Applied Econometrics using MATLAB

James P. LeSage
Department of Economics
University of Toledo

October, 1999

Preface

This text describes a set of MATLAB functions that implement a host of econometric estimation methods. Toolboxes are the name given by the MathWorks to related sets of MATLAB functions aimed at solving a particular class of problems. Toolboxes of functions useful in signal processing, optimization, statistics, finance and a host of other areas are available from the MathWorks as add-ons to the standard MATLAB software distribution. I use the term *Econometrics Toolbox* to refer to the collection of function libraries described in this book.

The intended audience is faculty and students using statistical methods, whether they are engaged in econometric analysis or more general regression modeling. The MATLAB functions described in this book have been used in my own research as well as teaching both undergraduate and graduate econometrics courses. Researchers currently using Gauss, RATS, TSP, or SAS/IML for econometric programming might find switching to MATLAB advantageous. MATLAB software has always had excellent numerical algorithms, and has recently been extended to include: sparse matrix algorithms, very good graphical capabilities, and a complete set of object oriented and graphical user-interface programming tools. MATLAB software is available on a wide variety of computing platforms including mainframe, Intel, Apple, and Linux or Unix workstations.

When contemplating a change in software, there is always the initial investment in developing a set of basic routines and functions to support econometric analysis. It is my hope that the routines in the *Econometrics Toolbox* provide a relatively complete set of basic econometric analysis tools. The toolbox also includes a number of functions to mimic those available in Gauss, which should make converting existing Gauss functions and applications easier. For those involved in vector autoregressive modeling, a complete set of estimation and forecasting routines is available that implement a wider variety of these estimation methods than RATS software. For example, Bayesian Markov Chain Monte Carlo (MCMC) estimation of VAR models that robustify against outliers and accommodate heteroscedastic disturbances have been implemented. In addition, the estimation functions for error correction models (ECM) carry out Johansen's tests to determine the number of cointegrating relations, which are automatically incorporated in the model. In the area of vector autoregressive forecasting, routines are available for VAR and ECM methods that automatically handle data transformations (e.g. differencing, seasonal differences, growth rates). This allows users to work with variables in raw levels form. The forecasting functions carry out needed transformations for estimation and return forecasted values in level form. Comparison of forecast accuracy from a wide variety of vector autoregressive, error correction and other methods is quite simple. Users can avoid the difficult task of unraveling transformed forecasted values from alternative estimation methods and proceed directly to forecast accuracy comparisons.

The collection of around 300 functions and demonstration programs are organized into libraries that are described in each chapter of the book. Many faculty use MATLAB or Gauss software for research in econometric analysis, but the functions written to support research are often suitable for only a single problem. This is because time and energy (both of which are in short supply) are involved in writing more generally applicable functions. The functions described in this book are intended to be re-usable in any number of applications. Some of the functions implement relatively new Markov Chain Monte Carlo (MCMC) estimation methods, making these accessible to undergraduate and graduate students with absolutely no programming involved on the students part. Many of the automated features available in the vector autoregressive, error correction, and forecasting functions arose from my own experience in dealing with students using these functions. It seemed a shame to waste valuable class time on implementation details when these can be handled by well-written functions that take care of the details.

A consistent design was implemented that provides documentation, example programs, and functions to produce printed as well as graphical presentation of estimation results for all of the econometric functions. This was accomplished using the "structure variables" introduced in MATLAB Version 5. Information from econometric estimation is encapsulated into a single variable that contains "fields" for individual parameters and statistics related to the econometric results. A thoughtful design by the MathWorks allows these structure variables to contain scalar, vector, matrix, string, and even multi-dimensional matrices as fields. This allows the econometric functions to return a single structure that contains all estimation results. These structures can be passed to other functions that can intelligently decipher the information and provide a printed or graphical presentation of the results.

The *Econometrics Toolbox* should allow faculty to use MATLAB in undergraduate and graduate level econometrics courses with absolutely no programming on the part of students or faculty. An added benefit to using MATLAB and the *Econometrics Toolbox* is that faculty have the option of implementing methods that best reflect the material in their courses as well as their own research interests. It should be easy to implement a host of ideas and methods by: drawing on existing functions in the toolbox, extending these functions, or operating on the results from these functions. As there is an expectation that users are likely to extend the toolbox, examples of how to accomplish this are provided at the outset in the first chapter. Another way to extend the toolbox is to download MATLAB functions that are available on Internet sites. (In fact, some of the routines in the toolbox originally came from the Internet.) I would urge you to re-write the documentation for these functions in a format consistent with the other functions in the toolbox and return the results from the function in a "structure variable". A detailed example of how to do this is provided in the first chapter.

In addition to providing a set of econometric estimation routines and documentation, the book has another goal. Programming approaches as well as design decisions are discussed in the book. This discussion should make it easier to use the toolbox functions intelligently, and facilitate creating new functions that fit into the overall design, and work well with existing toolbox routines. This text can be read as a manual for simply using the existing functions in the toolbox, which is how students tend to approach the book. It can also be seen as providing programming and design approaches that will help implement extensions for research and teaching of econometrics. This is how I would think faculty would approach the text. Some faculty in Ph.D. programs expect their graduate students to engage in econometric problem solving that requires programming, and certainly this text would eliminate the burden of spending valuable course time on computer programming and implementation details. Students in Ph.D. programs receive the added benefit that functions implemented for dissertation work can be easily transported to another institution, since MATLAB is available for almost any conceivable hardware/operating system environment.

Finally, there are obviously omissions, bugs and perhaps programming errors in the *Econometrics Toolbox*. This would likely be the case with any such endeavor. I would be grateful if users would notify me when they encounter problems. It would also be helpful if users who produce generally useful functions that extend the toolbox would submit them for inclusion.

Much of the econometric code I encounter on the internet is simply too specific to a single research problem to be generally useful in other applications. If econometric researchers are serious about their newly proposed estimation methods, they should take the time to craft a generally useful MATLAB function that others could use in applied research. Inclusion in the *Econometrics Toolbox* would also have the benefit of introducing the method to faculty teaching econometrics and their students.

The latest version of the *Econometrics Toolbox* functions can be found on the Internet at: http://www.econ.utoledo.edu under the MATLAB gallery icon. Instructions for installing these functions are in an Appendix to this text along with a listing of the functions in the library and a brief description of each.

Acknowledgements

First I would like to acknowledge my students who suffered through previous buggy and error filled versions of the toolbox functions. I owe a great deal to their patience and persistence in completing projects under such conditions.

I would like to thank Michael R. Dowd would was brave enough to use early versions of the vector autoregressive library routines in his research. In addition to providing feedback regarding the functions and reading early drafts of the text, Mike also provided LaTeX assistance to produce this text. David C. Black is another faculty colleague who tested many of the functions and read early drafts of the text. Michael Magura who has always been a great editor and helpful critic also read early drafts of the text.

Some of the functions in the *Econometrics Toolbox* represent algorithms crafted by others who unselfishly placed them in the public domain on the Internet. The authors names and affiliations have been preserved in these functions. In all cases, the documentation was modified to match the format of the *Econometrics Toolbox* and in many cases, the arguments returned by the function were converted to MATLAB structure variables. In other cases, functions written in other languages were converted to the MATLAB language. I am indebted to those who implemented these routines as well.

The following persons have generously contributed their functions for inclusion in the *Econometrics Toolbox*. C. F. Baum from Boston College, Mike Cliff from UNC, Matt Hergott a Duke B-school graduate and Tony E. Smith from University of Pennsylvania. Tony also spent a great deal of time looking at the spatial econometrics functions and reporting problems and bugs.

Many others have sent e-mail comments regarding errors and I am grateful to all of you.

Contents

1 Introduction **1**

2 Regression using MATLAB **5**
2.1 Design of the regression library 6
2.2 The ols function 8
2.3 Selecting a least-squares algorithm 12
2.4 Using the results structure 17
2.5 Performance profiling the regression toolbox 28
2.6 Using the regression library 30
2.6.1 A Monte Carlo experiment 31
2.6.2 Dealing with serial correlation 32
2.6.3 Implementing statistical tests 38
2.7 Chapter summary 41

Chapter 2 Appendix **42**

3 Utility Functions **45**
3.1 Calendar function utilities 45
3.2 Printing and plotting matrices 49
3.3 Data transformation utilities 65
3.4 Gauss functions 69
3.5 Wrapper functions 73
3.6 Chapter summary 76

Chapter 3 Appendix **77**

4 Regression Diagnostics **80**
4.1 Collinearity diagnostics and procedures 80
4.2 Outlier diagnostics and procedures 94
4.3 Chapter summary 100

Chapter 4 Appendix **101**

5 VAR and Error Correction Models **103**
5.1 VAR models 103
5.2 Error correction models 113
5.3 Bayesian variants 125
5.3.1 Theil-Goldberger estimation of these models 138
5.4 Forecasting the models 139
5.5 Chapter summary 145

Chapter 5 Appendix **148**

6 Markov Chain Monte Carlo Models **151**
6.1 The Bayesian Regression Model 154
6.2 The Gibbs Sampler 156
6.2.1 Monitoring convergence of the sampler 159
6.2.2 Autocorrelation estimates 163
6.2.3 Raftery-Lewis diagnostics 163
6.2.4 Geweke diagnostics 165
6.3 A heteroscedastic linear model 169
6.4 Gibbs sampling functions 175
6.5 Metropolis sampling 184
6.6 Functions in the Gibbs sampling library 190
6.7 Chapter summary 197

Chapter 6 Appendix **199**

7 Limited Dependent Variable Models **204**
7.1 Logit and probit regressions 206
7.2 Gibbs sampling logit/probit models 211
7.2.1 The probit_g function 218
7.3 Tobit models 220
7.4 Gibbs sampling Tobit models 224
7.5 Chapter summary 227

Chapter 7 Appendix **228**

8 Simultaneous Equation Models **230**
8.1 Two-stage least-squares models 230
8.2 Three-stage least-squares models 235
8.3 Seemingly unrelated regression models 240

8.4 Chapter summary 244

Chapter 8 Appendix 246

9 Distribution functions library 247
9.1 The pdf, cdf, inv and rnd functions 248
9.2 The specialized functions 249
9.3 Chapter summary 256

Chapter 9 Appendix 257

10 Optimization functions library 260
10.1 Simplex optimization 261
10.1.1 Univariate simplex optimization 261
10.1.2 Multivariate simplex optimization 268
10.2 EM algorithms for optimization 269
10.3 Multivariate gradient optimization 278
10.4 Chapter summary 287

Chapter 10 Appendix 288

11 Handling sparse matrices 289
11.1 Computational savings with sparse matrices 289
11.2 Estimation using sparse matrix algorithms 297
11.3 Gibbs sampling and sparse matrices 304
11.4 Chapter summary 309

Chapter 11 Appendix 310

References 313

Appendix: Toolbox functions 320

List of Examples

1.1 Demonstrate regression using the ols() function 8
2.2 Least-squares timing information 12
2.3 Profiling the ols() function 28
2.4 Using the ols() function for Monte Carlo 31
2.5 Generate a model with serial correlation 33
2.6 Cochrane-Orcutt iteration 34
2.7 Maximum likelihood estimation 37
2.8 Wald's F-test 38
2.9 LM specification test 40
3.1 Using the cal() function 47
3.2 Using the tsdate() function 48
3.3 Using cal() and tsdates() functions 49
3.4 Reading and printing time-series 49
3.5 Using the lprint() function 51
3.6 Using the tsprint() function 56
3.7 Various tsprint() formats 58
3.8 Truncating time-series and the cal() function 60
3.9 Common errors using the cal() function 60
3.10 Using the tsplot() function 61
3.11 Finding time-series turning points with fturns() 64
3.12 Seasonal differencing with sdiff() function 67
3.13 Annual growth rates using growthr() function 68
3.14 Seasonal dummy variables using sdummy() function 68
3.15 Least-absolute deviations using lad() function 71
4.1 Collinearity experiment 82
4.2 Using the bkw() function 86
4.3 Using the ridge() function 88
4.4 Using the rtrace() function 90
4.5 Using the theil() function 92
4.6 Using the dfbeta() function 94

4.7 Using the robust() function 97
4.8 Using the pairs() function 99
5.1 Using the var() function 105
5.2 Using the pgranger() function 107
5.3 VAR with deterministic variables 110
5.4 Using the lrratio() function 111
5.5 Impulse response functions 112
5.6 Using the adf() and cadf() functions 116
5.7 Using the johansen() function 119
5.8 Estimating error correction models 123
5.9 Estimating BVAR models 128
5.10 Using bvar() with general weights 130
5.11 Estimating RECM models 137
5.12 Forecasting VAR models 140
5.13 Forecasting multiple related models 141
5.14 A forecast accuracy experiment 143
6.1 A simple Gibbs sampler 157
6.2 Using the coda() function 160
6.3 Using the raftery() function 165
6.4 Geweke's convergence diagnostics 166
6.5 Using the momentg() function 168
6.6 Heteroscedastic Gibbs sampler 173
6.7 Using the ar_g() function 182
6.8 Metropolis within Gibbs sampling 188
6.9 Bayesian model averaging with the bma_g() function 192
7.1 Logit and probit regression functions 206
7.2 Demonstrate Albert-Chib latent variable 212
7.3 Gibbs vs. maximum likelihood logit 214
7.4 Gibbs vs. maximum likelihood probit 215
7.5 Heteroscedastic probit model 217
7.6 Tobit regression function 222
7.7 Gibbs sampling tobit estimation 225
8.1 Two-stage least-squares 231
8.2 Monte Carlo study of ols() vs. tsls() 234
8.3 Three-stage least-squares 237
8.4 Using the sur() function 242
9.1 Beta distribution function example 248
9.2 Random Wishart draws 252
9.3 Left- and right-truncated normal draws 253
9.4 Rejection sampling of truncated normal draws 254

10.1 Simplex maximum likelihood for Box-Cox model 265
10.2 EM estimation of switching regime model 275
10.3 Maximum likelihood estimation of the Tobit model 281
10.4 Using the optsolv() function 286
11.1 Using sparse matrix functions 291
11.2 Solving a sparse matrix system 292
11.3 Symmetric minimum degree ordering operations 295
11.4 Solving for rho using the far() function 303
11.5 Gibbs sampling with sparse matrices 307

List of Figures

2.1 Histogram of $\hat{\beta}$ outcomes 33

3.1 Output from tsplot() function 62
3.2 Graph of turning point events 66

4.1 Ridge trace plot 91
4.2 Dfbeta plots 95
4.3 Dffits plots 96
4.4 Pairwise scatter plots 99

5.1 Impulse response graphs 113
5.2 Prior means and precision for important variables 136
5.3 Prior means and precision for unimportant variables 137

6.1 Prior V_i distributions for various values of r 171
6.2 Mean of V_i draws 176
6.3 Distribution of $\phi_1 + \phi_2$ 184
6.4 $|I_n - \rho W|$ as a function of ρ 187

7.1 Cumulative distribution functions compared 206
7.2 Actual y vs. mean of latent y-draws 213
7.3 Posterior mean of v_i draws with outliers 219

9.1 Beta distribution demonstration program plots 250
9.2 Histograms of truncated normal distributions 253
9.3 Contaminated normal versus a standard normal distribution . 255

10.1 Plots from switching regime regression 277

11.1 Sparsity structure of W from Pace and Berry 291
11.2 An illustration of fill-in from matrix multiplication 294

11.3 Minimum degree ordering versus unordered Pace and Barry matrix 296

List of Tables

2.1 Timing (in seconds) for Cholesky and QR Least-squares . . . 13
2.2 Digits of accuracy for Cholesky vs. QR decomposition 16

4.1 Variance-decomposition proportions table 84
4.2 BKW collinearity diagnostics example 85
4.3 Ridge Regression for our Monte Carlo example 88

6.1 A Comparison of FAR Estimators 190

Chapter 1

Introduction

The *Econometrics Toolbox* contains around 50 functions that implement econometric estimation procedures, 20 functions to carry out diagnostic and statistical testing procedures, and 150 support and miscellaneous utility functions. In addition, there are around 100 demonstration functions covering all of the econometric estimation methods as well as the diagnostic and testing procedures and many of the utility functions. Any attempt to describe this amount of code must be organized.

Chapter 2 describes the design philosophy and mechanics of implementation using least-squares regression. Because regression is widely-understood by econometricians and others working with statistics, the reader should be free to concentrate on the 'big picture'. Once you understand the way that the *Econometric Toolbox* functions encapsulate estimation and other results in the new MATLAB Version 5 'structure variables', you're on the way to successfully using the toolbox.

Despite the large (and always growing) number of functions in the toolbox, you may not find exactly what you're looking for. From the outset, examples are provided that illustrate how to incorporate your own functions in the toolbox in a well-documented, consistent manner. Your functions should be capable of using the existing printing and graphing facilities to provide printed and graphical display of your results.

Chapters 3 through 10 focus more directly on description of functions according to their econometric purpose. These chapters can be read as merely a software documentation manual, and many beginning students approach the text in this manner. Another approach to the text is for those interested in MATLAB programming to accomplish research tasks. Chapters 3 through 10 describe various design challenges regarding the task of passing information to functions and returning results. These are used to illustrate alternative design and programming approaches. Additionally, some estimation procedures provide an opportunity to demonstrate coding that solves problems likely to arise in other estimation tasks. Approaching the text from this viewpoint, you should gain some familiarity with a host of alternative coding tricks, and the text should serve as a reference to the functions that contain these code fragments. When you encounter a similar situation, simply examine (or re-use) the code fragments modified to suit your particular problem.

Chapter 3 presents a library of utility functions that are used by many other functions in the *Econometrics Toolbox*. For example, all printing of results from econometric estimation and testing procedures is carried out by a single function that prints matrices in a specified decimal or integer format with optional column and row labels. Many of the examples throughout the text also rely on this function named **mprint**.

Regression diagnostic procedures are the topic of Chapter 4. Diagnostics for collinearity and influential observations from texts like Belsley, Kuh and Welsch (1980) and Cook and Weisberg (1982) are discussed and illustrated.

Chapter 5 turns attention to vector autoregressive and error correction models, as well as forecasting. Because we can craft our own functions in MATLAB, we're not restricted to a limited set of Bayesian priors as in the case of RATS software. It is also possible to ease the tasks involved with cointegration testing for error correction models. These tests and model formation can be carried out in a single function with virtually no intervention on the part of users. A similar situation exists in the area of forecasting. Users can be spared the complications that arise from data transformations prior to estimation that require reverse transformations to the forecasted values.

A recent method that has received a great deal of attention in the statistics literature, Markov Chain Monte Carlo, or MCMC is covered in Chapter 6. Econometric estimation procedures are also beginning to draw on this approach, and functions are crafted to implement these methods. Additional functions were devised to provide convergence diagnostics (that are an integral part of the method) as well as presentation of printed and graphical results. These functions communicate with each other via the MATLAB structure variables.

Chapter 7 takes up logit, probit and tobit estimation from both a maximum likelihood as well as MCMC perspective. Recent MCMC approaches to limited dependent variable models hold promise for dealing with nonconstant variance and outliers, and these are demonstrated in this chapter.

Simultaneous equation systems are the subject of Chapter 8, where we face the challenge of encapsulating input variables for a system of equations when calling our toolbox functions. Although MATLAB allows for variables that are global in scope, no such variables are used in any *Econometric Toolbox* functions. We solve the challenges using MATLAB structure variables.

Chapter 9 describes a host of functions for calculating probability densities, cumulative densities, quantiles, and random deviates from twelve frequently used statistical distributions. Other more special purpose functions dealing with statistical distributions are also described.

The subject of optimization is taken up in Chapter 10 where we demonstrate maximum likelihood estimation. Alternative approaches and functions are described that provide a consistent interface for solving these types of problems.

The final chapter discusses and illustrates the use of MATLAB sparse matrix functions. These are useful for solving problems involving large matrices that contain a large proportion of zeros. MATLAB has a host of algorithms that can be used to operate on this type of matrix in an intelligent way the conserves on both time and computer memory.

Readers who have used RATS, TSP or SAS should feel very comfortable using the *Econometrics Toolbox* functions and MATLAB. The procedure for producing econometric estimates is very similar to these other software programs. First, the data files are "loaded" and any needed data transformations to create variables for the model are implemented. Next, an estimation procedure is called to operate on the model variables and a command is issued to print or graph the results.

The text assumes the reader is familiar with basic MATLAB commands introduced in the software manual, *Using MATLAB Version 5* or *The Student Edition of MATLAB, Version 5 User's Guide* by Hanselmann and Littlefield (1997). Gauss users should have little trouble understanding this text, perhaps without reading the MATLAB introductory manuals, as the syntax of MATLAB and Gauss is very similar.

All of the functions in the *Econometrics Toolbox* have been tested using MATLAB Version 5.2 on Apple, Intel/Windows and Sun Microsystems computing platforms. The functions also work with Versions 5.0 and 5.1, but some printed output that relies on a new string justification option in Version 5.2 may not appear as nicely as it does in Version 5.2.

The text contains 71 example programs that come with the *Econometric Toolbox* files. Many of these examples generate random data samples, so the results you see will not exactly match those presented in the text. In addition to the example files, there are demonstration files for all of the econometric estimation and testing functions and most of the utility functions.

To conserve on space in the text, the printed output was often edited to eliminate 'white space' that appears in the MATLAB command window. In some cases output that was too wide for the text pages was also altered to eliminate white space.

As you study the code in the *Econometric Toolbox* functions, you will see a large amount of repeated code. There is a trade-off between writing functions to eliminate re-use of identical code fragments and clarity. Functions after all hide code, making it more difficult to understand the operations that are actually taking place without keeping a mental note of what the sub-functions are doing. A decision was made to simply repeat code fragments for clarity. This also has the virtue of making the functions more self-contained since they do not rely on a host of small functions.

