{y} - \mathbf{X}\beta) + \nu s^2 \right) h \sim \chi_{T+\nu}^2$$

If we call the term in parentheses `RSSPLUS`, we can get a draw for h with

```
compute hu = %ranchisqr(nu+%nobs)/rssplus
```

Note that this draws the *reciprocal* of the variance. That's what `%RANMVPOSTCMOM` wants as its second parameter.

The sum of squared residuals $(\mathbf{y} - \mathbf{X}\beta)'(\mathbf{y} - \mathbf{X}\beta)$ can be computed using `SSTATS` if you have the residuals already computed (into, say, the series `U`):

```
sstats / u^2>>rssbeta
```

In most cases, however, it will be easier to use the convenience function `%RSSCMOM(CMOM, Beta)`. Like `%RANMVPOSTCMOM`, `%RSSCMOM` works with the cross product matrix of $\mathbf{X}, \mathbf{y}$ ($\mathbf{X}$ variables first). A complete sequence of drawing h given β and drawing β given h is:

```
compute rssplus = nu*s2 + %rsscmom(%cmom,beta)  
compute hu      = %ranchisqr(nu+%nobs)/rssplus  
compute beta    = %ranmvpostcmom(%cmom,hu,priorh,priorb)
```

Multivariate Normal Regression (VAR)—Jeffreys Prior

Write a vector autoregression (or any multivariate regression with identical explanatory variables) as

$$(16) \quad \mathbf{y}_t = \beta \mathbf{X}_t + \mathbf{u}_t, t = 1, \dots, T$$

$\mathbf{X}_t$ is a p -vector of explanatory variables, $\mathbf{y}_t, \mathbf{u}_t$ are n -vectors (all these are column vectors) and β is the $p \times n$ matrix of coefficients. We use a matrix for β in this discussion as much as possible since that's the arrangement that is used by the functions `%MODELGETCOEFFS` and `%MODELSETCOEFFS`. The $\mathbf{u}_t$ are assumed i.i.d. over time with distribution $N(0, \Sigma)$. Let the OLS estimates for β and Σ be $\mathbf{B}$ and $\mathbf{S}$. With the standard diffuse (Jeffreys) prior of

$$(17) \quad f(\beta, \Sigma) \propto |\Sigma|^{-(n+1)/2}$$

the posterior is

$$(18) \quad \Sigma \sim IW\left[(T\mathbf{S})^{-1}, T - p\right] \text{ and, given } \Sigma,$$

$$(19) \quad \text{vec}(\beta) \sim N\left[\text{vec}(\mathbf{B}), \Sigma \otimes (\mathbf{X}'\mathbf{X})^{-1}\right]$$

Σ has an inverse Wishart distribution, which can be drawn using `%RANWISHARTI`. This takes as its arguments a factor of $(T\mathbf{S})^{-1}$ and the degrees of freedom. If the VAR is estimated using `ESTIMATE`, the following will generate a draw for Σ .

```
compute fwish    = %decomp(inv(%nobs*%sigma))  
compute wishdof = %nobs - %nreg  
compute sigmad  = %ranwisharti(fwish,wishdof)
```

To draw from (19), note that

$$(20) \quad (\mathbf{F}_\Sigma \otimes \mathbf{F}_{\mathbf{xx}})(\mathbf{F}_\Sigma \otimes \mathbf{F}_{\mathbf{xx}})' = \Sigma \otimes \mathbf{X}'\mathbf{X} \quad \text{where } \mathbf{F}_\Sigma \mathbf{F}_\Sigma' = \Sigma \quad \text{and} \quad \mathbf{F}_{\mathbf{xx}} \mathbf{F}_{\mathbf{xx}}' = (\mathbf{X}'\mathbf{X})^{-1}$$

so a factor of the (possibly) very large covariance matrix can be obtained from factors of the component matrices. And $\mathbf{F}_{\mathbf{xx}}$ is a function just of the observable data, and so can be computed just once. Given the component factors, the function `%RANMVKRON(FSIGMA,FXX)` will draw a matrix which is the “unvec”ed draw from (19) with a mean of zero. Adding the $\mathbf{B}$ matrix will give a draw for β . The following sequence draws Σ (as `SIGMAD`) and β (as `BETADRAW`). The first five lines depend only upon the data and can (and should) be done outside the simulation loop. Only the last three need to be done within it.

```
estimate
compute fwish    = %decomp(inv(%nobs*%sigma))
compute wishdof  = %nobs - %nreg
compute fxx      = %decomp(%xx)
compute betaols  = %modelgetcoeffs(varmodel)

compute sigmad   = %ranwisharti(fwish,wishdof)
compute fsigma   = %decomp(sigmad)
compute betadraw = betaols + %ranmvkron(fsigma,fxx)
```

Multivariate Normal Regression (VAR)—General Priors

The most time-consuming part of drawing a multivariate Normal vector of high dimension is factoring the covariance matrix. If we have a six variable VAR with 20 coefficients per equation, factoring the 120×120 covariance matrix and generating a draw based upon that takes roughly 30 times as long as using `%RANMVKRON` with the 20×20 component already factored. And this gap gets larger as the size of the covariance matrix increases. The Kroneker product form saves time both by avoiding factoring a full-sized matrix, and also by taking advantage of the structure of the matrix in multiplying out to get the final draw.

However, unless the prior on the coefficients takes a very specific form, the posterior covariance matrix will not have a convenient structure (see Kadiyala and Karlsson, 1997). And Σ will not have an unconditional distribution as it does in (18).

If we keep the standard diffuse prior for Σ , but now have the informative multivariate Normal prior for β :

$$(21) \quad f(\beta, \Sigma) \propto |\Sigma|^{-(n+1)/2} \exp \left(-\frac{1}{2} \text{vec}(\beta - \bar{\mathbf{B}})' \bar{\mathbf{H}} \text{vec}(\beta - \bar{\mathbf{B}}) \right)$$

the posterior for Σ is given by

$$(22) \quad \Sigma | \beta \sim IW \left((\mathbf{Y} - \mathbf{X}\beta)' (\mathbf{Y} - \mathbf{X}\beta), T \right)$$

and the posterior for $\text{vec}(\beta) | \Sigma$ is a multivariate Normal with precision

$$(23) \quad \Sigma^{-1} \otimes \mathbf{X}'\mathbf{X} + \bar{\mathbf{H}} \text{ and mean}$$

$$(24) \quad (\Sigma^{-1} \otimes \mathbf{X}'\mathbf{X} + \bar{\mathbf{H}})^{-1} \left((\Sigma^{-1} \otimes \mathbf{X}'\mathbf{X}) \text{vec}(\mathbf{B}) + \bar{\mathbf{H}} \text{vec}(\bar{\mathbf{B}}) \right)$$

While it's possible to apply %RANMVPOST to this, there is an even greater advantage in using a pair of cross-moment based functions to avoid unnecessary calculations. These are %SIGMACMOM(CMOM, B), which computes $(\mathbf{Y} - \mathbf{X}\beta)' (\mathbf{Y} - \mathbf{X}\beta)$ from (22) and %RANMVKRONCMOM(CMOM, SINV, H, B), which, given Σ^{-1} , generates a draw from the posterior given by (23) and (24). Note that if the B argument (the prior mean) in %RANMVKRONCMOM is $p \times n$, so will be the draw; while if B is stacked in vector form, the draw will be as well. The cross product matrix in this case should have the X variables first, then all the Y's. This is the order that CMOM(MODEL=varmodel) will create. The following will create BDRAW given SIGMAD and SIGMAD given BDRAW. The CMOM will be outside the loop, the others inside of it. Note that this is an example of Gibbs sampling (Section 16.7).

```
cmom(model=varmodel)
compute bdraw = %ranmvkroncmom(%cmom,inv(sigmad),hprior,bprior)
compute rssmat = %sigmacmom(%cmom,bdraw)
compute sigmad = %ranwisharti(%decomp(inv(rssmat)),%nobs)
```

Probabilities

The natural prior for the probability p in a binomial random variable is the beta. If you have data with n observations and k successes, and a beta prior with parameters a, b , the posterior is beta with parameters $a + k, b + n - k$. In most cases, the easiest way to get the count value k is with the instruction **SSTATS** applied to a logical or relational expression. For instance, the following counts (and puts into the variable COUNT) the number of times that S_{t-1} was 1 and S_t was 2. The number of observations in the sample, that is, the number of times $S_{t-1} = 1$, will be put in %NOBS. The **COMPUTE** then makes a draw from the posterior (here for the probability of staying in regime 1) combining this with a beta prior with parameters γ_{11}, γ_{12} .

```
sstats(smpl=(s{1}==1)) 2 * s==2>>count
compute p11 = %ranbeta(gamma11+%nobs-count,gamma12+count)
```

If there are more than two alternatives, you would use the Dirichlet distribution. %RANDIRICHLET returns a vector with the draw for the probabilities of each of the states. The same basic setup with three states is done with

```
sstats(smpl=(s{1}==1)) 2 * s==2>>count12 s==3>>count13
compute plx = %randirichlet(||gamma11+%nobs-count12-count13,$
gamma12+count12,gamma13+count13||)
```

16.5 Monte Carlo Integration

Monte Carlo integration uses randomization techniques to compute approximations to the value of integrals for which analytical methods can't be applied and standard numerical methods, such as quadrature, are slow and difficult. Suppose the integral to be computed can be written

$$(25) \quad I = \int h(x)f(x)dx$$

where $f(x)$ is a density function for a (convenient) probability distribution. Then I is the expected value of h over that distribution. If h is well-behaved, then the law of large numbers will apply to an independent random sample from the density f , and I can be approximated by

$$(26) \quad \hat{I} = \frac{1}{n} \sum h(x_i)$$

This approximation improves with the size of the random sample n . The accuracy of the Monte Carlo estimates is governed by the Central Limit Theorem: if we use

$$(27) \quad \frac{1}{n} \sum h^*(x_i) \text{ to approximate } Eh(x)$$

for some random variable h^* , the variance of the numerical error is approximately

$$(28) \quad \frac{1}{n} \text{var } h^*$$

While this process can be applied outside of statistical work (need to integrate a function g ?—find a clever hf combination), it is mainly used in Bayesian statistics where a complex posterior distribution can be “mapped out” by reporting the expected values of interesting functions. The practical examples in the earlier parts of this chapter are actually all fairly trivial instances of Monte Carlo integration. For instance, a probability is just the expected value of an indicator variable on a particular event: h is one or zero depending upon whether the draw for x satisfies a condition.

The following prices a vanilla European call using Monte Carlo, estimates the standard deviation of the error in this calculation, and compares it with the Black-Scholes value. Because only the value at expiration matters, this can be simulated without generating a full path for the security price. This is on the example file `MCPriceEurope.RPF`.

```
compute expire = 5.0/12.0, strike=52.0, price=50.0
compute rate   = .1, sigma=.40
compute hdrift = (rate-.5*sigma^2)*expire
compute hsigma = sigma*sqrt(expire)
compute ndraws = 10000
compute total  = 0.0, total2=0.0
```

```
do draw=1,ndraws
    compute payoff = exp(-rate*expire)*$
                    %max(0,price*exp(hdrift+%ran(hsigma))-strike)
    compute total = total+payoff,total2=total2+payoff^2
end do draw
compute mcvalue = total/ndraws
compute mcvar   = (total2/ndraws-mcvalue^2)/ndraws
disp "Value by MC" total/ndraws "Std Dev" sqrt(mcvar)

compute d1 = (log(price/strike)+expire*(rate+.5*sigma^2))/($
              (sigma*sqrt(expire)))
compute d2 = d1-sigma*sqrt(expire)

compute value = price*%cdf(d1)-strike*exp(-rate*expire)*%cdf(d2)
disp "Value by Black-Scholes" value
```

Antithetic Acceleration

In *antithetic acceleration*, we use a clever choice of h^* to get a reduction in the variance. Suppose that the density f is symmetric around x_0 . Because of the symmetry, the function

$$(29) \quad h^*(x) = (1/2)(h(x) + h(2x_0 - x))$$

has the same expected value as h . And its variance will be lower as long as there is a correlation between $h(x)$ and $h(2x_0 - x)$. In particular, if h is linear in x , $\text{var } h^*$ is zero: every draw for x will produce $h^*(x) = h(x_0)$, which is, of course, the correct expected value. If h is close to being linear over the region where f is high, we would see a very substantial reduction in variance for the antithetical method versus independent draws.

The option pricing example above seems a good candidate for antithetic acceleration, as flipping the sign of the draw will change a high security price to a low one and vice versa. Since you're doing two function evaluations per draw, antithetic acceleration shows signs of working if the variance is smaller than 1/2 the standard MC.

```
compute total=0.0,total2=0.0
do draw=1,ndraws
    compute u = %ran(hsigma)
    compute payoff1 = exp(-rate*expire)*$
                    %max(0,price*exp(hdrift+u)-strike)
    compute payoff2 = exp(-rate*expire)*$
                    %max(0,price*exp(hdrift-u)-strike)
    compute total   = total + .5*(payoff1+payoff2)
    compute total2  = total2 + .25*(payoff1+payoff2)^2
end do draw
compute mcvalue = total/ndraws
compute mcvar   = (total2/ndraws-mcvalue^2)/ndraws
disp "Value by MC-Antithetic" total/ndraws "Std Dev" sqrt(mcvar)
```

16.5.1 Monte Carlo Integration (VAR)

The first widespread use of simulation techniques in econometrics was Monte Carlo integration for VAR's and it probably remains the most common introduction to simulation methods for most students.

The impulse responses from a VAR are highly non-linear functions of the coefficients. Monte Carlo integration is one way to examine the distribution of these, so we can properly assess the statistical significance of the point values that are generated by **IMPULSE**. Other calculations routinely done with VAR's, such as forecasts, also are simple only if the coefficients are assumed to be known. Like IRF's, allowing for uncertainty in the lag coefficients creates a non-linear function which cannot easily be analyzed except by simulation methods. One very nice property of Monte Carlo methods is that the same basic method for simulating the VAR can be applied to any subsequent calculation, whether it's for impulse responses, error decompositions or forecasting.

What we will discuss in this section is the application of the techniques of “Multivariate Normal Regression (VAR)—Jeffreys Prior” to a (full) VAR with a diffuse (Jeffrey's) prior. A VAR with a prior, a near-VAR, or an overidentified structural VAR are more difficult because the posterior doesn't “factor”—those require some combination of the techniques in Section 16.6 (Importance Sampling), Section 16.7 (Gibbs Sampling), and Section 16.8 (Metropolis-Hastings). Those are more complicated to set up and require more care to ensure that you get good results. By contrast, the method here is straightforward and has very good Monte Carlo properties.

As is the case with many applications of simulation techniques, there is actually more programming involved in extracting the information from the simulations than in doing the simulations themselves. Because the desired organization of the impulse response graphs is generally the same regardless of the method used to generate the draws, we've written a special procedure **@MCGraphIRF** to do the graphs.

%%RESPONSES

To use **@MCGraphIRF**, the draws for the impulse responses must be saved in a particular way, in a `VECT[RECT]` called `%%RESPONSES`, which has the following structure:

- The dimension for the `VECT` is `NDRAWS`, with each element of the `VECT` giving the entire generated response for one draw.
- For each draw, the response is put into a `RECT` with `NVAR*NSHOCKS` rows and `NSTEPS` columns. `NVAR` is the number of endogenous variables in the model, `NSTEPS` is the number of steps of responses. `NSHOCKS` is the number of shocks, which is usually the same as `NVAR`, but doesn't have to be.
- The rows in the `RECT` for saving the draw are blocked by shocks, so the first set of `NVAR` elements in a column are the responses to the first shock, the second set are the responses to the second shock.

It's relatively simple to set this up properly. Do the following outside the loop:

```
declare vect[rect] %%responses (ndraws)
```

Inside the loop, you can use the **FLATTEN** option on **IMPULSE** to put a properly constructed matrix into element **DRAW** of **%%RESPONSES**:

```
impulse (noprint,model=varmodel,factor=fsigma,$  
  flatten=%%responses (draw),steps=nstep)
```

If you need to make some adjustment to what is saved, the following will pack the information in a **RECT[SERIES]** filled with impulse responses into the correct form. To do this the **RECT[SERIES]** needs to be dimensioned number of variables by number of shocks:

```
dim %%responses (draw) (nvar*nshocks,nstep)  
ewise %%responses (draw) (i,j)=ix=%vec(%xt(impulses,j)),ix(i)
```

In the **EWISE** instruction, the **%XT (IMPULSES, J)** extracts the $NVAR \times NSHOCKS$ matrix of responses at step **J**, the **%VEC** then “flattens” that into a **VECTOR** with the proper organization. The elements of that **VECTOR** are then copied into the proper slots in the **%%RESPONSES (DRAW)** matrix.

Generating Draws: MONTEVAR.RPF example

Under the convenient assumptions for the full VAR with the Jeffrey's prior, the Σ matrix can be drawn unconditionally, then the lag coefficients can be drawn conditional on Σ .

This is extremely quick since the factor of the biggest matrix $(\mathbf{X}'\mathbf{X})^{-1}$ can be computed just once, outside the loop. In **MONTEVAR.RPF**, after the VAR has been set up and estimated, the following computes factors of two key matrices into **FXX** and **FWISH**, saves the OLS estimates into **BETAOLS** and saves two other useful pieces of information.

```
estimate  
compute nvar      = %nvar  
compute fxx       = %decomp(%xx)  
compute fwish     = %decomp(inv(%nobs*%sigma))  
compute wishdof   = %nobs-%nreg  
compute betaols   = %modelgetcoeffs(varmodel)
```

Inside the loop, this uses antithetic acceleration by drawing a new value of Σ and β on the odd draws, then flipping the sign of the deviation between the draw and the OLS estimates (the mean of the posterior) on the even draws. **SIGMAD** is a draw for the covariance matrix (which requires only the sample values of **FWISH** and **WISHDOF**, since it has an unconditional distribution), **FSIGMA** is the factor of it and **BETAU** is the randomly drawn Normal addition to the OLS estimates to get the (odd-valued) draw. On the even numbered draws, the previous value for Σ is retained, and the antithetic value for the coefficients is generated by subtracting **BETAU** rather than adding.


```

if %clock(draw,2)==1 {
    compute sigmad   =%ranwisharti(fwish,wishdof)
    compute fsigma   =%decomp(sigmad)
    compute betau    =%ranmvkron(fsigma,fx)
    compute betadraw=betaols+betau
}
else
    compute betadraw=betaols-betau

```

The following resets the coefficient matrix for the working VAR:

```
compute %modelsetcoeffs(varmodel,betadraw)
```

and this computes and saves the impulse responses using the Cholesky factor shocks:

```

impulse(noprint,model=varmodel,factor=fsigma,$
results=impulses,steps=nstep,flatten=%responses(draw))

```

This *can* be adapted to any just-identified structural VAR by replacing the **FACTOR** option on the **IMPULSE** with some other factorization of **SIGMAD**. However, it will almost always be easier to use the **@MONTEVAR** or **@MCVARDoDraws** procedures with an **FFUNCTION** option to do that (page UG–522). Where the program here is helpful is when you need to generate draws from this type of model for other reasons (such as analyzing out-of-sample forecasts).

@MCGRAPHIRF Procedure

@MCGraphIRF has quite a few options. The default behavior, which we use here, will generate an $n \times n$ “matrix” of graphs, with the responses of a variable in a row, the response to a variable in a column. Following the recommendation of Sims and Zha (1999), this displays percentile bands, not standard errors. The 16% and 84% quantiles correspond to one standard deviation if we were doing symmetrical error bands based upon estimates of the variance. If you want standard error bands instead, you can use the **STDDEV** option. In our case, we do:

```
@mcgraphirf(model=varmodel)
```

Other useful options are **SHOCKLABELS** and **VARLABELS**, which let you relabel the shocks and endogenous variables to something more informative. (By default, both use the labels of the endogenous variables).

@MCPROCESSIRF Procedure

If the options of **@MCGRAPHIRF** can’t create the type of graph you want (or if you want some non-graphical output), you can use **@MCPROCESSIRF** to get the same information, such as **RECT[SERIES]** with the upper and lower bounds, but not the graphs. Like **@MCGRAPHIRF**, this also requires that the responses be packed into the **%RESPONSES** matrix (page UG–519).

@MCFEVDTABLE Procedure

@MCFEVDTABLE uses the `%%RESPONSES` information to produce a table with error bands for the decomposition of variance. Note that this assumes that the shocks used in generating `%%RESPONSES` are orthogonalized and produce a complete factorization of the covariance matrix. It cannot be used with isolated or non-orthogonalized shocks. Also note that the results can seem quite odd particularly with a larger model—it's not uncommon for the point estimates to be outside the error bands due to how highly asymmetric the variance decomposition can be. It is not, and should not be, considered standard practice to include this.

@MONTEVAR and @MCVARDoDraws procedures

In practice, you wouldn't adapt the `MONTEVAR.RPF` program to do the IRF error bands with a different data set—the **@MONTEVAR** procedure can be applied immediately after the **ESTIMATE** instruction to do the remainder of the calculations. In this case, it would be done with

```
@montevar (model=varmodel , step=nstep , draws=ndraws)
```

@MONTEVAR also has a `FFUNCTION` option which allows you to input a function to do an alternative factorization. If you want to control the graphics, instead of using **@MONTEVAR**, you can use **@MCVARDoDraws** which generates the `%%RESPONSES` array for use with **@MCGraphIRF** or **@MCProcessIRF**. Like **@MONTEVAR**, this has an `FFUNCTION` option. These all use the same methods shown in this section.

Note that if you do any calculation with long-run restrictions, you need to recompute the sum of lag coefficients (with the `%MODELLAGSUMS` function—`%VARLAGSUMS` is calculated at the OLS estimates only and doesn't reflect the fact that you have just drawn a new set). If you want to do non-orthogonal shocks, you can also do that by using a `FACTOR` or `SHOCK` option that has the shape that you want. For instance, if you use `SHOCKS=%UNITV(NVAR,1)`, you'll get just unit shocks to the first variable.

However, note that the calculations done by the `MONTEVAR.RPF` program or by these procedures *can't* be used for an overidentified structural VAR—that requires the use of importance sampling (Section 16.6) or Metropolis-Hastings (Section 16.8). For some just-identified factorizations which can't be done with simple matrix operations (if you need, for instance, **CVMODEL** to calculate the parameters), you may better off also using one of those techniques rather than estimating the SVAR every draw with **CVMODEL**.

16.6 Importance Sampling

Monte Carlo integration is fairly straightforward if the density f is an easy one from which to generate draws. However, many models do not produce the convenient Normal-gamma or Normal-Wishart posteriors that we saw in Section 16.4. One method which can be used for more difficult densities is known as *importance sampling*. For technical details beyond those provided here, see Geweke (1989).

Importance sampling attacks an expectation over an unwieldy density function using

$$\begin{aligned}
 E_f(h(x)) &= \int h(x)f(x)dx \\
 (30) \quad &= \int h(x)(f(x)/g(x))g(x)dx \\
 &= E_g(hf/g)
 \end{aligned}$$

where f and g are density functions, with g having convenient Monte Carlo properties.

The main problem is to choose a proper g . In most applications, it's crucial to avoid choosing a g with tails that are too thin relative to f . If f and g are true density functions (that is, they integrate to 1), we can use

$$(31) \quad \hat{h} = (1/n) \sum h(x_i)f(x_i)/g(x_i)$$

to estimate the desired expectation, where the x_i are drawn independently from g .

If $\int |h(x)|f(x)dx < \infty$, then $\int |h(x)(f(x)/g(x))|g(x)dx < \infty$

so the strong law of large numbers applies to (31), and $\hat{h}$ will converge a.s. to $E_f(h(x))$. However, while the sample means may converge, if the variance of hf/g doesn't exist, the convergence may be extremely slow. If, on the other hand, that variance *does* exist, the Central Limit Theorem will apply, and we can expect convergence at the rate $n^{1/2}$.

To take a trivial example, where the properties can be determined analytically, suppose $h(x) = x$, and f is a Normal with mean zero and variance 4. Suppose we choose as g a standard Normal. Then

$$(32) \quad f/g = \frac{1}{2} \exp\left(-\frac{3}{8}x^2\right) \quad \text{and} \quad \int (h(x)f(x)/g(x))^2 g(x)dx = \int \frac{x^2}{4\sqrt{2\pi}} \exp\left(-\frac{1}{4}x^2\right)dx = \infty$$

Convergence of $\hat{h}$ with this choice of g will be painfully slow.

Suppose now that f is a Normal with mean zero and variance 1/4 and again we choose as g a standard Normal. Now

$$(33) \quad \int (h(x)f(x)/g(x))^2 g(x)dx = \int \frac{4x^2}{\sqrt{2\pi}} \exp\left(-\frac{7}{2}x^2\right)dx = \frac{4}{7\sqrt{7}} \approx .216$$

For this particular h , not only does the importance sampling work, but, in fact, it works better than independent draws from the true f density. The standard error of the importance sampling estimate is $\sqrt{.216/n}$, while that for draws from f would be $\sqrt{.25/n}$. This result depends on the shape of the h function—in this case, the importance function gives a lower variance by oversampling the values where h is larger. If h goes to zero in the tails, sampling from g will still work, but won't do better than draws from f . (A thin-tailed g works respectably only for such an h .)

The lesson to be learned from this is that, in practice, it's probably a good idea to be conservative in the choice of g . When in doubt, scale variances up or switch to fatter-tailed distributions or both. For instance, the most typical choice for an importance function is the asymptotic distribution from maximum likelihood estimates. You're likely to get better results if you use a fatter-tailed t rather than the Normal.

All of the above was based upon f and g being true density functions. In reality, the integrating constants are either unknown or very complicated. If we don't know f/g , just $w = f^*/g^* = cf/g$, where c is an unknown constant, then

$$\begin{aligned}
 (1/n) \sum h(x_i) w(x_i) &= (1/n) \sum h(x_i) cf(x_i)/g(x_i) \rightarrow cE_f(h(x_i)) \\
 (1/n) \sum w(x_i) &= (1/n) \sum cf(x_i)/g(x_i) \rightarrow cE_f(1) = c \\
 (1/n) \sum (h(x_i) w(x_i))^2 &= (1/n) \sum (h(x_i) cf(x_i)/g(x_i))^2 \rightarrow c^2 E_g(h(x_i) f(x_i)/g(x_i))^2
 \end{aligned}
 \tag{34}$$

Using the second of these to estimate c gives the key results

$$\begin{aligned}
 \hat{h} &= \sum h(x_i) w(x_i) / \sum w(x_i) \\
 s_h^2 &= \left[\sum (h(x_i) w(x_i))^2 / (\sum w(x_i))^2 \right] - \hat{h}^2 / n
 \end{aligned}
 \tag{35}$$

There's one additional numerical problem that needs to be avoided in implementing this. Particularly for large parameter spaces, the omitted integrating constants in the density functions can be huge. As a result, a direct calculation of w can produce machine overflows or underflows. (The typical computer can handle real numbers up to around 10^{300}). To avoid this, we would advise computing w by

$$\exp(\log f^*(x) - \log f_{\max}^* - \log g^*(x) + \log g_{\max}^*)
 \tag{36}$$

where $f_{\max}^*$ and $g_{\max}^*$ are the maximum values taken by the two kernel functions.

Now all of the above shows how to compute the expectation of a measurable function of the random variable x . If you want to compute quantiles, you need to use the function %WFRACTILES. Quantiles are estimated by sorting the generated values and locating the smallest value for which the cumulated normalized weights exceeds the requested quantile. The proof of this is in the Geweke article. If you want to estimate a density function, add the WEIGHT option to your **DENSITY** instruction.

Importance Sampling: GARCHIMPORT.RPF example

GARCHIMPORT.RPF applies importance sampling to a GARCH model. The log likelihood for a GARCH model has a non-standard form, so it isn't possible to draw directly from the posterior distribution, even with "flat" priors. Importance sampling is one way to conduct a Bayesian analysis of such a model; we'll show another on page UG-537.

The importance function used here is a multivariate Student- t , with the mean being the maximum likelihood estimate and the covariance matrix being the estimated covariance matrix from **GARCH**. This is fattened up by using 5 degrees of freedom.

The true density function is computed using **GARCH** with an input set of coefficients and METHOD=EVAL which does a single function evaluation at the initial guess values. Most instructions which might be used in this way (**BOXJENK**, **CVMODEL**, **FIND**, **MAXIMIZE**, **NLLS**) have METHOD=EVAL options.

The example uses a GARCH model on the US 3-month Treasury Bill rate. The "mean model" is a one lag autoregression. This estimates a linear regression to get its residual variance for use as the pre-sample value throughout the analysis.

```
linreg ftbs3
# constant ftbs3{1}
compute h0=%sigmasq
```

Estimate a GARCH with Normally distributed errors

```
garch(p=1,q=1,reg,presample=h0,distrib=normal) / ftbs3
# constant ftbs3{1}
```

Set up for importance sampling.

FXX is the factor of the covariance matrix of coefficients

XBASE is the estimated coefficient vector

FBASE is the final log likelihood

```
compute fxx =%decomp(%xx)
compute xbase=%beta
compute fbase=%logl
```

This is the number of draws and the degrees of freedom of the multivariate t distribution from which we're drawing.

```
compute ndraws=10000
compute drawdf=5.0
```

Initialize vectors for the first and second moments of the coefficients

```
compute sumwt=0.0, sumwt2=0.0
compute [vect] b = %zeros(%nreg,1)
compute [symm] bxx = %zeros(%nreg,%nreg)
dec vect u(%nreg) betau(%nreg)
```

This is used for an estimated density of the persistence measure (ALPHA+BETA). We need to keep the values of the draws and the observation weights in order to use the **DENSITY** instruction.

```
set persist 1 ndraws = 0.0
set weights 1 ndraws = 0.0
```

```
do draw=1,ndraws
```

Draw an %NREG vector from a t with DRAWDF degrees of freedom, mean 0 and identity scale matrix.

```
compute u = %rant(drawdf)
```

Premultiply by FXX to correct the scale matrix and add the mean to get the draw for the coefficients

```
compute betau = xbase+fxx*u
```

Compute the density of the draw relative to the density at the mode. The log of this is returned by the value of %RANLOGKERNEL() from the %RANT function.

```
compute gx = %ranlogkernel()
```

Compute the log likelihood at the drawn value of the coefficients

```
garch(p=1,q=1,reg,presamp=h0,method=eval,init=betau) / ftbs3
# constant ftbs3{1}
```

Compute the difference between the log likelihood at the draw and the log likelihood at the mode.

```
compute fx = %logl-fbase
```

It's quite possible for %LOGL to be NA in the GARCH model (if the GARCH parameters go sufficiently explosive). If it is, make the weight zero. Otherwise, $\exp(\dots)$ the difference in the log densities.

```
compute weight = %if(%valid(%logl),exp(fx-gx),0.0)
```

Update the accumulators. These are weighted sums.

```
compute sumwt = sumwt+weight, sumwt2 = sumwt2+weight^2
compute b = b+weight*betau, bxx = bxx+weight*%outerxx(betau)
```

Save the drawn value for PERSIST and WEIGHT.

```
compute persist(draw) = betau(4)+betau(5)
compute weights(draw) = weight
end do draw
```

The efficacy of importance sampling depends upon function being estimated, but the following is a simple estimate of the number of effective draws.

```
disp "Effective Sample Size" sumwt^2/sumwt2
```

Transform the accumulated first and second moments into mean and variance

```
compute b = b/sumwt, bxx = bxx/sumwt-%outerxx(b)
```

Create a table with the Monte Carlo means and standard deviations

```
report(action=define)
report(atrow=1,atcol=1,fillby=cols) "a" "b" "gamma" "alpha" "beta"
report(atrow=1,atcol=2,fillby=cols) b
report(atrow=1,atcol=3,fillby=cols) %sqrt(%xdiag(bxx))
report(action=show)
```

Estimate the density of the persistence measure and graph it

```
density(weights=weights,smpl=weights>0.00,smoothing=1.5) $
    persist 1 ndraws xx dx
scatter(style=line,header=$
    "Density of Persistence Measure (alpha+beta)")
# xx dx
```

16.7 Gibbs Sampler

The Gibbs sampler (Gelfand and Smith, 1990) is one of several techniques developed recently to deal with posterior distributions which not long ago were considered to be intractable. Monte Carlo integration for impulse responses (16.5.1) is able to work well despite the large number of parameters in the underlying vector autoregression because, with the help of a convenient choice of prior, it is fairly easy to make draws from the posterior distribution. Unfortunately, there are very few multivariate distributions for which this is true. Even in a simple linear regression model, all it takes is a slight deviation from “convenience” in the prior to produce a tangled mess in the posterior distribution, making direct draws with the basic toolkit of random Normals and gammas impossible.

The Gibbs sampler can be brought into play when the parameters can be partitioned so that, although an unconditional draw can’t be obtained directly, each partition can be drawn conditional on the parameters outside its partition. The standard result is that if we draw sequentially from the conditional distributions, the resulting draws are, in the limit, from the unconditional distribution. Because this is a limit result, it is common for practitioners to ignore a certain number of early draws (called the “burn-in”) which might not be representative of the unconditional distribution.

If importance sampling (Section 16.6) is able to work successfully in a given situation, it usually should be chosen over Gibbs sampling, because the draws are independent rather than correlated. However, it often becomes quite hard to find a workable importance function as the dimension of the parameter space gets larger.

A General Framework

The following is a general set-up that we’ve found useful for Gibbs sampling (and which is created using the *Help—Pseudo-Code Generator—Monte Carlo/Gibbs Loop* wizard)

```
compute nburn=# of burn-in draws           Note: as written this does nburn+1
compute ndraws=# of accepted draws
```

Set initial values for parameters. Do any calculations which don’t depend upon the draws. Initialize bookkeeping information

```
infobox(action=define,progress,lower=-nburn,upper=ndraws) $
  "Gibbs Sampler"
do draw=-nburn,ndraws
  infobox(current=draw)
```

Do the next draw

```
if draw<=0
  next
```

Update bookkeeping information

```
end do draw
infobox(action=remove)
```


Mean-Variance Blocking

The most common blocking for the Gibbs sampler is between the regression parameters and the variance or precision. In Section 16.4 for the standard Normal linear model, we can draw β conditional on h and h conditional on β . For the VAR with a general prior, we can draw β conditional on Σ and Σ conditional on β . We'll show the most efficient way to do Gibbs draws from a linear regression. Outside the loop, do

```
cmom
# x variables    y
linreg(cmom) y
# x variables
compute beta=%beta
```

to calculate the cross moment matrix and get an initial value for the coefficients. Also, set up the prior mean (BPRIOR) and precision (HPRIOR) and the degrees of freedom (NUPRIOR) and scale factor (S2PRIOR) for the residual precision. Inside the loop, do

```
compute rssplus = nuprior*s2prior + %rsscmom(%cmom,bdraw)
compute hdraw   = %ranchisqr(nuprior+%nobs)/rssplus
compute bdraw   = %ranmvpostcmom(%cmom,hdraw,hprior,bprior)
```

Unobservable States (Parameter Augmentation)

Gibbs sampling can be also be used in situations where there are unobservable variables at each time period. For instance, the Markov switching models (Section 11.7) have the unobservable S_t in addition to the model's other parameters. These are added to the parameter set in a process known as *augmentation*. Note that the number of parameters now exceeds the number of data points—this is possible only because of the use of priors.

Getting draws for regression parameters given the regimes is quite simple if the model is otherwise linear. The tricky part is getting the draws for the latent variables. The simplest way to do this is to draw them one at a time; that is, draws are done sequentially (as t goes from 1 to T) for

$$(37) \quad S_t | S_1, \dots, S_{t-1}, S_{t+1}, \dots, S_T, \theta$$

where θ are the other parameters. Note that this can be *very* time-consuming. In addition to the cost of doing (at least) T function evaluations per Gibbs sweep, because of the high correlation between the S values (in most cases), the chain also has a high degree of correlation from one step to the next, so it requires a long burn-in time and many draws. (As a general rule, Gibbs sampling works best when the blocks of parameters are close to being independent of each other, and worst when correlated parameters are drawn separately). A more efficient (but more complicated) way to do the draws for the states is known as *forward-filter-backwards-sampling*. This has many similarities to Kalman smoothing (Section 10.2) as it takes a forward pass through the data to determine the joint distribution of the states, then uses that to take draws starting at the end, working towards the start of the data set.

@MCMCPOSTPROC

There is often more work to do *after* getting the simulations than there is doing them in the first place. We've provided a special procedure for doing a standard set of post-processing calculations for Gibbs sampling. That's **@MCMCPostProc**. The input to this is a **SERIES [VECTOR]** which has the results of the kept draws from the sampler. You can use the procedure to calculate the sample means and standard errors for each component of the vector and also to compute some measures of the performance of the sampler. Remember that the draws here aren't independent, and if there's too high a correlation, the results are likely to be much less precise than we would like.

@MCMCPostProc (NDRAWS=# of draws, other options) *stats*

stats is a **SERIES [VECTOR]**. You want to create one (or more) of these to hold the statistics of interest at each draw that you keep. Inside the draw loop, organize a **VECTOR** with the statistics that you want, and save a copy of it into entry "draw" of the **SERIES [VECTOR]**.

Note that if the number of draws is large (more than 10000), it can take quite a while to do the diagnostics and the percentiles.

ndraws=# of draws (length of *stats*) (REQUIRED)

mean=(output) [VECTOR] *averages of each component*

stderrs=(output) [VECTOR] *standard errors of each component*

cv=(output) [SYMMETRIC] *estimated covariance matrix*

nse=(output) [VECTOR] *numerical standard errors of each component*

The **MEAN** and **STDERRS** are the sample mean and standard errors for each element of the **VECTOR** across the draws, so if you do **MEAN=BMEANS**, **BMEANS(1)** will be the sample mean across draws for the first statistic. **CV** is the full sample covariance matrix of the saved statistics. **NSE** are corresponding estimates of the standard errors of the **MEAN** component, which are computed using HAC estimators since the Markov Chain estimates in general are at least somewhat autocorrelated.

percentiles=(input) ||desired percentiles|| [not used]

quantiles=(output) VECT[VECTOR] *with percentiles*

If you want percentiles of the distribution, use the **PERCENTILES** to make the request and **QUANTILES** (which is a **VECTOR [VECTOR]**) for the output. Each outer element of **QUANTILES** is for a statistic, and the inner elements are the computed percentiles in the same order as the request. So **PERCENTILES=||.16,.50,.84||**, **QUANTILES=Q** will mean that **Q(1)** will be a **VECTOR** with the .16,.50 and .84 quantiles of the first saved statistic, similarly **Q(2)** for the second statistic.

```
cd=(output) [VECTOR] Geweke CD measures
bw=(output) [VECTOR] between-within measures
blocks=number of blocks for BW [10]
```

CD and BW are diagnostics. CD is a comparison of early draws against late draws and is mainly a measure of how well the chain has been “burned-in”. Asymptotically, it’s $N(0,1)$ component by component with neither sign expected over the other. BW computes the ratio of between to within variation within the chain, using the number of blocks given by the BLOCKS option, that is, if you have 25000 draws and the default BLOCKS=10, it will look at 10 blocks of 2500. If the draws are independent, this will have an F distribution. A large value indicates that the draws tend to be highly correlated for long stretches.

In the next example, we use the following:

```
@mcmcpostproc (ndraws=ndraws, mean=bmean, $
               stderrs=bstderrs, cd=bcd, nse=bnse) bgibbs
```

The saved statistics (in this case, the simulated regression coefficients) are in the SERIES [VECTOR] named BGIBBS. We compute the mean (into BMEAN), standard errors (into BSTDERRS), the CD measure (described in Koop(2003)) which is for testing the adequacy of the number of burn-in draws, and the numerical standard errors, which estimate the precision of the sample means.

Gibbs Sampling (Linear Regression): GIBBS.RPF example

GIBBS.RPF is based upon an example from Koop (2003). It’s a linear regression, estimating a hedonic regression for housing prices. The explanatory variables are lot size, number of bedrooms, number of bathrooms and a measure of storage. The prior used is fairly loose. It’s usually easier to think about priors based upon a mean and standard deviation, so the standard deviations need to be converted into prior precisions by squaring and then inverting that matrix. The prior here has a diagonal covariance which is fairly typical—we’re likely to have an idea about how large each coefficient might be, but very little about the specifics regarding correlations among them.

This is the regression of interest:

```
linreg price
# constant lot_siz bed bath stor
```

Prior for variance.

```
compute s2prior = 5000.0^2
compute nuprior = 5.0
```

This is the mean and variance for the prior on the regression coefficients themselves. Convert the variance to precision by inverting.

```
compute [vector] bprior = ||0.0,10.0,5000.0,10000.0,10000.0||
```

```
compute [symm]    vprior = $
    %diag(||10000.0^2,5.0^2,2500.0^2,5000.0^2,5000.0^2||)
compute [symm]    hprior = inv(vprior)
```

Compute the cross product matrix from the regression. This will include the dependent variable as the final row. The cross product matrix has all the sufficient statistics for the likelihood (except the number of observations, which is %NOBS).

```
cmom(lastreg)
```

The two obvious places to start the Gibbs sampler are the OLS estimates and the prior. We'll use OLS.

```
compute bdraw = %beta
compute s2draw = %seesq
```

```
compute nburn = 1000
compute ndraws = 10000
```

This sets up and initializes the two locations where we are storing information generated by the draws. BGIBBS saves the coefficient vectors, while HGIBBS saves the draws for the precision (reciprocal of the variance).

```
dec series[vect] bgibbs
dec series          hgibbs
gset bgibbs 1 ndraws = %zeros(%nreg,1)
set  hgibbs 1 ndraws = 0.0
```

```
do draw=-nburn,ndraws
```

Draw residual precision conditional on previous beta

```
compute rssplus = nuprior*s2prior + %rsscmom(%cmom,bdraw)
compute hdraw = %ranchisqr(nuprior+%nobs)/rssplus
```

Draw betas given hdraw

```
compute bdraw = %ranmvpostcmom(%cmom,hdraw,hprior,bprior)
if draw<=0
    next
```

Do the bookkeeping here.

```
compute bgibbs(draw) = bdraw
compute hgibbs(draw) = hdraw
end do draw
```

Do the post-processing, computing the means, standard errors, numerical standard errors and CD measures:

```
@mcmcpostproc(ndraws=ndraws,mean=bmean,$
    stderrs=bstderrs,cd=bcd,nse=bnse) bgibbs
```

```

report(action=define)
report(atrow=1,atcol=1,align=center) "Variable" "Coeff" $
  "Std Error" "NSE" "CD"
do i=1,%nreg
  report(row=new,atcol=1) %eqnreglabels(0)(i) bmean(i) $
    bstderrs(i) bnse(i) bcd(i)
end do i
report(action=format,atcol=2,tocol=3,picture="*.###")
report(action=format,atcol=4,picture="*.##")
report(action=show)

```

Gibbs Sampling (Near-VAR): MONTESUR.RPF

A near-VAR or any other type of multiple equation linear model with non-matching explanatory variables has to be done using the techniques described for general priors in multivariate systems from page UG–515. The difference between a technically correct analysis of a near-VAR system and a VAR with a prior is quite minor—the block exclusion in the near-VAR is really just an (infinitely) tight prior on some of the coefficients.

This requires inverting the precision matrix for the entire system. This could be extremely time-consuming for a large model—with the time required for inversion going up with the cube of the size, a 1200 coefficient model will take (roughly) 64 times as long as a 300 coefficient model.

Although the procedure for drawing from a near-VAR is equivalent to estimating a SUR model each time you get a new covariance matrix, the **SUR** instruction, in addition to having to invert the big matrix, also has to compute the cross product matrix of the data. Since the data information is the same from sweep to sweep, we can eliminate that time cost by organizing the data information in advance. That can be done with the **@SURGIBBSSETUP** procedure group which handles the interactions between the drawing procedure and the data. If you are interested in the technical details, they are covered as part of the *Bayesian Econometrics* e-course.

The MONTESUR.RPF example does a two variable near-VAR, where one variable is excluded from the other's equation. However, the same basic program can be adapted to create draws for any other (linear) SUR model. The draws are used to compute impulse response functions, which are saved and later graphed as described in Section 16.5.1. The following sets up the model by defining the two equations and GROUPING them into a model. See page UG–262 for different ways to handling this step that might be more appropriate for larger systems.

```

equation gdpeq gdph
# gdph{1 to lags} fm1{1 to lags} constant
equation m1eq fm1
# fm1{1 to lags} constant
group surmodel m1eq gdpeq

```

This takes the model and sets up all the information required for interacting with the data. It figures out what series and lags are needed and computes the cross product matrix of them and organizes information for later use.

```
@SURGibbsSetup surmodel
```

This uses **SUR** to initialize the sampler—as we have the sampler set up, all that’s needed for that are the coefficients (into **BDRAW**).

```
sur(model=surmodel)  
compute ntotal=%nreg  
compute bdraw=%beta  
compute wishdof=%nobs
```

Inside the draw loop, the first step is to compute the covariance matrix of residuals at the current beta. This uses one of the other procedures that is pulled in by **@SURGIBBSSETUP**.

```
compute covmat=SURGibbsSigma(bdraw)
```

This does a draw for the precision matrix (inverse of the covariance matrix) conditional on the current coefficients.

```
compute hdraw=%ranwishartf(%decomp(inv(covmat)),wishdof)
```

Compute the information obtained by interacting the precision matrix and the data. These are precision matrix for the coefficients (**HDATA**) and precision matrix times the mean (**HBDATA**). If you had an actual prior (besides the zero restrictions), you would adjust **HDATA** and **HBDATA** to take those into account before continuing.

```
@SURGibbsDataInfo hdraw hdata hbdata
```

This is the “hot spot” of the calculation, as this requires (internally) inverting the (potentially very) large precision matrix **HDATA**.

```
compute bdraw=%ranmvposthb(hdata,hbdata)
```

The following is the “bookkeeping” section, which is executed once we’re past the burn-in. The calculations above have all used only the precision matrix for the residuals, so we have to invert that now to get the covariance matrix (into **SIGMAD**). This does Cholesky factor shocks. As with the full var, you can use any other “**FACTOR**” matrix here to deal with just-identified or non-orthogonalized shocks. However, the (relatively) simple Gibbs sampler can’t be used with an overidentified structural model.

```
compute %modelsetcoeffs(surmodel,bdraw)  
compute sigmad=inv(hdraw)  
impulse(noprint,model=surmodel,factor=%decomp(sigmad),  
steps=nsteps,flatten=%responses(draw))
```

16.8 Metropolis-Hastings

The Gibbs sampler (Section 16.7) requires an ability to generate draws from the conditional distributions. And again, there are many cases where the conditional distributions don't have convenient Monte Carlo properties. It's still possible to generate a Markov chain which will converge to the correct distribution, by using techniques similar to importance sampling (Section 16.6). The main difference between these is that importance sampling generates independent draws and weights them to achieve the correct distribution. Because the Gibbs sampler is only approximating the distribution at the end of a chain, a weighting system won't work. Instead, the "weighting" is done by keeping high-density draws for multiple sweeps.