Another issue is error checking inside the functions. An almost endless amount of error checking code could be written to test for a host of possible mistakes in the user input arguments to the functions. The *Econometric Toolbox* takes a diligent approach to error checking for functions like least-squares that are apt to be the first used by students. All functions check for the correct number of input arguments, and in most cases test for structure variables where they are needed as inputs. My experience has been that after students master the basic usage format, there is less need for extensive error checking. Users can easily interpret error messages from the function to mean that input arguments are not correct. A check of the input arguments in the function documentation against the user's MATLAB command file will usually suffice to correct the mistake and eliminate the error message. If your students (or you) find that certain errors are typical, it should be fairly straightforward to add error checking code and a message that is specific to this type of usage problem. Another point in this regard is that my experience shows certain types of errors are unlikely. For example, it is seldom the case that users will enter matrix and vector arguments that have a different number of observations (or rows). This is partly due to the way that MATLAB works. Given this, many of the functions do not check for this type of error, despite the fact that we could add code to carry out this type of check.

Chapter 2

Regression using MATLAB

This chapter describes the design and implementation of a *regression function library*. Toolboxes are the name given by the MathWorks to related sets of MATLAB functions aimed at solving a particular class of problems. Toolboxes of functions useful in signal processing, optimization, statistics, finance and a host of other areas are available from the MathWorks as add-ons to the standard MATLAB distribution. We will reserve the term *Econometrics Toolbox* to refer to the collection of function libraries discussed in each chapter of the text. Many of the function libraries rely on a common *utility function library* and on other function libraries. Taken together, these constitute the *Econometrics Toolbox* described in this book.

All econometric estimation methods were designed to provide a consistent user-interface in terms of the MATLAB help information, and related routines to print and plot results from various types of regressions. Section 2.1 discusses design issues and the use of MATLAB structures as a way to pass results from the various regression estimation functions to associated routines. Implementation of a least-squares function is discussed in Section 2.2 and the choice of least-squares algorithm is discussed in Section 2.3 from a speed and numerical accuracy standpoint. Section 2.4 takes up the design of related functions for printing and plotting regression results. The performance profiling capabilities of MATLAB are illustrated in Section 2.5 and Section 2.6 demonstrates the use of alternative regression functions from the library in an applied setting. All of the regression functions implemented in the library are documented in an appendix to the chapter. Some of these functions are discussed and illustrated in later chapters.

2.1 Design of the regression library

In designing a regression library we need to think about organizing our functions to present a consistent user-interface that packages all of our MATLAB regression functions in a unified way. The advent of 'structures' in MATLAB Version 5 allows us to create a host of alternative regression functions that all return 'results structures'.

A structure in MATLAB allows the programmer to create a variable containing what MATLAB calls 'fields' that can be accessed by referencing the structure name plus a period and the field name. For example, suppose we have a MATLAB function to perform ordinary least-squares estimation named **ols** that returns a structure. The user can call the function with input arguments (a dependent variable vector y and explanatory variables matrix x) and provide a variable name for the structure that the **ols** function will return using:

```
result = ols(y,x);
```

The structure variable 'result' returned by our **ols** function might have fields named 'rsqr', 'tstat', 'beta', etc. These fields might contain the R-squared statistic, $t-$statistics for the $\hat{\beta}$ estimates and the least-squares estimates $\hat{\beta}$. One virtue of using the structure to return regression results is that the user can access individual fields of interest as follows:

```
bhat = result.beta;
disp('The R-squared is:');
result.rsqr
disp('The 2nd t-statistic is:');
result.tstat(2,1)
```

There is nothing sacred about the name 'result' used for the returned structure in the above example, we could have used:

```
bill_clinton = ols(y,x);
result2      = ols(y,x);
restricted   = ols(y,x);
unrestricted = ols(y,x);
```

That is, the name of the structure to which the **ols** function returns its information is assigned by the user when calling the function.

To examine the nature of the structure in the variable 'result', we can simply type the structure name without a semi-colon and MATLAB will present information about the structure variable as follows:

```
result =
     meth: 'ols'
        y: [100x1 double]
     nobs: 100.00
     nvar: 3.00
     beta: [  3x1 double]
     yhat: [100x1 double]
    resid: [100x1 double]
     sige: 1.01
    tstat: [  3x1 double]
     rsqr: 0.74
     rbar: 0.73
       dw: 1.89
```

Each field of the structure is indicated, and for scalar components the value of the field is displayed. In the example above, 'nobs', 'nvar', 'sige', 'rsqr', 'rbar', and 'dw' are scalar fields, so their values are displayed. Matrix or vector fields are not displayed, but the size and type of the matrix or vector field is indicated. Scalar string arguments are displayed as illustrated by the 'meth' field which contains the string 'ols' indicating the regression method that was used to produce the structure. The contents of vector or matrix strings would not be displayed, just their size and type. Matrix and vector fields of the structure can be displayed or accessed using the MATLAB conventions of typing the matrix or vector name without a semi-colon. For example,

```
result.resid
result.y
```

would display the residual vector and the dependent variable vector y in the MATLAB command window.

Another virtue of using 'structures' to return results from our regression functions is that we can pass these structures to another related function that would print or plot the regression results. These related functions can query the structure they receive and intelligently decipher the 'meth' field to determine what type of regression results are being printed or plotted. For example, we could have a function **prt** that prints regression results and another **plt** that plots actual versus fitted and/or residuals. Both these functions take a regression structure as input arguments. Example 2.1 provides a concrete illustration of these ideas.

The example assumes the existence of functions **ols, prt, plt** and data matrices y, x in files 'y.data' and 'x.data'. Given these, we carry out a regression, print results and plot the actual versus predicted as well as residuals with the MATLAB code shown in example 2.1. We will discuss the **prt** and **plt** functions in Section 2.4.

```
% ----- Example 2.1 Demonstrate regression using the ols() function
load y.data;
load x.data;
result = ols(y,x);
prt(result);
plt(result);
```

2.2 The ols function

Now to put these ideas into practice, consider implementing an **ols** function. The function code would be stored in a file 'ols.m' whose first line is:

```
function results=ols(y,x)
```

The keyword 'function' instructs MATLAB that the code in the file 'ols.m' represents a callable MATLAB function.

The help portion of the MATLAB 'ols' function is presented below and follows immediately after the first line as shown. All lines containing the MATLAB comment symbol '%' will be displayed in the MATLAB command window when the user types 'help ols'.

```
function results=ols(y,x)
% PURPOSE: least-squares regression
%---------------------------------------------------
% USAGE: results = ols(y,x)
% where: y = dependent variable vector (nobs x 1)
%        x = independent variables matrix (nobs x nvar)
%---------------------------------------------------
% RETURNS: a structure
%        results.meth  = 'ols'
%        results.beta  = bhat
%        results.tstat = t-stats
%        results.yhat  = yhat
%        results.resid = residuals
%        results.sige  = e'*e/(n-k)
%        results.rsqr  = rsquared
%        results.rbar  = rbar-squared
%        results.dw    = Durbin-Watson Statistic
%        results.nobs  = nobs
%        results.nvar  = nvars
%        results.y     = y data vector
```

```
% -------------------------------------------------
% SEE ALSO: prt(results), plt(results)
%--------------------------------------------------
```

All functions in the econometrics toolbox present a unified documentation format for the MATLAB 'help' command by adhering to the convention of sections entitled, 'PURPOSE', 'USAGE', 'RETURNS', 'SEE ALSO', and perhaps a 'REFERENCES' section, delineated by dashed lines.

The 'USAGE' section describes how the function is used, with each input argument enumerated along with any default values. A 'RETURNS' section portrays the structure that is returned by the function and each of its fields. To keep the help information uncluttered, we assume some knowledge on the part of the user. For example, we assume the user realizes that the '.residuals' field would be an (nobs x 1) vector and the '.beta' field would consist of an (nvar x 1) vector.

The 'SEE ALSO' section points the user to related routines that may be useful. In the case of our **ols** function, the user might what to rely on the printing or plotting routines **prt** and **plt**, so these are indicated. The 'REFERENCES' section would be used to provide a literature reference (for the case of more exotic regression procedures) where the user could read about the details of the estimation methodology.

To illustrate what we mean by consistency in the user documentation, the following shows the results of typing 'help ridge', that provides user documentation for the **ridge** regression function in the regression library.

```
 PURPOSE: computes Hoerl-Kennard Ridge Regression
---------------------------------------------------
 USAGE: results = ridge(y,x,theta)
 where: y = dependent variable vector
        x = independent variables matrix
    theta = an optional ridge parameter
            (default: best theta value ala Hoerl-Kennard)
---------------------------------------------------
 RETURNS: a structure
        results.meth  = 'ridge'
        results.beta  = bhat
        results.theta = theta (input or HK determined value)
        results.tstat = t-stats
        results.yhat  = yhat
        results.resid = residuals
        results.sige  = e'*e/(n-k)
        results.rsqr  = rsquared
        results.rbar  = rbar-squared
        results.dw    = Durbin-Watson Statistic
```

```
        results.nobs  = nobs
        results.nvar  = nvars
        results.y     = y data vector
 --------------------------------------------------
 SEE ALSO: rtrace, prt, plt
---------------------------------------------------
 REFERENCES: David Birkes, Yadolah Dodge, 1993, Alternative Methods of
 Regression and Hoerl, Kennard, Baldwin, 1975 ‘Ridge Regression: Some
 Simulations’, Communcations in Statistics
```

Now, turning attention to the actual MATLAB code for estimating the ordinary least-squares model, we begin processing the input arguments to carry out least-squares estimation based on a model involving y and x. We first check for the correct number of input arguments using the MATLAB ‘nargin’ variable.

```
if (nargin ~= 2); error('Wrong # of arguments to ols');
else
 [nobs nvar] = size(x); [nobs2 junk] = size(y);
 if (nobs ~= nobs2); error('x and y must have same # obs in ols');
 end;
end;
```

If we don’t have two input arguments, the user has made an error which we indicate using the MATLAB **error** function. The **ols** function will return without processing any of the input arguments in this case. Another error check involves the number of rows in the y vector and x matrix which should be equal. We use the MATLAB **size** function to implement this check in the code above.

Assuming that the user provided two input arguments, and the number of rows in x and y are the same, we can pass on to using the input information to carry out a regression.

The ‘nobs’ and ‘nvar’ returned by the MATLAB **size** function are pieces of information that we promised to return in our results structure, so we construct these fields using a ‘.nobs’ and ‘.nvar’ appended to the ‘results’ variable specified in the function declaration. We also fill in the ‘meth’ field and the ‘y’ vector fields.

```
results.meth = 'ols';
results.y = y;
results.nobs = nobs;
results.nvar = nvar;
```

The decision to return the actual y data vector was made to facilitate the **plt** function that will plot the actual versus predicted values from the regression along with the residuals. Having the y data vector in the structure makes it easy to call the **plt** function with only the structure returned by a regression function.

We can proceed to estimate least-squares coefficients $\hat{\beta} = (X'X)^{-1}X'y$, but we have to choose a solution method for the least-squares problem. The two most commonly used approaches are based on the Cholesky and qr matrix decompositions. The regression library **ols** function uses the qr matrix decomposition method for reasons that will be made clear in the next section. A first point to note is that we require more than a simple solution for $\hat{\beta}$, because we need to calculate $t-$statistics for the $\hat{\beta}$ estimates. This requires that we compute $(X'X)^{-1}$ which is done using the MATLAB 'slash' operator to invert the $(X'X)$ matrix. We represent $(X'X)$ using $(r'r)$, where r is an upper triangular matrix returned by the qr decomposition.

```
[q r] = qr(x,0);
xpxi = (r'*r)\eye(nvar);
results.beta = r\(q'*y);
```

An alternative solution based on the Cholesky decomposition is faster, but less accurate for ill-conditioned X data matrices. The Cholesky solution could be implemented as:

```
xpxi = (x'*x)\eye(k);
results.beta = xpxi*(x'*y);
```

Given either of these solutions, we are in a position to use our estimates $\hat{\beta}$ to compute the remaining elements of the **ols** function results structure. We add these elements to the structure in the '.yhat, .resid, .sige,' etc., fields.

```
results.yhat = x*results.beta;
results.resid = y - results.yhat;
sigu = results.resid'*results.resid;
results.sige = sigu/(nobs-nvar);
tmp = (results.sige)*(diag(xpxi));
results.tstat = results.beta./(sqrt(tmp));
ym = y - mean(y);
rsqr1 = sigu; rsqr2 = ym'*ym;
results.rsqr = 1.0 - rsqr1/rsqr2; % r-squared
rsqr1 = rsqr1/(nobs-nvar);
rsqr2 = rsqr2/(nobs-1.0);
results.rbar = 1 - (rsqr1/rsqr2); % rbar-squared
ediff = results.resid(2:nobs) - results.resid(1:nobs-1);
results.dw = (ediff'*ediff)/sigu; % durbin-watson
```

2.3 Selecting a least-squares algorithm

In this section, we explore the speed and accuracy of the Cholesky and qr approaches to solving the least-squares problem. Using a program like that shown in example 2.2 we can examine execution times for the Cholesky and qr solution methods. Execution times are found using the MATLAB **tic** and **toc** functions. MATLAB begins timing execution when it encounters the **tic** command and stops timing when the **toc** command is encountered, printing out the execution time in the MATLAB command window.

```
% ----- Example 2.2 Least-squares timing information
n = 10000; k = 10;
e = randn(n,1); x = randn(n,k); b = ones(k,1);
y = x*b + e;
disp('time needed for Cholesky solution');
tic;
xpxi = (x'*x)\eye(k); % solve using the Cholesky decomposition
bhatc = xpxi*(x'*y);
toc;
disp('time needed for QR solution');
tic;
[q r] = qr(x,0);      % solve using the qr decomposition
xpxi = (r'*r)\eye(k);
bhatq = r\(q'*y);
toc;
```

Using this type of program, we explored the timing differences between the Cholesky and qr solution methods for different numbers of observations and explanatory variables. The timing results (presented in seconds of time required) from this exploration are shown in Table 2.1. These results were produced on a MacIntosh G3 computer that contains a 266 Mhz PowerPC 750 chip. From the third part of the table that shows the ratio of times for the qr versus Cholesky algorithms, we seen that for smaller problems (100 to 1000 observations and 10 or 50 explanatory variables) the qr routine takes 2 to 3.5 times as long as the Cholesky approach. As the number of observations increases to 10000 or the number of variables equals 100, the qr takes 1.5 to 2 times as long as the Cholesky method.

As indicated above, speed is not our only concern. Accuracy in the face of ill-conditioning is also an important consideration in designing a least-squares algorithm. A question that arises is — how sensitive are the computer solutions (the estimates) to small perturbations in the data contained in the X matrix? We would like to believe that small changes in the elements of the X matrix would not lead to large changes in the solution vector $\hat{\beta}$. If small changes in the data elements lead to large changes in the solution to an estimation problem, we say that the problem is *ill-conditioned*.

Table 2.1: Timing (in seconds) for Cholesky and QR Least-squares

		Cholesky	
nobs/nvar	10	50	100
100	0.0009	0.0128	0.0603
1000	0.0072	0.1112	0.6572
10000	0.2118	2.6344	8.8462
		QR	
nobs/nvar	10	50	100
100	0.0026	0.0250	0.0919
1000	0.0261	0.2628	1.1174
10000	0.4289	4.9914	17.0524
		QR/Cholesky	
nobs/nvars	10	50	100
100	2.8618	1.9535	1.5244
1000	3.5064	2.2806	1.7004
10000	2.0249	1.8947	1.9276

We can quantify the conditioning of a problem by calculating a *condition number*, which in the case of the least-squares problem is the ratio of the largest to the smallest eigenvalue of the data matrix X. The larger this ratio, the more ill-conditioned is the least-squares problem. Belsley, Kuh, and Welsch (1980, p. 114) use the condition number to state the following: If the data are known to d significant digits and the condition number is on the order of magnitude of 10^r, then a small change in the last place of the data can (but need not) affect the solution for $\hat{\beta}$ in the $(d-r)th$ place. A proof of this statement along with a more detailed discussion can be found in Belsley, Kuh, and Welsch (1980).

To see how this information is useful, consider an example where the explanatory variables data are trusted to 4 digits, that is we have no faith in our ability to measure these variables beyond 4 decimal digits. In this case, a shift in the fifth decimal place of the X matrix data would produce an *observationally equivalent* data set — one that cannot be distinguished from the original data set given our measurement limitations. It would be highly desirable that two observationally equivalent data sets produce the same least-squares estimates of the β parameters. Continuing this example, suppose the condition number of the X matrix is $1,000$ which can be written as 10^3. Then a shift in the fifth decimal place of the X matrix data *could* affect the least-squares parameter estimates of β in its $(5-3) = 2$nd digit.

The implication is that only the first digit is numerically accurate, meaning that an estimate of $\hat{\beta} = .83$ only informs us that the parameter is between .80 and .90!

These statements are based on some theoretical bounds calculated in Belsley, Kuh, and Welsch (1980), leading us to conclude that the results *could* be affected in the ways stated. The theoretical bounds are upper bounds on the potential problems, reflecting the worst that could happen. To examine the numerical accuracy of the Cholesky and qr approaches to solving the least-squares problem, we rely on a "benchmark" data set for which we know the true parameter values. We can then test the two regression algorithms to see how accurately they compute estimates of the true parameter values. Such a benchmark is shown in (2.1).

$$X = \begin{bmatrix} 1 & 1+\gamma & 1+\gamma & \dots & 1+\gamma \\ 1 & \gamma+\epsilon & \gamma & \dots & \gamma \\ 1 & \gamma & \gamma+\epsilon & \dots & \gamma \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \gamma & \gamma & \dots & \gamma+\epsilon \\ 1 & \gamma & \gamma & \dots & \gamma \end{bmatrix} \quad y = \begin{bmatrix} (n-1)+(n-2)\gamma+\epsilon \\ (n-2)\gamma+\epsilon \\ (n-2)\gamma+\epsilon \\ \vdots \\ (n-2)\gamma+\epsilon \\ (n-1)+(n-2)\gamma-\epsilon \end{bmatrix} \tag{2.1}$$

This is a modification of a benchmark originally proposed by Wampler (1980) set forth in Simon and LeSage (1988a, 1988b). The n by $(n-1)$ matrix X and the nx1 vector y in (2.1) represent the Wampler benchmark with the parameter γ added to the last $(n-2)$ columns of the X matrix, and with $(n-2)\gamma$ added to the y vector. When $\gamma = 0$ this benchmark is equivalent to the original Wampler benchmark. The modified benchmark shares the Wampler property that its solution is a column of ones for all values of $\epsilon > 0$, and for all values of γ, so the coefficient estimates are unity irrespective of ill-conditioning. This property makes it easy to judge how accurate the least-squares computational solutions for the estimates are. We simply need to compare the estimates to the true value of one.

The parameters γ and ϵ in (2.1) control the severity of two types of near-linear relationships in the data matrix X. The parameter ϵ controls the amount of collinearity between the last $(n-2)$ columns of the data matrix X. As the parameter ϵ is decreased towards zero, the last $(n-2)$ columns move closer to becoming perfect linear combinations with each other. The implication is that the parameter ϵ acts to control the amount of collinearity, or the severity of the near linear combinations among the last $(n-2)$ columns of the data matrix. As we make the value of ϵ smaller we produce an increasingly ill-conditioned least-squares problem.

The γ parameter serves to control a near linear relationship between the intercept column and the last $(n-2)$ columns of the data matrix. As the parameter γ is increased, the last $(n-2)$ columns of the matrix X become more collinear with the intercept column producing a near linear combination between the intercept term and the last $(n-2)$ columns. This type of collinear relationship reflects a situation where each independent variable becomes more nearly constant, exhibiting a near linear relationship with the constant term vector.

Using this benchmark data set, we can examine the numerical accuracy of regression algorithms while continuously varying the amount of ill-conditioning. In addition, we can examine the impact of two different types of collinear relationships that one might encounter in practice. The first type of collinearity — that controlled by the ϵ parameter — involves the last $(n-2)$ columns of the data matrix representing the explanatory variables in our least-squares model. The second type of collinearity — controlled by the γ parameter — represents cases where the intercept term is involved in a near linear relationship with the entire set of explanatory variables, the last $(n-2)$ columns.

We will measure the numerical accuracy of the coefficient estimates produced by the Cholesky and qr regression algorithms using a formula from Wampler (1980). This formula, can be thought of as representing the number of digits of accuracy and is shown in (2.2).

$$\mathrm{acc}_j = -\log_{10}(|1.0 - \hat{\beta}_j|) \tag{2.2}$$

In (2.2) acc_j represents the accuracy of the estimate $\hat{\beta}_j$ in estimating the true value which is known to be 1.0, $\log_{10}$ represents the base 10 logarithm, and the symbol $|$ denotes that we are taking the absolute value of the quantity $1.0 - \hat{\beta}_j$. As an example, consider a case where $\hat{\beta}_j = 1.001$, according to (2.2) we have three digits of accuracy.

Table 2.2 presents the digits accuracy results from the Cholesky and qr algorithms over a range of values for the control parameters γ and ϵ. Values of $n = 10, 25$ and 50 were used to produce numerous estimates. The symbol '**' is used in the table to designate negative digits of accuracy or inability to solve the least-squares problem due to ill-conditioning. Accuracy for the intercept term estimate is presented along with results for the 'most' and 'least' accurate slope estimates.

From Table 2.2 we conclude that the qr algorithm was capable of solving the least-squares problem for all values of γ and ϵ, producing at least 6

Table 2.2: Digits of accuracy for Cholesky vs. QR decomposition

	Least accurate slope Cholesky			
$\gamma\backslash\epsilon$	.01	.001	.0001	.00005
10	7.34	5.95	3.87	3.36
100	5.54	3.69	1.59	0.77
1000	3.56	1.48	**	**
5000	2.39	0.56	**	**
	Least accurate slope qr			
$\gamma\backslash\epsilon$	.01	.001	.0001	.00005
10	11.61	9.70	8.04	7.17
100	11.09	9.73	7.58	6.87
1000	10.11	8.74	7.61	6.95
5000	9.35	8.13	7.04	7.05
	Most accuracy slope Cholesky			
$\gamma\backslash\epsilon$	.01	.001	.0001	.00005
10	8.29	7.85	6.38	4.46
100	6.44	5.20	2.91	2.51
1000	4.68	2.76	**	0.17
5000	3.48	2.71	**	**
	Most accurate slope qr			
$\gamma\backslash\epsilon$	.01	.001	.0001	.00005
10	12.70	11.12	8.92	8.02
100	12.19	10.89	8.50	7.79
1000	10.95	10.56	8.25	8.60
5000	10.16	8.71	7.94	8.68
	Cholesky intercept accuracy			
$\gamma\backslash\epsilon$	.01	.001	.0001	.00005
10	6.43	4.90	2.54	2.53
100	3.85	2.22	0.44	**
1000	0.72	**	**	**
5000	**	**	**	**
	qr intercept accuracy			
$\gamma\backslash\epsilon$	.01	.001	.0001	.00005
10	12.17	12.39	11.91	12.33
100	10.67	10.06	9.86	10.42
1000	8.63	8.29	8.72	8.16
5000	6.95	7.93	6.66	7.28

decimal digits of accuracy. In contrast, the Cholesky produced negative decimal digits of accuracy (or was incapable of solving the least-squares problem) in 4 of the 16 least accurate slope estimate cases, 3 of 16 cases for the most accurate slope estimate, and 8 of the 16 cases for the intercept estimates. The conclusion is that if we care about numerical accuracy in the face of ill-conditioned data sets, we should rely on the qr algorithm to solve the least-squares problem. With the advent of faster computers, the timing differences do not appear as dramatic as the numerical accuracy differences.