While there are quite a few variations of this procedure, there are two principal methods of generating a test draw: you can either draw from a fixed distribution (similar to what is done with importance sampling), or you can draw from a distribution centered around the last draw.

The distribution from which you make the draw is known as the *proposal distribution* or *jumping distribution*. Let x be the previous draw and y be the test draw. You compute a jumping probability α . The "pseudo-code" then would be:

```
if %ranflip(alpha) {
    accept y
}
else {
    keep x
}
```

It's possible for ALPHA to be bigger than 1. If that's the case, the draw will always be accepted.

Fixed proposal distribution (Independence Chain)

Let f be the target density and g be the proposal density. (These only need to be known up to a constant of integration). Then

$$(38) \quad \alpha = \frac{f(y)g(x)}{f(x)g(y)}$$

As in importance sampling, you need to watch for overflows in the calculation. To safeguard against this, compute this (if possible) as

$$(39) \quad \alpha = \exp(\log f(y) - \log f(x) + \log g(x) - \log g(y))$$

Random Walk Metropolis

If y is drawn from a density which is conditional on x , let $g(x, y)$ denote the density for the transition from x to y . The general formula is

$$(40) \quad \alpha = \frac{f(y)g(x, y)}{f(x)g(y, x)}$$

However if the transition density has $y | x \sim N(x, \Sigma)$ where Σ doesn't depend upon x , then $g(y, x) = g(x, y)$, so (39) simplifies to

$$(41) \quad \alpha = \frac{f(y)}{f(x)}$$

This is known as Random Walk Metropolis. This simplification will happen whenever $g(y, x) = g(x, y)$, which generally would mean a Normal or t centered at the previous draw. Note that if you do any linearization to get the proposal density, that will make the density dependent upon x for more than the mean. If you do that, you have to do the analogous linearization around y to be able to compute $g(y, x)$.

Choosing a Proposal Density

This can be quite difficult, and may require quite a bit of experimentation, particularly if you're applying this to a multi-dimensional x . With either type of proposal, if the distribution is too broad, you don't move very often because most of the draws hit low densities. If it's too narrow, you might move only very slowly once you've landed in a high density area. It's a good idea to keep track of the percentage of times you jump. If this is too small, it may indicate that you need to make some type of adjustment. But taking many jumps doesn't necessarily indicate a successful set-up—you can move around quite a bit in one area, but fail to move beyond it.

Displaying the Acceptance Rate

If you are doing a time-consuming chain, it's helpful to get quicker feedback on the chain's acceptance probability than you would get by running an entire set of draws first. You can do that by replacing the **INFOBOX** inside the loop with:

```
infobox(current=draw) %strval(100.0*accept/(draw+nburn+1), "##.##")
```

This puts a second line in the progress box (page UG–501) showing the percentage of jumps made. This is updated after each sweep, so if it looks like it's running too low, you can cancel the loop, and try some different settings.

Metropolis-Hastings (GARCH model): GARCHGIBBS.RPF example

GARCHGIBBS.RPF uses Metropolis-Hastings for estimating the same GARCH model used in the GARCHIMPORT.RPF example (page UG–525). This blocks the parameters into the mean AR(1) parameters and the GARCH parameters. To illustrate both methods, the AR(1) parameters are handled with a fixed proposal and the GARCH are done by Random Walk Metropolis.

```
open data haversample.rat
calendar(m) 1957
data(format=rats) 1957:1 2006:12 ftbs3
```

Estimate a linear regression to get its residual variance for use as the pre-sample value.

```
linreg ftbs3
# constant ftbs3{1}
compute h0 = %sigmasq
```

Estimate a GARCH

```
garch(p=1,q=1,reg,presample=h0,hseries=h) / ftbs3
# constant ftbs3{1}
```

Pull out the regression coefficients (1 and 2). This forms block one of the Metropolis/Gibbs. Get a decomp of its submatrix of the covariance matrix for use in the proposal density. Do a draw from this to initialize the Gibbs sampler and save the log kernel of the density at the draw.

```
compute xbeta0 = %xsubvec(%beta,1,2)
compute fbeta = %decomp(%xsubmat(%xx,1,2,1,2))
compute xbeta = xbeta0 + %ranmvnormal(fbeta)
compute gx = %ranlogkernel()
```

Pull out the GARCH coefficients (3,4 and 5). This forms block two. Again, get the decomp of its covariance matrix. Because these are being done by Random Walk Metropolis, we don't need the proposal densities.

```
compute fgarch = %decomp(%xsubmat(%xx,3,5,3,5))
compute xgarch = %xsubvec(%beta,3,5)
```

```
compute nburn=1000, ndraws=10000
```

Initialize vectors for the first and second moments of the coefficients

```
compute sumwt=0.0, sumwt2=0.0
compute [vect] b = %zeros(%nreg,1)
compute [syymm] bxx = %zeros(%nreg,%nreg)
dec vect betau(%nreg) betadraw ygarch
```

Counters for the number of jumps

```
compute bjumps=0, gjumps=0
```

```
infobox(action=define,progress,lower=-nburn,upper=ndraws) $  
  "Random Walk Metropolis"  
do draw=-nburn,ndraws
```

Drawing regression parameters. Evaluate the log likelihood (into FX) at the values for XBETA and XGARCH.

```
garch(p=1,q=1,reg,method=eval,presample=h0,$  
  initial=xbeta~~xgarch) / ftbs3  
# constant ftbs3{1}  
set u = %resids  
compute fx = %logl
```

Draw a test vector from the (fixed) proposal distribution. Recalculate the log likelihood (into FY) and fetch the log density at the draw (into GY).

```
compute ybeta=xbeta0+%ranmvnormal(fbeta)  
compute gy=%ranlogkernel()  
garch(p=1,q=1,reg,method=eval,presample=h0,$  
  initial=ybeta~~xgarch) / ftbs3  
# constant ftbs3{1}  
compute fy = %logl
```

Compute the jump alpha. If we need to move, reset XBETA and GX, and copy the residuals from Y into U.

```
compute alpha=exp(fy-fx+gy-gx)  
if %ranflip(alpha) {  
  compute bjumps = bjumps + 1  
  compute xbeta=ybeta, gx=gy  
  set u = %resids  
}
```

Evaluate the log likelihood of a GARCH model (into FX) on the residuals at the current settings for XGARCH.

```
garch(p=1,q=1,nomean,method=eval,presample=h0,$  
  initial=xgarch) / u  
compute fx = %logl
```

Draw from the proposal distribution centered around XGARCH and evaluate the log likelihood into FY.

```
compute ygarch = xgarch + %ranmvnormal(fgarch)  
garch(p=1,q=1,nomean,method=eval,presample=h0,$  
  initial=ygarch) / u  
compute fy = %logl
```

Compute the jump alpha. (Because draws for the GARCH parameters can turn the model explosive, check for an invalid FY).

```
compute alpha = %if(%valid(fy), exp(fy-fx), 0.0)
```

```
if %uniform(0.0,1.0)<alpha {  
    compute gjumps=gjumps+1  
    compute xgarch=ygarch  
}  
infobox(current=draw)  
if draws<=0  
    next
```

Update the accumulators if we're past the burn-in

```
    compute betau = xbeta~~xgarch  
    compute b=b+betau, bxx =bxx+%outerxx(betau)  
end do draw  
infobox(action=remove)
```

```
disp "Percentage of jumps for mean parms" $  
    100.0*float(bjumps)/(nburn+ndraws+1)  
disp "Percentage of jumps for GARCH parms" $  
    100.0*float(gjumps)/(nburn+ndraws+1)
```

Transform the accumulated first and second moments into mean and variance

```
compute b=b/ndraws, bxx=bxx/ndraws-%outerxx(b)  
report(action=define)  
report(atarow=1,atcol=1,fillby=cols) "a" "b" "gamma" "alpha" "beta"  
report(atarow=1,atcol=2,fillby=cols) b  
report(atarow=1,atcol=3,fillby=cols) %sqrt(%xdiag(bxx))  
report(action=show)
```

16.9 Bootstrapping and Resampling Methods

All of the methods in the previous sections draw data from a continuous distribution. Bootstrapping and other resampling techniques instead reshuffle an existing set of numbers. The key instruction is **BOOT**, which draws a random set of integers. By default, **BOOT** draws *with* replacement, but with the option **NOREPLACE** will do the draws without replacement.

A draw of random integers will give you the entry numbers you need to sample from a data set. You then use **SET** instructions to generate the resampled data. For instance, the following draws entry numbers between 1950:1 and 2017:12 (the estimation range) and puts them into 2018:1 to 2018:12 of the **SERIES [INTEGER]** named **ENTRIES**. The series **PATH1**, **PATH2** and **PATH3** will then be random samples from **RESIDS1**, **RESIDS2** and **RESIDS3** over 1950–2017. Note that the three residual series are sampled together. The resampling thus retains any contemporaneous relationship that is present in the data set.

```
group  model5 x1eq>>fx1 x2eq>>fx2 x3eq>>fx3 x4id>>fx4 x5id>>fx5
smp1 2018:1 2018:12
do draw=1,ndraws
```

Choose random entries between 1950:1 and 2017:12

```
boot  entries / 1950:1 2017:12
set path1 = resids1(entries)
set path2 = resids2(entries)
set path3 = resids3(entries)
forecast(model=model5,paths)
# path1 path2 path3
```

... bookkeeping ...

```
end do draw
smp1
```

If you have a single equation, and just want a simple bootstrap based upon the residuals, you can use **UFORECAST** with the **BOOTSTRAP** option. For instance, in the example above, if we just wanted to bootstrap **X1EQ**, we could just do

```
do draw=1,ndraws
  uforecast(boot,equation=x1eq) fx1 2018:1 2018:12
```

... bookkeeping ...

```
end do draw
```

Block Bootstrapping

In its simplest use, **BOOT** samples uniformly from the indicated range. This is appropriate when the observations are independent, or when you are resampling some component of the data (typically residuals), that have been transformed to be (ap-

proximately) independent. However, in some applications with correlated data, you may not have available a model which will transform the data to independent components. In order to maintain the observed form of dependence, you need to resample the data in blocks. This is done using **BOOT** with the `BLOCK=block size` option. There are three methods for doing block bootstrapping; you choose this using the `METHOD` option. The default is `METHOD=OVERLAP`. This will allow any block of *block size* entries from the resampling zone to be chosen. `METHOD=NOOVERLAP` partitions the resampling zone into disjoint blocks and only selects from those.

`METHOD=STATIONARY` doesn't sample by blocks. Instead, when sampling for an entry, it either takes the next item in the resampling zone, or (with probability) $1/\text{block size}$, starts a new block. With this, the data are resampled with blocks with an *expected* size of *block size*, while in the other two methods, they are exactly *block size*. See Politis and Romano (1994).

There are also some applications where it might be desired to oversample more recent values. You can do this with **BOOT** with the option `GEOMETRIC=decay rate`, where *decay rate* is the rate at which the probability of being chosen declines, with the last entry in the resampling zone having the highest probability. The closer *decay rate* is to 1, the closer this comes to uniform sampling.

In addition to the **BOOT** instruction, there are several functions which choose a random integer or set of integers.

%raninteger (1,u)	draws an integer uniformly from $\{l, l+1, \dots, u\}$
%ranpermute (n)	returns a <code>VECTOR[INTEGER]</code> with a random permutation (ordering) of the numbers $\{1, \dots, n\}$
%rancombo (n,k)	returns a <code>VECTOR[INTEGER]</code> with a random combination of <i>k</i> values from $\{1, \dots, n\}$ drawn without replacement.
%ranbranch (p)	returns a randomly selected “branch” from $\{1, \dots, \text{dim}(\mathbf{p})\}$ where p is a <code>VECTOR</code> giving the (relative) probabilities of the different branches.

For instance, the following draws a “poker hand”, a combination of 5 numbers from 1 to 52:

```
compute cards = %rancombo (52,5)
```

If `PPOST` is an *n*-vector with relative probabilities of a “break” at a given location in the interval `LOWER` and `LOWER+N-1`, then

```
compute break = %ranbranch (ppost)+lower-1
```

will choose a random break point in that range with probabilities weighted according to the values in `PPOST`.

Bootstrapping (GARCH Model): GARCHBOOT.RPF example

GARCHBOOT.RPF does a Value at Risk (VaR) calculation for the dollar/yen exchange rate using a bootstrapped GARCH(1,1) model to generate the simulated returns. (See, for instance, Tsay (2010) for more on calculation of VaR).

You can't simply take bootstrap draws of the residuals from a GARCH process because of the serial dependence of the variance process. Instead, the estimated residuals are standardized by dividing by the square root of their estimated variance, and the GARCH process is simulated out of sample with the bootstrapped standardized residuals scaled up by the simulated variance.

Estimate the GARCH(1,1) model.

```
garch(p=1,q=1,resids=u,hseries=h) / x
```

Generate a forecasting formula from the results of the GARCH estimation. GSTART and GEND are the regression range, which we need for drawing standardized residuals.

```
compute gstart=%regstart(), gend=%regend()  
compute b0=%beta(1), chat=%beta(2), ahat=%beta(3), $  
        bhat=%beta(4)  
frm1 hf = chat + bhat*h{1} + ahat*u{1}^2
```

Standardize the historical residuals

```
set ustandard gstart gend = u/sqrt(h)
```

SPAN is the number of periods over which returns are to be computed. NDRAWS is the number of bootstrapping draws

```
compute span=10  
compute ndraws=10000
```

Extend out the H series (values aren't important—this is just to get the extra space).

```
set h gend+1 gend+span = h(gend)
```

```
dec vect returns(ndraws)  
do draw=1,ndraws
```

This draws standardized u's from the USTANDARD series

```
boot entries gend+1 gend+span gstart gend
```

Simulate the GARCH model out of sample, scaling up the standardized residuals by the square root of the current H.

```
set udraw gend+1 gend+span = ustandard(entries)  
set u        gend+1 gend+span = (h(t)=hf(t)), udraw(t)*sqrt(h(t))
```

Figure out the cumulative return over the span. As written, this allows for the con-

tinuation of the sample mean return. If you want to look at zero mean returns, take the B0 out.

```
sstats gend+1 gend+span b0+u>>returns(draw)
end do draw
```

Compute desired quantiles of the returns

```
compute [vect] pvals=||.01,.05,.10||
compute [vect] VaR=%fractiles(returns,pvals)
report(action=define,hlabels=||"P","VaR/$100"||)
do i=1,3
  report(atrow=i,atcol=1) pvals(i) VaR(i)
end do i
report(action=show)
```

Approximate Randomization: RANDOMIZE.RPF example

Approximate randomization is a technique for testing “unrelatedness” in a fairly general way. It can be applied in place of analysis of variance tests and the like. The null hypothesis to be tested is that some variable X is unrelated to another variable Y. The method of attack is to take random permutations of the sample X’s. If X is, indeed, unrelated to Y, then the actual sample should be fairly typical of the population of permutations. Choose an appropriate test statistic and count the number of times the permuted samples produce a more extreme statistic than the actual sample. (In “exact” randomization, *all* permutations are examined. That is clearly only possible for very small sample sizes).

As an example, RANDOMIZE.RPF takes another look at the heteroscedasticity tests done in the HETEROTEST.RPF example page UG–77. The null hypothesis is that the variance is unrelated to lot size. We’ll look at two different choices for the test statistic: the first is the ratio of variances between the two subsamples (which would be testing against the alternative that the variance for small lots is smaller than those for large ones) and the second is the rank correlation between lot size and the squared residuals (which tests more generally that the variance increases with lot size).

```
order(ranks=lotranks) lotsize
linreg price
# constant lotsize sqrft bdrms
```

The first test statistic is the ratio between the sum of the residuals squared over the final 36 observations (large lot size) to that over the first 36 (small lots), skipping the middle 16. The second is the correlation between the ranks of the lot size and the squared residual.

```
set ressq = %resids^2
sstats / ressq*(lotranks<=36)>>sum1 ressq*(lotranks>=53)>>sum2
compute refer1=sum2/sum1
order(rank=vrank) ressq
```

```
compute refer2=%corr(lotranks,vrank)
```

COUNT1 and COUNT2 are the number of times we get a more extreme value after reshuffling. We do 999 shuffles.

```
compute count1=count2=0.0
compute ns=999
do draw=1,ns
```

Use **BOOT** with **NOREPLACE** to come up with a new permutation of the **RESSQR** variable. Recompute the test statistic for these.

```
boot(noreplace) entries 1 88
set shuffle = ressq(entries(t))

sstats / shuffle*(lotranks<=36)>>sum1 $
        shuffle*(lotranks>=53)>>sum2
compute teststat=sum2/sum1
compute count1=count1+(teststat>refer1)

order(rank=vrank) shuffle
compute teststat=%corr(lotranks,vrank)
compute count2=count2+(teststat>refer2)
end do draw
```

The p -values for the tests are computed by taking $(\text{count}+1)/(\text{draws}+1)$. The +1's are needed because we are, in effect, adding the actual sample in with the draws and seeing where it ends up.

```
disp "Test 1, p-value" (count1+1)/(ns+1)
disp "Test 2, p-value" (count2+1)/(ns+1)
```


Bibliography

- Aielli, G. P. (2013). "Dynamic Conditional Correlation: On Properties and Estimation," *Journal of Business & Economic Statistics*, vol 31, no 3, pp, 282–299.
- Akaike, H. (1973). "Information Theory and the Extension of the Maximum Likelihood Principle." In *2nd International Symposium on Information Theory*, B.N. Petrov and F. Csaki, eds., Budapest.
- Andersen, T.G., and T. Bollerslev (1998). "Answering the Skeptics: Yes, Standard Volatility Models Do Provide Accurate Forecasts", *International Economic Review*, 39, pp 885-905.
- Andrews, D.W.K., and P. Guggenberger (2003). "A Bias–Reduced Log–Periodogram Regression Estimator for the Long–Memory Parameter." *Econometrica*, Vol. 71, pp. 675-712.
- Arellano, M. and S. Bond (1991). "Some Tests of Specification for Panel Data: Monte Carlo Evidence and an Application to Employment Equations." *Review of Economic Studies*, Vol. 58, pp. 277-297.
- Bai, J and P. Perron (2003). "Computation and analysis of multiple structural change models." *Journal of Applied Econometrics*, Vol. 18, no. 1, pp 1-22.
- Baltagi, B.H. (2008), *Econometric Analysis of Panel Data, 4th Edition*. Chichester, UK: Wiley.
- Bernanke, B. (1986). "Alternative Explanations of the Money-Income Correlation." *Carnegie-Rochester Conference Series on Public Policy*, Vol. 25, pp. 49-100.
- Berndt, E.K., B.H. Hall, R.E. Hall and J.A. Hausman (1974). "Estimation and Inference in Nonlinear Structural Models." *Annals of Economic and Social Measurement*, Vol. 3/4, pp. 653-665.
- Blanchard, O. and D. Quah (1989). "The Dynamic Effects of Aggregate Demand and Supply Disturbances." *American Economic Review*, Vol. 79, pp. 655-673.
- Bollerslev, T. (1986). "Generalized Autoregressive Conditional Heteroskedasticity." *Journal of Econometrics*, Vol. 31, pp. 307-327.
- Bollerslev, T. and J.M. Wooldridge (1992). "Quasi-Maximum Likelihood Estimation and Inference in Dynamic Models with Time Varying Covariances." *Econometric Reviews*.
- Box, G.E.P., G.M. Jenkins, and G.C. Reinsel (2008). *Time Series Analysis, Forecasting and Control, 4th ed.* Hoboken: Wiley.
- Breitung, J. (2000). "The local power of some unit root tests for panel data", from Baltagi, Fomby, Hill (eds), *Nonstationary Panels, Panel Cointegration, and Dynamic Panels (Advances in Econometrics, Volume 15)*, Emerald Group Publishing, pp.161-177.
- Breusch, T.S. (1978). "Testing for Autocorrelation in Dynamic Linear Models." *Australian Economics Papers*, Vol. 17, pp. 334-355.
- Breusch, T.S. and A.R. Pagan (1979). "A Simple Test for Heteroscedasticity and Random Coefficient Variation." *Econometrica*, Vol. 47, pp. 1287-1294.
- Brockwell, P.J. and R.A. Davis (2002). *Introduction to Time Series Forecasting, 2nd Edition*. New York: Springer–Verlag.
- Brockwell, P.J. and R.A. Davis (1991). *Time Series: Theory and Methods, 2nd Edition*. New York: Springer–Verlag.

Bibliography

- Brown, B. and S. Maital (1981). "What Do Economists Know? An Empirical Study of Experts' Expectations." *Econometrica*, Vol. 49, pp. 491-504.
- Brown, R.L., J. Durbin and J.M. Evans (1975). "Techniques for Testing the Constancy of Regression Relationships over Time." *Journal of the Royal Statistical Society, Series B*, Vol. 37, pp. 149-192.
- Burg, J.(1967), "Maximum Entropy Spectral Analysis", *Proceedings of the 37th Meeting of the Society of Exploration Geophysicists*.
- Cai, J. (1994). "A Markov Model of Switching-Regime ARCH." *Journal of Business and Economic Statistics*, vol 12(3), pp 309-316.
- Campbell, J.Y., A. Lo, and A.C. MacKinlay (1997). *The Econometrics of Financial Markets*. Princeton: Princeton University Press.
- Canova, F. and G. De Nicolo (2002). "Monetary Disturbances Matter for Business Fluctuations in the G-7," *Journal of Monetary Economics*, Vol. 49(6), pp. 1131-1159.
- Carter, C.K. and R. Kohn (1994), "On Gibbs Sampling for State Space Models." *Biometrika*, Vol. 81, No. 3, pp. 541-553
- Chan, K.C., A. Karolyi, F. Longstaff and A. Sanders(1992), "An Empirical Comparison of Models of the ShortTerm Interest Rate." *Journal of Finance*, vol 47, no 3, pp. 1209-1227.
- Clark, T. E. & M.W. McCracken (2001), "Tests of equal forecast accuracy and encompassing for nested models," *Journal of Econometrics*, vol. 105, no. 1, pp 85-110.
- Commandeur, J.J.F. and S.J. Koopman (2007). *An Introduction to State Space Time Series Analysis*. Oxford: Oxford University Press.
- Corana, A., M. Marchesi, C. Martini and S. Ridella (1987), "Minimizing Multimodal Functions of Continuous Variables with the Simulated Annealing Algorithm", *ACM Transactions on Mathematical Software*, Vol. 13, No. 3, pp 262-280.
- Davidson, R. and J. MacKinnon (1981). "Several Tests for Model Specification in the Presence of Alternative Hypotheses." *Econometrica*, Vol. 49, pp. 781-793.
- Davidson, R and J. MacKinnon (1993). *Estimation and Inference in Econometrics*. Oxford: Oxford University Press.
- DeLurgio, S. (1998). *Forecasting Principles and Applications*. Boston: Irwin McGraw-Hill.
- Dempster, A.P., N.M. Laird and D.B. Rubin(1977). "Maximum Likelihood from Incomplete Data via the EM Algorithm," *Journal of the Royal Statistical Society. Series B*, Vol. 39, No. 1, pp. 1-38
- Dickey, D. and W.A. Fuller (1979). "Distribution of the Estimators for Time Series Regressions with a Unit Root." *Journal of the American Statistical Association*, Vol. 74, pp. 427-431.
- Diebold, F.X. (2004). *Elements of Forecasting, 3rd Edition*. Cincinnati: South-Western.
- Diebold, F.X. and R.S. Mariano(1995). "Comparing Predictive Accuracy." *Journal of Business & Economic Statistics*, Vol 13, no 3, pp 253-63.
- Doan, T.A. (2010). "Practical Issues with State-Space Models with Mixed Stationary and Non-Stationary Dynamics," Estima Technical Paper, (1)

- Doan, T., R. Litterman, and C.A. Sims (1984). "Forecasting and Conditional Projection Using Realistic Prior Distributions." *Econometric Reviews*, Vol. 3, pp. 1-100.
- Dueker, M.S.(1997). "Markov Switching in GARCH Processes and Mean-Reverting Stock-Market Volatility." *Journal of Business and Economic Statistics*, vol 15(1), pp 26–34.
- Durbin, J. (1969), "Tests for Serial Correlation in Regression Analysis Based on the Periodogram of Least Squares Residuals", *Biometrika*, vol 56, pp 1-16.
- Durbin, J. (1970). "Testing for Serial Correlation in Least-Squares Regression When Some of the Regressors Are Lagged Dependent Variables." *Econometrica*, Vol. 38, pp. 410-421.
- Durbin, J. and S.J. Koopman (2012). *Time Series Analysis by State Space Methods*, 2nd ed. Oxford: Oxford University Press.
- Eicker, F. (1967). "Limit Theorems for Regression with Unequal and Dependent Errors." *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*. Berkeley: University of California Press, pp. 59-82.
- Elliott, G., T. Rothenberg, and J. Stock (1996). "Efficient Tests for an Autoregressive Unit Root." *Econometrica*, vol 64, no 4, pp 813–836.
- Enders, W. (2014). *Applied Econometric Time Series, 4th Edition*. Hoboken: Wiley.
- Engle, R. (1974). "Band Spectrum Regression." *International Economic Review*, Vol. 15, pp. 1-11.
- Engle, R. (1982). "Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation." *Econometrica*, Vol. 50, pp. 987-1008.
- Engle, R. (2002). "Dynamic Conditional Correlation—A Simple Class of Multivariate GARCH Models." *Journal of Business and Economic Statistics*, Vol. 20, Number 3, pp. 339-350.
- Engle, R. and C.W.J. Granger (1987). "Co-Integration and Error Correction: Representation, Estimation and Testing." *Econometrica*, Vol. 55, pp. 251-76.
- Engle, R. and K.F. Kroner (1995). "Multivariate Simultaneous Generalized ARCH." *Econometric Theory*, Vol. 11, pp. 122-150.
- Engle, R., D.M. Lilien, and R.P. Robins (1987). "Estimating Time Varying Risk Premia in the Term Structure: The ARCH-M Model." *Econometrica*, Vol. 55, pp. 391-407.
- Fair, R.C. (1970). "The Estimation of Simultaneous Equation Models with Lagged Endogenous Variables and First Order Serially Correlated Errors." *Econometrica*, Vol. 38, pp. 507-516.
- Fair, R.C. (1979). "An Analysis of the Accuracy of Four Macroeconomic Models." *Journal of Political Economy*, Vol. 87, pp. 701-718.
- Fox, R. and M. S. Taqqu (1986). "Large-Sample Properties of Parameter Estimates from Strongly Dependent Stationary Gaussian Time Series." *The Annals of Statistics*, Vol. 14, pp. 517-532.
- Fuller, W.A. (1976). *Introduction to Statistical Time Series*. New York: Wiley.
- Gelfand, A. E. and A.F.M. Smith (1990). "Sampling-Based Approaches to Calculating Marginal Densities", *Journal of the American Statistical Association*, Vol. 85, pp. 398-409.

Bibliography

- Geweke, J., R. Meese and W. Dent (1982). "Comparing Alternative Tests of Causality in Temporal Systems." *Journal of Econometrics*, Vol. 21, pp. 161-194.
- Geweke, J. (1989). "Bayesian Inference in Econometric Models Using Monte Carlo Integration," *Econometrica*, November 1989, pp. 1317-1339.
- Geweke, J. and S. Porter-Hudak (1983). "The Estimation and Application of Long Memory Time Series models." *Journal of Time Series Analysis*, Vol. 4, pp. 221-238.
- Glosten, L., R. Jagannathan and D. Runkle (1993). "On the Relation between the Expected Value and the Volatility of the Nominal Excess Return on Stocks." *Journal of Finance*, Vol. 48, pp. 1779-1801.
- Godfrey, L.G. (1978). "Testing Against General Autoregressive and Moving Average Error Models When the Regressors Include Lagged Dependent Variables." *Econometrica*, Vol. 46, pp. 1293-1302.
- Gomez, V. and A. Maravall(2001). "Automatic Modeling Methods for Univariate Series", in Peña, Tiao and Tsay, eds., *A Course in Time Series Analysis*, New York: Wiley.
- Granger, C.W.J. (1969). "Investigating Causal Relations by Econometric Models and Cross-Spectral Models." *Econometrica*, Vol. 37, pp. 424-438.
- Granger, C.W.J. and R. Joyeux (1980), "An Introduction to Long-Memory Time Series Models and Fractional Differencing." *Journal of Time Series Analysis*, Vol. 1, pp. 15-39.
- Granger, C.W.J. and P. Newbold (1986). *Forecasting Economic Time Series*. New York: Academic Press.
- Granger, C.W.J. and P. Newbold (1974). "Spurious Regressions in Econometrics", *Journal of Econometrics*, Vol. 2, pp. 111-120.
- Greene, W.H. (2008). *Econometric Analysis, 6th Edition*. New Jersey: Prentice Hall.
- Greene, W.H. (2012). *Econometric Analysis, 7th Edition*. New Jersey: Prentice Hall.
- Gujarati, D.N. (2003). *Basic Econometrics, 4th Edition*. New York: McGraw-Hill.
- Hadri, K. (2000). "Testing for stationarity in heterogeneous panel data", *Econometrics Journal*, vol 3, no 2, 148-161.
- Hafner, C. and H. Herwartz(2006). "Volatility impulse responses for multivariate GARCH models: An exchange rate illustration", *Journal of International Money and Finance*, vol 25, no 5, pp 719-740.
- Hall, A. (2000). "Covariance Matrix Estimation and the Power of the Overidentifying Restrictions Test." *Econometrica*, Vol. 68, pp. 1517-1528.
- Hamilton, J. (1994). *Time Series Analysis*. Princeton: Princeton University Press.
- Hamilton, J and R. Susmel (1994). "Autoregressive conditional heteroskedasticity and changes in regime." *Journal of Econometrics*, Vol 64, no 1-2, pp 307-333.
- Hannan, E.J. (1963). "Regression for Time Series." In *Proceedings of Symposium on Time Series Analysis*, M. Rosenblatt, ed. New York: Wiley.

- Hansen, B. (1992). "Parameter Instability in Linear Models." *Journal of Policy Modeling*, 14 (4), pp. 517-533.
- Hansen, B. (1996). "Inference When a Nuisance Parameter is Not Identified Under the Null Hypothesis." *Econometrica*, 1996, Vol. 64, No. 2, pp. 413-430.
- Hansen, L.P. (1982): "Large Sample Properties of Generalized Method of Moments Estimators." *Econometrica*, Vol. 50, pp. 1029-1054.
- Hansen, L.P. and K.J. Singleton (1982). "Generalized Instrumental Variables Estimation of Non-Linear Rational Expectations Models." *Econometrica*, Vol. 50, pp. 1269-1286.
- Harris, R. and E. Tzavalis (1999). "Inference for unit roots in dynamic panels where the time dimension is fixed." *Journal of Econometrics*, vol 91, pp 201-226.
- Harvey, A. C. (1989). *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge: Cambridge University Press.
- Hausman, J.A. (1978). "Specification Tests in Econometrics." *Econometrica*, Vol. 46, pp. 1251-1272.
- Hausman, J.A. and W.E. Taylor (1981). "Panel Data and Unobservable Individual Effects." *Econometrica*, Vol. 49, pp. 1377-1398.
- Hayashi, F. (2000). *Econometrics*. Princeton, New Jersey: Princeton University Press
- Hayashi, F. and C.A. Sims (1983). "Nearly Efficient Estimation of Time Series Models with Predetermined, But Not Exogenous, Instruments." *Econometrica*, Vol. 51, pp. 783-798.
- Heckman, J. (1976). "The Common Structure of Statistical Models of Truncation, Sample Selection, and Limited Dependent Variables and a Simple Estimator for Such Models." *Annals of Economic and Social Measurement*, Vol. 5, pp. 475-492.
- Hill, R.C., W.E. Griffiths, and G.C. Lim (2008). *Principles of Econometrics, 3rd Edition*. New York: Wiley.
- Hodrick, R. and E. Prescott (1997). "Post-War U.S. Business Cycles: An Empirical Investigation." *Journal of Money, Credit and Banking*, vol 29, no. 1, pp 1-16.
- Holtz-Eakin, D., Newey, W. and S. Rosen (1988). "Estimating Vector Autoregressions with Panel Data." *Econometrica*, vol. 56, no 6, pp 1371-95.
- Hosking, J.R.M. (1981). "Fractional Differencing." *Biometrika*, Vol. 68, pp. 165-176.
- Hsiao, C. (1986). *Analysis of Panel Data*. Cambridge: Cambridge University Press.
- Im, K. S., M. H. Pesaran and Y. Shin (2003). "Testing for Unit Roots in Heterogeneous Panels", *J. of Econometrics*, vol 115, pp 53-74.
- Inclan, C. and G. Tiao (1994). "Use of Cumulative Sums of Squares for Retrospective Detection of Changes in Variance", *J of American Statistical Association*, vol 89, pp 913-923.
- Jacquier, E., N.G. Polson, and P.E. Rossi (1994). "Bayesian Analysis of Stochastic Volatility Models." *Journal of Business and Economic Statistics*, Vol. 12, No. 4, pp. 371-389.
- Jagannathan, R. and Z. Wang (1996). "The Conditional CAPM and the Cross-Section of Expected Returns", *Journal of Finance*, Vol. 51, No. 1, pp. 3-35.

Bibliography

- Johnston, J. and J. DiNardo (1997). *Econometric Methods, 4th Edition*. New York: McGraw Hill.
- Juselius, K. (2006). *The Cointegrated VAR Model.: Methodology and Applications*. New York: Oxford University Press.
- Kadiyala, K.R. and S. Karlsson (1997). "Numerical Methods for Estimation and Inference in Bayesian VAR-Models." *Journal of Applied Econometrics*. Vol. 12, pp. 99-132.
- King, R.G., C.I. Plosser, J.H. Stock and M.W. Watson (1991). "Stochastic Trends and Economic Fluctuations", *American Economic Review*, Vol. 81, pp. 819-840.
- Kim, C.-J. and C. R. Nelson (1999), *State-Space Models with Regime Switching*. MIT Press.
- Kim, S., N. Shephard, and S. Chib (1998). "Stochastic Volatility: Likelihood Inference and Comparison with ARCH Models", *Review of Economic Studies*, Vol. 65, pp. 361-93.
- Koenker, R. (1981). "A note on studentizing a test for heteroscedasticity," *Journal of Econometrics*, Vol. 17, no. 1, pp. 107-112.
- Koop, G. (2003). *Bayesian Econometrics*. Chichester: Wiley.
- Koopman, S.J. (1997). "Exact Initial Kalman Filtering and Smoothing for Nonstationary Time Series Models." *Journal of the American Statistical Association*. Vol. 92, pp. 1630-1638.
- Koopmans, L.H. (1974). *The Spectral Analysis of Time Series*. New York: Academic Press.
- Koutmos, G. (1996). "Modeling the Dynamic Interdependence of Major European Stock Markets," *Journal of Business, Finance and Accounting*, vol 23, pp 975-988.
- Kwiatkowski, D., P. Phillips, P. Schmidt, and Y. Shin (1992). "Testing the Null Hypothesis of Stationarity against the Alternative of a Unit Root." *Journal of Econometrics*, vol 54, nos 1-3, pp. 159-178.
- Lam, P. (1990). "The Hamilton model with a general autoregressive component: estimation and comparison with other models of economic time series." *Journal of Monetary Economics*, Vol. 26, no. 3, pp. 409-432.
- Lancaster, T. (1990). *The Econometric Analysis of Transition Data*. Cambridge: Cambridge University Press.
- Leamer, E. (1978). *Specification Searches*. New York: Wiley.
- Lee, J. and M. C. Strazicich (2003). "Minimum LM Unit Root Test with Two Structural Breaks," *Review of Economics and Statistics*, Vol. 85(4), pp. 1082-1089.
- Levin A., C.-F. Lin and C.-S. Chu(2002), "Unit root tests in panel data: Asymptotic and finite-sample properties", *Journal of Econometrics*, vol 108, pp 1-24.
- Ling, S. and M. McAleer (2003). "Asymptotic theory for a new vector ARMA-GARCH model." *Econometric Theory*, Vol. 19, pp. 280-310.
- Ljung, G.M. and G.E.P. Box (1978). "On a Measure of Lack of Fit in Time Series Models." *Biometrika*, Vol. 67, pp. 297-303.
- Luenberger, D.A. (1989). *Linear and Nonlinear Programming (2nd. Edition)*. Reading, Mass.: Addison-Wesley.

- Lumsdaine, R. and D. Papell(1997). "Multiple trend breaks and the unit root hypothesis." *Review of Economics and Statistics*, vol 79, 212-218.
- Lütkepohl, H. (2006). *New Introduction to Multiple Time Series*. Berlin: Springer.
- MacKinnon, J. (1991). "Critical Values for Cointegration Tests", *Long- Run Economic Relationships*, R.F. Engle and C.W.J. Granger, eds. London: Oxford University Press.
- MacKinnon, J., H. White, and R. Davidson (1983). "Tests for Model Specification in the Presence of Alternative Hypotheses: Some Further Results." *Journal of Econometrics*, Vol. 21, pp. 53-70.
- Marquardt, D.(1963). "An Algorithm for Least-Squares Estimation of Nonlinear Parameters." *SIAM Journal on Applied Mathematics*, vol 11, pp 431-441.
- Martin, V, S. Hurn and D. Harris(2013). *Econometric Modelling with Time Series: Specification, Estimation and Testing*. Cambridge: Cambridge University Press.
- McCurdy, T.H. and I.G. Morgan (1991). "Tests for Systematic Risk Components in Deviations from Uncovered Interest Rate Parity." *Review of Economic Studies*.
- McFadden, D. (1974). "Conditional Logit Analysis of Qualitative Choice Behavior." In *Frontiers in Econometrics*, P. Zarembka ed. New York: Academic Press.
- McLeod, A. and W. Li (1983). "Diagnostic checking ARMA time series models using squared residual autocorrelations." *Journal of Time Series Analysis*, Vol 4, pp 269–273.
- Newey, W. and K. West (1987). "A Simple Positive-Definite Heteroskedasticity and Autocorrelation Consistent Covariance Matrix." *Econometrica*, Vol. 55, pp. 703-708.
- Nelson, C.R. and C.I. Plosser (1982). "Trends and Random Walks in Macroeconomic Time Series." *Journal of Monetary Economics*, Vol. 10, pp. 139-162.
- Nelson, D. B. (1990). "Stationary and Persistence in the GARCH(1,1) Model." *Econometric Theory*, Vol. 6, pp 318-334.
- Nelson, D.B. (1991) "Conditional Heteroskedasticity in Asset Returns: A New Approach." *Econometrica*, Vol. 59, pp. 347-370.
- Novalés, A., E. Fernandez & J. Ruiz(2009), *Economic Growth: Theory and Numerical Solution Methods*, Springer-Verlag.
- Nyblom, Jukka (1989) "Testing for Constancy of Parameters Over Time", *Journal of the American Statistical Association*, Vol. 84, pp. 223-230.
- Olsen, R. (1978). "A Note on the Uniqueness of the Maximum Likelihood Estimator in the Tobit Model", *Econometrica*, Vol. 46, pp. 1211-1215
- Pagan, A. and A. Ullah (1999). *Nonparametric Econometrics*. Cambridge: Cambridge University Press.
- Patton, Andrew J. (2011). "Volatility forecast comparison using imperfect volatility proxies," *Journal of Econometrics*, vol. 160, no 1, pp 246-256.
- Perron, P. (1989). "The Great Crash, the Oil Price Shock, and the Unit Root Hypothesis." *Econometrica*, Econometric Society, Vol. 57(6), pp. 1361-1401.

Bibliography

- Perron, P. (1990). "Testing for a Unit Root in a Time Series with a Changing Mean," *Journal of Business and Economic Statistics*, Vol. 8(2), pp. 153-62.
- Perron, P. (2006), "Dealing with Structural Breaks," *Palgrave Handbook of Econometrics*, Vol. 1, pp 278-352.
- Pesaran, H. H. and Y. Shin (1998). "Generalized Impulse Response Analysis in Linear Multivariate Models," *Economics Letters*, Vol. 58(1), pp. 17-29.
- Phillips, P.C.B. (1987). "Time Series Regressions with a Unit Root." *Econometrica*, Vol. 55, pp. 277-301.
- Phillips, P.C.B. and P. Perron (1988). "Testing for a Unit Root in Time Series Regressions." *Biometrika*, Vol. 65, pp. 335-346.
- Pindyck, R. and D. Rubinfeld (1998). *Econometric Models and Economic Forecasts, 4th Edition*. New York: McGraw-Hill.
- Politis, D. N., J. P. Romano (1994). "The Stationary Bootstrap". *Journal of the American Statistical Association*, vol. 89 , pp. 1303—1313.
- Press, W.H., B.P. Flannery, S.A. Teukolsky and W.T. Vetterling (2007). *Numerical Recipes, 3rd Edition*. New York: Cambridge University Press.
- Ramsey, J.B. (1969). "Tests for Specification Errors in Classical Linear Least Squares Regression Analysis." *Journal of the Royal Statistical Society B*, Vol. 32, pp. 350-371.
- Rubio-Ramirez, J.F., D.F. Waggoner, and T. Zha (2010). "Structural Vector Autoregressions: Theory of Identification and Algorithms for Inference," *Review of Economic Studies*, Vol. 77, No. 2, pp. 665-696.
- Schmidt, P. and P.C.B. Phillips (1992). "LM Test for a Unit Root in the Presence of Deterministic Trends," *Oxford Bulletin of Economics and Statistics*, Vol 54, pp. 257-287.
- Schwarz, G. (1978). "Estimating the Dimension of a Model." *Annals of Statistics*, Vol. 6, pp. 461-464.
- Schwert, G.W. (1989). "Tests for Unit Roots: A Monte Carlo Investigation." *Journal of Business and Economic Statistics*, Vol. 7, pp. 147-159.
- Shiller, R.J. (1973). "A Distributed Lag Estimator Derived from Smoothness Priors." *Econometrica*, Vol. 41, pp. 775-788.
- Sims, C.A. (1972). "Money, Income and Causality." *American Economic Review*, Vol. 62, pp. 540-552.
- Sims, C.A. (1974a). "Distributed Lags." In *Frontiers of Quantitative Economics*, M.D. Intriligator and D.A. Kendrick, eds. Amsterdam: North Holland.
- Sims, C.A. (1974b). "Seasonality in Regression." *Journal of the American Statistical Association*, Vol. 69, pp. 618-626.
- Sims, C.A. (1980). "Macroeconomics and Reality." *Econometrica*, Vol. 48, pp. 1-49.
- Sims, C.A. (1986). "Are Forecasting Models Usable for Policy Analysis?" *Federal Reserve Bank of Minneapolis Quarterly Review*, Winter.

- Sims, C.A. (1988). "Bayesian Skepticism on Unit Root Econometrics." *Journal of Economic Dynamics and Control*, Vol. 12, pp. 463-474.
- Sims, C.A. (1993). "A 9 Variable Probabilistic Macroeconomic Forecasting Model." In *Business Cycles, Indicators, and Forecasting*, Stock and Watson, eds., University of Chicago Press.
- Sims, C.A. (2000). "Using a likelihood perspective to sharpen econometric discourse: Three examples." *Journal of Econometrics*, Vol. 95, Issue 2, pp. 443-462.
- Sims, C.A. (2002). "Solving Linear Rational Expectations Models", *Computational Economics*, Vol. 20, Nos. 1-2, pp. 1-20.
- Sims, C.A., J.H. Stock, and M.W. Watson (1990). "Inference in Linear Time Series Models with Some Unit Roots", *Econometrica*, Vol. 58, No. 1, pp. 113-144.
- Sims, C.A. and T. Zha (1999). "Error Bands for Impulse Responses." *Econometrica*, Vol. 67, pp. 1113-1156.
- Stock, J. (1987). "Asymptotic Properties of Least Squares Estimators of Cointegrating Vectors." *Econometrica*, Vol. 55, pp. 1035-56.
- Stock, J. and M. Watson (2011). *Introduction to Econometrics*, 3rd Edition. Boston: Pearson.
- Swamy, P. (1970) "Efficient Inference in a Random Coefficient Regression Model." *Econometrica*, Vol. 38, pp. 311-323.
- Terasvirta, T. (1994). "Specification, Estimation and Evaluation of Smooth Transition Autoregressive Models", *Journal of the American Statistical Association*, Vol. 89, No. 425, pp. 208-218.
- Theil, H. (1971). *Principles of Econometrics*. New York: Wiley.
- Tsay, R.S. (2010). *Analysis of Financial Time Series, 3rd Edition*. New York: Wiley.
- Uhlig, H. (2005). "What Are the Effects of Monetary Policy on Output? Results from an Agnostic Identification Procedure," *Journal of Monetary Economics*, Vol. 52, pp. 381-419.
- Verbeek, M. (2008). *A Guide to Modern Econometrics, 3rd Edition*. New York: Wiley.
- Watson, M.W. (1993). "Measures of Fit for Calibrated Models", *Journal of Political Economy*, Vol. 101, No. 6, pp. 1011-1041.
- West, K. and D. Cho(1995). "The predictive ability of several models of exchange rate volatility," *Journal of Econometrics*, vol. 69, no 2, pp 367-391.
- West, M. and J. Harrison (1997). *Bayesian Forecasting and Dynamic Models, 2nd Edition*. New York: Springer-Verlag.
- White, H. (1980). "A Heteroskedasticity-Consistent Covariance Matrix Estimator and Direct Test for Heteroskedasticity." *Econometrica*, Vol. 48, pp. 817-838.
- Wooldridge, J. (2009), *Introductory Econometrics, 4th edition*. Mason, Ohio: South-Western.
- Wooldridge, J. (2010). *Econometric Analysis of Cross Section and Panel Data, 2nd ed.* Cambridge, Mass.: The MIT Press.
- Wu, D.-M. (1973). "Alternative tests of independence between stochastic regressors and disturbances." *Econometrica*, vol 42, 529-546.