2.4 Using the results structure

To illustrate the use of the 'results' structure returned by our **ols** function, consider the associated function **plt_reg** which plots actual versus predicted values along with the residuals. The results structure contains everything needed by the **plt_reg** function to carry out its task. Earlier, we referred to functions **plt** and **prt** rather than **plt_reg**, but we will see in Chapter 3 that **prt** and **plt** are "wrapper" functions that call the functions **prt_reg** and **plt_reg** where the real work of printing and plotting regression results is carried out. The motivation for taking this approach is that separate smaller functions can be devised to print and plot results from all of the econometric procedures in the toolbox facilitating development. The wrapper functions eliminate the need for the user to learn the names of different printing and plotting functions associated with each group of econometric procedures, all results structures can be printed and plotted by simply invoking the **prt** and **plt** functions. The *Econometrics Toolbox* contains a host of printing and plotting functions for producing formatted output from vector autoregressions, cointegration tests, Gibbs sampling estimation, simultaneous equations estimators, etc. All of these functions can be accessed using the wrapper functions **prt** and **plt**, or by directly calling the individual function names such as **plt_reg**.

The **plt_reg** function is shown below. A check that the user supplied a regression results structure can be carried out using the MATLAB **isstruct** function that is true if the argument represents a structure. After this error check, we rely on a MATLAB programming construct called the 'switch-case' to provide the remaining error checking for the function.

```
function plt_reg(results);
% PURPOSE: plots regression actual vs predicted and residuals
%---------------------------------------------------
% USAGE: plt_reg(results);
% where: results is a structure returned by a regression function
```

```
%---------------------------------------------------
% RETURNS: nothing, just plots regression results
% --------------------------------------------------
% NOTE: user must supply pause commands, none are in plt_reg function
%       e.g. plt_reg(results);
%            pause;
%            plt_reg(results2);
% --------------------------------------------------
% SEE ALSO: prt_reg(results), prt, plt
%---------------------------------------------------

if ~isstruct(results), error('plt_reg requires a structure'); end;
nobs = results.nobs; tt=1:nobs; clf;

switch results.meth
case {'arma','boxcox','boxcox2','logit','ols','olsc','probit','ridge', ...
   'theil','tobit','hwhite','tsls','nwest'}
subplot(211), plot(tt,results.y,'-',tt,results.yhat,'--');
title([upper(results.meth), '   Actual vs. Predicted']);
subplot(212), plot(tt,results.resid);  title('Residuals');

case {'robust','olst','lad'}
subplot(311), plot(tt,results.y,'-',tt,results.yhat,'--');
title([upper(results.meth), '   Actual vs. Predicted']);
subplot(312), plot(tt,results.resid);  title('Residuals');
subplot(313), plot(tt,results.weight); title('Estimated weights');
otherwise
error('method not recognized by plt_reg');
end;
subplot(111);
```

The ‘switch’ statement examines the ‘meth’ field of the results structure passed to the **plt_reg** function as an argument and executes the plotting commands if the ‘meth’ field is one of the regression methods implemented in our function library. In the event that the user passed a result structure from a function other than one of our regression functions, the ‘otherwise’ statement is executed which prints an error message.

The switch statement also helps us to distinguish special cases of **robust, olst, lad** regressions where the estimated weights are plotted along with the actual versus predicted and residuals. These weights allow the user to detect the presence of outliers in the regression relationship. A similar approach could be used to extend the **plt_reg** function to accommodate other special regression functions where additional or specialized plots are desired.

A decision was made not to place the ‘pause’ command in the **plt_reg** function, but rather let the user place this statement in the calling program or function. An implication of this is that the user controls viewing regression plots in 'for loops' or in the case of multiple invocations of the **plt_reg** function. For example, only the second 'plot' will be shown in the following code.

```
result1 = ols(y,x1);
plt_reg(result1);
result2 = ols(y,x2);
plt_reg(result2);
```

If the user wishes to see the regression plots associated with the first regression, the code would need to be modified as follows:

```
result1 = ols(y,x1);
plt_reg(result1);
pause;
result2 = ols(y,x2);
plt_reg(result2);
```

The 'pause' statement would force a plot of the results from the first regression and wait for the user to strike any key before proceeding with the second regression and accompanying plot of these results.

Our **plt_reg** function would work with new regression functions that we add to the library provided that the regression returns a structure containing the fields '.y', '.yhat', '.resid', '.nobs' and '.meth'. We need simply add this method to the switch-case statement.

A more detailed example of using the results structure is the **prt_reg** function from the regression library. This function provides a printout of regression results similar to those provided by many statistical packages. The function relies on the 'meth' field to determine what type of regression results are being printed, and uses the 'switch-case' statement to implement specialized methods for different types of regressions.

A small fragment of the **prt_reg** function showing the specialized printing for the **ols** and **ridge** regression methods is presented below:

```
function prt_reg(results,vnames,fid)
% PURPOSE: Prints output using regression results structures
%---------------------------------------------------
% USAGE: prt_reg(results,vnames,fid)
% Where: results = a structure returned by a regression
%        vnames  = an optional vector of variable names
%        fid     = optional file-id for printing results to a file
%                  (defaults to the MATLAB command window)
%---------------------------------------------------
%  NOTES: e.g. vnames = strvcat('y','const','x1','x2');
```

```
%         e.g. fid = fopen('ols.out','wr');
%  use prt_reg(results,[],fid) to print to a file with no vnames
% --------------------------------------------------
%  RETURNS: nothing, just prints the regression results
% --------------------------------------------------
% SEE ALSO: prt, plt
%---------------------------------------------------

if ~isstruct(results) % error checking on inputs
 error('prt_reg requires structure argument');
elseif nargin == 1, nflag = 0; fid = 1;
elseif nargin == 2, fid = 1;   nflag = 1;
elseif nargin == 3, nflag = 0;
 [vsize junk] = size(vnames); % user may supply a blank argument
   if vsize > 0, nflag = 1; end;
else, error('Wrong # of arguments to prt_reg'); end;
nobs = results.nobs; nvar = results.nvar;
%  make up generic variable names
Vname = 'Variable';
 for i=1:nvar;
  tmp = ['variable ',num2str(i)]; Vname = strvcat(Vname,tmp);
 end;
if (nflag == 1) % the user supplied variable names
[tst_n nsize] = size(vnames);
 if tst_n ~= nvar+1
  warning('Wrong # of variable names in prt_reg -- check vnames argument');
  fprintf(fid,'will use generic variable names \n');
  nflag = 0;
 else,
 Vname = 'Variable';
 for i=1:nvar; Vname = strvcat(Vname,vnames(i+1,:)); end;
end; % end of nflag issue
switch results.meth
case {'ols','hwhite','nwest'} % <=== ols,white,nwest regressions
  if strcmp(results.meth,'ols')
   fprintf(fid,'Ordinary Least-squares Estimates \n');
  elseif strcmp(results.meth,'hwhite')
   fprintf(fid,'White Heteroscedastic Consistent Estimates \n');
  elseif strcmp(results.meth,'nwest')
   fprintf(fid,'Newey-West hetero/serial Consistent Estimates \n');
  end;
  if (nflag == 1)
   fprintf(fid,'Dependent Variable = %16s \n',vnames(1,:));
  end;
  fprintf(fid,'R-squared      = %9.4f \n',results.rsqr);
  fprintf(fid,'Rbar-squared   = %9.4f \n',results.rbar);
  fprintf(fid,'sigma^2        = %9.4f \n',results.sige);
  fprintf(fid,'Durbin-Watson  = %9.4f \n',results.dw);
  fprintf(fid,'Nobs, Nvars    = %6d,%6d \n',results.nobs,results.nvar);
```

```
  fprintf(fid,'*****************************************************\n');
% <================== end of ols,white, newey-west case
case {'ridge'} % <================== ridge regressions
  fprintf(fid,'Ridge Regression Estimates \n');
  if (nflag == 1)
   fprintf(fid,'Dependent Variable = %16s \n',vnames(1,:));
  end;
  fprintf(fid,'R-squared      = %9.4f \n',results.rsqr);
  fprintf(fid,'Rbar-squared   = %9.4f \n',results.rbar);
  fprintf(fid,'sigma^2        = %9.4f \n',results.sige);
  fprintf(fid,'Durbin-Watson  = %9.4f \n',results.dw);
  fprintf(fid,'Ridge theta    = %16.8g \n',results.theta);
  fprintf(fid,'Nobs, Nvars    = %6d,%6d \n',results.nobs,results.nvar);
  fprintf(fid,'*****************************************************\n');
% <================== end of ridge regression case
otherwise
  error('method unknown to the prt_reg function');
end;
tout = tdis_prb(results.tstat,nobs-nvar); % find t-stat probabilities
tmp = [results.beta results.tstat tout];  % matrix to be printed
% column labels for printing results
bstring = 'Coefficient'; tstring = 't-statistic'; pstring = 't-probability';
cnames = strvcat(bstring,tstring,pstring);
in.cnames = cnames; in.rnames = Vname; in.fmt = '%16.6f'; in.fid = fid;
mprint(tmp,in); % print estimates, t-statistics and probabilities
```

The function **mprint** is a utility function to produce formatted printing of a matrix with column and row-labels and is discussed in detail in Chapter 3. All printing of matrix results for the *Econometric Toolbox* functions is done using the **mprint** function.

The **prt_reg** function allows the user an option of providing a vector of fixed width variable name strings that will be used when printing the regression coefficients. These can be created using the MATLAB **strvcat** function that produces a vertical concatenated list of strings with fixed width equal to the longest string in the list. We give the user a break here, if the wrong number of names is supplied, a warning is issued to the user but results are still printed using generic variable names. I found this avoids some annoying situations where generic names and printed results are preferable to an error message with no printed output.

We can also print results to an indicated file rather than the MATLAB command window. Three alternative invocations of the **prt_reg** function illustrating these options for usage are shown below:

```
vnames = strvcat('y variable','constant','population','income');
res = ols(y,x);
```

```
prt_reg(res);               % print with generic variable names
prt_reg(res,vnames);        % print with user-supplied variable names
fid = fopen('ols.out','wr'); % open a file for printing
prt_reg(res,vnames,fid);    % print results to file 'ols.out'
```

The first use of **prt_reg** produces a printout of results to the MATLAB command window that uses 'generic' variable names:

```
Ordinary Least-squares Estimates
R-squared      =    0.8525
Rbar-squared   =    0.8494
sigma^2        =    0.6466
Durbin-Watson  =    1.8791
Nobs, Nvars    =    100,     3
***************************************************************
Variable        Coefficient      t-statistic    t-probability
variable 1         1.208077        16.142388         0.000000
variable 2         0.979668        11.313323         0.000000
variable 3         1.041908        13.176289         0.000000
```

The second use of **prt_reg** uses the user-supplied variable names. The MATLAB function **strvcat** carries out a vertical concatenation of strings and pads the shorter strings in the 'vnames' vector to have a fixed width based on the longer strings. A fixed width string containing the variable names is required by the **prt_reg** function. Note that we could have used:

```
vnames = ['y variable',
          'constant  ',
          'population',
          'income    '];
```

but, this takes up more space and is slightly less convenient as we have to provide the padding of strings ourselves. Using the 'vnames' input in the **prt_reg** function would result in the following printed to the MATLAB command window.

```
Ordinary Least-squares Estimates
Dependent Variable =      y variable
R-squared      =    0.8525
Rbar-squared   =    0.8494
sigma^2        =    0.6466
Durbin-Watson  =    1.8791
Nobs, Nvars    =    100,     3
***************************************************************
Variable        Coefficient      t-statistic    t-probability
constant           1.208077        16.142388         0.000000
population         0.979668        11.313323         0.000000
income             1.041908        13.176289         0.000000
```

The third case specifies an output file opened with the command:

```
fid = fopen('ols.out','wr');
```

The file 'ols.out' would contain output identical to that from the second use of **prt_reg**. It is the user's responsibility to close the file that was opened using the MATLAB command:

```
fclose(fid);
```

Next, we turn to details concerning implementation of the **prt_reg** function. The initial code does error checking on the number of input arguments, determines if the user has supplied a structure argument, and checks for variable names and/or an output file id. We allow the user to provide a file id argument with no variable names using the call: **prt_reg(result,[],fid)** where a blank argument is supplied for the variable names. We check for this case by examining the size of the vnames input argument under the case of nargin == 3 in the code shown below.

```
if ~isstruct(results) % error checking on inputs
 error('prt_reg requires structure argument');
elseif nargin == 1, nflag = 0; fid = 1;
elseif nargin == 2, fid = 1;   nflag = 1;
elseif nargin == 3, nflag = 0;
 [vsize junk] = size(vnames); % user may supply a blank argument
   if vsize > 0, nflag = 1; end;
else
 error('Wrong # of arguments to prt_reg');
end;
```

Variable names are constructed if the user did not supply a vector of variable names and placed in a MATLAB fixed-width string-array named 'Vname', with the first name in the array being the row-label heading 'Variable' which is used by the function **mprint**. For the case where the user supplied variable names, we simply transfer these to a MATLAB 'string-array' named 'Vname', again with the first element 'Variable' that will be used by **mprint**. We do error checking on the number of variable names supplied which should equal the number of explanatory variables plus the dependent variable (nvar+1). In the event that the user supplies the wrong number of variable names, we issue a warning and print output results using the generic variable names.

```
nobs = results.nobs; nvar = results.nvar;
%  make up generic variable names
```

```
Vname = 'Variable';
 for i=1:nvar;
  tmp = ['variable ',num2str(i)]; Vname = strvcat(Vname,tmp);
 end;
if (nflag == 1) % the user supplied variable names
[tst_n nsize] = size(vnames);
 if tst_n ~= nvar+1
  warning('Wrong # of variable names in prt_reg -- check vnames argument');
  fprintf(fid,'will use generic variable names \n');
  nflag = 0;
 else,
 Vname = 'Variable';
 for i=1:nvar; Vname = strvcat(Vname,vnames(i+1,:)); end;
end; % end of nflag issue
```

After constructing or transferring variable names, the 'switch-case' takes over sending our function to the appropriate customized segment of code for printing part of the regression results depending on the 'meth' field of the results structure.

In the case of 'ols', 'hwhite' or 'nwest' (ordinary least-squares, White's heteroscedastic consistent estimator or Newey and West's heteroscedastic-serial correlation consistent estimator), we rely on a great deal of common code. The title for the regression results printout will differ depending on which of these methods was used to produce the results structure, with everything else identical.

```
switch results.meth
case {'ols','hwhite','nwest'} % <====== ols,white,nwest regressions
 if strcmp(results.meth,'ols')
  fprintf(fid,'Ordinary Least-squares Estimates \n');
 elseif strcmp(results.meth,'hwhite')
  fprintf(fid,'White Heteroscedastic Consistent Estimates \n');
 elseif strcmp(results.meth,'nwest')
  fprintf(fid,'Newey-West hetero/serial Consistent Estimates \n');
 end;
 if (nflag == 1)
  fprintf(fid,'Dependent Variable = %16s \n',vnames(1,:));
 end;
 fprintf(fid,'R-squared      = %9.4f \n',results.rsqr);
 fprintf(fid,'Rbar-squared   = %9.4f \n',results.rbar);
 fprintf(fid,'sigma^2        = %9.4f \n',results.sige);
 fprintf(fid,'Durbin-Watson  = %9.4f \n',results.dw);
 fprintf(fid,'Nobs, Nvars    = %6d,%6d \n',results.nobs,results.nvar);
 fprintf(fid,'*******************************************************\n');
% <================== end of ols,white, newey-west case
```

A point to note is that use of the MATLAB 'fprintf' command with an input argument 'fid' makes it easy to handle both the case where the user wishes output printed to the MATLAB command window or to an output file. The 'fid' argument takes on a value of '1' to print to the command window and a user-supplied file name value for output printed to a file.

Finally, after printing the specialized output, the coefficient estimates, t-statistics and marginal probabilities that are in common to all regressions are printed. The marginal probabilities are calculated using a function **tdis_prb** from the *distributions library* discussed in Chapter 9. This function returns the marginal probabilities given a vector of $t-$distributed random variates along with a degrees of freedom parameter. The code to print coefficient estimates, t-statistics and marginal probabilities is common to all regression printing procedures, so it makes sense to move it to the end of the 'switch-case' code and execute it once as shown below. We rely on the function **mprint** discussed in Chapter 3 to do the actual printing of the matrix of regression results with row and column labels specified as fields of a structure variable 'in'. Use of structure variables with fields as input arguments to functions is a convenient way to pass a large number of optional arguments to MATLAB functions, a subject taken up in Chapter 3.

```
tout = tdis_prb(results.tstat,nobs-nvar); % find t-stat probabilities
tmp = [results.beta results.tstat tout];  % matrix to be printed
% column labels for printing results
bstring = 'Coefficient'; tstring = 't-statistic'; pstring = 't-probability';
cnames = strvcat(bstring,tstring,pstring);
in.cnames = cnames; in.rnames = Vname; in.fmt = '%16.6f'; in.fid = fid;
mprint(tmp,in); % print estimates, t-statistics and probabilities
```

The case of a ridge regression illustrates the need for customized code to print results for different types of regressions. This regression produces a ridge parameter estimate based on a suggested formula from Hoerl and Kennard (1970), or allows for a user-supplied value. In either case, the regression output should display this important parameter that was used to produce the coefficient estimates.

```
case {'ridge'} % <=================== ridge regressions
fprintf(fid,'Ridge Regression Estimates \n');
if (nflag == 1)
 fprintf(fid,'Dependent Variable = %16s \n',vnames(1,:));
end;
fprintf(fid,'R-squared      = %9.4f \n',results.rsqr);
fprintf(fid,'Rbar-squared   = %9.4f \n',results.rbar);
fprintf(fid,'sigma^2        = %9.4f \n',results.sige);
fprintf(fid,'Durbin-Watson  = %9.4f \n',results.dw);
fprintf(fid,'Ridge theta    = %16.8g \n',results.theta);
```

```
fprintf(fid,'Nobs, Nvars    = %6d,%6d \n',results.nobs,results.nvar);
fprintf(fid,'*******************************************************\n');
% <=================== end of ridge case
```

A design consideration here is whether to attempt a conditional statement that would print the ridge parameter based on an 'if' statement that checks for the ridge method inside the code for the **ols, white, nwest** case. Taking this approach, we could incorporate the same code used for printing the **ols, white, nwest** functions with the addition of a single additional statement to print the ridge parameter when we detect a **ridge** regression in the 'meth' field. The **prt_reg** function was designed to allow easy modification for printing results from new methods that are added to the regression library. There is a trade-off between repeating code and complicating the function so it becomes difficult to modify in the future.

An example of a more convincing case for writing separate code for different regression procedures is the Theil-Goldberger regression. Here we want to print prior means and standard deviations specified as input arguments by the user in the output information. This specialized information can be printed by the code segment handling the '**theil** method as shown below:

```
case {'theil'} % <=================== theil-goldberger regressions
 fprintf(fid,'\n');
 fprintf(fid,'Theil-Goldberger Regression Estimates \n');
 if (nflag == 1)
  fprintf(fid,'Dependent Variable = %16s \n',vnames(1,:));
 end;
 fprintf(fid,'R-squared      = %9.4f \n',results.rsqr);
 fprintf(fid,'Rbar-squared   = %9.4f \n',results.rbar);
 fprintf(fid,'sigma^2        = %9.4f \n',results.sige);
 fprintf(fid,'Durbin-Watson  = %9.4f \n',results.dw);
 fprintf(fid,'Nobs, Nvars    = %6d,%6d \n',results.nobs,results.nvar);
 fprintf(fid,'*******************************************************\n');
vstring = 'Variable'; bstring = 'Prior Mean'; tstring = 'Std Deviation';
tmp = [results.pmean results.pstd];
cnames = strvcat(bstring,tstring);
pin.cnames = cnames;
pin.rnames = Vname;
pin.fmt = strvcat('%16.6f','%16.6f');
pin.fid = fid;
mprint(tmp,pin);
fprintf(fid,'*******************************************************\n');
fprintf(fid,'      Posterior Estimates             \n');
```

As in the case of all other regression methods, the common code at the end of the **prt_reg** function would print out the posterior estimates, t-statistics and probabilities that should be in the '.beta, .tstat' fields of the structure returned by the **theil** function.

As an example of adding a code segment to handle a new regression method, consider how we would alter the **prt_reg** function to add a Box-Jenkins ARMA method. First, we need to add a 'case' based on the Box-Jenkins 'meth' field which is 'arma'. The specialized code for the 'arma' method handles variable names in a way specific to a Box-Jenkins arma model. It also presents output information regarding the number of AR and MA parameters in the model, log-likelihood function value and number of iterations required to find a solution to the nonlinear optimization problem used to find the estimates, as shown below.

```
case {'arma'} % <=============== box-jenkins regressions
 p = length(results.ar);
 q = length(results.ma);
 fprintf(1,'\n');
 fprintf(1,'Box-Jenkins ARMA Estimates \n');
 fprintf(1,'R-squared      = %9.4f \n',results.rsqr);
 fprintf(1,'Rbar-squared   = %9.4f \n',results.rbar);
 fprintf(1,'sigma^2        = %9.4f \n',results.sige);
 fprintf(1,'log-likelihood = %16.8g \n',results.like);
 fprintf(1,'Nobs           = %6d \n',results.nobs);
 fprintf(1,'AR,MA orders   = %6d,%6d \n',p,q);
 fprintf(1,'Iterations     = %6d \n',results.iter);
 fprintf(fid,'*******************************************************\n');
 Vname = 'Variable';
 % create special variable names for box-jenkins
if results.cterm == 0;
 for i=1:nvar
  if i <= p, tmp = str2mat(['AR ',num2str(i)]);
  Vname = strvcat(Vname,tmp);
  elseif i <= p+q, tmp = str2mat(['MA ',num2str(i-p)]);
  Vname = strvcat(Vname,tmp);
  end;
 end;
else
 for i=1:nvar
if i <= p, tmp = str2mat(['AR ',num2str(i)]);
Vname = strvcat(Vname,tmp);
    elseif i == p+1, Vname = strvcat(Vname,'Const');
    else, tmp = str2mat(['MA ',num2str(i-p-1)]);
Vname = strvcat(Vname,tmp);
    end;
 end;
end;
% <=================== end of boxjenkins case
```

Provided that we placed the regression estimates and t-statistics from the **arma** regression routine into structure fields 'results.beta' and 'results.tstat', the common code (that already exists) for printing regression results would work with this new function.

2.5 Performance profiling the regression toolbox

This section demonstrates how to use the MATLAB performance profiling function **profile** to examine regression functions and enhance their speed. The **profile** command is provided with a 'function' argument which we then execute. After executing the function, calling the **profile** function again with the argument 'report' produces a printed summary indicating how much time the function spent on each line of code . Example 2.3 illustrates profiling the **ols** function.

```
% ----- Example 2.3 Profiling the ols() function
nobs = 1000; nvar = 15;    beta = ones(nvar,1);
xmat = randn(nobs,nvar-1); x = [ones(nobs,1) xmat];
evec = randn(nobs,1);      y = x*beta + evec;
% profile the ols function
profile ols;      result = ols(y,x);  profile report;
% profile the prt_reg function
profile prt_reg;  prt_reg(result);    profile report;
```

This produced the following profile output for the **ols** function:

```
Total time in "ols.m": 0.1 seconds
100% of the total time was spent on lines:
[54 44 47]
             43:
0.04s, 40%   44: [q r] = qr(x,0);
             45: xpxi = (r'*r)\eye(nvar);
             46:
0.01s, 10%   47: results.beta = r\(q'*y);
             48: results.yhat = x*results.beta;
             53: results.tstat = results.beta./(sqrt(tmp));
0.05s, 50%   54: ym = y - mean(y);
             55: rsqr1 = sigu;
```

The total time spent to carry out the regression involving 1000 observations and 15 explanatory variables was 0.1 seconds. Three lines of code accounted for 100% of the time and these are listed in order as: [54 44 47]. Line #54 accounted for 50% of the total time, whereas the qr decomposition on line #44 only accounted for 40% of the time. Line #54 computes the mean of the y-vector used to determine the R-squared statistic. The third slowest computation involved line #47 where the backsolution for the coefficients took place, requiring 10% of the total time.

These results shed additional light on the speed differences between the Cholesky and qr decomposition methods discussed earlier in Section 2.3. Using either the Cholesky or qr method, we would still be spending the majority of time computing the mean of the y-vector and backsolving for the coefficients, and these computations account for 60% of the total time required by the **ols** function. Saving a few hundredths of a second using the Cholesky in place of the qr decomposition would not noticeably improve the performance of the **ols** function from the user's viewpoint.

The profile report on the **prt_reg** function was as follows:

```
Total time in "prt_reg.m": 0.47 seconds
100% of the total time was spent on lines:
[367 354 36 364 141 140 139 137 135 125]
0.03s,  6%   36: if ~isstruct(results)
             37:  error('prt_reg requires structure argument');
0.01s,  2%  125: fprintf(fid,'\n');
            126: if strcmp(results.meth,'ols')
            134: if (nflag == 1)
0.01s,  2%  135: fprintf(fid,'Dependent Variable = %16s \n',vnames(1,:));
            136: end;
0.01s,  2%  137: fprintf(fid,'R-squared      = %9.4f \n',results.rsqr);
            138: fprintf(fid,'Rbar-squared   = %9.4f \n',results.rbar);
0.01s,  2%  139: fprintf(fid,'sigma^2        = %9.4f \n',results.sige);
0.01s,  2%  140: fprintf(fid,'Durbin-Watson  = %9.4f \n',results.dw);
0.01s,  2%  141: fprintf(fid,'Nobs, Nvars    = %6d,%6d \n',results.nob
            142: fprintf(fid,'***************************************\n');
            353: tstat = results.tstat;
0.15s, 32%  354: tout = tdis_prb(tstat,nobs-nvar);
0.02s,  4%  364: fprintf(fid,'%16s %16s %16s %16s \n',vstring,bstring,tst
            366: for i=1:nvar;
0.21s, 45%  367: fprintf(fid,'%16s %16.6f %16.6f %16.6f\n',Vname{i},tmp(i,
            368: end;
```

Here we see that printing the results took 0.47 seconds, almost five times the 0.10 seconds needed to compute the regression results. It is not surprising that computation of the marginal probabilities for the t-statistics on line #354 took 32% of the total time. These computations require use of the incomplete beta function which in turn draws on the log gamma function, both of which are computationally intensive routines. Most of the time (45%) was spent actually printing the output to the MATLAB command window which is done in the 'for-loop' at line #367. (Note that we replaced the call to the

mprint function with the 'for loop' and explicit fprintf statements to make it clear that printing activity actually takes time.)

One conclusion we should draw from these profiling results is that the design decision to place computation of the marginal probabilities for the t-statistics in the **prt_reg** function instead of in the **ols** function makes sense. Users who wish to carry out Monte Carlo experiments involving a large number of least-squares regressions and save the coefficient estimates would be hampered by the slow-down associated with evaluating the marginal probabilities in the **ols** function.

A second conclusion is that if we are interested in least-squares estimates for β alone (as in the case of a two-stage least-squares estimation procedure), we might implement a separate function named **olsb**. Computing $\hat{\beta}$ coefficients using a specialized function would save time by avoiding computation of information such as the $R-$squared statistic, that is not needed.

The comments concerning speed differences between Cholesky and qr solutions to the least-squares problem are amplified by these profiling results. It would take the same amount of time to print results from either solution method, and as indicated, the time needed to print the results is five times that required to compute the results!

We could delve into the time-profile for the **tdis_prb** function which is part of the *distributions library* discussed in Chapter 9. This turns out to be un-enlightening as the routine spends 100% of its time in the MATLAB incomplete beta function (**betainc**) which we cannot enhance.

A final point regarding use of the MATLAB **profile** command is that you need to position MATLAB in the directory that contains the source file for the function you are attempting to profile. For example, to profile the **tdis_prb** function which is in the 'distrib' directory, we need to move to this directory before executing the **profile tdis_prb** and **profile report** commands.

2.6 Using the regression library

This section presents examples showing how some of the various regression functions in the library might be used to solve econometric estimation problems. It seems a good idea when creating a MATLAB function to provide an example program that illustrates use of the function. This has been done for most of the functions in the *Econometrics Toolbox*. Demonstration programs have an 'underscore d' attached to the function name. The file 'ols_d.m' would contain a demonstration program for the **ols** function, and the file 'ridge_d.m' a demo of the **ridge** function.

Basic use of the regression functions follows a canned format involving:

1. read in the sample data
2. perform any transformations or calculations necessary to form the set of explanatory variables and the dependent variable.
3. send the dependent and independent variables to the regression function for processing.
4. send the structure returned by the regression function to the **prt** or **plt** function to print or plot results.

For specific examples of this canned format you can examine the demonstration files in the *regression function library*.

In this section, we wish to go beyond simple demonstrations of the various estimation procedures to illustrate how the results structures can be useful in computing various econometric statistics and performing hypothesis tests based on regression results. This section contains sub-sections that illustrate various uses of the *regression function library* and ways to produce new functions that extend the library. Later chapters illustrate additional regression functions not discussed here.

2.6.1 A Monte Carlo experiment

As an initial demonstration of the **ols** function, consider a Monte Carlo experiment where we generate 100 different data sets based on the same explanatory variable matrix X and 100 different disturbance vectors ε that are used to produce 100 sample y vectors.

We wish to carry out 100 regressions and save the estimates $\hat{\beta}$ from each regression. We will use the 100 sets of estimates to compute a mean over the 100 trials, that should be close to the true values of β used to generate the 100 different sample y vectors. This experiment illustrates to introductory econometrics students the nature of the unbiasedness property associated with the least-squares estimates. Although the means over 100 different samples are close to the true values for β, the amount of dispersion in individual sample outcomes is usually a surprise to students.

```
% ----- Example 2.4 Using the ols() function for Monte Carlo
nobs = 100; nvar = 5; ntrials = 100;
b = ones(nvar,1);                          % true betas = 1
```

```
x = [ones(nobs,1) randn(nobs,nvar-1);     % fixed x-matrix
bout = zeros(ntrials,nvar);               % storage for estimates
for i=1:ntrials;                          % do ols in a for-loop
evec = randn(nobs,1); y = x*beta + evec;
out = ols(y,x);    bout(i,:) = out.beta'; % save bhat's
end;
bm = mean(bout); bs = std(bout);          % find mean and std of bhats
fprintf(1,'Mean of the bhats \n');
for i=1:nvar; fprintf(1,'%8.4f \n',bmean(1,i)); end;
fprintf(1,'Std deviation of the bhats \n');
for i=1:nvar; fprintf(1,'%8.4f \n',bstd(1,i));  end;
% provide a histogram for each bhat
hist(bout); ylabel('frequency of \beta outcomes');
xlabel('Estimated \beta values');
legend('\beta_1','\beta_2','\beta_3','\beta_4','\beta_5');
```

We recover the estimates $\hat{\beta}$ from the 'results' structure each time through the loop, transpose and place them in the 'ith' row of the matrix 'bsave'. After the loop completes, we compute the mean and standard deviations of the estimates and print these out for each of the 5 coefficients. MATLAB **mean** and **std** functions work on the columns of matrices, motivating our storage scheme for the 'bsave' matrix.

To provide a graphical depiction of our results, we use the MATLAB **hist** function to produce a histogram of the distribution of estimates. The **hist** function works on each column when given a matrix input argument, producing 5 separate histograms (one for each column) that are color-coded. We used the LaTeX notation, 'backslash beta' in the 'ylabel', 'xlabel' and 'legend' commands to produce a Greek symbol, β in the y and x labels and the LaTeX underscore to create the subscripted $\beta_i, i = 1, \ldots, 5$ symbols in the legend. Figure 2.1 shows the resulting 5-way histogram.

It is clear from Figure 2.1 that although the estimates are centered on the true value of unity, the distribution extends down to 0.75 and up to 1.25. The implication is that particular data samples may produce estimates far from truth despite the use of an unbiased estimation procedure.

2.6.2 Dealing with serial correlation

This discussion illustrates how one would construct Cochrane-Orcutt and maximum likelihood estimates in the face of first-order serial correlation in the disturbances using MATLAB functions and the regression library.

In example 2.5, we generate a regression model with serially correlated disturbances as in (2.3).

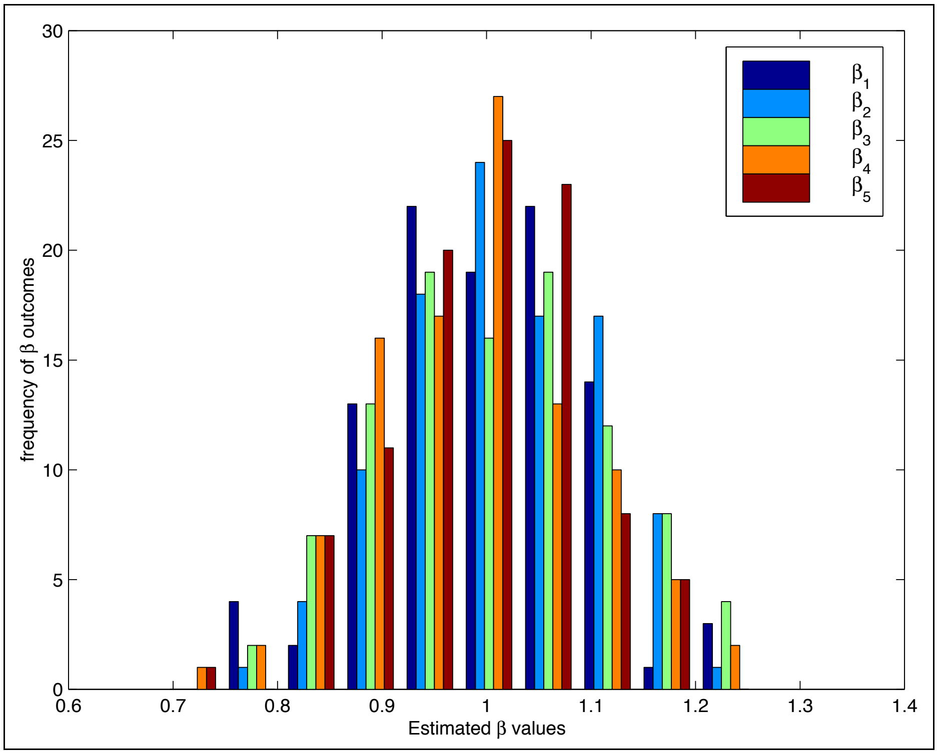

Figure 2.1: Histogram of $\hat{\beta}$ outcomes

$$\begin{aligned} y_t &= X_t\beta + u_t \\ u_t &= \rho u_{t-1} + \varepsilon_t \end{aligned} \quad (2.3)$$

Example 2.5 carries out least-squares estimation, prints and plots the results for the generated data set.

```
% ----- Example 2.5  Generate a model with serial correlation
n = 200; k = 3; evec = randn(n,1);
xmat = [ones(n,1) randn(n,k)]; y = zeros(n,1); u = zeros(n,1);
beta = ones(k,1); beta(1,1) = 10.0; % constant term
for i=2:n; % generate a model with 1st order serial correlation
u(i,1) = 0.4*u(i-1,1) + evec(i,1);
y(i,1) = xmat(i,:)*beta + u(i,1);
end;
% truncate 1st 100 observations for startup
```

```
yt = y(101:n,1); xt = xmat(101:n,:);
n = n-100; % reset n to reflect truncation
Vnames = strvcat('y','cterm','x2','x3');
result = ols(yt,xt); prt(result,Vnames); plt(result);
```

Cochrane-Orcutt estimation involves iteration using the least-squares residuals in a regression on the residuals lagged one period to estimate new values for ρ conditional on the least-squares estimates of β. The new value of ρ is used to produce an updated estimate of $\hat{\beta}$ and this process is iterated. When the values for ρ converge, we stop the iteration process. The iteration is carried out using a 'while loop' that checks for convergence every time through the loop. Example 2.6 relies on the variables **xt,yt** generated in the previous example to construct Cochrane-Orcutt estimates.

```
% ----- Example 2.6 Cochrane-Orcutt iteration
% NOTE: assumes xt,yt exist (from example 2.5)
converg = 1.0; rho = 0.0; iter = 1; xl = lag(xt,1); yl = lag(yt,1);
xlag = xl(2:n,:); ylag = yl(2:n,1);     % truncate to feed the lag
y = yt(2:n,1); x = xt(2:n,:); n = n-1;  % adjust n for truncation
Vnames = strvcat('ystar','istar','x2star','x3star');
disp('Cochrane-Orcutt Estimates');
while (converg > 0.0001),
 % step 1, using initial rho = 0, do OLS to get bhat
   ystar = y - rho*ylag; xstar = x - rho*xlag; res = ols(ystar,xstar);
 % compute residuals based on beta ols estimates
   e = y - x*res.beta; elag = lag(e);
 % truncate 1st observation to account for the lag
  et = e(2:n,1); elagt = elag(2:n,1);
 % step 2, update estimate of rho using residuals from step 1
  res_rho = ols(et,elagt); rho_last = rho; rho = res_rho.beta(1);
  converg = abs(rho - rho_last);
 fprintf(1,'rho = %8.5f, converg = %8.5f, iter = %2d \n',rho,converg,iter);
 iter = iter + 1;
end; % end of while loop
% after convergence produce a final set of estimates using rho-value
ystar = y - rho*ylag;   xstar = x - rho*xlag;
res = ols(ystar,xstar); prt(res,Vnames);
```

The output from the Cochrane-Orcutt estimation in example 2.6 looks as follows:

```
Cochrane-Orcutt Estimates
rho =  0.38148, converg =  0.38148, iter =  1
rho =  0.39341, converg =  0.01193, iter =  2
rho =  0.39363, converg =  0.00022, iter =  3
rho =  0.39364, converg =  0.00000, iter =  4
Ordinary Least-squares Estimates
```

```
Dependent Variable =          ystar
R-squared      =     0.711
Rbar-squared   =     0.705
sigma^2        =     0.904
Durbin-Watson  =     1.803
Nobs, Nvars    =     99,     3
***************************************************************
        Variable      Coefficient      t-statistic    t-probability
         istar         9.80742294      62.19721521       0.00000000
         x2star        0.97002428       9.99808285       0.00000000
         x3star        1.05493208      12.06190233       0.00000000
```

As an exercise, consider packaging this procedure as a MATLAB function that we could add to our regression library. This would involve:

1. Transferring the above code to a function and providing documentation for use of the function.

2. returning a regression function 'results structure' that is compatible with the **prt_reg** function from the library.

3. modifying the **prt_reg** function by adding specific code for printing the output. Your printing code should probably present the results from iteration which can be passed to the **prt_reg** function in the results structure.

You might compare your code to that in **olsc.m** from the *regression function library*.

Maximum likelihood estimation of the least-squares model containing serial correlation requires that we simultaneously minimize the negative of the log-likelihood function with respect to the parameters ρ, β and σ_ε in the problem. This can be done using a simplex minimization procedure that exists as part of the MATLAB toolbox. Other more powerful multidimensional optimization procedures are discussed in Chapter 10 which takes up the topic of general maximum likelihood estimation of econometric models.

We will use a MATLAB function that minimizes a function of several variables using a simplex algorithm. The function is named **fmins** and it has the following input format for our application:

```
[xout, ooptions] = fmins('ar1_like',xin,ioptions,[],y,x);
```

The string input argument 'ar1_like' is the name of a MATLAB function we must write to evaluate the negative of the log-likelihood function. The argument 'xin' is a vector of starting values for the parameters, 'ioptions' is a 4x1 vector of optimization input options, and the 'y,x' after the empty matrix input argument '[]' are the data vectors that will be passed to our function **ar1_like**. The **fmins** function produces an output vector 'xout' containing the vector of parameter values that minimize the negative of the log-likelihood function and a vector 'ooptions' containing information about the optimization problem. For example, the function value at the minimum, the number of iterations taken to find a minimum, and so on.

Turning attention to the function **ar1_like**, we have a few options here. Some authors refer to the likelihood function for this model conditional on the first observation as:

$$L(\rho, \beta) = \sum_{t=2}^{n} (e_t - \rho e_{t-1})^2 \tag{2.4}$$

But, this likelihood ignores the first observation and would only be appropriate if we do not view the serial correlation process as having been in operation during the past. The function we use is not conditional on the first observation and takes the form (see Green, 1997 page 600):

$$\begin{aligned}
L(\rho, \beta) &= -(n/2)\ln(y^* - X^*\beta)'(y^* - X^*\beta) + (1/2)\ln(1-\rho^2) \qquad (2.5) \\
y^* &= Py, \quad X^* = PX \\
P &= \begin{bmatrix} \sqrt{(1-\rho^2} & 0 & 0 & \dots & 0 & 0 \\ -\rho & 1 & 0 & \dots & 0 & 0 \\ 0 & -\rho & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & \dots & -\rho & 1 \end{bmatrix}
\end{aligned}$$

where σ_ε^2 has been concentrated out of the likelihood function. We compute this parameter estimate using $e^{*\prime}e^*/(n-k)$, where $e^* = y^* - X^*\hat{\beta}$.

The MATLAB function **ar1_like** to compute this log-likelihood for various values of the parameters ρ and β is shown below:

```
function llike = ar1_like(param,y,xmat)
% PURPOSE: evaluate log-likelihood for ols model with AR1 errors
%---------------------------------------------------
% USAGE:    like = ar1_like(b,y,x)
% where:     b = parameter vector (k x 1)
%            y = dependent variable vector (n x 1)
```

```
%           x = explanatory variables matrix (n x k-1)
%---------------------------------------------------
% NOTE: this function returns a scalar equal to -log(likelihood)
%       b(1,1) contains rho parameter
%       sige is concentrated out
%---------------------------------------------------
% REFERENCES: Green, 1997 page 600
%---------------------------------------------------
[n k] = size(x); rho = param(1,1); beta = param(2:2+k-1,1);
ys = y - rho*lag(y); xs = x - rho*lag(x);
ys(1,1) = sqrt(1-rho*rho)*y(1,1); xs(1,:) = sqrt(1-rho*rho)*x(1,:);
term1 = -(n/2)*log((ys - xs*beta)'*(ys - xs*beta));
term2 = 0.5*log(1-rho*rho);
like = term1+term2;
llike = -like;
```

Note that we do not strictly follow the mathematical formulas presented in (2.6) when coding **ar1_like** to evaluate the negative of the log likelihood function. Use of the large matrix P would be inefficient and require additional computer memory for storage of the sparse matrix P.