Index

Symbols

– operator, Int–40.
 for matrices, UG–27.
 –= operator, Int–40.
 ; multiple statements per line, Int–57.
 .AND. operator, Int–41.
 .EQ. operator, Int–41.
 .GE. operator, Int–41.
 .GT. operator, Int–41.
 .LE. operator, Int–41.
 .LT. operator, Int–41.
 .NE. operator, Int–41.
 .NOT. operator, Int–41.
 .OR. operator, Int–41.
 .^ operator, UG–27.
 .* operator, UG–27.
 ./ operator, UG–27.
 {...} for lags
 in expressions, Int–25, RM–11.
 in regressor lists, Int–79.
 @procedure, Int–33, UG–470, RM–167.
 *
 for comments, Int–74.
 for default entry, Int–31.
 with SMPL, RM–455.
 operator, Int–40.
 ** operator, Int–40.
 for matrices, UG–27.
 *= operator, Int–40.
 /
 for default entries, Int–31.
 operator, Int–40.
 for matrices, UG–27.
 for integers, Int–42.
 /= operator, Int–40.
 \\ for line breaks, Int–132.
 & symbol
 for strings, UG–23.
 + operator, Int–40.
 for matrices, UG–27.
 for MODELS, UG–15.
 for strings, UG–23.
 += operator, Int–40.
 < operator, Int–41.
 <= operator, Int–41.
 <> operator, Int–41.
 = operator
 assignment, UG–5.
 with matrices, UG–27.
 == operator, Int–41.
 > operator, Int–41.
 >= operator, Int–41.
 || for in-line matrices, Int–91, UG–26, RM–49.
 ~ operator
 for matrices, UG–27.
 ~\ operator
 for matrices, UG–27.
 ~~ operator
 for matrices, UG–27.
 \$ line continuation, Int–57, Int–76.

A

@ABLAGS procedure, UG–429.
 Acceptance-rejection method, UG–511.
 ACCUMULATE instruction, RM–2.
 ADAPTIVE.RPF example, UG–434.
 Add factors, UG–284.
 Akaike criterion, UG–60, UG–182.
 @BJEST procedure, UG–198.
 example, UG–61.
 for VARs, UG–209, UG–212.
 AKAIKE.RPF example, UG–61.

Alert dialog box, RM–321.
 Algorithms
 BFGS, UG–116, UG–146.
 BHHH, UG–116, UG–146.
 Cochrane-Orcutt, UG–275.
 Gauss-Seidel, UG–281.
 genetic, UG–118, UG–146.
 Hildreth-Lu, UG–275.
 Levenberg-Marquardt, RM–338.
 random numbers, RM–443.
 simplex, UG–117, UG–146, UG–152.
 Aliasing. *See* FOLD instruction.
 ALLOCATE instruction, RM–4.
 for cross-sectional data, UG–394.
 using series parameter, UG–22.
 Almon lags, UG–58.
 Analysis of variance (panel data), UG–420, RM–393.
 Andrews, D., UG–464.
 Annual data, RM–27.
 Antithetic acceleration, UG–518.
 Appending data to a file, RM–361.
 AR1
 fitted values, RM–9.
 forecasts, UG–163, RM–9.
 AR1 instruction, Int–78, UG–45, RM–6.
 FRML option, UG–279.
 instrumental variables, RM–9, RM–272.
 simultaneous equation models, UG–275.
 with panel data, UG–422.
 AR1.RPF example, UG–45.

- ARCH/GARCH Wizard, RM–207.
- Archiving data, RM–391.
- ARCH model, UG–290.
- GARCH instruction, RM–207.
- Markov switching, UG–390.
- testing for, UG–80.
- @ARCHTEST procedure, UG–80.
- ARDL model, UG–59.
- Arellano-Bond estimator, UG–429.
- ARELLANO.RPF example, UG–429.
- ARFIMA models, UG–464.
- %ARG function, UG–448.
- ARIMA models, Int–64, UG–176, UG–181, RM–15.
- defining with EQUATION, RM–143.
- example, UG–182.
- forecasting, UG–176, UG–181.
- initial estimates using INITIAL, RM–265.
- intervention terms, UG–200.
- non-consecutive lags, RM–16.
- procedures for, UG–192.
- residual autocorrelations, UG–197.
- selecting, UG–181.
- ARIMA.RPF example, UG–182.
- Arithmetic
- expressions, Int–40, RM–541.
- operators, RM–541.
- complex values, UG–448.
- @ARMADLM procedure, UG–329.
- ARMA model
- spectrum of, UG–459.
- @ARMASPECTRUM procedure, UG–459.
- ARMAX models, UG–203.
- Arrays, Int–91, UG–18. *See also* Matrix.
- ASSOCIATE instruction, RM–11.
- with EQUATION, RM–143.
- AUTOBOX.RPF example, UG–199.
- Autocorrelations, UG–181, RM–55.
- correction
- higher-order, RM–10.
- with AR1, RM–6.
- graphing, Int–145.
- inverse, RM–55, RM–57.
- partial, RM–57.
- residual, UG–197.
- Autocovariances, RM–55.
- B**
- Backpropagation, UG–441.
- @BAIPERRON procedure, UG–367.
- Banded matrices, RM–186.
- Bartlett window, UG–41.
- Basic structural model (BSM), UG–329.
- Batch mode
- graphs in, Int–153.
- Bayesian methods
- antithetic acceleration, UG–518.
- GARCH model, UG–525, UG–537.
- Gibbs sampling, UG–528, UG–530.
- importance sampling, UG–523.
- for Markov switching, UG–381.
- Metropolis-Hastings, UG–535, UG–537.
- Monte Carlo integration, UG–517.
- test for unit roots, UG–103.
- time-varying coefficients, UG–267.
- VAR, UG–253, RM–458.
- @BDINDTESTS procedure, UG–298.
- Beach-MacKinnon AR1 algorithm, RM–6.
- BEKK GARCH model, UG–303.
- Benchmarking, Int–62.
- Bernanke, B., UG–216, UG–218.
- Berndt, E. K., UG–114, UG–116.
- Berndt, Hall, Hall, Hausman (BHHH), UG–116.
- %BESTREP function, RM–109.
- Beta distribution
- random draws, UG–505, UG–506.
- Beta values for securities, UG–484.
- %BETA vector, UG–25, UG–35.
- BFGS method, UG–116.
- BHHH method, UG–116.
- Binary choice models, UG–395.
- fitted values for, RM–385.
- Binary data, Int–123.
- Biweekly data, RM–27.
- @BJAUTOFIT procedure, UG–198.
- @BJDIFF procedure, UG–193.
- @BJFORE procedure, UG–196.

- @BJIDENT procedure, UG–193.
 @BJTRANS procedure, UG–192.
 Blanchard-Quah decomposition, UG–223, UG–228.
 Block diagonal matrix creating, UG–27.
 Block exogeneity test, UG–214.
 Bollerslev, T., UG–290, UG–296.
 BONDS.RPF example, UG–150.
 BOOT instruction, UG–540, RM–13.
 Bootstrapping, UG–505, UG–540, RM–13, RM–189.
 computing a probability, UG–501.
 example, UG–542.
 %BOXCOX function, UG–147, RM–312.
 Box-Cox transformation, UG–147, RM–308.
 Box, G. E. P., UG–81, UG–181, RM–56, RM–265.
 BOXJENK instruction, UG–182, RM–15.
 intervention models, UG–200.
 Box-Jenkins wizard, RM–15.
 Box-Pierce Q, UG–81.
 Box plots, RM–216.
 %BQFACTOR function, UG–224.
 BRANCH instruction, RM–222.
 BREAK instruction, RM–24.
 Burn-in, UG–510.
 Business day data, RM–28.
- C**
- %CABS function, UG–448.
 CACCUMULATE instruction, RM–25.
 CADD instruction, RM–45.
 CAGAN.RPF example, UG–354.
 Calendar functions, RM–26, RM–31.
 CALENDAR instruction, Int–21, RM–26.
 for panel data, UG–413.
 saving/recalling settings, RM–30.
 Calendar wizard, Int–8, RM–27.
 CARDS utility, RM–150, RM–295.
 Carter-Kohn simulation, UG–332.
 Cass-Koopmans growth model, RM–128.
 CATALOG instruction, Int–111, RM–32.
 CATS software, UG–104.
 Cauchy distribution random draws, UG–507.
 Causality tests, UG–84, UG–214.
 CAUSAL.RPF example, UG–84.
 CDF instruction, UG–71, RM–33.
 CDIVIDE instruction, RM–45.
 %CDSTAT variable, UG–73.
 Censored samples, UG–403, RM–284.
 Census X11/X12, RM–537.
 %CEXP function, UG–448.
 CEXP instruction, RM–38.
 CHANGE instruction, RM–34.
 %CHISQR function, UG–71.
- Chi-squared distribution random draws, UG–506.
 %CHOICE function, RM–365.
 CHOICE instruction, UG–493, RM–320.
 Choleski factorization, UG–217, UG–218, UG–228.
 Chow test, UG–86.
 CHOWTEST.RPF example, UG–87.
 CLABELS instruction, RM–36.
 CLEAR instruction, RM–37.
 Clear Memory menu command, RM–138.
 CLN instruction, UG–449, RM–38.
 %CLOG function, UG–448.
 CLOSE instruction, RM–39.
 %CLOSESTDATE function, UG–30.
 %CLOSESTWEEKDAY function, UG–30.
 CLUSTER option, UG–128.
 CMASK instruction, RM–40.
 %CMOM array, UG–25.
 CMOMENT instruction, UG–25, UG–61, UG–516, RM–41.
 CMOMENT option, RM–41.
 %CMPLX function, UG–448.
 CMULTIPLY instruction, UG–449, UG–456, RM–45.
 CMVE instruction, RM–38.
 Cochrane-Orcutt, UG–45, UG–275.
 Coefficients
 assigning to an equation, RM–143.
 cross-equation restrictions, RM–487.
 of equation, RM–11.
 reading from a file, RM–12.

- sums of, UG-71, RM-483.
- testing, UG-71, RM-504.
- time-varying, UG-267.
- updating with Kalman filter, RM-274.
- VECTOR of, RM-292.
- %COEFF array, RM-5.
- Coherence, UG-458, RM-376.
- Cointegration, UG-104, UG-247.
- COINTTST.RPF example, UG-105.
- Comments, Int-74.
- Compacting data series, RM-75.
- Compiled section, UG-468.
 - line number of error in, UG-480.
 - procedures, RM-389.
- COMPLEX data type, UG-10, UG-14.
- Complex numbers
 - expressions using, UG-448.
 - operators on, RM-542.
- Complex series, UG-449.
 - converting from real, RM-426.
 - converting to real, RM-68.
 - creating, RM-66, RM-426.
 - displaying, UG-454.
 - Fourier transforms, RM-172.
 - graphing, UG-454, RM-68.
 - labeling, RM-36.
 - padding, UG-452.
 - preparation, UG-452.
 - printing, RM-59.
 - reading in, RM-427.
 - %Z variable, RM-66.
- COMPUTE instruction, Int-6, UG-2, UG-12, RM-47.
- alternative to INPUT, RM-267.
- assigning data types, UG-3.
- expressions in, RM-541.
- for matrices, Int-91.
- Concatenation
 - of matrices, UG-27.
 - of strings, Int-132, UG-23.
- %CONCAT function, UG-484.
- @CONDITION procedure, UG-243, UG-245.
- Conditional blocks, Int-93, UG-468.
 - ending, RM-138.
 - IF/ELSE, RM-253.
 - UNTIL loops, RM-520.
 - WHILE loops, RM-531.
- Conditional forecasts, UG-242.
- Condition number, RM-135.
- CONDITION.RPF example, UG-245.
- Confidence bands
 - for impulse responses, UG-231.
- %CONJG function, UG-448.
- CONJUGATE instruction, RM-38.
- Consistent covariance matrices
 - with MCOV instruction, RM-313, RM-548.
- CONSTANT.RPF example, UG-90.
- Constants, Int-40.
- CONSTANT series, Int-27.
- Constrained optimization
 - non-linear, UG-130.
- CONSUMER.RPF example, UG-145.
- Continuation lines, Int-57, Int-76.
- Contour plots, RM-220.
- Convergence
 - non-linear estimation, UG-110.
 - setting criterion with NL-PAR, RM-337.
 - simultaneous equation models, UG-282.
- Convolution, UG-460.
- COPY instruction, Int-96, Int-111, RM-51.
 - with spreadsheets, Int-119.
- COPY unit, RM-361.
- CORRELATE instruction, UG-182, RM-55.
- Correlation
 - auto, RM-55.
 - of series, RM-61.
- Count data model, RM-89.
- Counterfactual simulation, UG-241.
- Covariance
 - auto, RM-55.
 - of series, RM-61.
- Covariance matrix modelling, RM-70.
 - of coefficients, UG-66, RM-292.
 - consistent estimate of, UG-66, UG-67, UG-125, RM-313, RM-548.
 - of series, RM-526.
- VAR (via KFSET), RM-279.
- Covariance Matrix wizard, RM-526.
- CPRINT instruction, RM-59.
- CREATE option
 - with LINREG, UG-64.
 - with RESTRICT and MRESTRICT, UG-63.
- Cross-correlations, RM-61.

Cross Correlations wizard, RM-61.
 Cross-equation restrictions, RM-487.
 CROSS instruction, RM-61.
 Cross-moment matrix computing, RM-41.
 Cross-periodograms, RM-532.
 Cross sectional data, Int-66, UG-394.
 @CROSSPEC procedure, UG-458.
 Cross-spectral analysis, UG-456, RM-377.
 CRSP format, Int-98.
 CSAMPLE instruction, RM-65.
 CSCALE instruction, RM-38.
 CSET instruction, UG-449, RM-66.
 %CSQRT function, UG-448.
 CSQRT instruction, RM-38.
 CSUBTRACT instruction, RM-45.
 CTOR instruction, RM-68.
 @CUMPDGM procedure, UG-82, UG-83.
 Cumulated periodogram test, UG-82, RM-25.
 CUSUM test, , UG-91.
 example, Int-138.
 @CUSUMTESTS procedure, UG-92.
 CVMODEL instruction, Int-90, UG-219, UG-225, RM-70.
 %CVTOCORR function, UG-310.
 CXTREMUM instruction, RM-25, RM-74.

D

Daily data, RM-28.
 removing gaps, RM-428.
 DAMP option, UG-41.
 Data
 archiving, RM-391.
 changing frequency of, Int-104, RM-30.
 cross-section, Int-66.
 detrending, Int-48.
 interpolating, RM-30.
 reading from a file, Int-18, Int-96.
 setting frequency of, RM-26.
 subset of, Int-67.
 transformations, Int-25, RM-446.
 writing to a file, Int-96.
 Data/Graphics menu
 Calendar, Int-8.
 Data Browsers, Int-98.
 Data (Other Formats), Int-18.
 Data (RATS Format), Int-85.
 Filter/Smooth, Int-44.
 Graph, Int-46.
 Scatter (X-Y) Graph, Int-71.
 Transformations, Int-26.
 Trend/Seasonals/Dummies, Int-53.
 DATA instruction, Int-22, Int-96, RM-75.
 changing data frequencies, Int-103.
 with holidays omitted, Int-108.
 spreadsheet files, Int-116.
 DATA I/O unit, RM-361.
 Data types, UG-2, UG-10.
 basic, UG-14.

declaring variables, RM-96.
 Data wizards, Int-18, RM-75.
 (Other Formats), Int-43, Int-103.
 (RATS Format), Int-85, Int-110.
 %DATEANDTIME function, UG-23.
 %DATELABEL function, UG-23.
 Dates
 arithmetic using, UG-3.
 in expressions, RM-11.
 range of, Int-30, Int-82.
 referring to, Int-21.
 specifying, RM-552.
 on spreadsheet files, Int-115, Int-119.
 Davidson-MacKinnon test, UG-94.
 %DAYCOUNT function, UG-30.
 %DAY function, UG-30.
 DBOX instruction, UG-12, RM-80.
 DCC GARCH, UG-305.
 DDV instruction, Int-79, UG-395, UG-398, UG-401, UG-402, UG-409, RM-89.
 DEBUG instruction, RM-95.
 DECLARE instruction, UG-11, UG-12, RM-96.
 %DECOMP function, UG-507.
 Decomposition
 Blanchard-Quah, UG-223.
 Choleski, UG-217.
 eigen, RM-134.
 error components, RM-393.
 historical, UG-239.
 QZ, RM-398.

- Schur, RM–398.
- structural, UG–218.
- of variance, UG–261, RM–148, RM–152.
- DEDIT instruction, Int–111, RM–98.
- saving changes after, RM–430.
- Default
 - entry range, RM–455.
- %DEFINED function, UG–476, RM–169, RM–365.
- DEFINE option, Int–64.
- Degrees of freedom
 - VAR with a prior, UG–258.
- DELETE instruction, Int–111.
- Delta method
 - with SUMMARIZE, RM–486.
- Density
 - empirical, RM–100.
- DENSITY instruction, UG–432, RM–100.
- Dependent variable
 - of equation, RM–143.
 - in a VAR, RM–525.
- DETERMINISTIC instruction, UG–208, UG–263, RM–103.
- @DFUNIT procedure, UG–100.
- %DIAG function, UG–328.
- Dialog boxes
 - INFOBOX, RM–263.
 - MESSAGEBOX, RM–321.
 - user-defined, UG–489.
- Dickey-Fuller unit root test, UG–100.
- Diebold-Mariano test, UG–171.
- DIEBP210.RPF example, RM–518.
- DIEBP228.RPF example, Int–138.
- DIEBP235.RPF example, Int–138.
- DIFFERENCE instruction, UG–182, RM–104.
- Differencing, Int–62, RM–104.
- fractional, UG–464, RM–104.
- matrix operator, RM–186.
- prior to forecasting, UG–177.
- prior to VAR models, UG–209, UG–263.
- seasonal, RM–104.
- Differencing wizard, Int–26, RM–104.
- DIF files, Int–114.
- Diffuse initialization
 - for state space model, UG–335.
- Dimensioning arrays, RM–107.
- DIMENSION instruction, UG–12, UG–17.
- Dirichlet distribution
 - random draw, UG–509.
- @DISAGGREGATE procedure, Int–105.
- Discrete dependent variables, UG–395, RM–89.
- DDV instruction, RM–89.
- Displaying
 - arrays, RM–535.
 - calculations, UG–4.
 - complex series, UG–454.
 - data series, RM–383.
- DISPLAY instruction, Int–4, UG–4, UG–12, RM–108.
- for matrices, Int–91.
- DISTRIBLAG.RPF example, UG–57.
- Distributed lags, UG–57.
- graphing, UG–58.
- lag length, UG–60.
- polynomial, RM–136.
- sums of coefficients, RM–483.
- Distributions
 - chi-squared, RM–33.
 - Dirichlet, UG–509.
 - extreme value, UG–395, UG–396, RM–90, RM–387.
- F, RM–33.
- GED (generalized error), UG–292, RM–207.
- logistic, RM–387.
- logit, RM–90.
- normal, RM–33.
- probit, RM–90.
- t, UG–292, RM–33.
- Wishart, UG–514.
- DLM. *See* Dynamic Linear Models.
- DLMCYCLE.RPF example, UG–341.
- DLMEXAM2.RPF example, UG–331.
- DLMEXAM3.RPF example, UG–332.
- DLMEXAM4.RPF example, UG–333.
- DLM instruction, Int–90, RM–112.
- for solving DSGE, RM–129.
- @DLMIRF procedure, UG–349.
- @DMARIANO procedure, UG–171.
- Doan, T., UG–208, UG–268, UG–269, RM–280.
- DOFOR instruction, Int–92, UG–468, RM–123.
- %DO function, UG–411, RM–122, RM–163.
- DO instruction, Int–92, UG–468, RM–121.

- DO loops
 EWISE as alternative to, RM-164.
 %DOW function, UG-30.
 DSGE instruction, UG-343, RM-125.
 DUMMY instruction, RM-130.
 Dummy observation prior, UG-272.
 Dummy variables, Int-60, Int-61.
 panel data, UG-417.
 seasonal, RM-439.
 for subsamples, RM-455.
 Duration analysis, UG-410.
 Durbin h statistic, UG-83.
 Durbin, J., , UG-321, UG-83.
 Durbin-Watson statistic, Int-37.
 %DURBIN variable, RM-292.
 Dynamic forecasts, UG-161, RM-189.
 Dynamic Linear Models, Int-90, UG-321, RM-112.
 forecasting with, UG-176.
 spectrum of, UG-459.
 Dynamic OLS, UG-249.
 Dynamic Stochastic General Equilibrium Models, RM-125.
- E**
- %EASTER function, UG-30.
 %EBW variable, RM-534.
 ECT instruction, UG-247, RM-132.
 ECT.RPF example, UG-249.
 %EDF variable, RM-534.
 EDIT instruction, Int-111.
 Edit menu
 Comment-Out Lines, Int-75.
 Copy as TeX, Int-16.
 Show Last Error, Int-81.
 To Lower Case, Int-28.
 Uncomment Lines, Int-75.
 EGARCH model, UG-291.
 @EGTEST procedure, UG-106.
 EIGEN instruction, UG-26, RM-134.
 Eigenvalues/vectors, RM-134.
 ELSE instruction, Int-93, RM-253.
 EM algorithm, UG-153.
 for Markov switching, UG-380.
 EMEXAMPLE.RPF example, UG-154.
 ENCODE instruction, UG-58, UG-62, RM-136.
 Enders, W., UG-104, UG-181, UG-207, UG-218, UG-247, UG-286.
 END instruction, RM-138.
 END(SYSTEM), UG-258.
 Endogenous variables in a VAR, RM-525.
 swapping, UG-280.
 Engle, R., UG-104, UG-294, UG-510, UG-303, UG-80.
 ENTER instruction, UG-12, RM-139.
 Entry ranges, Int-30, Int-82.
 setting default, RM-455.
 ENVIRONMENT instruction, RM-141.
 %EQNXVECTOR function, RM-245.
 EQUATION data type, UG-10, UG-15.
 Equation/FRML wizard, Int-87, RM-143, RM-197.
 EQUATION instruction, UG-12, RM-143.
 Equations, Int-64, UG-15, RM-143.
 adding variables to, RM-524.
 coefficients
 assigning to, RM-11.
 reading from a file, RM-12.
 combining with COMPUTE, RM-50.
 converting to FRMLs, RM-197.
 estimating systems of, RM-487.
 estimating with LINREG, RM-288.
 forecasting, UG-163.
 getting current number of, RM-269.
 grouping into a MODEL, UG-280, RM-233.
 identity, UG-280.
 listing variables in, RM-323.
 modifying, RM-323.
 non-linear, UG-17.
 residuals
 setting, RM-11.
 substituting variables within, RM-529.
 system of, RM-103.
 variance, RM-11.
 EQV instruction, RM-147.
 Error correction models, UG-247.
 Error location, UG-480.
 Error messages, Int-80.
 suppressing, RM-141.
 ERRORS instruction, UG-211, RM-148.
 @ERSTEST procedure, UG-102.
 ESMOOTH instruction, UG-179, RM-153.
 example, UG-180.