The code shown in example 2.7 to implement maximum likelihood estimation sets initial values for the parameters ρ, β based on the Cochrane-Orcutt estimates produced by the code described previously. (They are assumed to exist in this example.) We then set some options for use in the optimization routine, and make a call to the **fmins** function using the data vector y and matrix X.

```
% ----- Example 2.7 Maximum likelihood estimation
% NOTE: assumes Cochrane-Orcutt estimates exist (from example 2.6)
param = zeros(k+2,1);          % initial values from Cochrane-Orcutt
param(1,1) = rho;              % initial rho
param(2:2+k-1,1) = res.beta;   % initial bhat's
options(1,1) = 0;              % no print of intermediate results
options(2,1) = 0.0001;         % simplex convergence criteria
options(3,1) = 0.0001;         % convergence criteria for function
options(4,1) = 1000;           % set # of function evaluations
[mrho ooptions] = fmins('ar1_like',param,options,[],y,x);
  niter = ooptions(1,10);      % recover some info on optimization
  llike = ooptions(1,8);
% find sig estimate based on ml rho and bhat's
ystar = y - rho*ylag; xstar = x - rho*xlag; bhat = mrho(2:2+k-1,1);
sig = (ystar - xstar*bhat)'*(ystar - xstar*bhat); sig = sig/(n-k);
fprintf(1,'Maximum likelihood estimate of rho   = %8.4f \n',mrho(1,1));
fprintf(1,'Maximum likelihood estimate of sigma = %8.4f \n',sige);
fprintf(1,'Maximum likelihood estimates of bhat = %8.4f \n',bhat);
fprintf(1,'negative of Log-likelihood value     = %8.4f \n\n',llike);
fprintf(1,'number of iterations taken           = %4d \n\n',niter);
```

The resulting parameter estimates for ρ, β are returned from the **fmins** function and used to compute an estimate of σ_ε^2. These results are then printed. For an example of a function in the regression library that implements this approach, see **olsar1**.

2.6.3 Implementing statistical tests

In this example, we demonstrate how the 'results' structure returned by the **ols** function can be used in a function named **waldf** that carries out the Wald F-test for a restricted versus unrestricted model. A joint F-test is used to test whether the zero restrictions produced by omitting variables from a model are consistent with the sample data.

We generate a data set containing ten explanatory variables and carry out a regression. A second model based on only the first five explanatory variables represents a restricted model. We wish to carry out a joint F-test for the significance of variables 6 through 10 in the model. This involves computing an F-statistic based on the residuals from the restricted ($e'e_r$) and unrestricted ($e'e_u$) models:

$$\frac{F = (e'e_r - e'e_u)/m}{e'e_u/(n-k)} \tag{2.6}$$

where m is the number of restrictions (five in our example) and n, k are the number of observations and number of explanatory variables in the unrestricted model respectively.

We use **ols** for the two regressions and send the 'results' structures to a function **waldf** that will carry out the joint F-test and return the results.

Assuming the existence of a function **waldf** to implement the joint F-test, the MATLAB code to carry out the test would be as shown in example 2.8.

```
% ----- Example 2.8 Wald's F-test
nobs = 200; nvar = 10; b = ones(nvar,1); b(7:nvar,1) = 0.0;
xmat1 = randn(nobs,nvar-1); evec = randn(nobs,1)*5;
x = [ones(nobs,1) xmat1];           % unrestricted data set
y = x*b + evec;       x2 = x(:,1:5); % restricted data set
resultu = ols(y,x);  prt(resultu);  % do unrestricted ols regression
resultr = ols(y,x2); prt(resultr);  % do restricted ols regression
% test the restrictions
[fstat fprob] = waldf(resultr,resultu);
disp('Wald F-test results'); [fstat fprob]
```

All of the information necessary to carry out the joint F-test resides in the two results structures, so the function **waldf** needs only these as input arguments. Below is the **waldf** function.

```
function [fstat, fprb] = waldf(resultr,resultu)
% PURPOSE: computes Wald F-test for two regressions
%---------------------------------------------------
% USAGE: [fstat fprob] = waldf(resultr,resultu)
% Where: resultr = results structure from ols() restricted regression
%        resultu = results structure from ols() unrestricted regression
%---------------------------------------------------
% RETURNS: fstat = {(essr - essu)/#restrict}/{essu/(nobs-nvar)}
%          fprb  = marginal probability for fstat
% NOTE:  large fstat => reject as inconsistent with the data
%---------------------------------------------------
if nargin ~= 2 % flag incorrect arguments
    error('waldf: Wrong # of input arguments');
elseif isstruct(resultu) == 0
error('waldf requires an ols results structure as input');
elseif isstruct(resultr) == 0
error('waldf requires an ols results structure as input');
end;
% get nobs, nvar from unrestricted and restricted regressions
nu = resultu.nobs;  nr = resultr.nobs;
ku = resultu.nvar;  kr = resultr.nvar;
if nu ~= nr
error('waldf: the # of obs in the results structures are different');
end;
if (ku - kr) < 0 % flag reversed input arguments
error('waldf: negative dof, check for reversed input arguments');
end;
% recover residual sum of squares from .sige field of the result structure
epeu = resultu.sige*(nu-ku);  eper = resultr.sige*(nr-kr);
numr = ku - kr;              % find # of restrictions
ddof = nu-ku;                % find denominator dof
fstat1 = (eper - epeu)/numr; % numerator
fstat2 = epeu/(nu-ku);       % denominator
fstat = fstat1/fstat2; fprb = fdis_prb(fstat,numr,ddof);
```

The only point to note is the use of the function **fdis_prb** which returns the marginal probability for the F-statistic based on numerator and denominator degrees of freedom parameters. This function is part of the *distribution functions library* discussed in Chapter 9.

As another example, consider carrying out an LM specification test for the same type of model based on restricted and unrestricted explanatory variables matrices. This test involves a regression of the residuals from the restricted model on the explanatory variables matrix from the unrestricted model. The test is then based on the statistic computed using $n * R^2$ from this regression, which is chi-squared (χ^2) distributed with degrees of freedom equal to the number of restrictions.

We can implement a function **lm_test** that takes as input arguments the 'results' structure from the restricted regression and an explanatory variables matrix for the unrestricted model. The **lm_test** function will: carry out the regression of the residuals from the restricted model on the variables in the unrestricted model, compute the chi-squared statistic, and evaluate the marginal probability for this statistic. We will return a regression 'results' structure from the regression performed in the **lm_test** function so users can print out the regression if they so desire. Example 2.9 shows a program that calls the **lm_test** function to carry out the test.

```
% ----- Example 2.9 LM specification test
nobs = 200; nvar = 7;   beta = ones(nvar,1);
beta(6:nvar,1) = 0.0;   evec = randn(nobs,1)*5;
xmat1 = randn(nobs,nvar-1);
x = [ones(nobs,1) xmat1]; % unrestricted data set
xr = x(:,1:5);            % restricted data set
y = x*beta + evec;
resultr = ols(y,xr);    prt(resultr);           % restricted ols
[lmstat lmprob result_lm] = lm_test(resultr,x); % do LM - test
disp('LM-test regression results'); prt(result_lm);
disp('LM test results'); [lmstat lmprob]
```

Note that one of the arguments returned by the **lm_test** function is a regression 'results' structure that can be printed using the **prt** function. These regression results reflect the regression of the residuals from the restricted model on the unrestricted data matrix. The user might be interested in seeing which particular explanatory variables in the unrestricted model matrix exhibited a significant influence on the residuals from the restricted model. The results structure is passed back from the **lm_test** function just as if it were any scalar or matrix argument being returned from the function.

The function **lm_test** is shown below.

```
function [lmstat, lmprob, reslm] = lm_test(resr,xu)
% PURPOSE: computes LM-test for two regressions
%---------------------------------------------------
% USAGE: [lmstat lmprob, result] = lm_test(resultr,xmatrixu)
% Where: resultr  = matrix returned by ols() for restricted regression
%        xmatrixu = explanatory variables matrix from unrestricted model
%---------------------------------------------------
% RETURNS:  lmstat = calculated chi-squared statistic
%           lmprob = marginal probability for lmstat
```

```
%           result = ols() results structure
%---------------------------------------------------
% NOTE:   Expected value of (lmstat) = #restrictions
%---------------------------------------------------
if nargin ~= 2,             error('lm_test: Wrong # of inputs');
elseif isstruct(resr) == 0, error('lm_test: ols structure needed');
end;
nobs1 = resultr.nobs;   [nobs rsize] = size(xu);
if nobs1 ~= nobs
 error('lm_test: both inputs should have same # of obs');
end;
er = resultr.resid; % recover residuals from restricted model
reslm = ols(er,xu); % step 1) regress er on all the x's
rsqr = reslm.rsqr;  % step 2) recover R^2
lmstat = rsqr*nobs; % compute chi-squared statistic
% determine the # of restrictions
[junk ku] = size(xu); kr = resultr.nvar; nrestrict = ku - kr;
lmprob = 1.0 - chis_prb(lmstat,nrestrict);
```

2.7 Chapter summary

We designed a general framework for regression functions that share common functions for printing and plotting results. The use of MATLAB Version 5 structure variables allowed us to pass regression estimation results between these functions using a single structure variable. This design also allows us to implement auxiliary functions to carry out traditional econometric statistical tests.

In Chapter 3 we will see how a single function **prt** and another **plt** can be constructed to print and plot results from all functions in the *Econometrics Toolbox*, including those in the regression library. This facility also derives from the powerful possibilities associated with MATLAB structure variables.

Chapter 2 Appendix

The *regression function library* is in a subdirectory **regress**.

```
regression function library

------- regression program functions -----------

ar_g         - Gibbs sampling Bayesian autoregressive model
bma_g        - Gibbs sampling Bayesian model averaging
boxcox       - Box-Cox regression with 1 parameter
boxcox2      - Box-Cox regression with 2 parameters
egarchm      - EGARCH(p,q)-in-Mean regression model
emhergott    - EM estimates of Hamilton's markov switching model
garchs       - garchs(1,1,1) model using non-central t-distribution
hmarkov_em   - Hamilton's markov switching model
hwhite       - Halbert White's heteroscedastic consistent estimates
lad          - least-absolute deviations regression
lm_test      - LM-test for two regression models
logit        - logit regression
mlogit       - multinomial logit regression
nwest        - Newey-West hetero/serial consistent estimates
ols          - ordinary least-squares
ols_g        - Gibbs sampling Bayesian linear model
olsar1       - Maximum Likelihood for AR(1) errors ols model
olsc         - Cochrane-Orcutt AR(1) errors ols model
olst         - regression with t-distributed errors
probit       - probit regression
probit_g     - Gibbs sampling Bayesian probit model
ridge        - ridge regression
robust       - iteratively reweighted least-squares
rtrace       - ridge estimates vs parameters (plot)
sur          - seemingly unrelated regressions
switch_em    - switching regime regression using EM-algorithm
theil        - Theil-Goldberger mixed estimation
thsls        - three-stage least-squares
tobit        - tobit regression
tobit_g      - Gibbs sampling Bayesian tobit model
tsls         - two-stage least-squares
```

```
waldf        - Wald F-test

-------- demonstration programs -----------------

ar_gd        - demonstration of Gibbs sampling ar_g
bma_gd       - demonstrates Bayesian model averaging
boxcox_d     - demonstrates Box-Cox 1-parameter model
boxcox2_d    - demonstrates Box-Cox 2-parmaeter model
demo_all     - demos most regression functions
egarchm_d    - demos egarchm function
garchs_d     - demos garchs function
hmarkov_emd  - demos Hamilton's model
hmarkov_emd2 - another demo of Hamilton's model
hwhite_d     - H. White's hetero consistent estimates demo
lad_d        - demos lad regression
lm_test_d    - demos lm_test
logit_d      - demonstrates logit regression
mlogit_d     - demonstrates multinomial logit
nwest_d      - demonstrates Newey-West estimates
ols_d        - demonstrates ols regression
ols_d2       - Monte Carlo demo using ols regression
ols_gd       - demo of Gibbs sampling ols_g
olsar1_d     - Max Like AR(1) errors model demo
olsc_d       - Cochrane-Orcutt demo
olst_d       - olst demo
probit_d     - probit regression demo
probit_gd    - demo of Gibbs sampling Bayesian probit model
ridge_d      - ridge regression demo
robust_d     - demonstrates robust regression
sur_d        - demonstrates sur using Grunfeld's data
switch_emd   - demonstrates switching regression
theil_d      - demonstrates theil-goldberger estimation
thsls_d      - three-stage least-squares demo
tobit_d      - tobit regression demo
tobit_d2     - tobit right-censoring demo
tobit_gd     - demo of Gibbs sampling Bayesian tobit model
tobit_gd2    - Bayesian tobit right-censoring demo
tsls_d       - two-stage least-squares demo
waldf_d      - demo of using wald F-test function

-------- Support routines -----------------------

ar1_like     - used by olsar1   (likelihood)
bmapost      - used by bma_g
box_lik      - used by box_cox  (likelihood)
box_lik2     - used by box_cox2 (likelihood)
chis_prb     - computes chi-squared probabilities
dmult        - used by mlogit
egarchm_lik  - likelihood used by egarchm
```

```
garchs_llf   - likelihood used by garchs
herg_llf     - likelihood used by emhergott
herg_llf2    - likelihood used by emhergott
hmarkov_llf  - likelihood used by hmarkov_em
hmarkov_llf2 - likelihood used by hmarkov_em
fdis_prb     - computes F-statistic probabilities
find_new     - used by bma_g
grun.dat     - Grunfeld's data used by sur_d
grun.doc     - documents Grunfeld's data set
hessian      - used by tobit to determine numerical hessian
lo_like      - used by logit    (likelihood)
mcov         - used by hwhite
mderivs      - used by mlogit
mlogit_lik   - used by mlogit
nmlt_rnd     - used by probit_g
nmrt_rnd     - used by probit_g, tobit_g
norm_cdf     - used by probit, pr_like
norm_pdf     - used by prt_reg, probit
olse         - ols returning only residuals (used by sur)
plt_eqs      - plots equation systems
plt_reg      - plots regressions
pr_like      - used by probit   (likelihood)
prt_eqs      - prints equation systems
prt_gibbs    - prints Gibbs sampling models
prt_reg      - prints regressions
prt_swm      - prints switching regression results
sample       - used by bma_g
stdn_cdf     - used by norm_cdf
stdn_pdf     - used by norm_pdf
tdis_prb     - computes t-statistic probabilities
to_like      - used by tobit    (likelihood)
```

Chapter 3

Utility Functions

This chapter presents utility functions that we will use throughout the remainder of the text. We can categorize the various functions described here into functions for:

1. Working with time-series that contain dates. MATLAB is relatively weak in this regard when compared to standard econometric programs for working with economic time-series like RATS and TSP.

2. General functions for printing and plotting matrices as well as producing LaTeX formatted output of matrices for tables.

3. Econometric data transformations.

4. Some functions that mimic Gauss functions, allowing us to more easily re-code the many econometric procedures available for Gauss to MATLAB functions.

The material in this chapter is divided into four sections that correspond to the types of utility functions enumerated above. A final section discusses development of "wrapper" functions that call other functions to make printing and plotting econometric procedure results simpler for the user.

3.1 Calendar function utilities

Working with time-series data often requires that we associate observations with calendar dates, where the dates vary depending on the frequency of the data. We construct three functions that help in this task.

The first function **cal** stores information regarding the calendar dates covered by our time-series data in a MATLAB structure variable. We would use the function at the outset of our analysis to package information regarding the starting year, period and frequency of our data series. This packaged structure of calendar information can then be passed to other time series estimation and printing functions.

For example, consider a case where we are dealing with monthly data that begins in January, 1982, we would store this information using the call:

```
cal_struct = cal(1982,1,12);
```

which returns a structure, 'cal_struct' with the following fields for the beginning year, period and frequency.

```
cal_struct.beg_yr
cal_struc.beg_per
cal_struc.freq
```

The field 'beg_yr' would equal 1982, 'beg_per' would equal 1 for January, and 'freq' would equal 12 to designate monthly data. Beyond setting up calendar information regarding the starting dates for our time-series data, we might want to know the calendar date associated with a particular observation. The **cal** function can provide this information as well, returning it as part of the structure. The documentation for the function is:

```
PURPOSE: create a time-series calendar structure variable that
         associates a date with an observation #
-----------------------------------------------------
USAGE:       result = cal(begin_yr,begin_per,freq,obs)
        or:  result = cal(cstruc,obs)
where:    begin_yr  = beginning year, e.g., 1982
          begin_per = beginning period, e.g., 3
              freq  = frequency, 1=annual,4=quarterly,12=monthly
              obs   = optional argument for an observation #
          cstruc    = a structure returned by a previous call to cal()
-----------------------------------------------------
RETURNS: a structure:
            result.beg_yr  = begin_yr
            result.beg_per = begin_period
            result.freq    = frequency
            result.obs     = obs            (if input)
            result.year    = year for obs   (if input)
            result.period  = period for obs (if input)
```

Calling **cal** with an observation number argument, will produce a return structure that associates that particular observation with a year and period, recorded in the '.year' and '.period' fields. We also include the observation number in the field '.obs' of the returned structure. To illustrate use of this function, consider that our data series start in January, 1982 and we wish to determine the calendar year and period associated with observation number 83, we would make the call:

```
cal_struct = cal(1982,1,12,83)
ans =
     beg_yr: 1982
    beg_per: 1
       freq: 12
        obs: 83
       year: 1988
     period: 11
```

which informs us that observation number 83 is associated with November, 1988.

Another option allows you to call **cal** with a structure returned by a previous call to the function itself. This might be useful if you set up a calendar structure at the outset of your program and later wish to find observation numbers associated with dates. Here is an example of this type of usage.

```
% ----- Example 3.1 Using the datesf() function
load test.dat; % monthly mining employment for il,in,ky,mi,oh,pa,tn,wv
dates = cal(1982,1,12);          % data covers 1982,1 to 1996,5
begf = ical(1995,6,dates);       % beginning forecast period
nfor = 12;                       % number of forecast periods
datesf = cal(dates,begf+nfor-1); % add end of forecast period
```

Note that we could use: **datesf = cal(1982,1,12,begf+nfor-1)**, so the ability to use the previously returned structure 'dates' simply saves us some typing.

The function **ical** used in the example above, serves as an inverse to the **cal** function, returning an observation number associated with a particular calendar date. It utilizes the information stored in the structure returned by **cal** to determine this information. Forecasting provides an example where we might use this function. Suppose our data series begins in January 1980 and we wish to produce a forecast beginning in June, 1991 we could utilize the following calls to **cal** and **ical** to accomplish this task.

```
cstr = cal(1980,1,12);
begf = ical(1991,6,cstr);
```

This would set begf=138, which we can utilize when calling our forecasting function, or simply printing the data series observations for this particular date.

The third function, **tsdate** allows us to display and print date strings associated with annual, quarterly or monthly time-series, which is helpful for printing results and forecasted values with date labels. Documentation for this function is:

```
PURPOSE: produce a time-series date string for an observation #
         given beginning year, beginning period and frequency of the data
---------------------------------------------------
USAGE: out = tsdate(beg_yr,beg_period,freq,obsn);
       or:   tsdate(beg_yr,beg_period,freq,obsn);
       or:   tsdate(cal_struct,obsn);
where: beg_yr     = beginning year, e.g., 1974
       beg_period = beginning period, e.g., 4 for april
       freq       = 1 for annual, 4 for quarterly 12 for monthly
       obsn       = the observation #
       cal_struct = a structure returned by cal()

e.g., tsdate(1974,1,12,13) would print: Jan75
      tsdate(1974,1,4,13)  would print: Q1-77
      tsdate(1974,1,1,13)  would print 1986
      out = tsdate(1974,1,12,13); would return a string 'Jan75'
      cstr = cal(1974,1,12);
      tsdate(cstr,13);      would print Jan75
---------------------------------------------------
RETURNS: a string, or: simply displays the date associated
         with obsn (observation #) in the command window
```

The function uses MATLAB 'nargout' to determine whether the user has requested an output argument or not. In the event that no output argument is requested, the function calls the MATLAB **datestr** function with no semicolon after the call to produce a printed string in the MATLAB command window. For cases where the user desires more control over the printing format of the date string, or wishes to include the date string in an output file, we can call the function with an output argument. As an illustration of the first case, consider the following code:

```
% ----- Example 3.2 Using the tsdate() function
cstruct = cal(1982,1,4);
for i=1:2;
tsdate(cstruct,i);
end;
```

which would produce the following output in the MATLAB command window:

```
Q1-82
Q2-82
```

On the other hand, we could make a series of calls providing a MATLAB cell-variable array as an output argument:

```
% ----- Example 3.3 Using cal() and tsdates() functions
cstruct = cal(1982,1,12);
for i=1:6;
fdates{i} = tsdate(cstruct,i);
end;
```

that would place each date in the cell-array for use later in printing. The cell-array looks as follows:

```
fdates =
 'Jan82'    'Feb82'    'Mar82'    'Apr82'    'May82'    'Jun82'
```

Further illustrations will be provided in Chapter 5 that demonstrate how these utility functions are used in conjunction with the vector autoregressive modeling functions. To summarize, we provide the following example program that utilizes the functions to read and print time-series data.

```
% ----- Example 3.4 Reading and printing time-series
dates = cal(1982,1,12);
load test.dat; % monthly time-series data starting in Jan,1982
[nobs nvar] = size(test);
endd = tsdate(dates,nobs);      % date of the last observation
fprintf(1,'The ending date of the sample is %10s \n\n',endd);
begin_prt = ical(1990,1,dates); % print data with dates for 1990
end_prt   = ical(1990,12,dates);
fprintf(1,'The data for 1990 is: \n\n');
for i=begin_prt:end_prt; fprintf(1,'%10s ',tsdate(dates,i));
for j=1:nvar fprintf(1,'%8.0f ',test(i,j)); end;
fprintf(1,'\n'); end;
```

One comment is that we construct a function **tsprint** in the next section that makes it easier to print time-series data than the approach demonstrated in the example above.

3.2 Printing and plotting matrices

Printing matrices and time-series is a frequent occurrence in econometric work, as is the task of producing tabular results for inclusion in word-processing. This section presents three utility functions to help with this task. We also provide a function to plot time-series with dates on the time-axis.

The function **lprint** transforms a MATLAB matrix by adding formatting symbols needed to produce a table in LaTeX, a widely-used mathematical typesetting program.

```
 PURPOSE: print an (nobs x nvar) matrix in LaTeX table format
---------------------------------------------------
 USAGE:     lprint(x,info)
 where:
 x         = (nobs x nvar) matrix (or vector) to be printed
 info      = a structure containing printing options
 info.begr = beginning row to print,    (default = 1)
 info.endr = ending row to print,       (default = nobs)
 info.begc = beginning column to print, (default = 1)
 info.endc = ending column to print,    (default = nvar)
 info.cnames = an (nvar x 1) string of names for columns (optional)
               e.g. info.cnames = strvcat('col1','col2');
               (default = no column headings)
 info.rnames = an (nobs+1 x 1) string of names for rows (optional)
               e.g. info.rnames = strvcat('Rows','row1','row2');
               (default = no row labels)
 info.fmt    = a format string, e.g., '%12.6f' or '%12d' (default = %10.4f)
               or an (nvar x 1) string containing formats
               e.g., info.fmt=strvcat('%12.6f','%12.2f','%12d');
 info.fid    = file-id for printing results to a file
               (defaults to the MATLAB command window)
               e.g. fid = fopen('file.out','w');
 info.rflag  = 1  row #'s printed, 0 no row #'s (default = 0)
---------------------------------------------------
 e.g.   in.cnames = strvcat('col1','col2');
        in.rnames = strvcat('rowlabel','row1','row2');
        lprint(y,in), prints entire matrix, column and row headings
        in2.endc = 3; in2.cnames = strvcat('col1','col2','col3');
    or: lprint(y,in2), prints 3 columns of the matrix, just column headings
    or: lprint(y), prints entire matrix, no column headings or row labels
 NOTES: - defaults are used for info-elements not specified
        - wrapping occurs at (80/format) columns, which varies with
          format used, e.g. %10.2f will wrap after 8 columns
---------------------------------------------------
 SEE ALSO: tsprint, mprint, lprint_d
---------------------------------------------------
```

This is the first example of a function that uses the MATLAB structure variable as an input argument. This allows us to provide a large number of input arguments using a single structure variable. It also simplifies parsing the input arguments in the function and setting default options.

As an example of using the function, consider the following program that generates a series of matrices containing normally distributed random numbers and transforms them to LaTeX format using **lprint** and various formatting options. Note that you can name the structure variable used to input the options anything — it is the fieldnames that the function **lprint** parses to find the options.

```
% ----- Example 3.5 Using the lprint() function
table = randn(5,3);          fprintf(1,'using no options \n');
lprint(table);               % no printing options used
table2 = round(table)*1000;  fprintf(1,'using fmt option \n');
option.fmt = '%10.0f';       lprint(table2,option); % format option used
fprintf(1,'using column names and fmt option \n');
inarg.cnames = strvcat('Illinois','Ohio','Indiana');
inarg.fmt = '%12.3f';  lprint(table2,inarg);      % format, column names
fprintf(1,'row and column labels \n');
inarg.rnames = strvcat('Rows','row1','row2','row3','row4','row5');
lprint(table2,inarg);                             % adding row-labels
fprintf(1,'wrapped output for large matrices \n');
vnames = strvcat('IL','OH','IN','WV','PA','AK','HI','NY','MS','TN');
table3 = randn(5,10); option2.fmt = '%20.4f'; option2.cnames = vnames;
lprint(table3,option2);                           % wrapping matrices
fprintf(1,'generic row labels \n');
table4 = randn(4,4)*100; option3.fmt = '%10.4f'; option3.rflag = 1;
option3.cnames = strvcat('column1','column2','column3','column4');
lprint(table4,option3);                           % generic row-labels
fprintf(1,'variable column formats \n');
table4(:,1) = round(table4(:,1)); table4(:,3) = round(table4(:,3));
option4.fmt = strvcat('%10d','%8.3f','%10d','%16.3f');
option4.cnames = cnames; lprint(table4,option4);  % variable column formats
fprintf(1,'demo of printing selected rows and columns \n');
clear in; table = randn(5,10);
in.begc = 5; in.endc = 9;    % specify selected columns to print
in.begr = 2; in.endr = 5;    % specify selected rows to print
cnames = strvcat('col1','col2','col3','col4');
cnames = strvcat(cnames,'col5','col6','col7','col8','col9','col10');
rnames = strvcat('Rows','row1','row2','row3','row4','row5');
% NOTE we need column and row names for all rows and columns
%      not just those selected for printing
in.cnames = cnames; in.rnames = rnames;
lprint(table,in);                 % print selected rows and columns
```

The output from this example is shown below. Matrices that have a large number of columns are automatically wrapped when printed, with wrapping determined by both the numeric format and column names. To understand how wrapping works, note that the **lprint** function examines the width of column names as well as the width of the numeric format supplied. The numeric format is padded at the left to match column names that are wider than the format. If column names are not as wide as the numeric format, these are right-justified. For example, if you supply a 15 character column name and a '10.6f' numeric format, the columns will be 15 characters wide with the numbers printed in the last 10 character places in the '10.6f' format. On the other hand, if you supply 8 character column names and '10.6f' numeric format, the columns will be 10 characters wide with the column names right-justified to fill the right-most 8 characters.

Wrapping occurs after a number of columns determined by 80 divided by the numeric format width. Depending on the numeric format, you can achieve a different number of columns for your table. For example, using a format '10f.3' would allow 8 columns in your table, whereas using '5d' would allow up to 16 columns. The format of '20.4f' in the example, produces wrapping after 4 columns. Note that the width of the printout may be wider than 80 columns because of the column name padding considerations discussed in the previous paragraph.

```
using no options
    -0.2212 &     0.3829 &    -0.6628 \\
    -1.5460 &    -0.7566 &    -1.0419 \\
    -0.7883 &    -1.2447 &    -0.6663 \\
    -0.6978 &    -0.9010 &     0.2130 \\
     0.0539 &    -1.1149 &    -1.2154 \\
using fmt option
         -0 &          0 &      -1000 \\
      -2000 &      -1000 &      -1000 \\
      -1000 &      -1000 &      -1000 \\
      -1000 &      -1000 &          0 \\
          0 &      -1000 &      -1000 \\
using column names and fmt option
  Illinois  &       Ohio  &     Indiana  \\
    -0.000  &      0.000  &   -1000.000  \\
 -2000.000  &  -1000.000  &   -1000.000  \\
 -1000.000  &  -1000.000  &   -1000.000  \\
 -1000.000  &  -1000.000  &       0.000  \\
     0.000  &  -1000.000  &   -1000.000  \\
row and column labels
Rows     Illinois  &       Ohio  &     Indiana  \\
row1       -0.000  &      0.000  &   -1000.000  \\
row2    -2000.000  &  -1000.000  &   -1000.000  \\
row3    -1000.000  &  -1000.000  &   -1000.000  \\
row4    -1000.000  &  -1000.000  &       0.000  \\
row5        0.000  &  -1000.000  &   -1000.000  \\
wrapped output for large matrices
        IL  &                  OH  &                  IN  &                  WV  &
   -0.6453  &              0.6141  &              0.1035  &              1.5466  &
    1.0657  &             -0.6160  &              0.4756  &              0.6984  &
   -0.1516  &              1.0661  &              1.2727  &              0.8227  &
    0.8837  &             -0.8217  &              1.6452  &              1.1852  &
    0.8678  &             -1.0294  &              0.8200  &              0.4893  &
```

```
      PA  &             AK  &             HI  &             NY  &
  0.7344  &         0.0380  &         0.2682  &        -0.0662  &
 -0.7534  &         0.3395  &        -0.8468  &         0.8473  &
 -1.2320  &         0.1657  &        -0.4726  &         0.4700  &
  1.1433  &        -1.7245  &        -1.4360  &        -1.4543  &
  1.0019  &        -0.0296  &         0.0132  &         1.1019  &
      MS  &             TN  \\
 -0.3047  &         1.8723  \\
  1.5178  &        -0.6233  \\
  0.6368  &        -0.3576  \\
  0.3657  &        -0.0250  \\
  0.9989  &        -0.1637  \\
generic row labels
 Obs#    column1  &    column2  &    column3  &    column4  \\
    1     2.9275  &    71.3611  &   133.5464  &   -53.9951  \\
    2     9.2330  &   -76.0785  &   139.0374  &   -55.8497  \\
    3   -42.6711  &    94.7476  &  -107.4995  &   -78.1427  \\
    4   -61.9797  &   -15.0058  &    40.9104  &    52.3172  \\
variable column formats
   column1  &  column2  &    column3  &          column4  \\
         3  &   71.361  &        134  &          -53.995  \\
         9  &  -76.079  &        139  &          -55.850  \\
       -43  &   94.748  &       -107  &          -78.143  \\
       -62  &  -15.006  &         41  &           52.317  \\
demo of printing selected rows and columns
Rows       col5  &       col6  &       col7  &       col8  &      col9   \\
row2    -1.0790  &    -0.8061  &     0.0273  &    -0.5726  &    0.5509   \\
row3     2.0281  &     1.2631  &     2.3123  &    -0.4425  &   -1.6558   \\
row4     0.0198  &    -0.0496  &    -0.2238  &    -0.2303  &    1.5463   \\
row5     0.7006  &    -0.0253  &    -1.1709  &     1.2747  &    0.4729   \\
```

There is also a function **lprintf** contributed by Mike Cliff that prints a matrix of data with a criteria-based symbol next to each element. Data are formatted globally or column-wise, as specified. This function was designed to all indicating significance levels of parameter estimates in LaTeX tables.

For printing general matrices to the MATLAB command window or a file, we provide the function **mprint**. This function works identically to **lprint**, but does not provide the LaTeX formatting symbols. Both **lprint** and **mprint** permit only integer and decimal formats such as '%6d' and '%10.4f'. The documentation is:

```
 PURPOSE: print an (nobs x nvar) matrix in formatted form
---------------------------------------------------
 USAGE:     mprint(x,info)
 where:
 x         = (nobs x nvar) matrix (or vector) to be printed
 info      = a structure containing printing options
 info.begr = beginning row to print,    (default = 1)
 info.endr = ending row to print,       (default = nobs)
 info.begc = beginning column to print, (default = 1)
 info.endc = ending column to print,    (default = nvar)
```

```
 info.cnames = an (nvar x 1) string of names for columns (optional)
               e.g. info.cnames = strvcat('col1','col2');
               (default = no column headings)
 info.rnames = an (nobs+1 x 1) string of names for rows (optional)
               e.g. info.rnames = strvcat('Rows','row1','row2');
               (default = no row labels)
 info.fmt    = a format string, e.g., '%12.6f' or '%12d' (default = %10.4f)
               or an (nvar x 1) string containing formats
               e.g., info.fmt=strvcat('%12.6f','%12.2f','%12d');
 info.fid    = file-id for printing results to a file
               (defaults to the MATLAB command window)
               e.g. fid = fopen('file.out','w');
 info.rflag  = 1 row #'s printed, 0 no row #'s (default = 0)
 info.width  = # of columns before wrapping occurs (default = 80)
---------------------------------------------------
 e.g.   in.cnames = strvcat('col1','col2');
        in.rnames = strvcat('rowlabel','row1','row2');
        mprint(y,in), prints entire matrix, column and row headings
        in2.endc = 3; in2.cnames = strvcat('col1','col2','col3');
    or: mprint(y,in2), prints 3 columns of the matrix, with headings
    or: mprint(y), prints entire matrix, no headings
 NOTES: - defaults are used for info-elements not specified
        - wrapping occurs at 80 columns, which varies depending on
          format used, e.g. %10.2f will wrap after 8 columns
---------------------------------------------------
 SEE ALSO: tsprint, mprint_d, lprint
---------------------------------------------------
```

The use of a structure variable to input arguments to functions is a useful MATLAB programming construct that we employ in many *Econometric Toolbox* functions. To see how this is accomplished consider the following code from **mprint** that parses the input structure fields for the structure variable named 'info' in the function declaration.

```
function mprint(y,info)
% note structure variable named info in argument declaration
[nobs nvars] = size(y);
fid = 1; rflag = 0; cflag = 0; rnum = 0; nfmts = 1; % setup defaults
begr = 1; endr = nobs; begc = 1; endc = nvars; fmt = '%10.4f'; cwidth = 80;
if nargin == 1
% rely on defaults
elseif nargin == 2
  if ~isstruct(info)
    error('mprint: you must supply the options as a structure variable');
  end;
fields = fieldnames(info);
nf = length(fields);
for i=1:nf
```

```
    if strcmp(fields{i},'fmt')
        fmts = info.fmt; [nfmts junk] = size(fmts);
        if nfmts <= nvars, fmt = fmts;
        else
         error('mprint: wrong # of formats in string -- need nvar');
        end;
    elseif strcmp(fields{i},'fid'),  fid = info.fid;
    elseif strcmp(fields{i},'begc'), begc = info.begc;
    elseif strcmp(fields{i},'begr'), begr = info.begr;
    elseif strcmp(fields{i},'endc'), endc = info.endc;
    elseif strcmp(fields{i},'endr'), endr = info.endr;
    elseif strcmp(fields{i},'cnames'), cnames = info.cnames; cflag = 1;
    elseif strcmp(fields{i},'rnames'), rnames = info.rnames; rflag = 1;
    elseif strcmp(fields{i},'rflag'), rnum = info.rflag;
elseif strcmp(fields{i},'width'); cwidth = info.width;
    end;
end;
else
error('Wrong # of arguments to mprint');
end; % end of if-elseif input checking
```

After setting up default values, we use the MATLAB function **fieldnames** to extract the fields from the structure variable into a cell-array named 'fields'. These are parsed in a 'for-loop' over the length of the 'fields' cell-array using the MATLAB function **strcmp**. Input fields specified by the user will be detected by the string comparisons and extracted to overwrite the default argument values set prior to parsing the input structure variable.

As with the case of **lprint**, Mike Cliff has provided an enchanced version of **mprint** in the form of a function named **mprint3** which prints a set of matrices together by stacking the (i,j) elements of each. This allows for separate formatting of each matrix so one can print parameter estimates, standard errors and t-statistics.

To facilitate printing time-series data we have a function **tsprint** that will print time-series with calendar dates as labels. The function documentation is:

```
 PURPOSE: print time-series data with dates and column labels
---------------------------------------------------
 USAGE:     tsprint(y,cstruc,begp,endp,vnames,fmt)
        or: tsprint(y,cstruct,vnames), prints entire series with names
        or: tsprint(y,cstruct), entire series, no variable names
        or: tsprint(y,cstruct,fmt) entire series, using fmt
        or: tsprint(y,cstruct,vnames,fmt) entire series w/names and fmt
        or: tsprint(y,cstruct,begp,endp) partial series no names or fmt
```

```
  where: y       = matrix (or vector) of series to be printed
         cstruc  = a structure returned by cal()
         begp    = the beginning observation to print
         endp    = the ending period to print,
         vnames  = a string matrix of names for a header (optional)
         fmt     = a format string, e.g., '%12.6f' or '%12d'
 ---------------------------------------------------
NOTES:    cstr = cal(1980,1,12);
          tsprint(y,cstr,13,24), would print data for 1981
      or: tsprint(y,cstr,ical(1981,1,cstr),ical(1981,12,cstr)),
          which would print the same data for 1981
 ---------------------------------------------------
```

This function takes a dates structure, begin and end period and format as arguments. As indicated in the function documentation, the user can call this function using a number of different input arguments in almost any order. Example 3.6 demonstrates some of the alternatives.

```
% ----- Example 3.6 Using the tsprint() function
names =strvcat('Illinois','Ohio','Indiana','West Virginia','Pennsylvania');
d1 = randn(120,5); dates = cal(1980,1,12); fmt = '%14.3f';
begp = ical(1985,1,dates); endp = ical(1985,12,dates);
fprintf(1,'all options \n');        tsprint(d1,dates,begp,endp,names,fmt);
fprintf(1,'default format  \n');    tsprint(d1,dates,begp,endp,names);
fprintf(1,'integer format and printing the whole time-series \n');
fmt = '%13d'; d2 = round(data)*100; tsprint(d2,dates,names,fmt);
fprintf(1,'d format with floating point #s \n'); tt=1:90;
d3 = [tt' d1(:,2:5)]; fmt = '%13d'; tsprint(d3,dates,begp,endp,fmt);
fprintf(1,'format option, monthly dates and wrapping \n');
d4 = randn(12,10); fmt = '%16.8f';  tsprint(d4,dates,fmt);
fprintf(1,'format option, quarterly dates and wrapping \n');
dates2 = cal(1980,1,4);             tsprint(d4,dates2,fmt);
fprintf(1,'default format, annual dates and wrapping \n');
dates3 = cal(1980,1,1);             tsprint(d4,dates3);
```

This flexibility to allow user input of any number of the options in almost any order is achieved using the MATLAB **varargin** variable. This is a generally useful approach to crafting functions, so we provide the details of how this is accomplished. The function declaration contains the MATLAB keyword 'varargin' which is a cell-array that we parse inside the function. It is our responsibility to correctly deduce the number and order of the input arguments supplied by the user.

```
function tsprint(y,cstruc,varargin)
% NOTE the use of varargin keyword in the function declaration
[nobs nvar] = size(y); fmt = '%10.4f';  % set defaults
```

```
begp = 1; endp = nobs; nflag = 0;
nargs = length(varargin);               % find the # of input arguments
if nargs == 0                 % no user-supplied vnames or dates or fmt
% rely on defaults
elseif nargs == 1             % no dates but vnames or fmt
 [testf testg] = size(varargin{1});
 if testf == 1                % we have a format
  fmt = varargin{1};          % replace default fmt with user fmt
 else                         % we have vnames
  nflag = 1;                  % set flag for vnames
  vnames = varargin{1};       % pull-out user variable names
 end;
elseif nargs == 2;            % either begp,endp or names and fmt
 [testf testg] = size(varargin{1});
 if testf == 1                % we have a format or begp
    if isnumeric(varargin{1}) % we have begp, endp
     begp = varargin{1};      % pull-out begp
     endp = varargin{2};      % pull-out endp
    end;
 else                         % we have vnames, fmt
    vnames = varargin{1};     % pull-out vnames
    fmt = varargin{2};        % pull-out format
    nflag = 1;                % set flag for vnames
 end;
elseif nargs == 3             % begp,endp with either vnames or fmt
 [testf testg] = size(varargin{3});
 if testf == 1                % we have a format
  begp = varargin{1}; endp = varargin{2}; fmt = varargin{3};
 else                         % we have vnames
  nflag = 1; begp = varargin{1}; endp = varargin{2}; vnames = varargin{3};
 end;
elseif nargs == 4             % we have everything
 nflag = 1; begp = varargin{1}; endp = varargin{2};
 vnames = varargin{3}; fmt = varargin{4};
end; % end of input checking
```

Our deductions are based on logic and the use of MATLAB functions **isnumeric** to detect ‘begp’ input arguments and the **size** command to distinguish formats from variable names supplied as input arguments. Needless to say, there are limits to what can be accomplished with this approach depending on the nature of the input arguments. In addition, writing this type of function is more prone to errors than the use of the structure variable as an input argument demonstrated by the **mprint** and **lprint** functions. The advantage is that the user need not define an input structure, but can pass input arguments directly to the function. Passing input arguments directly may be more convenient when typing commands to the MATLAB COMMAND WINDOW, which is likely to be the case when using the **tsprint** function. After implementing time-series data transformations on a matrix, the user may wish to verify the transformations using **tsprint** typed in the COMMAND WINDOW.

It might be of interest that **tsprint** calls the **mprint** function to do the printing. This is done after parsing the input and forming a set of row-names using our **tsdate** function as follows:

```
rnames = 'Date';
for k=begp:endp;
rnames = strvcat(rnames,tsdate(cstruc,k));
end;
in.rnames = rnames;
in.fmt = fmt;
in.cnames = vnames;
mprint(y(begp:endp,:),in);
```

Because **tsprint** relies on **mprint**, the wrapping of time-series matrices containing a large number of columns takes the same format as in **mprint** and **lprint**. The number of columns printed will vary depending on the numeric format used. Example 3.7 illustrates using various numeric formats and the wrapping results.

```
% ----- Example 3.7 Various tsprint() formats
vnames  = strvcat('illinos','indiana','kentucky','michigan','ohio',  ...
                  'pennsylvania','tennessee','west virginia');
dates = cal(1982,1,12);    load test.dat; y = test;
begp = ical(1990,1,dates); endp = ical(1990,12,dates);
fmt = '%16.0f'; tsprint(y,dates,begp,endp,vnames,fmt);
vnames  = strvcat('IL','IN','KY','MI','OH','PA','TN','WV');
fmt = '%6.2f';  tsprint(y,dates,begp,endp,vnames,fmt);
fmt = '%20d';   tsprint(y,dates,begp,endp,vnames,fmt);
```

The output varies due to the different formats and variable names supplied as shown below.

```
Date       illinos        indiana       kentucky       michigan            ohio
Jan90          192             76            348             97             171
Feb90          190             76            350             96             170
Mar90          192             78            356             95             172
Apr90          192             80            357             97             174
May90          194             82            361            104             177
Jun90          199             83            361            106             179
Jul90          200             84            354            105             181
Aug90          200             84            357             84             181
Sep90          199             83            360             83             179
Oct90          199             83            353             82             177
Nov90          197             82            350             81             176
Dec90          196             82            353             96             171
```

Date	pennsylvania	tennessee	west virginia
Jan90	267	61	352
Feb90	266	57	349
Mar90	271	62	350
Apr90	273	62	351
May90	277	63	355
Jun90	279	63	361
Jul90	279	63	359
Aug90	281	63	358
Sep90	280	62	358
Oct90	277	62	357
Nov90	268	61	361
Dec90	264	59	360

Date	IL	IN	KY	MI	OH	PA	TN	WV
Jan90	192.00	76.00	348.00	97.00	171.00	267.00	61.00	352.00
Feb90	190.00	76.00	350.00	96.00	170.00	266.00	57.00	349.00
Mar90	192.00	78.00	356.00	95.00	172.00	271.00	62.00	350.00
Apr90	192.00	80.00	357.00	97.00	174.00	273.00	62.00	351.00
May90	194.00	82.00	361.00	104.00	177.00	277.00	63.00	355.00
Jun90	199.00	83.00	361.00	106.00	179.00	279.00	63.00	361.00
Jul90	200.00	84.00	354.00	105.00	181.00	279.00	63.00	359.00
Aug90	200.00	84.00	357.00	84.00	181.00	281.00	63.00	358.00
Sep90	199.00	83.00	360.00	83.00	179.00	280.00	62.00	358.00
Oct90	199.00	83.00	353.00	82.00	177.00	277.00	62.00	357.00
Nov90	197.00	82.00	350.00	81.00	176.00	268.00	61.00	361.00
Dec90	196.00	82.00	353.00	96.00	171.00	264.00	59.00	360.00

Date	IL	IN	KY	MI
Jan90	192	76	348	97
Feb90	190	76	350	96
Mar90	192	78	356	95
Apr90	192	80	357	97
May90	194	82	361	104
Jun90	199	83	361	106
Jul90	200	84	354	105
Aug90	200	84	357	84
Sep90	199	83	360	83
Oct90	199	83	353	82
Nov90	197	82	350	81
Dec90	196	82	353	96

Date	OH	PA	TN	WV
Jan90	171	267	61	352
Feb90	170	266	57	349
Mar90	172	271	62	350
Apr90	174	273	62	351
May90	177	277	63	355
Jun90	179	279	63	361
Jul90	181	279	63	359
Aug90	181	281	63	358
Sep90	179	280	62	358
Oct90	177	277	62	357
Nov90	176	268	61	361
Dec90	171	264	59	360

A warning about using the **cal** and **tsprint** functions. When you truncate time-series to account for transformations, you should re-set the calendar with another call to the **cal** function based on the new dates associated with the truncated series. Example 3.8 provides an illustration of this.

```
% ----- Example 3.8 Truncating time-series and the cal() function
dates = cal(1982,1,12); load test.dat; y = growthr(test,12);
vnames  = strvcat('IL','IN','KY','MI','OH','PA','TN','WV');
% define beginning and ending print dates
begp = ical(1983,1,dates); endp = ical(1984,12,dates);
tsprint(y,dates,begp,endp,vnames);
ynew = trimr(y,dates.freq,0); % truncate initial observations
tdates = cal(1983,1,12);      % reset the calendar after truncation
% re-define beginning and ending print dates based on the new calendar
begp = ical(1983,1,tdates); endp = ical(1983,12,tdates);
tsprint(ynew,tdates,begp,endp,vnames);
```

In the code above, **growthr** (a function discussed in the next section) transforms data to annual growth rates setting observations for the initial year to zero. The function **trimr** (also discussed in the next section) trims off the rows associated with the initial year, so our data matrix 'ynew' starts in 1983, not 1982 like 'y'.