- ESTAR model, UG–370.
- ESTIMATE instruction, Int–79, UG–210, RM–158.
- CMOM option, UG–258.
- DUMMY option, UG–258.
- with prior, UG–258.
- Estimators
 - ARCH, RM–207.
 - Arellano-Bond, UG–429.
 - ARMA (alternative to BOXJENK), RM–265.
 - Bayesian, RM–41.
 - censored samples, UG–403.
 - Cochrane-Orcutt, UG–45.
 - convergence criterion, RM–337.
 - density function, UG–432.
 - discrete dependent variables, RM–89.
 - distributed lags, UG–57.
 - fixed effects, UG–421, UG–425, RM–372.
 - GARCH, RM–207.
 - generalized method of moments, UG–142, RM–313, RM–340, RM–548.
 - HAC, RM–313, RM–548.
 - Heckit, UG–407.
 - heteroscedasticity consistent, RM–313, RM–548.
 - Hildreth-Lu, UG–45.
 - iterated least squares, UG–407.
 - Kalman filter, RM–274.
 - least absolute deviations (LAD), , UG–67.
 - limited dependent variable, RM–284.
 - linear regression, RM–288.
 - with linear restrictions, UG–74, RM–136, RM–324, RM–411.
 - logit, UG–395, RM–89.
 - maximum likelihood, UG–146, RM–308.
 - minimum absolute deviations (MAD), RM–422.
 - multinomial logit, UG–398.
 - multivariate non-linear least squares, RM–340.
 - neural network, UG–439, RM–348.
 - non-linear least squares, RM–331, UG–136, UG–137, UG–136.
 - non-linear systems, UG–142.
 - non-parametric, UG–432, RM–357.
 - panel data, UG–420, RM–378.
 - probit, UG–395, RM–89.
 - quantile regression, RM–422.
 - random effects, UG–421, UG–425.
 - recursive least squares, RM–418.
 - ridge, RM–41.
 - robust, UG–67.
 - seemingly unrelated regressions (SUR), UG–53, UG–54.
 - simultaneous equation models, UG–276.
 - stepwise regression, RM–479.
 - systems of equations, RM–487.
 - three-stage least squares, UG–53, UG–276.
 - non-linear, UG–142, RM–340, UG–276.
 - tobit, UG–403.
 - truncated samples, UG–403.
 - two-stage least squares, UG–275, RM–272, RM–331.
- Estrella, A., RM–94.
- Eventual forecast function, UG–371.
- EVIEWS work files, Int–99.
- EWIS instruction, UG–12, UG–25, RM–163.
- Exact line search method, RM–337.
- EXAMPLEFIVE.RPF example, Int–66.
- EXAMPLEFOUR.RPF example, Int–43.
- Example programs, Int–160.
- EXAMPLESIX.RPF example, Int–85.
- EXAMPLETHREE.RPF example, Int–17.
- Excel files, Int–99, Int–114.
- EXCLUDE instruction, UG–71, RM–165.
- Exclusion restrictions, UG–71.
- EXECUTE instruction, UG–470, RM–167.
- Execution order. *See* Program control.
- Exogenous variables in a VAR, RM–103.
- in forecasts, UG–242, RM–189.
- EXP instruction, RM–170.
- Exponential smoothing, Int–48, UG–179, RM–153.
- forecasting, UG–176.
- technical information, RM–156.
- Exponential Smoothing wizard, Int–48, RM–153.
- Exponents, RM–541.
- complex, RM–38.
- Exporting data, RM–51, RM–384.

- Export menu command, RM-391.
- Entry numbers
in expressions, UG-3.
- Expressions, Int-40.
complex numbers, UG-448.
displaying results, RM-108, RM-535.
evaluating with COMPUTE, RM-47.
logical, Int-41.
scalar, UG-2.
syntax, RM-541.
with series, RM-446.
- EXPSMOOTH1.RPF example, UG-180.
- EXPSMOOTH2.RPF example, UG-180.
- Extreme values
across multiple series, RM-500.
of a complex series, RM-74.
of a series, RM-171.
- EXTREMUM instruction, RM-171.
- ## F
- Factor analysis, UG-222, RM-70.
- FACTOR option, UG-228.
- Fair AR1 for 2SLS, UG-275.
- Fame format, Int-98.
- Fan chart, Int-142.
- FFT instruction, UG-451, UG-456, RM-172.
- File menu
Clear Memory, Int-18.
Directory, Int-34.
New Editor/Text Window, Int-3, Int-18.
- Files
appending data to, RM-361.
binary, Int-123.
closing, RM-39.
RATS format, RM-397.
formats, Int-22.
FORTRAN formats, RM-53.
free-format text, RM-53.
graph, RM-243.
list of available, RM-449.
opening, RM-361.
organization of data on, Int-22.
outputting scalars and arrays to, RM-108.
reading data from, RM-75, RM-401.
reading strings from, Int-132.
rewinding, RM-417.
spaces in names, Int-34.
spreadsheet, Int-114.
temporary, RM-39.
writing series to, RM-51, RM-383.
- Filtering, Int-44, RM-174.
frequency domain, UG-460.
Henderson, RM-174.
Hodrick-Prescott, UG-339, RM-174.
Spencer, RM-174.
- FILTER instruction, RM-174.
- Filter/Smooth wizard, Int-26, Int-48, RM-174.
- FIML. See Full-information maximum likelihood.
- FIND instruction, Int-90, RM-180, UG-152.
- Fitted values, Int-64, RM-385.
for non-linear regression, Int-89.
- Fixed effects estimators, UG-421, UG-425, RM-372.
with PREGRESS, RM-378.
- FIXED instruction, RM-184.
- FIX function, Int-42, UG-3.
- FLOAT function, Int-42.
- %FLOATINGDATE function, UG-30.
- @FLUX procedure, UG-357.
- FMATRIX instruction, UG-26, RM-186.
- @FM procedure, UG-249.
- FOLD instruction, RM-188.
- Fonts
for graphs, Int-155, RM-235.
- @FORCEDFACTOR procedure, UG-226.
- FORECAST instruction, UG-162, UG-182, UG-505, RM-189.
saving results, UG-165.
- Forecasts, Int-53, Int-64, UG-160.
ARIMA models, UG-176, UG-181, RM-15.
example, UG-182.
- Box-Jenkins models, UG-176, UG-181.
@BJFORE procedure, UG-196.
conditional, UG-242, UG-245, RM-189.
dynamic, UG-161, RM-189.
error statistics, RM-473.
exogenous variables in, RM-189.
exponential smoothing, UG-176, UG-179, RM-153.

- GARCH model, UG–301, UG–318.
- graphing, Int–147, UG–166.
- identities, UG–164.
- performance statistics, UG–169, UG–170, RM–507.
- rolling regressions, UG–167.
- with shocks, RM–189.
- spectral methods, UG–176, UG–205, UG–462.
- standard errors of, UG–169, RM–148.
- static, Int–64, UG–161, RM–385, RM–473.
- time series methods, UG–161, UG–176.
- troubleshooting problems, UG–172.
- univariate, RM–517.
- with VARs, UG–253, UG–261.
- FORMAT option, Int–22, RM–75.
 - CDF, Int–114.
 - CITIBASE, Int–98.
 - CRSP, Int–98.
 - DIF, Int–114.
 - DTA, Int–99.
 - FAME, Int–98.
 - FREE, Int–120, RM–53.
 - HAVER, Int–98.
 - MATLAB, Int–100.
 - ODBC, Int–99.
 - PORTABLE, Int–98.
 - PRN, Int–114.
 - RATS, Int–110.
 - TSD, Int–114.
 - WF1, Int–99.
 - WKS, Int–114.
 - XLS, Int–114.
 - XLSX, Int–114.
- FORM=CHISQUARED, UG–65, UG–66.
- Formulas, Int–64, UG–10, UG–17, UG–279, RM–197.
 - associating with an equation, RM–12.
 - creating from regression, Int–64, UG–133.
 - defined in loops, UG–134.
 - forecasting models with, RM–191.
- MAXIMIZE instruction, UG–146, RM–308.
- MODELS of, UG–280, RM–233.
- for non-linear estimation, UG–131.
- self-referencing, RM–201.
- for simultaneous equation models, UG–279.
- FORTRAN formats, RM–53.
- Fourier transforms, UG–451, RM–172.
 - tapering prior to, RM–502.
- Fox, R., UG–465.
- FPE (Final Prediction Error), UG–60.
- Fractiles
 - from bootstrapping/simulations, UG–503.
 - moving window, RM–326.
 - of array, UG–503.
 - of series, RM–469.
- %FRACTILES function, UG–503.
- Fractional
 - differencing, UG–464.
 - integration, UG–464.
- Fractional differencing, RM–104.
- FRED database, Int–98.
- FREQDESEASON.RPF example, UG–460.
- %FREQSIZE function, UG–453, RM–195.
- FREQUENCY instruction, UG–449, UG–452, RM–195.
- Frequency of data series, RM–26.
 - changing, Int–103, RM–75, RM–428.
 - differing, Int–34.
- FRML data type. *See* Formulas; *See* Formulas.
- FRML instruction, Int–86, UG–12, UG–131, RM–197, RM–308, RM–331.
 - simultaneous equations, UG–279.
- %FTEST function, UG–71.
- F-tests, UG–8, RM–393.
 - CDF instruction, RM–33.
 - for zero coefficients, RM–165.
- Fuller, W. A., , UG–100.
- Full-information maximum likelihood, UG–278.
- Fully-modified least squares, UG–249.
- FUNCTION instruction, UG–470, RM–95, RM–204.
 - END statement, RM–138.
 - example, UG–150.
 - in non-linear estimation, UG–149.
- Functions
 - calendar/date, UG–30, RM–31.
 - complex/spectral, UG–448.
 - matrix, UG–28.
 - random draws, UG–506.
 - string, UG–23.
 - using, Int–41, UG–6.

G

Gain

- cross spectral, RM-514.
- Gamma distribution
random draws, UG-506.
- GARCHBACKTEST.RPF
example, UG-360.
- GARCHBOOT.RPF exam-
ple, UG-542.
- GARCHFLUX.RPF exam-
ple, UG-357.
- GARCHGIBBS.RPF exam-
ple, UG-537.
- GARCHIMPORT.RPF ex-
ample, UG-525.
- GARCH instruction, Int-90,
RM-207.
- GARCH model, RM-207,
RM-308.
backtesting, UG-360.
Bayesian analysis, UG-
525, UG-537.
bootstrapping, UG-542.
forecasting, UG-301,
UG-318.
semiparametric, UG-432.
- GARCHMVMAX.RPF ex-
ample, UG-319.
- GARCHMV.RPF exam-
ple, UG-302.
- GARCHSEMIPARAM.RPF
example, UG-432.
- GARCHUVMAX.RPF ex-
ample, UG-299.
- GARCHUV.RPF exam-
ple, UG-290.
- Gauss-Newton algo-
rithm, UG-115.
used by NLLS, RM-331.
- Gauss-Seidel algo-
rithm, UG-279, UG-
281.
- GBOX instruction, RM-216.
- GCONTOUR instruc-
tion, RM-220.
- GCONTOUR.RPF exam-
ple, Int-143.
- Gelfand, A. E., UG-528.
- Generalized impulse re-
sponses, UG-227.
- Generalized least squares
with BOXJENK, RM-18.
for heteroscedasticity, UG-
42.
joint (Zellner), UG-53.
panel data transforma-
tions, UG-420.
for serial correlation, UG-
45.
- Generalized Method of Mo-
ments, UG-39, UG-
142, UG-145.
Arellano-Bond estima-
tor, UG-429.
multivariate, RM-340,
UG-145.
univariate, UG-138.
- Generalized residuals, UG-
405.
- Genetic algorithm, UG-118,
RM-337.
- Geweke, J., UG-85, UG-
464.
- GIBBS.RPF example, UG-
530.
- Gibbs sampling, UG-528,
UG-530.
- %GIRFMATRIX func-
tion, UG-227.
- GIV.RPF example, UG-139.
- Glosten, L., UG-293.
- GLS. *See* Generalized least
squares.
- @GMAUTOFIT proce-
dure, UG-199.
- GMM. *See* Generalized
Method of Moments.
- Godfrey-Breusch test, UG-
83.
- Goldfeld-Quandt test, UG-
79.
- Goodness of fit, Int-35.
- GOTO instruction, RM-222.
- GRANGERBOOTSTRAP.
RPF example, UG-85.
- Granger, C. W. J., UG-84,
UG-104, UG-181,
UG-464.
- Granger-Sims causality
tests, UG-84, UG-211.
- GRAPHBOXPLOT.RPF
example, Int-144.
- GRAPHFORECAST.RPF
example, Int-147.
- GRAPHFUNCTION.RPF
example, Int-140.
- GRAPHHIGHLOW.RPF
example, Int-146.
- GRAPH instruction, Int-126,
RM-223.
with complex series, UG-
454.
with SPGRAPH, RM-462.
- GRAPHOVERLAY.RPF ex-
ample, Int-135.
- Graphs, RM-223.
adding text to, RM-239.
appearance options, RM-
235.
autocorrelations, Int-145.
background, Int-151.
box plots, RM-216.
contour plots, RM-220.
copying and pasting, Int-
129.
displaying, Int-46, Int-127,
RM-141.
exporting, Int-129, RM-
243.
fan charts, Int-142.
fonts on, Int-155.
highlighting entries, Int-
141.
high-low-close, Int-146.
impulse response, UG-
231.
instructions for, Int-126.

keys/legends, Int–132.
labeling, Int–131, RM–235, RM–239.
multiple per page, RM–462.
orientation of, Int–129, RM–235.
overlay/two-scale, Int–134, RM–229.
patterns, Int–152.
preparing for publication, Int–129.
printing, RM–141.
saving, RM–141, RM–243, RM–361.
scatter (X-Y) plots, Int–71, RM–431.
shading, Int–141.
size of, RM–235.
special, RM–462.
window, Int–56.
wizard, Int–46.
Graph window, Int–56, Int–127.
Graph wizard, Int–46, RM–223, RM–431.
GROUP instruction, UG–12, UG–280, RM–233.
Growth rates, Int–62.
GRPARM instruction, Int–126, RM–235.
GRTEXT instruction, Int–126, RM–239.
GSAVE instruction, Int–153, RM–243.
GSET instruction, UG–12, RM–245.

H

HAC covariance matrix, UG–41, UG–46, RM–313, RM–548.
HALT instruction, RM–247.

Hamilton, J., UG–218, UG–247, UG–379, UG–388.
HAMILTON.RPF example, UG–388.
Hamilton switching model, UG–356.
Hannan-Quinn criterion, UG–60.
Hansen, B., UG–86, UG–90, UG–98.
Hansen, L. P., UG–39, UG–95, RM–344, RM–490.
HANSEN.RPF example, UG–99.
Hansen specification test, UG–98.
Hansen stability test, UG–86, UG–90.
Harvey-Collier test, UG–52.
HASH aggregator, UG–486.
Hausman, J. A., UG–95, UG–96, UG–425.
HAUSMAN.RPF example, UG–96.
Hausman specification test, UG–95.
Hazard models, UG–410.
Heckit models, UG–403, UG–407.
Heckman, J., UG–66, UG–407.
Henderson filter, RM–174.
Hessian matrix, UG–116, UG–125.
HETERO.RPF example, UG–43.
Heteroscedasticity, UG–42. *See also* ARCH model; *See also* GARCH model.
consistent covariance matrix, RM–313, RM–548.
testing for, UG–77, UG–80.

HETEROTEST.RPF example, UG–77.
Hildreth-Lu, UG–45, UG–275.
Hill-climbing methods, UG–114.
Histograms, RM–100.
Historical decompositions, UG–239, RM–248.
Historical simulations, UG–283.
HISTORY instruction, UG–211, UG–239, RM–248.
HISTORY.RPF example, UG–240.
Hodrick-Prescott filter, UG–339, RM–174.
Holidays, RM–30.
on data file, Int–108.
Holt-Winters model, RM–154.
Hosking, J. R. M., UG–464.
HPFILTER.RPF example, UG–339.
Hsiao, C., UG–420.
@HURST procedure example, RM–14.
Hypothesis tests, RM–324.
with BOXJENK, RM–21.
coefficient values, RM–504.
with consistent covariance matrices, RM–36.
exclusion restrictions, RM–165.
formulas for, UG–74.
instructions for, UG–71.
linear restrictions, RM–411.
for logit and probit, RM–94.
results of, Int–11.
with SUR, UG–55.
variables defined, UG–73.

- with VARs, UG–212, RM–399.
 - with NLLS, UG–137.
- I**
- Identities, RM–233.
 - adding to a model, UG–280.
 - %IF function, Int–41, Int–63, Int–107.
 - used in multinomial logit, UG–398.
 - used with SET, Int–63.
 - IF instruction, Int–93, RM–253.
 - %IF function as alternative to, RM–253.
 - IFT instruction, UG–451, RM–172.
 - IGARCH model, UG–291.
 - %IMAG function, UG–448.
 - Impact response, UG–216.
 - @IMPACTSIGNFLIP procedure, UG–224.
 - Importance sampling, UG–523.
 - IMPULSE instruction, UG–211, RM–256.
 - alternative to, RM–149.
 - Impulse responses, UG–234, RM–256.
 - confidence bands for, UG–231.
 - cumulative, RM–2.
 - fractiles, UG–519.
 - generalized, UG–227.
 - graphing, UG–231.
 - standard errors of, UG–519.
 - IMPULSES.RPF example, UG–234.
 - INCLUDE instruction, Int–111.
 - alternative to, RM–475.
 - %INDIV function, UG–415.
 - Individual effects, UG–419, RM–393.
 - means, RM–369.
 - Inequality constraints
 - linear/quadratic programming, RM–301.
 - non-linear estimation, RM–354.
 - INFOBOX instruction, UG–489, RM–263.
 - example using, UG–501.
 - Information criteria, UG–60.
 - INITIAL instruction, RM–265.
 - Innovations process, UG–216.
 - Input
 - changing source of, RM–34.
 - echoing of, RM–141.
 - SOURCE instruction, RM–457.
 - INPUT instruction, UG–12, RM–266.
 - with complex series, RM–427.
 - INPUT I/O unit, RM–361.
 - Inputting
 - arrays, RM–266.
 - variables, RM–266.
 - Input window, Int–7.
 - INQUIRE instruction, UG–477, RM–269.
 - Instructions
 - echoing, RM–141.
 - input files of, RM–361.
 - locations in compiled code, UG–480.
 - long (splitting up), Int–57, Int–76.
 - multiple per line, Int–57.
 - secondary input files, RM–457.
 - Instrumental variables, UG–48, RM–272, RM–494.
 - with LINREG, RM–288.
 - non-linear least squares, RM–331.
 - simultaneous equation models, UG–274.
 - INSTRUMENT.RPF example, UG–50.
 - INSTRUMENTS instruction, Int–78, UG–48, UG–274.
 - Integer
 - numbers, Int–42.
 - values in expressions, UG–3.
 - INTEGER data type, UG–3, UG–10, UG–14.
 - Interactive procedures, UG–489.
 - dialog boxes, UG–489, RM–263, RM–321.
 - input from user with QUERY, RM–395.
 - menus, UG–489, RM–320.
 - with USERMENU, RM–521.
 - selecting from a list, RM–444.
 - using PROCEDURE, RM–389.
 - viewing/editing arrays, RM–317.
 - Interpolating data, Int–105, RM–30.
 - Interrupt program, Int–7.
 - Interval estimation, UG–405.
 - Intervention models, UG–200, RM–15.
 - INTERVENTION.RPF example, UG–201.
 - Intraday data, RM–28.
 - Inverse
 - autocorrelations, RM–57.
 - I/O units, RM–361.
 - closing, RM–39.
 - reserved, RM–361.
 - rewinding, RM–417.

standard, RM-34.
switching, RM-34.
tips on, Int-123.
user-defined, RM-361.
Iterations
non-linear estimation, UG-117.

J

Jarque-Bera Normality test, RM-469.
%JDF variable, UG-140.
Jenkins, G. M., UG-181, RM-265.
@JOHMLE procedure, UG-106.
JOHNP121.RPF example, RM-420.
Joyeux, R., UG-464.
JROBUST option, UG-140.
%JSIGNIF variable, UG-140.
%JSTAT variable, UG-140.
%JULIAN function, UG-30.

K

Kadiyala, K. R., UG-515.
Kalman filter, RM-274.
equation variances, RM-281.
estimating parameters, RM-180.
initial values, UG-265, RM-158.
likelihood function, RM-280.
recursive residuals, UG-266.
sequential estimation, UG-265.
setting up, RM-279, RM-497.
state-space models, UG-332, RM-112.

time-varying coefficients, UG-267, RM-515.
updating formula, UG-264.
VAR models, UG-264.
KALMAN instruction, RM-274.
Kalman smoothing, UG-339, RM-112.
Karlsson, S., UG-515.
Kernel methods
density estimation, UG-432, RM-100.
non-parametric regression, UG-432, RM-357.
KEYBOARD I/O unit, RM-361.
KFSET instruction, UG-267, RM-279.
King-Plosser-Rebelo model, UG-350.
KLEIN.RPF example, UG-49.
Koopman, S. J., UG-321.
@KPSS procedure, UG-103.
Kroner, K. F., UG-303.
Kurtosis, RM-469.

L

LABEL data type, UG-10, UG-14.
%LABEL function, UG-23.
Labels, UG-23, UG-484.
of complex series, RM-36.
on data files, Int-115.
expressions using, RM-49, RM-543.
for graphs, Int-131.
series, RM-282.
LABELS instruction, RM-282.

vs. EQV instruction, RM-147.
LAD estimator, UG-67.
Lag length
methods for choosing, UG-60.
VAR
choice of, UG-255.
testing, UG-212.
Lagrange multiplier
constrained optimization, UG-122.
%LAGRANGE variable, RM-346.
tests, UG-75, UG-83.
Lags
non-consecutive, with BOXJENK, RM-16.
with panel data, UG-417.
LAGS instruction, UG-208, RM-283.
LAGS option
for serial correlation, UG-45.
Landscape mode, Int-129, RM-235.
LDV instruction, Int-79, UG-403, UG-404, UG-409, RM-284.
Least squares, Int-27.
iterated, UG-407.
non-linear, RM-331.
recursive, RM-418.
three-stage, , UG-276.
Lee, J., UG-377.
%LEFT function, UG-23.
Levenberg-Marquardt algorithm, RM-338.
%L function, Int-132, UG-21, UG-23.
LGT instruction, UG-395.
Likelihood function
hazard models, UG-410.
Kalman filter, UG-269.

- limited dependent variables, UG-404.
- %LOGL variable, RM-292.
- maximum likelihood estimation, UG-146.
- multivariate, UG-148.
- partial, UG-410.
- probit and logit, UG-395, UG-400.
- structural VAR, UG-218.
- Likelihood ratio test, UG-71, UG-212, RM-399.
- Lilien, D. M., UG-294.
- Limited dependent variables, UG-403, RM-284.
- Limited/Discrete Dependent Variables wizard, Int-79, RM-89, RM-284.
- Limited information maximum likelihood, UG-277.
- LIML. *See* Limited information maximum likelihood.
- @LIML procedure, UG-277.
- Linear
 - filters, RM-174.
 - probability model, UG-401, UG-402.
 - programming, UG-435, RM-301.
 - regressions, Int-27.
 - annotated output, Int-35.
 - restrictions, RM-136, RM-411.
- Linear Regressions wizard, RM-6, RM-288, RM-422, RM-479.
- Line numbers of errors, UG-480.
- Least squares with LINREG, RM-288.
- LINREG instruction, Int-28, RM-288.
- CMOMENT option, RM-41.
- CREATE option, UG-64.
- FRML option, UG-279.
- instrumental variables, RM-272.
- UNRAVEL option, UG-58.
- with VAR's, UG-210.
- LIST aggregator, UG-487.
- LISTEXAMPLE.RPF example, UG-488.
- LIST instruction, RM-295.
- example, RM-150, RM-259.
- Litterman, R., UG-208, UG-268, UG-269, RM-280.
- Ljung-Box Q, RM-55, RM-61, RM-292.
- Ljung, G. M., UG-81, RM-56.
- @LOCALDLM procedure, UG-329.
- LOCAL instruction, UG-475, RM-297.
- Local optima, UG-111.
- Local variables, UG-470, UG-475.
- Logical
 - expressions, RM-541.
 - with IF instruction, RM-253.
 - operators, Int-41, RM-541.
 - creating dummies using, Int-61.
 - with complex values, UG-448.
- LOG instruction, RM-299.
- %LOGISTIC function, UG-371.
- Logit models, UG-395, UG-401, UG-402, RM-89.
- fitted values for, RM-385.
- samples with repetition, UG-400.
- Log likelihood. *See* Likelihood function; *See also* Maximum likelihood.
- Long memory models estimation, UG-464.
- Long-run restriction, UG-223.
- Looping instructions
 - DO, RM-121.
 - DOFOR, RM-123.
 - LOOP, RM-300.
 - NEXT, RM-330.
 - UNTIL, RM-520.
 - WHILE, RM-531.
- LOOP instruction, Int-92, UG-468, RM-300.
- Loops, UG-468.
- branching out of, RM-222.
- breaking out of, RM-24.
- conditional, RM-520, RM-531.
- DO, RM-121.
- ending, RM-138.
- exiting, RM-24, RM-222.
- indefinite, RM-300.
- jumping to next pass, RM-330.
- line number of error in, UG-480.
- nested, RM-121.
- quitting RATS from within, RM-247.
- LOWESS estimator, UG-432, RM-359.
- LOWESS.RPF example, RM-360.
- @LPUNIT procedure, UG-377.
- LQPROG instruction, Int-90, UG-435, RM-301.
- LSTAR model, UG-370.
- @LSUNIT procedure, UG-377.

Lumsdaine, R., UG–377.
LWINDOW option, UG–41.

M

MacKinnon-White-Davidson
test, UG–94.

MAKE instruction, UG–26,
UG–28, RM–305.

Marginal significance lev-
els, Int–11, UG–38,
UG–71, UG–73, RM–
33.

Mariano, R., UG–171.

Markov chain, UG–381.

Markov switching mod-
els, UG–356, UG–379.
ARCH, UG–390.

Marquardt algorithm, RM–
338.

MATLAB data files, Int–99.

Matrix

||...|| notation, UG–26.

banded, RM–186.

copying data to series
from, RM–307.

correlation, RM–41.

creating, UG–11.

from data series, RM–
305.

cross-moment, RM–41.

declaring, RM–96.

defined by RATS instruc-
tions, UG–25.

dimensioning, RM–107,
RM–305.

displaying, UG–12, RM–
108, RM–535.

DOFOR lists, RM–123.

eigen decomposition, RM–
134.

elementwise opera-
tions, RM–163.

expressions, UG–25.

instructions, UG–25.

using in lists, UG–18.

one-dimensional, UG–17.

printing, RM–535.

QZ decomposition, RM–
398.

reading data into, RM–266,
RM–401.

reading/writing complex
series as, RM–427.

rectangular, UG–10.

RECTANGULAR data
type, UG–16.

Schur decomposition, RM–
398.

of series, UG–21.

setting, UG–12, RM–305.

symmetric, UG–10.

SYMMETRIC data
type, UG–16.

vector, UG–10.

VECTOR data type, UG–
17.

viewing/editing with
MEDIT, UG–489,
RM–317.

%MAXENT variable, RM–
171.

Maximization. *See* MAXI-
MIZE instruction.

using FIND, UG–152,
RM–180.

MAXIMIZE instruction, UG–
146, RM–308.

for censored and truncated
data, RM–21.

formulas for, RM–197.

for GARCH model, UG–
299, UG–319.

for logit and probit, UG–
400.

parameters for, RM–354.

Maximum likelihood, RM–
308.

AR1 models, RM–6.

ARMA models, RM–16.

censored and truncated
samples, UG–403,
UG–404.

estimation, UG–146.

full information, UG–278.

limited information, UG–
277.

tobit models, UG–403.

Maximum value

moving window, RM–326.

of a series, RM–469,
RM–468, RM–171.

%MAXIMUM variable, RM–
171.

across multiple se-
ries, RM–500.

%MCERGODIC func-
tion, UG–383.

@MCFEVDTABLE proce-
dure, UG–522.

@MCGRAPHIRF proce-
dure, UG–519, UG–
521.

@MCLEODLI proce-
dure, UG–298.

@MCMCPOSTPROC proce-
dure, UG–530.

MCOV instruction, UG–26,
UG–66, UG–138,
RM–313, RM–548.

MCPRICEEUROPE.RPF
example, UG–517.

@MCPROCESSIRF proce-
dure, UG–521.

%MCSTATE function, UG–
383.

@MCVARDODRAWS proce-
dure, UG–522.

Mean

individual (panel
data), UG–419.

of a series, RM–469,
RM–499.

@MEANGROUP.SRC proce-
dure, UG–427.

- Median
of a series, RM-469.
- MEDIT instruction, UG-12,
UG-489, UG-492,
UG-497, RM-317.
- Memory
available, RM-449.
clearing, RM-138.
freeing, RM-404.
- %MENUCHOICE vari-
able, RM-522.
- MENU instruction, UG-493,
RM-320.
- Menus (user-defined), UG-
490, RM-320.
MENU/CHOICE instruc-
tions, UG-493.
pull-down, RM-521.
USERMENU, UG-489.
- MESSAGEBOX instruc-
tion, UG-489, RM-321.
- Metropolis-Hastings meth-
od, UG-535.
- %MID function, UG-23.
- Mills ratio, UG-403, UG-
407, RM-387.
- %MINENT variable, RM-
171.
- %MINIMALREP func-
tion, RM-109.
- Minimization. *See* MAXIMIZE
instruction.
linear/quadratic program-
ming, UG-435, RM-
301.
using FIND, UG-152,
RM-180.
- Minimum value
moving window, RM-326.
of a series, RM-469,
RM-468, RM-171.
- %MINIMUM variable, RM-
171.
across multiple se-
ries, RM-500.
- Missing values, Int-62,
Int-63, RM-30.
on data files, Int-107.
in expressions, Int-42,
Int-63.
in forecasts, RM-180.
%NA constant, Int-40.
removing from a se-
ries, RM-428.
- Mixture models, UG-379.
@MIXVAR procedure, UG-
262.
- MODEL data type, UG-10,
UG-15, UG-279.
- %MODELLAGSUMS func-
tion, UG-223.
- Models
building with GROUP, RM-
233.
combining with +, RM-50.
defining with SYS-
TEM, RM-497.
dynamic linear, RM-112.
forecasting, UG-279,
RM-189.
simulations, RM-450.
state-space, RM-112.
- Model selection
stepwise regressions, RM-
479.
- %MODELSETCOEFFS func-
tion, UG-514.
- MODIFY instruction, ,
UG-279.
- MONTEARCH.RPF exam-
ple, UG-510.
- Monte Carlo
antithetic accelera-
tion, UG-518.
bookkeeping for, UG-500.
importance sampling, UG-
523.
impulse response standard
errors, UG-519.
integration, UG-517.
tests, UG-510, RM-450.
- MONTESUR.RPF exam-
ple, UG-533.
- @MONTEVAR proce-
dure, UG-522.
- MONTEVAR.RPF exam-
ple, UG-519.
- %MONTH function, UG-30.
- Monthly data, RM-27.
- Mean
moving window, RM-326.
- Median
moving window, RM-326.
- Moving average, Int-25.
adding lags to equa-
tion, RM-524.
alternative to, RM-326.
representation, UG-216,
RM-256.
serial correlation correc-
tion, RM-313.
- Moving Window Statistics
wizard, RM-326,
RM-329.
- %MQFORM function, UG-
66.
- MRESTRICT instruc-
tion, UG-63, UG-71,
RM-324.
- %MSUPDATE function, UG-
383.
- @MSVARSETUP proce-
dure, UG-385.
- Multipliers, UG-283.
- @MVGARCHFORE proce-
dure, UG-318.
- @MVQSTAT proce-
dure, UG-315.
- MVSTATS instruction, RM-
326.
- MWHP366.RPF exam-
ple, Int-142.

N

Nadaraya-Watson estimator, UG-432, RM-359.

Names

for numbered series, RM-147, RM-282.

legal form, UG-11.

%NA missing value, Int-40.

Near-VARs, UG-251.

Neural networks, Int-90, UG-439.

convergence of, UG-443.

fitted values/predictions, RM-353.

limit values of, UG-444.

tips on fitting, UG-442.

training/fitting, RM-348.

NEURAL.RPF example, UG-446.

Newbold, P., UG-181.

Newey-West estimator, UG-41.

Newton-Raphson, UG-115.

NEXT instruction, RM-330.

NLLS instruction, Int-88, RM-331.

FRMLs for, RM-197.

instrumental variables, RM-272.

technical details, UG-127.

NLLS.RPF example, UG-137.

NLPAR instruction, RM-337.

NLSYSTEM instruction, UG-142, RM-340, UG-276.

FRMLs for, RM-197.

instrumental variables, RM-272.

technical details, UG-127, UG-142.

NNLEARN instruction, Int-90, UG-440, RM-348.

NNTEST instruction, Int-90, UG-440, RM-353.

Non-differentiable functions optimizing, UG-152.

Non-linear

estimation, Int-85, RM-308.

convergence, UG-110, RM-337.

forecasting, UG-164.

formulas for, RM-197.

initial values, Int-87, UG-111.

introduction to, UG-110.

maximum likelihood, UG-146.

method of moments, UG-138.

parameters for, UG-129, RM-354.

user-defined functions, UG-149.

least squares, UG-136, RM-331.

example of, UG-137.

formulas for, UG-131.

multivariate, RM-340.

systems

estimation, UG-142.

solving, UG-279.

NONLINEAR.RPF example, UG-152.

NONLIN instruction, Int-86, UG-12, UG-129, RM-308, RM-331, RM-354.

Non-parametric regressions, RM-357.

example, UG-433.

Normal distribution, RM-387.

CDF instruction, RM-33.

random draws, UG-500, UG-506.

multivariate, UG-506, UG-507.

statistics, RM-385.

test for, RM-469.

truncated, UG-506.

Normalizing series, Int-62.

NPREG instruction, Int-90, UG-432, RM-357.

NPREG.RPF example, UG-433.

Numbered series

labels for, RM-36.

naming, RM-282.

O

OK/Cancel dialog box, RM-321.

OLSMENU.RPF example, UG-490.

ONLYIF option, UG-135.

OPEN instruction, Int-34, Int-96, RM-54, RM-78, RM-361.

Open menu command

RATS format data

files, RM-391, RM-475.

Operators

arithmetic, RM-541.

label and string, RM-543.

logical, RM-541.

matrix, UG-27.

precedence, Int-40, RM-542.

Optimal control

with DSGE, RM-127.

linear-quadratic, RM-112.

OPTIMALWEIGHTS option, UG-139.

Optimization

linear/quadratic programming, RM-301.

using FIND, RM-180.

OPTION instruction, RM-363.

Options

defining in procedures, RM-363.

procedure, UG-470, RM-167.

using expressions with, UG-476.

Orderings

choosing for VAR, UG-233.

ORDER instruction, RM-366.

ORGANIZATION option, Int-22, RM-75.

with free-format, Int-121.

Orthogonalization, UG-216, RM-256.

Choleski factorization, UG-217.

%OUTERXX function, UG-504.

Outliers

ARIMA model, RM-18.

Output

changing destination of, RM-34.

data series, RM-391.

displaying variables, RM-108.

formatted reports, RM-405.

formatting, UG-5, RM-108, RM-535.

opening file for, RM-361.

in table form, UG-31.

width, RM-141.

OUTPUT unit, RM-361.

Output window, Int-7.

Overflow, UG-110.

Overidentifying restrictions, UG-95, UG-220, UG-222.

Overlay graphs, Int-134, RM-229.

P

Padding complex series, UG-452.

Panel data

analysis of variance tests, RM-393.

AR1 regression, UG-422.

CALENDAR for, UG-413.

date expressions, RM-552.

dummy variables, UG-417.

fixed/random effects estimation using PREGRESS, RM-378.

forming series into, RM-373.

lags in, UG-417.

organization of, UG-413.

random coefficients model, UG-427.

reading from disk files, UG-415.

regressions, UG-420, RM-369, RM-378.

seasonal dummies, RM-441.

Swamy's random coefficients model, UG-427.

transformations, UG-419, RM-369.

unbalanced, RM-373.

variance components, RM-379.

Panel Data Regressions wizard, Int-79, RM-378.

PANEL instruction, UG-419, UG-426, RM-369.

%PANELOBS function, UG-414.

PANEL.RPF example, UG-426.

%PANELSIZE function, UG-414.

Papell, D., UG-377.

Parameters

non-linear estimation, RM-354.

procedure, UG-470, RM-168, RM-516.

by address/by value, UG-474.

%DEFINED function, RM-365.

formal vs actual, UG-473.

using INQUIRE with, RM-270.

Parentheses

in expressions, RM-542.

PARMSET data type, UG-10, UG-15, UG-129, UG-500, RM-354.

%PARMSPEEK function, UG-130.

%PARMSPOKE function, UG-130.

Parsimony, UG-178.

Partial autocorrelations, RM-55.

Partial likelihood hazard models, UG-411.

Partial sums

of complex series, RM-25.

of series, RM-2.

PDL.RPF example, UG-58, RM-137.

%PERIOD function, UG-30, UG-415.

Periodograms, UG-456.

smoothing, RM-532.

%PERP function, UG-224.

@PERRONBREAKS procedure, UG-377.

- Perron, P., UG–367, UG–376, UG–102.
- PFORM instruction, UG–418, RM–373.
- Phase lead, UG–458, RM–376.
- Phillips-Perron unit root test, UG–100.
- %PI constant, Int–40, UG–448.
- PICT format, RM–141.
- Picture codes, Int–5, UG–5, RM–109.
on PRINT, RM–383.
- PLOT I/O unit, RM–361.
- Poisson
count data model, RM–89.
- POLAR instruction, UG–458, RM–376.
- Polynomial
distributed lags, RM–137.
- Porter-Hudak, S., UG–464.
- Portfolio optimization, UG–438.
- PORTFOLIO.RPF example, UG–438.
- Portrait mode, Int–130, RM–235.
- Position codes, RM–109.
- PostScript format, RM–141.
- @PPUNIT procedure, UG–102.
- Prais-Winsten, RM–6.
- PRBIT instruction, UG–395.
- Precedence
of operators, Int–40, RM–541.
- Precision of numbers, UG–110.
- Predetermined variables, RM–272.
- Preferences menu command, UG–471.
- PREGRESS instruction, UG–425, UG–426, RM–378.
- PRESAMPLE option
for DLM, UG–335.
for GARCH, UG–295.
- PRG file type, Int–58.
- PRINTER I/O unit, RM–361.
- Printing
complex series, UG–454, RM–59.
data from a RATS file, RM–391.
data series, RM–383.
variables, RM–108.
- PRINT instruction, Int–59, RM–383.
using COPY instead, RM–36.
- Prior
VAR, UG–254, UG–256, UG–258, UG–259, RM–458.
mean, UG–255.
tightness, UG–256.
- %PRIOR variable, UG–258.
- %PRJCDF variable, UG–396.
- %PRJDENSITY variable, UG–396.
- %PRJFIT variable, UG–396.
- PRJ instruction, Int–64, RM–385.
for censored/truncated data, UG–407.
forecasting with, Int–64.
- PRN files, Int–114, RM–75.
- Probit models, UG–395, UG–401, UG–402, RM–89.
fitted values for, RM–385.
samples with repetitions, UG–400.
sequential, UG–400.
- PROBIT.RPF example, UG–401.
- PROCEDURE instruction, UG–470, RM–389.
- Procedures, Int–32, UG–468, UG–470, RM–389.
%DEFINED function, UG–476, RM–365.
development tools, RM–95.
END statement, RM–138.
executing, RM–167.
local variables in, RM–297.
nested, RM–168.
options, RM–167, RM–363.
parameters, RM–516.
reading from a separate file, RM–457.
returning from, RM–416.
search directory, UG–471.
storing in separate files, UG–471.
supplementary cards, RM–139.
- Progress bars, UG–501, RM–263.
- Proportional hazard models, UG–410.
- PRSETUP.SRC file, UG–278.
- PRTDATA instruction, Int–111, RM–391.
- %PSDINIT function, UG–334.
- PSTATS instruction, UG–420, RM–393.

Q

- QMLE, UG–128.
- QPROG.RPF example, UG–438.
- %QSIGNIF variable, RM–57.

%QSTAT variable, RM-57.
 Q test
 cross correlation, RM-61.
 serial correlation, RM-547.
 Quadratic programming, UG-435, RM-301.
 Quah, D., UG-223.
 Quantile regression, RM-422.
 Quarterly data, RM-27.
 Quasi-Maximum Likelihood Estimation, UG-128.
 QUERY instruction, UG-12, UG-489, UG-492, RM-395.
 QUIT instruction, Int-111, RM-397.
 QZ instruction, UG-26, RM-398.

R

Ramsey, J., UG-93.
 %RANBETA function, UG-506, UG-516.
 %RANCHISQR function, UG-506, UG-513.
 %RANDIRICHLET function, UG-509, UG-516.
 Random coefficients model panel data, UG-427.
 Random effects estimators, UG-421, UG-425, RM-378.
 Randomization tests, UG-543.
 RANDOMIZE.RPF example, UG-543.
 Random numbers, UG-500.
 algorithms, RM-443.
 functions for, UG-506.
 generating series of, RM-450.
 initializing, RM-442.

 multivariate distributions, UG-513.
 reproducing, RM-442.
 techniques for, UG-511.
 uniform integers, RM-13.
 %RANFLIP function, UG-511.
 %RAN function, UG-500, UG-506.
 %RANGAMMA function, UG-506.
 Range of entries
 changing default, RM-455.
 Rank correlation, RM-366.
 %RANLOGKERNEL function, UG-509.
 %RANLOGNORMAL function, UG-506.
 %RANMAT function, UG-506.
 %RANMVKRONCMOM function, UG-516.
 %RANMVKRON function, UG-515.
 %RANMVNORMAL function, UG-507.
 %RANMVPOSTCMOM function, UG-513.
 %RANMVPOST function, UG-507, UG-513, UG-516.
 %RANMVPOSTHB function, UG-513.
 %RANMVT function, UG-508.
 %RANSPIHERE function, UG-509.
 %RANT function, UG-506.
 %RANTRUNCATE function, UG-506.
 %RANWISHARTF function, UG-508.
 %RANWISHART function, UG-508.

%RANWISHARTI function, UG-508, UG-514.
 RATIO instruction, UG-71, RM-399.
 RATS Data File window, Int-112.
 RATSDATA program, Int-113.
 RATS format files
 abort editing/close file, RM-397.
 editing/creating, RM-98.
 finding series length, RM-269.
 instructions for, Int-111.
 listing contents of, RM-32.
 older versions of, Int-113.
 panel data, UG-415.
 printing series, RM-391.
 reading data from, RM-75.
 saving data to, RM-430, RM-475.
 RATS forum, Int-161.
 RBC model, RM-129.
 READ instruction, UG-12, RM-401.
 Ready/Local button, Int-7.
 Real Business Cycle model, RM-129.
 REAL data type, UG-3, UG-10, UG-14.
 %REAL function, UG-448.
 RECTANGULAR arrays, UG-10, UG-16.
 Recursive least squares, UG-51, RM-418.
 tests using, UG-91.
 Recursive Least Squares wizard, Int-79.
 Recursive residuals, UG-51, UG-91.
 graphing example, Int-138.
 VARs, UG-266.

- @REGACTFIT procedure, Int-32.
- RegARIMA models
 - with BOXJENK, RM-18.
- REGARIMA.RPF example, UG-204.
- @REGCORRS procedure, UG-82, UG-197.
- @REGCRITS procedure, UG-60.
- @REGRESET procedure, UG-93.
- Regression format, Int-79.
- Regressions
 - AR1 errors, RM-6.
 - with panel data, UG-422.
 - fitted values, RM-385.
 - forecasting, UG-160.
 - non-parametric, RM-357.
 - ordinary least squares, RM-288.
 - output
 - annotated, Int-35.
 - panel data, RM-378.
 - quantile, RM-422.
 - recursive least squares, RM-418.
 - reporting results, UG-31.
 - restricted, UG-62, UG-71, RM-136, RM-324, RM-411.
 - robust, RM-422.
 - rolling, UG-4, RM-391.
 - seemingly unrelated, UG-53, RM-487.
 - statistics
 - accessing, UG-8.
 - defined by LINREG, RM-292.
 - stepwise, RM-479.
 - testing restrictions, RM-165.
 - using SWEEP, RM-494.
- Regressions wizard. *See* Linear Regressions wizard.
- Regression Tests wizard, Int-69, UG-73, RM-324, RM-411, RM-165.
- Regressor lists
 - building with ENTER, RM-139.
- @REGWHITETEST procedure, UG-77.
- @REGWUTEST procedure, UG-98.
- Rejection method, UG-511.
- REJECT option, UG-124.
- Relational operators, Int-41.
 - complex values, UG-448.
- RELEASE instruction, RM-404.
- RENAME instruction, Int-111.
- Repetitive tasks, UG-483.
 - arrays of series, UG-21, UG-483.
 - DOFOR loops, RM-123.
- REPORT data type, UG-10.
- REPORT instruction, UG-12, UG-31, RM-405.
- Reports
 - generating, UG-31, RM-405.
- Report window, Int-11, Int-15, Int-29.
- Resampling, UG-540.
- RESET test, UG-93.
- %RESIDS series, Int-76.
- Residuals, Int-76.
 - bootstrap, RM-14.
 - generalized, UG-405.
 - recursive, UG-266.
 - structural, UG-229.
 - sum of squares, RM-292.
- %%RESPONSES, UG-519.
- RESTRICT instruction, UG-63, UG-71, RM-411.
 - alternative to, RM-324.
- Restrictions
 - across equations, UG-56.
 - coefficient vector, UG-74.
 - covariance matrix, UG-74.
 - exclusion, UG-71, RM-165.
 - on non-linear parameters, UG-130.
 - regressions, UG-62, UG-71, RM-136, RM-324, RM-411.
 - tests, UG-71.
- RETURN instruction, UG-470, RM-416.
- REWIND instruction, RM-417.
- %RIGHT function, UG-23.
- RLS. *See* Recursive least squares.
- RLS instruction, RM-418.
- Robust
 - estimation, UG-67.
 - regression, Int-78.
- ROBUSTERRORS option, UG-40, UG-66, RM-313, RM-548.
 - for heteroscedasticity, UG-43.
 - for serial correlation, UG-45.
- @ROBUSTLMTEST procedure, UG-76.
- ROBUST option, Int-76.
- ROBUST.RPF example, UG-67.
- Rolling regressions, UG-4, UG-167, UG-358.
- @ROLLREG procedure, UG-358.
- Root mean square error
 - for forecasts, RM-507.
- Roots
 - using FIND, UG-152, RM-180.
- %ROUND function, Int-42.
- Rounding, Int-5.