If you don't take responsibility for doing this, the date labels on your printouts or plots may be incorrect. Some effort has been made to place error-checking in the function **ical** so that you cannot make a call with a 'year' argument that is less than the 'beg_yr' contained in the structure returned by the **cal** function. This should protect against the type of mistake shown in example 3.9.

```
% ----- Example 3.9 Common errors using the cal() function
dates = cal(1982,1,12); % define calendar to start in 1982,1
load test.dat; y = growthr(test,12); y = trimr(y,12,0);
dates = cal(1983,1,12); % re-define calendar to start in 1983,1
% try to define beginning and ending print dates
% forgetting that the calendar now starts in 1983,1
begp = ical(1982,1,dates);
% ical will produce an error message in the above call
```

One nice feature of using the **cal** and **ical** functions is that dates information is documented in the file. It should be fairly clear what the estimation, forecasting, truncation, etc. dates are — just by examining the file.

In addition to printing time-series data we might wish to plot time-series variables. A function **tsplot** was created for this purpose. The usage format is similar to **tsprint** and relies on a structure from **cal**. The documentation for this function is:

```
 PURPOSE: time-series plot with dates and labels
---------------------------------------------------
 USAGE:     tsplot(y,cstruc,begp,endp,vnames)
       or: tsplot(y,cal_struc,vnames), which plots the entire series
       or: tsplot(y,cal_struc), entire series no variable names

 where:  y      = matrix (or vector) of series to be plotted
         cstruc = a structure returned by cal()
         begp   = the beginning observation to plot
         endp   = the ending observation to plot
       vnames  = a string matrix of names for a legend (optional)
---------------------------------------------------
 NOTES:   cstr = cal(1970,1,12);
          tsplot(y,cstr); would plot all data
     or: tsplot(y,cstr,ical(1981,1,cstr),ical(1981,12,cstr)),
          which would plot data for 1981 only
---------------------------------------------------
```

tsplot produces graphs that look like that shown in Figure 3.1. (Colors are used to distinguish the lines in MATLAB). The time-axis labeling is not ideal, but this is due to limitations in MATLAB. The function attempts to distinguish between graphs that involve fewer observations and those that involve a large number of observations. A vertical grid is used for the case where we have a small number of observations.

If you invoke multiple calls to **tsplot**, you are responsible for placing MATLAB **pause** statements as in the case of the **plt** function from the regression library. You can add titles as well as use the MATLAB **subplot** feature with this function. Example 3.10 produces time-series plots using the **subplot** command to generate multiple plots in a single figure.

```
% ----- Example 3.10 Using the tsplot() function
dates = cal(1982,1,12); load test.dat; y = growthr(test,12);
yt = trimr(y,dates.freq,0);
dates = cal(1983,1,12); % redefine the calendar
vnames  = strvcat('IL','IN','KY','MI','OH','PA','TN','WV');
subplot(211), % a plot of the whole time-series sample
tsplot(yt,dates,vnames);
title('Entire data sample'); % add a title to the first plot
subplot(212), % produce a plot showing 1990,1 to 1992,12
% find obs # associated with 1990,1 and 1992,12
begp  = ical(1990,1,dates);  endp  = ical(1992,12,dates);
tsplot(yt,dates,begp,endp,vnames);
xlabel('a sub-sample covering 1990-1992'); % add a label in bottom plot
```

Note, we never really used the 'dates' structure returned by the first call to **cal**. Nonetheless, this command in the file provides documentation that

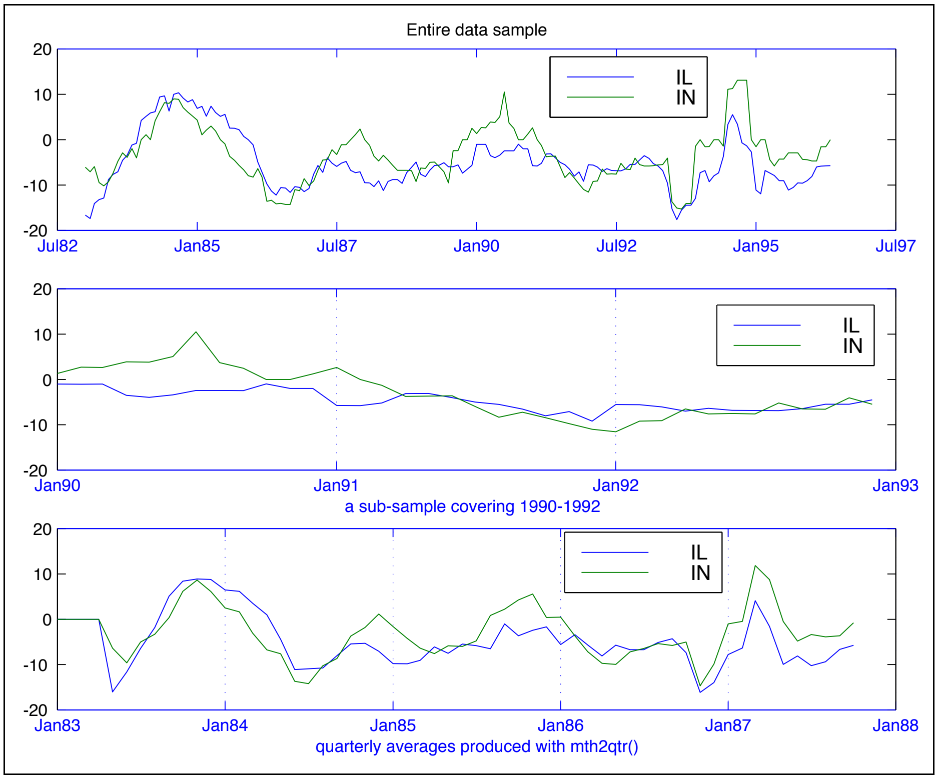

Figure 3.1: Output from tsplot() function

the file 'test.dat' contains monthly data beginning in January, 1982, so it does serve a purpose.

In addition to **tsplot**, there is a function **fturns** that finds turning points in time-series and **plt** that uses the 'result' structure variable from **fturns** to produce a graphical presentation of the turning points.

Zellner, Hong and Gulati (1988), and Zellner and Hong (1988) formulated the problem of forecasting turning points in economic time series using a Bayesian decision theoretic framework. The innovative aspect of the Zellner and Hong study was the use of time-series observations along with an explicit definition of a turning point, either a downturn (DT) or upturn (UT). This allows for a Bayesian computation of probabilities of a DT or UT given the past data and a model's predictive probability density function (pdf) for future observations.

The function **fturns** allows one to define a turning point event based on a sequence of time-series observations, an idea originating with Wecker (1979). For example, let the conditions for a downturn event be:

$$y_{t-k} < \ldots < y_{t-2} < y_{t-1} < y_t \tag{3.1}$$

Which indicates that when we observe a strictly increasing sequence of observations in our time-series, the conditions are right for a downturn event. The relationship between y_t and future values, y_{t+j} is used to determine if a downturn event actually took place. As an example, we might define a downturn event based on the following future sequence:

$$y_{t+j} < \ldots < y_{t+1} < y_t \tag{3.2}$$

This indicates that we define a downturn event at time t as a sequence of future observations falling strictly below the current observation at time t. Our definition is conditional on the initial rising sequence up to time t, followed by falling values after time t. If future values do not fall below those at time t, we define a "no downturn event".

A similar approach can be taken to define an upturn event, for example the necessary conditions for an upturn event might be a sequence of strictly falling time-series observations:

$$y_{t-k} > \ldots > y_{t-2} > y_{t-1} > y_t \tag{3.3}$$

followed by a sequence of strictly rising observations after time t.

$$y_{t+j} > \ldots > y_{t+2} > y_{t+1} > y_t \tag{3.4}$$

As in the case of a downturn event, if the future observations do not rise after time t, we specify a "no upturn event".

The function **fturns** allows the user to define the number of observations to examine prior to and after an upturn and downturn event as well as whether a strict sequence of rising or falling values is to be used. A non-sequential definition of the conditions necessary for a turning point event would be defined as:

$$y_{t-k}, \ldots, y_{t-2}, y_{t-1} > y_t \tag{3.5}$$

The documentation for **fturns** is:

```
 PURPOSE: finds turning points in a time-series
---------------------------------------------------
 USAGE: result = fturns(y,in)
```

```
 where:  y = an (nobs x 1) time-series vector
        in = a structure variable with options
          in.but,  # of down-periods before an upturn  (default = 4)
          in.aut,  # of up-periods   after  an upturn  (default = 1)
          in.bdt,  # of up-periods   before a downturn (default = 4)
          in.adt,  # of down-periods after  a downturn (default = 1)
          in.seq,  1 = sequence inequality,    default = 1
                   0 = simple inequality
          in.eq    1 = inequality with <=, >=, default = 1
                   0 = strict inequality <, >
   e.g. in.seq=0, in.eq=1 in.but = 3, in.aut = 1 would define:
   y(t-3), y(t-2), y(t-1), >= y(t)     [but=3, eq=1, seq=0]
   and y(t+1) > y(t) as UT             [aut=1]
   and y(t+1) <=  y(t) as NUT
   e.g. in.seq=0, in.eq=1 in.bdt = 3, in.adt = 2 would define:
   y(t-3), y(t-2), y(t-1), <= y(t)     [bdt=3, eq=1, seq=0]
   and y(t+2), y(t+1) < y(t) as DT     [adt=2]
   and y(t+2), y(t+1) >= y(t) as NDT
   e.g. in.seq=1, in.eq=1, in.but = 3, in.aut = 1 would define:
   y(t-3) >= y(t-2) >= y(t-1) >= y(t)  [but=3, eq=1, seq=1]
   and y(t+1) > y(t) as UT             [aut=1]
   and y(t+1) <=  y(t) as NUT
   e.g. in.seq=1, in.eq=0, in.bdt = 3, in.adt = 2 would define:
   y(t-3) < y(t-2) < y(t-1) < y(t)     [bdt=3, eq=0, seq=1]
   and y(t+2) > y(t+1) < y(t) as DT    [adt=2]
   and y(t+2) >= y(t+1) >= y(t) as NDT
---------------------------------------------------
 RETURNS:
        results = a structure variable
        results.ut  = (nobs x 1) vector with 1 = UT periods
        results.dt  = (nobs x 1) vector with 1 = DT periods
        results.nut = (nobs x 1) vector with 1 = NUT periods
        results.ndt = (nobs x 1) vector with 1 = NDT periods
        results.y   = time-series vector input
 (NUT = no upturn, NDT = no downturn)
--------------------------------------------------
 SEE ALSO: plt_turns (which will plot turning points)
--------------------------------------------------
```

An example showing various settings for the input structure variable best illustrates the use of the alternative turning point definitions that can be set using the function.

```
% ----- Example 3.11 Finding time-series turning points with fturns()
dates = cal(1982,1,12); load test.dat; y = growthr(test,dates);
yt = trimr(y,dates.freq,0); % truncate initial zeros
tdates = cal(1983,1,12);    % update calendar for truncation
ytime = yt(:,1);            % pull out state 1 time-series
```

```
% DT definition: y(t-4), y(t-3), y(t-2), y(t-1) <= y(t)
% If y(t+1) < y(t)  we have a downturn
in.bdt = 4; in.adt = 1; in.eq = 1; in.seq = 0;
% UT definition: y(t-4), y(t-3), y(t-2), y(t-1) >= y(t)
% If y(t+1) > y(t)  we have an upturn
in.but = 4; in.aut = 1;
results = fturns(ytime,in);  plt(results,tdates,'employment');
title('loose non-sequential definition --- finds lots of turns'); pause;
in.seq = 1; % Now, change to a sequential definition
% DT definition: y(t-4) <= y(t-3) <= y(t-2) <= y(t-1) <= y(t)
% If y(t+1) < y(t)  we have a downturn
% UT definition: y(t-4) >= y(t-3) >= y(t-2) >= y(t-1) >= y(t)
% If y(t+1) > y(t)  we have an upturn
results = fturns(ytime,in); plt(results,tdates,'employment');
title('sequential definition --- produces fewer turns'); pause;
% Now, illustrate requiring many points after the turns
% (should rule out some of the turning points in the above graph)
in.seq = 1; in.aut = 4; % 4 periods of up after an upturn
in.adt = 4;             % 4 periods of down after a downturn
% DT definition: y(t-4), y(t-3), y(t-2), y(t-1) <= y(t)
% If y(t+4) < y(t+3) < y(t+2) < y(t+1) < y(t) we have a downturn
% UT definition: y(t-4), y(t-3), y(t-2), y(t-1) >= y(t)
% If y(t+4) > y(t+3) > y(t+2) > y(t+1) > y(t) we have an upturn
results = fturns(ytime,in);  plt(results,tdates,'employment');
title('sequential aut=4, adt=4 definition --- finds less turns'); pause;
% Now turn off the sequential requirement
% ( should produce more turns that meet this looser definition)
in.seq = 0;
results = fturns(ytime,in);  plt(results,tdates,'employment');
title('non-sequential definition aut=4, adt=4 --- finds more turns'); pause;
```

An example of the graphs produced by submitting the 'results' structure from **fturns** to **plt_turns** or **plt** is shown in Figure 3.2.

3.3 Data transformation utilities

Typical data transformations for time-series analysis involve constructing lagged values of variables, differences, growth-rate transformations and seasonal differencing. A problem encountered with these transformations is that they produce result vectors with missing observations for the initial periods. For example, when constructing a lagged variable, y_{t-1}, we face an initialization problem where the lagged value for the first observation in our sample data represents a period prior to the beginning of the sample.

A design decision must be made regarding how to handle this situation. There are two options, one is to return a lagged variable vector or matrix

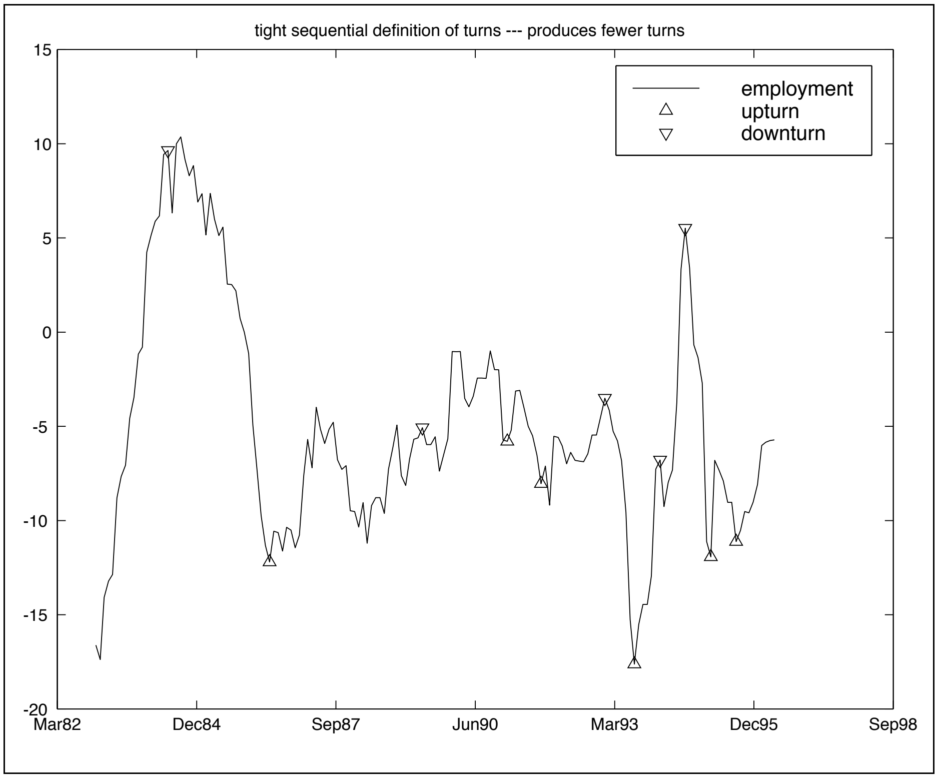

Figure 3.2: Graph of turning point events

that is truncated to "feed the lags". This approach would return different length vectors or matrices depending on the type of transformation and frequency of the data. For example, in the case of seasonal differencing, we loose the first 'freq' observations where freq=4 or 12 depending on whether we are working with quarterly or monthly data. When we consider that after implementing a seasonal differencing transformation, we might then construct lagged values of the seasonally differenced series, it becomes clear that this approach has the potential to raise havoc.

The second design option is to always return the same size vector or matrix that is used as an input argument to the function. This requires that the user take responsibility for truncating the vector or matrix returned by the function. We supply an auxiliary function, **trimr** than can help with the truncation task facing the user.

An example to illustrate using a seasonal differencing function **sdiff** and

trimr to accomplish truncation is:

```
% ----- Example 3.12 Seasonal differencing with sdiff() function
dates = cal(1982,1,12); load test.dat;
ysdiff = sdiff(test,dates);          % seasonally difference the data
ysdiff = trimr(ysdiff,dates.freq,0); % truncate first freq observations
```

The function **trimr** is designed after a similar function from the Gauss programming language, its documentation is:

```
PURPOSE: return a matrix (or vector) x stripped of the specified rows.
-----------------------------------------------------
USAGE: z = trimr(x,n1,n2)
where: x = input matrix (or vector) (n x k)
      n1 = first n1 rows to strip
      n2 = last  n2 rows to strip
NOTE: modeled after Gauss trimr function
-----------------------------------------------------
RETURNS: z = x(n1+1:n-n2,:)
-----------------------------------------------------
```

Note that we utilize our **cal** function structure in the **sdiff** function, but an integer 'freq' argument works as well. That is: **ysdiff = sdiff(test,12);** would return the same matrix of seasonally differenced data as in the example above.

In addition to the **sdiff** function, we create an ordinary differencing function **tdiff**. (Note, we cannot use the name **diff** because there is a MATLAB function named **diff**). This function will produce traditional time-series differences and takes the form:

```
ydiff = tdiff(y,k);
```

which would return the matrix or vector y differenced k times. For example: $k = 1$ produces: $y_t - y_{t-1}$, and $k = 2$ returns: $\Delta^2 y_t = y_t - 2y_{t-1} + y_{t-2}$.

Another time-series data transformation function is the **lag** function that produces lagged vectors or matrices. The format is:

```
xlag1 = lag(x);     % defaults to 1-lag
xlag12 = lag(x,12); % produces 12-period lag
```

where here again, we would have to use the **trimr** function to eliminate the zeros in the initial positions of the matrices or vectors xlag1, and xlag12 returned by the **lag** function.

For VAR models, we need to create matrices containing a group of contiguous lags that are used as explanatory variables in these models. A special lag function **mlag** is used for this purpose. Given a matrix Y_t containing two time-series variable vectors:

$$Y_t = (y_{1t},\ y_{2t}) \tag{3.6}$$

the **mlag** function call: **ylag = mlag(y,k);** will return a matrix 'ylag' equal to:

$$ylag = (y_{1t-1}, y_{1t-2}, \ldots, y_{1t-k}, y_{2t-1}, y_{2t-2}, \ldots, y_{2t-k}) \tag{3.7}$$

which takes the form of a typical set of explanatory variables for these models.