RPF file type, Int-58.
 RREG instruction, Int-78,
 RM-422.
 R-squared statistics, Int-30,
 RM-292.
 %RSSCMOM function, UG-
 514.
 RTOC instruction, UG-452,
 RM-426.
 Rubio-Ramirez, J., UG-219.
 Run button, Int-7.

S

SAMPLE instruction, RM-
 428.
 Sample range, RM-455.
 Sample selection bias, UG-
 407.
 Sandwich estimator, UG-41.
 SAVE instruction, Int-111,
 RM-430.
 %SCALAR function, UG-
 328.
 Scalars
 calculations, RM-47.
 variables, UG-2.
 SCATTER instruction, Int-
 71, Int-126, RM-431.
 with complex series, UG-
 454.
 OVERLAY option, RM-
 360.
 with SPGRAPH, RM-462.
 Scatter plots, Int-71, RM-
 431.
 multiple per page, RM-
 462.
 multiple scales, RM-462.
 Scatter (X-Y) Graph wiz-
 ard, Int-71.
 Schur decomposition, RM-
 398.
 Schwarz criterion, UG-60,
 UG-61, UG-182.

@BJEST procedure, UG-
 198.
 for VARs, UG-209,
 UG-212.
 Scientific notation, Int-40.
 SCREEN I/O unit, RM-361.
 Seasonal
 adjustment, Int-52.
 Census X11/X12, RM-
 537.
 exponential smooth-
 ing, RM-153.
 differencing, RM-104.
 dummies, Int-61, RM-439.
 forecasting models, UG-
 178.
 masks, RM-40.
 removing with linear re-
 gression, RM-174.
 @SEASONALDLM proce-
 dure, UG-329.
 SEASONAL instruc-
 tion, RM-439.
 SEED instruction, RM-442.
 Seemingly unrelated re-
 gression, UG-53,
 RM-487. *See also* SUR
 instruction.
 non-linear, UG-142,
 RM-340.
 with panel data, UG-413,
 UG-422.
 SELECT instruction, UG-
 489, RM-444.
 Selection bias, UG-403.
 Sequential probit, UG-400.
 Serial correlation
 consistent covariance ma-
 trix, RM-313, RM-548.
 estimation with, UG-45,
 RM-6, RM-200.
 estimation with high or-
 der, RM-174.
 estimation with panel
 data, UG-422.

periodogram test for, RM-
 25.
 tests for, Int-37, UG-81,
 RM-55.
 Serial number, Int-162.
 Series, Int-9.
 of arrays, RM-245.
 arrays of, UG-21, UG-
 483, RM-305.
 clearing, RM-37.
 compacting frequency, Int-
 104.
 compressing, RM-428.
 creating, Int-9, Int-25,
 RM-37, RM-446.
 creating from array, RM-
 307.
 creating from label
 (%S), UG-484.
 cross-sectional, UG-394.
 data type, UG-16, UG-19.
 aggregate, UG-10,
 UG-16.
 default length, RM-4.
 displaying, RM-51.
 dummy variables, Int-61.
 editing, Int-13.
 empirical density func-
 tion, RM-100.
 expanding frequency, Int-
 105.
 extreme values of, RM-
 171.
 filtering, RM-174, RM-
 186.
 forecasting, RM-189.
 fractiles, RM-469.
 frequency of, Int-103, RM-
 26, RM-30, RM-75,
 RM-428.
 graphing, RM-223.
 histogram, RM-100.
 labels, RM-282.
 listing all, RM-449.

- looping over list of, UG–482.
- making an array from a set of, RM–305.
- maximum of, RM–469.
- mean of, RM–326, RM–469, RM–499.
- median of, RM–469, RM–499.
- minimum of, RM–469.
- missing values, Int–62, Int–107.
- names, Int–22, RM–147, RM–282.
- numbers, UG–19, UG–469, UG–483, RM–282.
- panel data, RM–369, RM–373.
- printing, Int–59, RM–51, RM–383.
 - from a RATS file, RM–391.
- quantiles, RM–469.
- ranks, RM–366.
- reading from a file, Int–96, RM–75.
- sampling at an interval, RM–428.
- scalar computations, UG–23.
- scatter plots of, RM–431.
- selecting from list, UG–489, RM–444.
- setting by elements, RM–49, RM–446.
- sorting, RM–366.
- statistics on, RM–326, RM–469, RM–499.
- storing on RATS format files, RM–98, RM–475.
- transformations, Int–25, RM–446.
 - writing to files, RM–51.
- Series Edit window, Int–9, Int–13.
- Series window, Int–23, Int–38, RM–383, RM–475, RM–526.
- SETAR model, UG–369.
- SET instruction, Int–25, RM–446.
- %S function, UG–21, UG–23.
 - creating series with, UG–484.
- Shiller, R. J., UG–58.
- Shocks
 - adding to forecasts, RM–193.
 - responses to, RM–256.
- SHOCKS option
 - example, UG–284.
- @SHORTANDLONG procedure, UG–224.
- SHOW instruction, RM–449.
- %SIGMACMOM function, UG–516.
- %SIGMA variable, UG–210, RM–160, RM–346, RM–491.
- Significance levels
 - computing, UG–71, RM–33.
 - interpreting, UG–38.
- %SIGNIF variable, UG–73.
- SIMPLERBC.RPF example, RM–129.
- Simplex method
 - non-linear optimization, UG–117.
- Sims, C. A., UG–58, UG–84, UG–103, UG–208, UG–212, UG–216, UG–232, UG–268, UG–269, UG–218, RM–399.
- Sims exogeneity test, UG–84.
- SIMULADD.RPF example, UG–284.
- SIMULATE instruction, UG–500, UG–505, RM–450.
- Simulations, Int–65, UG–505, RM–450.
 - bookkeeping for, UG–500.
 - using FORECAST, RM–189.
 - historical, UG–283.
 - probability (computing), UG–501.
- SIMULEST.RPF example, UG–274.
- SIMULFOR.RPF example, UG–282.
- SIMULMUL.RPF example, UG–283.
- Simultaneous equation models
 - accuracy statistics, UG–283.
 - add factors, UG–284.
 - algorithm for solving, UG–281.
 - constructing models, RM–233.
 - convergence, UG–282.
 - formulas, UG–279.
 - historical simulations, UG–283.
 - identities, RM–197.
 - multipliers, UG–283.
 - out-of-sample forecasts, UG–282.
 - preliminary transformations, UG–274.
 - three-stage least squares, UG–53.
- Single-Equation Forecasts wizard, Int–54, RM–385, RM–517.
- Skewness, RM–469.
- SKIPSAVE option, RM–194.
- Smoothing, Int–44.
 - complex series, RM–532.

- exponential, Int-48, UG-176, RM-153.
- SMPL instruction, Int-82, RM-455.
 - getting current setting, RM-269.
- SMPL option, Int-67.
 - with Kalman filter, RM-274.
 - using SMPL instruction, RM-455.
- Solving models, UG-281.
- Sorting
 - data series, RM-366.
- SOURCE instruction, UG-471, UG-489, RM-457.
- Spearman's rho, RM-366.
- @SPECFORE procedure, UG-205, UG-462.
- SPECFORE.RPF example, UG-206.
- Specification tests, UG-84, UG-95, UG-140.
 - Harvey-Collier, UG-52.
- SPECIFY instruction, UG-256, RM-458.
- Spectral analysis, UG-452.
 - complex expressions, UG-448.
 - density, UG-456, UG-458, RM-188, RM-377.
 - forecasting, UG-176, UG-205.
 - window, UG-456, RM-532.
- @SPECTRUM procedure, UG-458.
- Spencer filter, RM-174.
- SPGRAPH instruction, Int-126, RM-216, RM-462.
- SPGRAPH.RPF example, Int-136.
- Splines, UG-58.
- SPREAD option, UG-42.
- Spreadsheets
 - adding dates to files, Int-119.
 - reading data from, Int-114.
- @SPUNIT procedure, UG-102.
- @SSMSPECTRUM procedure, UG-459.
- SSTATS Instruction, UG-23, RM-3, RM-467.
- Stability tests, UG-86, UG-90.
 - CUSUM, RM-419.
- @STABTEST procedure, UG-86, UG-90.
- Standard error
 - of autocorrelations, RM-547.
 - of coefficients, vector of, RM-292.
 - of forecast, RM-180.
 - of projection, RM-385.
 - of series, RM-499.
- STAR model, UG-370.
- @STARTEST procedure, UG-371.
- START option, UG-411.
 - for estimation instructions, UG-135.
- Stata data files, Int-99.
- State-space models. *See* Dynamic Linear Models.
- Static forecasts, UG-161.
- Statistics
 - forecast performance, RM-507.
 - marginal significance levels, RM-33.
 - moving window, RM-326.
 - regression, RM-292.
 - for series, RM-469, RM-499.
- STATISTICS instruction, Int-24, RM-467, RM-469.
- Statistics menu
 - Limited/Discrete Dependent Variables, Int-79.
 - Panel Data Regressions, Int-79.
 - Recursive Least Squares, Int-79.
 - Regressions, Int-27.
 - Regression Tests, Int-69.
 - Univariate Statistics, Int-24.
- %STDERRS vector, UG-25, UG-35.
- STEPS instruction, UG-162, RM-473.
 - saving results, UG-165.
- Stepwise regression, Int-78, RM-479.
- Stochastic volatility model, UG-340.
- Stock, J., UG-104.
- Stop button, Int-7.
- STORE instruction, Int-111, RM-475.
- Strazicich, M., UG-377.
- %STRCMP function, UG-23.
- %STRCMPNC function, UG-23.
- %STRFIND function, UG-23.
- STRING data type, UG-10, UG-14.
- %STRING function, UG-23.
- Strings
 - concatenating, Int-132.
 - expressions, RM-49.
 - line breaks in, Int-132.
 - operations with, RM-543.
 - reading from files, Int-132.
 - selecting from list, RM-444.
 - variables, UG-23.
- %STRLEN function, UG-23.

- %STRLOWER function, UG–23.
- %STRMATCH function, UG–23.
- %STRREP function, UG–23.
- %STRTRIM function, UG–23.
- @StructResids procedure, UG–229.
- Structural breaks
 - in ARIMA models, UG–199.
 - Bai-Perron algorithm, UG–367.
 - in unit root tests, UG–376.
- Structural VAR, UG–218, UG–228, RM–70.
- %STRUPPER function, UG–23.
- %STRVAL function, UG–23.
- STWISE instruction, Int–78, RM–479.
- Style Sheets
 - graph, Int–149.
- Sub-FRML's, RM–200.
- Subiterations Limit Exceeded, UG–156.
- Subprograms, UG–470.
- Sum
 - partial, RM–2.
- SUMMARIZE instruction, UG–71, UG–70, UG–57.
- Supplementary cards, Int–28.
 - in procedures, RM–139.
 - sets of, RM–295.
- SUR. *See* Seemingly unrelated regression.
- @SURGIBBSSETUP procedure, UG–533.
- SUR instruction, Int–79, UG–53, RM–487.
 - example of, UG–54.
 - technical information, UG–53.
 - three stage least squares, UG–276.
 - with VAR's, UG–211.
 - SUR.RPF example, UG–54.
 - Susmel, R., UG–390.
 - SV.RPF example, UG–340.
 - @SWAMY procedure, UG–427.
 - SWAMY.RPF example, UG–427.
 - Swamy's random coefficients, UG–427.
 - SWARCH.RPF example, UG–390.
 - @SWDOLS procedure, UG–249.
 - SWEEP instruction, UG–421, RM–494.
 - SYMMETRIC data type
 - aggregate, UG–10, UG–16.
 - Syntax errors, Int–80.
 - SYSTEM instruction, UG–208, RM–497.
 - Systems of equations
 - estimating, RM–487.
 - forecasting, RM–189.
 - linear estimation, UG–53.
- T**
- TABLE instruction, Int–23, RM–499.
- TAPER instruction, UG–457, RM–502.
- Tapers, UG–457, RM–502.
- Taqqu, M. S., UG–465.
- TAR model, UG–52, UG–369.
- t distribution
 - CDF instruction, RM–33.
 - random draws, UG–506.
 - multivariate, UG–508.
- Technical support, Int–162.
- Test
 - Chow predictive, UG–90.
- TEST instruction, UG–71, RM–504.
- Tests
 - analysis of variance, RM–393.
 - ARCH, UG–80.
 - causality, UG–84.
 - Chow, UG–86, UG–87.
 - coefficient values, RM–504.
 - cumulated periodogram, UG–82, RM–25.
 - Diebold-Mariano, UG–171.
 - exogeneity, UG–84.
 - Goldfeld-Quandt, UG–79.
 - Hansen specification, UG–98.
 - Harvey-Collier, UG–52.
 - Hausman specification, UG–95.
 - heteroscedasticity, UG–77.
 - lag length, UG–212.
 - Lagrange multiplier, UG–75.
 - likelihood ratio, RM–399.
 - linear restrictions, RM–324, RM–411.
 - MacKinnon-White-Davidson, UG–94.
 - marginal significance levels, UG–71.
 - Monte Carlo, RM–450.
 - normality, RM–469.
 - RESET, UG–93.
 - serial correlation, UG–81, RM–547.
 - specification, UG–95.
 - structural stability, UG–86.
 - unit roots, UG–100.
 - White heteroscedasticity, UG–78.
- TeX

- copy to, Int-16.
 Textbook examples, Int-160.
 Theil, H., UG-144, UG-252, UG-258.
 THEIL instruction, UG-190, UG-259, UG-283, RM-507.
 %THEIL matrix, RM-511.
 Theil U statistic, UG-170, UG-259, RM-507.
 Three-stage least squares, , UG-276.
 non-linear, RM-340.
 Threshold autoregression, UG-52.
 @THRESHTEST procedure, UG-371.
 Time effects, UG-419, RM-393.
 computing means of, RM-369.
 Time Series menu
 Exponential Smoothing, Int-48.
 Single-Equation Forecasts, Int-54.
 Time-varying coefficients, UG-267, RM-515.
 estimating with KALMAN, RM-274.
 more general, UG-270.
 Tobit models, UG-403, UG-409.
 TOBIT.RPF example, UG-409.
 %TODAY function, UG-30.
 Toolbar icons
 series edit window, Int-14.
 series window, Int-38.
 TRACE option, UG-113.
 %TRADEDAY function, UG-30, RM-22.
 Transfer function models, RM-15.
 spectral, RM-513.
 Transformations wizard, Int-26, RM-2, RM-299, RM-446.
 Trend
 creating, RM-446.
 and forecasts, UG-177.
 removing, RM-104.
 with HP filter, UG-339, RM-174.
 with linear regression, RM-174.
 Trend/Seasonals/Dummies wizard, Int-26, Int-53, Int-61, RM-439.
 Trend series, Int-53, Int-60.
 TRFUNCTION instruction, , UG-452.
 %TRSQUARED variable, UG-75.
 %TRSQ variable, UG-75.
 Truncated samples, UG-403, RM-284.
 TSAYP149.RPF example, RM-519.
 @TSAYTEST procedure, UG-371.
 t-statistics, RM-469.
 vector of coefficient, RM-292.
 %TSTATS vector, UG-36.
 %TTEST function, UG-71.
 T variable for SET, RM-447.
 @TVARSET procedure, UG-267.
 TVARYING instruction, UG-267, RM-515.
 TVARYING.RPF example, UG-267.
 Two-stage least squares, UG-48, UG-138.
 with AR1 errors, RM-6.
 instruments list, RM-272.
 non-linear, RM-331.
 TYPE instruction, UG-474, RM-516.
- ## U
- UC model, UG-322.
 UFORECAST instruction, Int-55, UG-505, RM-517.
 @UFOREERRORS procedure, UG-170, UG-190.
 Unbalanced panel data sets, RM-373.
 Underflow, UG-110.
 Uniform distribution
 random draws, UG-500, UG-506.
 %UNIFORM function, UG-500, UG-506.
 UNION.RPF example, UG-402.
 UNITROOT.RPF example, UG-101.
 Unit root tests, UG-100.
 Bayesian, UG-103.
 with structural breaks, UG-376.
 Univariate Statistics wizard, RM-55, RM-469, RM-171.
 Unobservable components, UG-322.
 UNRAVEL option, UG-58, UG-62, RM-136.
 UNTIL instruction, Int-92, UG-468, RM-520.
 UPDATE instruction, Int-111.
 USERMENU instruction, UG-489, RM-521.
 %UZWZU variable, UG-140, RM-292.

V

VADD instruction, RM-323, RM-524.
%VALID function, Int-41, Int-63.
Value at Risk (VaR), UG-542.
%VALUE function, UG-23.
VAR
 Bayesian, UG-253, RM-458.
 Blanchard-Quah, UG-223, UG-228.
 block exogeneity tests, RM-399.
 Choleski factorization, UG-217, UG-228.
 conditional forecasts, UG-242, UG-245.
 counterfactual simulation, UG-241.
 decomposition of variance, RM-148.
 defining, RM-103, RM-283.
 deterministic variables, RM-103.
 differencing variables, UG-209.
 dummy observation prior, UG-272.
 error correction, UG-247.
 estimating, UG-210, RM-158.
 with LINREG, UG-210.
 with SUR, UG-211.
 exogenous variables, RM-103.
 forecasting, UG-253, RM-189.
 historical decompositions, UG-239, RM-248.
 hypothesis tests, UG-212.

impulse responses, UG-234, RM-149, RM-256.
Kalman filter, UG-264.
lag length, UG-209, UG-212, UG-255.
lags, setting list of, RM-283.
long-run restriction, UG-223.
Markov switching, UG-385.
moving average representation, UG-216.
near, UG-208, UG-211.
orderings, UG-233.
orthogonalization, UG-216.
partial, UG-262.
preliminary transformations, UG-251.
prior, UG-254, UG-256, UG-259.
random draw for coefficients, UG-514.
seasonal data, UG-263.
selection of variables, UG-261.
setting up a, RM-497, RM-525.
SPECIFY instruction, UG-256.
structural, UG-218, UG-228, RM-70.
 residuals, UG-229.
time-varying coefficients, UG-267, RM-274, RM-279, RM-515.
trends, UG-209.
variance decomposition, UG-232.
VARCAUSE.RPF example, UG-214.

VAR (Forecast/Analyze) wizard, UG-211, RM-148, RM-189, RM-248, RM-256.

Variables

accessible, UG-8.
data types, UG-10.
declaring, UG-11, RM-96.
defined by RATS, UG-8.
displaying values of, UG-4, RM-108.
global, RM-297.
input/output of, UG-12.
local in procedures, UG-470, UG-475, RM-297.
names (form of), UG-11.
reading values into, RM-266, RM-401.
scalar, UG-2.
setting with COMPUTE, RM-47.
user input for, RM-395.
VARIABLES instruction, UG-208, RM-525.

Variance

components model, UG-425, RM-393.
decomposition of, UG-232, RM-148.
for EQUATION, RM-11.
for FRML, RM-198.
moving window, RM-326.
of series, RM-469.
@VARIRF procedure, UG-231, UG-234.
VARLAG.RPF example, UG-212.
@VARLAGSELECT procedure, UG-212.
%VARLAGSUMS array, UG-223.

- VAR (Setup/Estimate) wizard, UG-208, RM-103, RM-132, RM-158, RM-458, RM-497, RM-525.
- VCV instruction, RM-526.
- VECM, RM-132.
- VECTOR data type, RM-97.
- aggregate, UG-10, UG-17.
- Vector Error Correction Model, RM-132.
- View menu, Int-10.
- Change Layout Report Window, Int-15.
- Series Edit Windows, Int-13.
- Data Table, Int-25.
- Series Window, Int-12, Int-38.
- Statistics, Int-10, Int-23.
- Time Series Graph, Int-10.
- VREPLACE instruction, , RM-323, UG-279.
- W**
- Waggoner, D., UG-219.
- Wald test, UG-74.
- for logit, probit, RM-94.
- %WEEKDAY function, UG-30.
- Weekly data, RM-27.
- Weighted least squares, UG-42, RM-288.
- with panel data, UG-420.
- Weight matrix
- instrumental variables estimator, UG-138.
- West, K., UG-41.
- %WFRACILES function, UG-504.
- WHILE instruction, Int-92, UG-468, RM-531.
- White
- covariance matrix, UG-43, RM-313, RM-548.
- heteroscedasticity test, UG-78.
- White, H., UG-78.
- Window, Int-13.
- change layout, Int-13.
- data, RM-502.
- directing output to, RM-361.
- displaying data in, UG-12.
- displaying information in, UG-489.
- Graph, Int-56, Int-127.
- Input, Int-7.
- Output, Int-7.
- RATS Data File, Int-112.
- Report, Int-11.
- Series, Int-23.
- toolbar, Int-38.
- Series Edit, Int-9.
- WINDOW instruction, UG-456, RM-532.
- MASK option, RM-40.
- Window menu
- Close All Graphs, Int-56.
- Report Windows, Int-29.
- Use for Input, Int-7.
- Use for Output, Int-7.
- WINDOW option
- saving forecasts, UG-166.
- Windows Metafile format, RM-141.
- Window (spectral)
- leakage, RM-502.
- smoothing, RM-532.
- Wishart distribution
- random draws, UG-508.
- Wizards
- ARCH/GARCH, RM-207.
- Box-Jenkins (ARIMA models), RM-15.
- Calendar, RM-27.
- Covariance Matrix, RM-526.
- Cross Correlations, RM-61.
- Data, RM-75.
- Data (Other Formats), Int-18, Int-103.
- Data (RATS Format), Int-110.
- Differencing, Int-26, RM-104.
- Equation/FRML, Int-87, RM-143, RM-197.
- Exponential Smoothing, Int-48, RM-153.
- Filter/Smooth, Int-26, Int-48, RM-174.
- Graph, Int-46, RM-223.
- Limited/Discrete Dependent Variables, Int-79, RM-89, RM-284.
- Linear Regressions, Int-35, RM-6, RM-288, RM-422, RM-479.
- Moving Window Statistics, RM-326, RM-329.
- Panel Data Regressions, Int-79, RM-378.
- Recursive Least Squares, Int-79.
- Regression Tests, Int-69, UG-73, RM-324, RM-411, RM-165.
- Scatter (X-Y) Graph, Int-71.
- Series Window, RM-383, RM-475, RM-526.
- Single-Equation Forecasts, Int-54, RM-385, RM-517.
- Transformations, Int-26, RM-2, RM-299, RM-446.

Trend/Seasonals/Dummies, Int-26, Int-53, RM-439.
Univariate Statistics, Int-24, RM-55, RM-469, RM-171.
VAR (Forecast/Analyze), UG-211, RM-148, RM-189, RM-248, RM-256.
VAR (Setup/Estimate), UG-208, RM-103, RM-132, RM-158, RM-458, RM-497, RM-525.
X11, RM-537.
WKS files, Int-114, RM-75.
%WMATRIX matrix, UG-139.
WMATRIX option, UG-138.
Wooldridge, J. M., UG-296.
Wraparound effect, UG-461.
WRITE instruction, UG-12, RM-535.
with complex series, RM-427.

X

X11 instruction, RM-537.
X11 wizard, RM-537.
%X array, RM-5.
%XX matrix, UG-25.

Y

%YEAR function, UG-30.
Yule-Walker equations, RM-265.

Z

Zellner GLS, RM-487.
Zha, T., UG-219, UG-232.
%Z matrix, RM-66.
%ZTEST function, UG-71.
ZUMEAN option, UG-141.