Another data transformation we might wish to employ in time-series analysis is one that converts the time-series to the form of annual growth-rates, i.e., for quarterly time-series, $[(y_t - y_{t-4})/y_{t-4}] * 100$. The function **growthr** carries out this task. Again, we design this function to use either the 'freq' argument or the structure returned by the **cal** function. Examples of using this function would be:

```
% ----- Example 3.13 Annual growth rates using growthr() function
% Using the cal() structure
dates = cal(1982,1,12);    % initialize a calendar
load test.dat;             % load some data
y = growthr(test,dates);   % transform to growth rates
y = trimr(y,dates.freq,0); % truncate the first freq observations

% Using 'freq=12' as an input argument
y = growthr(test,12);      % transform to growth rates
y = trimr(y,12,0);         % truncate the first freq observations
```

Seasonal dummy variables can be created using the **sdummy** function. It returns a matrix of seasonal dummy variables based on the frequency of the time-series. The function call is:

```
% ----- Example 3.14 Seasonal dummy variables using sdummy() function
% Using the cal() structure
dates = cal(1982,1,12); % initialize a calendar
nobs = ical(1995,12,dates);
dummies = sdummy(nobs,dates);
% Using an input 'freq'
freq = 12; dummies = sdummy(nobs,freq); in.fmt = '%6.0f';
mprint(dummies,in);
```

Finally, a function **mth2qtr** converts monthly time-series to quarterly averages or sums. The documentation for the function is:

```
 PURPOSE: converts monthly time-series to quarterly averages
---------------------------------------------------
 USAGE:  yqtr = mth2qtr(ymth,flag)
 where:  ymth = monthly time series vector or matrix (nobs x k)
         flag = 0 for averages (default) and 1 for sums
---------------------------------------------------
 RETURNS: yqtr = quarterly time-series vector or matrix
                 [floor(nobs/3) + 1] in length by k columns
---------------------------------------------------
 NOTES:  the last observation is the actual month or
         the average (sum) of the last 2 months in cases where
         nobs/3 has a remainder
---------------------------------------------------
```

3.4 Gauss functions

There are a great number of Gauss routines in the public domain available on the Internet. To make it easier to convert these functions to MATLAB syntax, we provide a set of functions that operate in an identical fashion to functions by the same name in Gauss.

We have already seen the function **trimr** that behaves in a fashion identical to the function provided in the Gauss programming language. Other functions that work in an identical fashion to Gauss functions of the same name are:

rows - returns the # of rows in a matrix or vector

cols - returns the # of columns in a matrix or vector

trimr - trims rows of a matrix

trimc - trims columns of a matrix

invpd - inverse of a positive-definite matrix

cumprodc - returns cumulative product of each column of a matrix

cumsumc - returns cumulative sum of each column of a matrix

prodc - returns product of each column of a matrix

sumc - returns sum of each column

delif - select matrix values for which a condition is false

selif - select matrix values for which a condition is true

indexcat - extract indices equal to a scalar or an interval

recserar - computes recursive autoregressive series

recsercp - computes recursive series involving products

seqa - produces a sequence of numbers with a beginning and increment

stdc - standard deviations of the columns of a matrix returned as a column vector

matdiv - divides matrices that are not of the same dimension but are row or column compatible. (NOTE: there is no Gauss function like this, but Gauss allows this type of operation on matrices.)

matmul - multiplies matrices that are not of the same dimension but are row or column compatible.

matadd - adds matrices that are not of the same dimension but are row or column compatible.

matsub - divides matrices that are not of the same dimension but are row or column compatible.

To illustrate the use of these functions we provide an example of converting a Gauss function for least-absolute deviations regression to a MATLAB function. The Gauss version of the function is shown below, with line #'s added to facilitate our discussion.

```
/* Least Absolute Deviation
Author: Ron Schoenberg rons@u.washington.edu
Date: May 29, 1995
Provided without guarantee for public non-commercial use.
Gauss code for least absolute deviation
estimation: given y_i and x_i, i = 1,...,n,
estimate b such that \sum |y_i - b*x_i| is minimized.
The following code solves this as a re-iterated weighted least squares
problem where the weights are the inverse of the absolute values of the
residuals.*/
1    rndseed 45456;
2    nobs = 100;
3    numx = 3;
4    b = .5 * ones(numx+1,1);
5    x = ones(nobs,1)~rndn(nobs,numx);
6    y = x * b + rndn(nobs,1)^4; /* 4th power for kurtotic error term */
7    print "LS estimates   " (y / x)';
```

```
8    print;
9    b_old = 0;
10   b_new = b;
11   w = x;
12   do until abs(b_new - b_old) < 1e-3;
13       b_old = b_new;
14       b_new = invpd(w'x)*(w'y);
15       w = x./abs(y - x * b_new);
16   endo;
17   print "LAD estimates  "  b_new';
```

Some things to note about the Gauss code are:

1. use of matrix divisions that are non-conformable are allowed in Gauss if the two arguments are conformable in either the column or row dimensions. This is not the case in MATLAB, so we have a function **matdiv** that carries out these operations. This type of operation occurs on line #15 when scaling the explanatory variables matrix x by the residual vector abs$(y - x * b_{new})$.

2. The function **invpd** on line #14 examines a matrix for positive-definite status and inverts it if positive definite. If it is not positive definite, augmentation of the smallest eigenvalues occurs to make the matrix positive definite, then inversion takes place. We have a MATLAB equivalent function **invpd** to carry out this task, but it should be noted that it most likely doesn't operate the same as the Gauss function.

3. Gauss doesn't require asterisk symbols for all matrix multiplications, so we have statements like $w'x$ and $w'y$ on line #14 that we need to change.

4. Gauss carries out regression estimation using: (y/x) on line #7, which we need to replace with a MATLAB 'backslash' operator.

5. We need to replace the **do until** loop with a **while** loop.

The MATLAB program corresponding to the Gauss program is shown in example 3.15.

```
% ----- Example 3.15 Least-absolute deviations using lad() function
randn('seed',45456); nobs = 100; numx = 3; b = .5 * ones(numx+1,1);
x = [ones(nobs,1) randn(nobs,numx)];
y = x * b + randn(nobs,1).^4; % 4th power for kurtotic error term
fprintf('LS estimates = %12.6f   \n',x\y);
b_old = zeros(numx+1,1); b_new = b; w = x;
```

```
while max(abs(b_new - b_old)) > 1e-3;
    b_old = b_new;
b_new = invpd(w'*x)*(w'*y);
    w = matdiv(x,abs(y - x * b_new));
end;
fprintf('LAD estimates %12.6f \n',b_new);
```

Of course, beyond simply converting the code, we can turn this into a full-blown regression procedure and add it to the regression library. The code for this is shown below.

```
function results = lad(y,x,maxit,crit)
% PURPOSE: least absolute deviations regression
% --------------------------------------------------
% USAGE: results = lad(y,x,itmax,convg)
% where:       y = dependent variable vector (nobs x 1)
%              x = explanatory variables matrix (nobs x nvar)
%          itmax = maximum # of iterations (default=500)
%          convg = convergence criterion (default = 1e-15)
% --------------------------------------------------
% RETURNS: a structure
%        results.meth = 'lad'
%        results.beta  = bhat
%        results.tstat = t-stats
%        results.yhat  = yhat
%        results.resid = residuals
%        results.sige  = e'*e/(n-k)
%        results.rsqr  = rsquared
%        results.rbar  = rbar-squared
%        results.dw    = Durbin-Watson Statistic
%        results.nobs  = nobs
%        results.nvar  = nvars
%        results.y     = y data vector
%        results.iter  = # of iterations
%        results.conv  = convergence max(abs(bnew-bold))
% --------------------------------------------------
% NOTES: minimizes sum(abs(y - x*b)) using re-iterated weighted
%        least-squares where the weights are the inverse of
%        the absolute values of the residuals
if nargin == 2 % error checking on inputs
 crit = 1e-15; maxit = 500;
elseif nargin == 3, crit = 1e-15;
elseif nargin == 4, % do nothing
else, error('Wrong # of arguments to lad');
end;
[nobs nvar] = size(x);
      b_old = zeros(nvar,1); % starting values
      b_new = ones(nvar,1); iter = 0; w = x;
```

```
      conv = max(abs(b_new-b_old));
          while (conv > crit) & (iter <= maxit);
          b_old=b_new;              b_new = invpd(w'*x)*(w'*y);
          resid = (abs(y-x*b_new)); ind = find(resid < 0.00001);
          resid(ind) = 0.00001;     w = matdiv(x,resid);
          iter = iter+1;            conv = max(abs(b_new-b_old));
          end;
results.meth = 'lad'; results.beta = b_new;
results.y = y;        results.nobs = nobs;
results.nvar = nvar;  results.yhat = x*results.beta;
results.resid = y - results.yhat;
sigu = results.resid'*results.resid;
results.sige = sigu/(nobs-nvar);
tmp = (results.sige)*(diag(inv(w'*x)));
results.tstat = results.beta./(sqrt(tmp));
ym = y - ones(nobs,1)*mean(y); rsqr1 = sigu; rsqr2 = ym'*ym;
results.rsqr = 1.0 - rsqr1/rsqr2;  % r-squared
rsqr1 = rsqr1/(nobs-nvar); rsqr2 = rsqr2/(nobs-1.0);
results.rbar = 1 - (rsqr1/rsqr2);  % rbar-squared
ediff = results.resid(2:nobs) - results.resid(1:nobs-1);
results.dw = (ediff'*ediff)/sigu'; % durbin-watson
results.iter = iter; results.conv = conv;
```

3.5 Wrapper functions

When developing a set of econometric estimation routines, it is simplest to develop a corresponding function to print the results structures returned by the estimation functions. For example, if we were developing vector-autoregressive estimation routines, we might devise a function **prt_var** to print the results structures returned by these estimation functions, and have another function **prt_eqs** to print results from simultaneous equations estimation functions, and so on. Of course, this places a burden on the user to remember the function names associated with printing results from each of these different types of estimation procedures. It would be desirable to have a single function, say **prt** that prints results from all econometric estimation and testing functions.

Developing a single function **prt** to do this is inconvenient and becomes unwieldy. Another option is to develop a "wrapper function" named **prt** that examines the 'meth' field of the structure it is passed and then calls the appropriate printing function associated with the method.

As an illustration, consider the following **plt** function that works to call the appropriate plotting function for the user. Note, we should still leave error checking on inputs in the individual functions like **plt_reg**, as some users may directly call these functions in lieu of using the generic **plt** function.

```
function plt(results,vnames)
% PURPOSE: Plots results structures returned by most functions
%          by calling the appropriate plotting function
%---------------------------------------------------
% USAGE: plt(results,vnames)
% Where: results = a structure returned by an econometric function
%        vnames  = an optional vector of variable names
% --------------------------------------------------
% NOTES: this is simply a wrapper function that calls another function
% --------------------------------------------------
% RETURNS:  nothing, just plots the results
% --------------------------------------------------

if ~isstruct(results) % error checking on inputs
error('plt: requires a structure input');
elseif nargin == 3, arg = 3;
elseif nargin == 2, arg = 2;
elseif nargin == 1, arg = 1;
else, error('Wrong # of inputs to plt');
end;
method = results(1).meth;
% call appropriate plotting routine
switch method
case {'arma','hwhite','lad','logit','nwest','ols','olsc','olsar1','olst',...
      'probit','ridge','robust','theil','tobit','tsls'}
     % call plt_reg
     if arg == 1,     plt_reg(results);
     elseif arg == 2, plt_reg(results,vnames);
     end;
case {'thsls','sur'}
     % call prt_eqs
     if arg == 1,     plt_eqs(results);
     elseif arg == 2, plt_eqs(results,vnames);
     end;
case {'var','bvar','rvar','ecm','becm','recm'}
     % call prt_var
     if arg == 1,     plt_var(results);
     elseif arg == 2, plt_var(results,vnames);
     end;
otherwise
error('results structure not known by plt function');
end;
```

Note that the function **plt** contains no code to carry out plotting, it consists of a large switch statement with multiple case arguments that key on the 'meth' field of the structure variable input. This allows you to construct your own plotting functions that plot information contained in 'results structures' and add another 'case' to the large switch statement in the function **plt**. We illustrate how to do this using another wrapper function **prt** below.

In addition to plotting results, many of the econometric estimation and testing methods produce results that are printed using an associated **prt_reg**, **prt_var**, etc., function. A wrapper function named **prt** seems appropriate here as well and we have constructed one as part of the *Econometrics Toolbox*. All functions that provide 'results structures' with information that can be printed have been added to the function **prt**, so a simple call to **prt** with the 'results structure' will provide printed results. As in the case of **plt**, the function **prt** contains no code for actually printing results, it is simply a large 'switch' statement that queries the 'meth' field of the structure variable input and contains many 'case statements' where the appropriate printing functions are called.

All regression results structures and statistical tests results can be printed using **prt(results)**, or **prt(results,vnames,fid)**. This was accomplished by providing a specialized printing function to provide formatted output for each regression or statistical testing function. A call to the specialzed function is then added as a 'case' argument in the wrapper function **prt**. You should of course take the same approach when constructing your own functions.

As an illustration of this, consider the functions **johansen, adf, cadf, phillips** that implement cointegration tests. A function **prt_coint** was constructed that provides nicely formatted printing of the results structures returned by each of these functions. We then add to a case to the switch statement in the **prt** function that looks as follows:

```
case {'johansen','adf','cadf','phillips'}
     % call prt_coint
     if arg == 1
     prt_coint(results);
     elseif arg == 2
     prt_coint(results,vnames);
     elseif arg == 3
     prt_coint(results,vnames,fid);
     else
     prt_coint(results,[],fid);
     end;
```

This ensures that a call to the function **prt** will produce printed results for the results structures produced by these four cointegration testing functions by calling the appropriate **prt_coint** function. It saves users the trouble of knowing or remembering the name of your printing function.

When users submit econometric functions for inclusion in the *Econometrics Toolbox*, they should provide a printing or plotting function where appropriate and I will add a 'case' statement to the **prt** or **plt** function in the toolbox to call your printing or plotting functions.

3.6 Chapter summary

Basic utilities for printing and plotting matrices will be used often, so considerable effort should be directed towards allowing flexible forms of input arguments in these functions. Use of the MATLAB cell-array variable 'varargin' places the burden on the programmer to parse a variable number of input arguments into variables used by the function. This effort seems worthwhile for frequently used basic utility functions.

We also demonstrated using structure variables as input arguments to functions that contain a large number of alternative input options. Default options can be set in the function and user inputs can be parsed easily with the user of the MATLAB **fieldnames** and **strcmp** functions.

To handle time-series dates associated with observations we devised functions that package this information in a structure variable that can be passed as an argument to various time-series related functions. An alternative design would be to use 'global scope' variables that are available in MATLAB. This would eliminate the need to explicitly pass the structure variables as arguments to the functions.

An overall design decision was made to completely avoid the use of global variables in the *Econometrics toolbox*. My experience has been that students can produce the most creative and difficult to detect bugs when global scope variables are at their disposal.

Wrapper functions **prt** and **plt** were described that allow users to provide their own printing and plotting functions which can be accessed by users by simply calling **prt** or **plt** with the 'results' structure returned by the function. This enabled easy extention of the toolbox to include new estimation and statistical testing functions submitted by others that provide printed or plotted output using the single **prt** or **plt** functions.

Chapter 3 Appendix

The utility functions discussed in this chapter (as well as others not discussed) are in a subdirectory **util**.

```
utility function library

 -------- utility functions -------------

accumulate  - accumulates column elements of a matrix
blockdiag   - creates a block diagonal matrix
cal         - associates obs # with time-series calendar
ccorr1      - correlation scaling to normal column length
ccorr2      - correlation scaling to unit column length
cols        - returns the # of columns in a matrix or vector
crlag       - circular lag function
cumprodc    - returns cumulative product of each column of a matrix
cumsumc     - returns cumulative sum of each column of a matrix
delif       - select matrix values for which a condition is false
diagrv      - replaces main diagonal of square matrix with vector
findnear    - finds matrix element nearest a scalar value
fturns      - finds turning-points in a time-series
growthr     - converts time-series matrix to growth rates
ical        - associates time-series dates with obs #
indexcat    - extract indices equal to a scalar or an interval
indicator   - converts a matrix to indicator variables
invccorr    - inverse for ccorr1, ccorr2
invpd       - makes a matrix positive-definite, then inverts
kernel_n    - normal kernal density estimates
lag         - generates a lagged variable vector or matrix
levels      - generates factor levels variable
lprint      - prints a matrix in LaTeX table-formatted form
lprintf     - enhanced lprint function
matadd      - adds non-conforming matrices, row or col compatible.
matdiv      - divides non-conforming matrices, row or col compatible.
matmul      - multiplies non-conforming matrices, row or col compatible.
matsub      - divides non-conforming matrices, row or col compatible.
mlag        - generates a var-type matrix of lags
mode        - calculates the mode of a distribution
```

```
mprint      - prints a matrix
mprint3     - prints coefficient, t-statistics matrices
mth2qtr     - converts monthly to quarterly data
nclag       - generates a matrix of non-contiguous lags
plt         - wrapper function, plots all result structures
prodc       - returns product of each column of a matrix
prt         - wrapper function, prints all result strucutres
recserar    - recursive AR series (like Gauss)
recsercp    - recursive series product (like Gauss)
roundoff    - rounds matrix to fixed number of decimal digits
rows        - returns the # of rows in a matrix or vector
sacf        - sample autocorrelation function estimates
sdiff       - seasonal differencing
sdummy      - generates seasonal dummy variables
selif       - select matrix values for which a condition is true
seqa        - a sequence of numbers with a beginning and increment
shist       - plots spline smoothed histogram
spacf       - sample partial autocorrelation estimates
stdc        - std deviations of columns  returned as a column vector
sumc        - returns sum of each column
tally       - computes frequencies of distinct levels
tdiff       - time-series differencing
trimc       - trims columns of a matrix (or vector) like Gauss
trimr       - trims rows of a matrix (or vector) like Gauss
tsdates     - time-series dates function
tsprint     - print time-series matrix
unsort      - unsorts a sorted vector or matrix
vec         - turns a matrix into a stacked vector
vech        - matrix from lower triangular columns of a matrix
xdiagonal   - spreads x(nxk) out to X(n*n x n*k) diagonal matrix
yvector     - repeats y(nx1) to form Y(n*n x 1)

-------- demonstration programs -------------

cal_d.m     - demonstrates cal function
fturns_d    - demonstrates fturns and plt
ical_d      - demonstrates ical function
lprint_d    - demonstrates lprint function
lprintf_d   - demonstrates lprintf function
mprint_d    - demonstrates mprint function
mprint3_d   - demonstrates mprint3 function
sacf_d      - demonstrates sacf
spacf_d     - demonstrates spacf
tsdate_d    - demonstrates tsdate function
tsprint_d   - demonstrates tsprint function
util_d      - demonstrated some of the utility functions
```

The graphing functions are in a subdirectory **graphs**.

```
graphing function library

 -------- graphing programs ---------------------------

pairs       - scatter plot (uses histo)
plt         - plots all results structures
pltdens     - density plots
tsplot      - time-series graphs

-------- demonstration programs ---------------------------

pairs_d     - demonstrates pairwise scatter
pltdens_d   - demonstrates pltdens
tsplot_d    - demonstrates tsplot

------- support routines used

histo       - used by pairs
plt_turns   - used by plt to plot turning points from fturns function
```

Chapter 4

Regression Diagnostics

This chapter presents functions from a *regression diagnostics library* that can be used to diagnose collinearity and outliers in regression. Two common references to this literature are: *Regression Diagnostics*, Belsley, Kuh and Welsch (1980) and *Residuals and Influence in Regression*, Cook and Weisberg (1982).

The first section of this chapter introduces functions for diagnosing and correcting collinearity problems. The last section discusses functions to detect and correct for outliers and influential observations in regression problems.

4.1 Collinearity diagnostics and procedures

Simply stated, the collinearity problem is that near linear relations among the explanatory variable vectors tends to degrade the precision of the estimated parameters. Degraded precision refers to a large variance associated with the parameter estimates. Why should this be a matter of concern? Quite often the motivation for estimating an economic model is hypothesis testing to draw inferences about values of the model parameters. A large variance, or a lack of precision, may inhibit our ability to draw inferences from the hypothesis tests.

One way to illustrate the increase in dispersion of the least-squares estimates is with a Monte Carlo experiment. We generate a set of y vectors from a model where the explanatory variables are reasonably orthogonal, involving no near linear dependencies. Alternative sets of y vectors are then generated from a model where the explanatory variables become increasingly collinear. An examination of the variances of the $\hat{\beta}$ estimates from these experimental models should illustrate the increase in dispersion of the estimates arising from increasing the severity of the collinear relations between the explanatory variables.

The specific experiment involved using three explanatory variables in a model shown in (4.1).

$$Y = \alpha\iota + \beta X_1 + \gamma X_2 + \theta X_3 + \varepsilon \tag{4.1}$$

Initially, the three explanatory variables X_1, X_2, X_3, were generated as random numbers from a uniform distribution. This ensures that they will be reasonably orthogonal or independent, not involved in any near linear dependencies. We followed a typical Monte Carlo procedure, producing 1000 different y vectors by adding a normally distributed random ε vector to the *same* three fixed X's multiplied times the parameters β, γ, θ, whose values were set to unity.

After estimating the parameters for the 1000 data sets we find the mean and variance of the distribution of the 1000 sets of estimates. The mean and variance of each parameter will characterize the distribution of outcomes from parameter estimation for a model that obeys the Gauss-Markov assumptions and contains no collinearity problems. This also provides a benchmark against which to judge the impact on the distribution of estimates from near linear dependencies that we introduce next.

To create collinear relations we used the scheme shown in (4.2) where we no longer generate the X_2 and X_3 vectors independently.

$$X_2 = X_3 + u \tag{4.2}$$

Instead, we generate the X_2 vector from the X_3 vector with an added random error vector u. Equation (4.2) represents X_2 as a near linear combination of X_3 where the strength of the linear dependency is determined by the size of the u vector. To generate data sets with an increasing amount of collinearity between X_2 and X_3, we adopted the following strategy:

1. First set the variance of the random normal error vector u at 1.0 and generate the X_2 vector from the X_3 vector.

2. Use the three vectors X_1, X_2, X_3 to generate a set of 1000 Y vectors by adding the exact same ε vector that we used in the benchmark generation to these three fixed X's. The virtue of using the ε vector from the benchmark is that, we hold the noise in the data generation process constant. This should provide a ceteris paribus experiment

where the only change between these 1000 Y vectors and those from the benchmark generation is the collinear nature of the X_2 and X_3 vectors.

3. Two additional sets of 1000 Y vectors were generated in the same manner based on the same X_3 and X_1 vectors, but with two new versions of the X_2 vector generated from X_3. The new X_2 vectors were produced by decreasing the variance of the random vector u to 0.5 and 0.1, respectively.

In summary, we have four sets of 1000 Y vectors, one benchmark set, where the three explanatory variables are reasonably independent, and three sets where the collinear relation between the X_2 and X_3 vectors becomes increasingly severe.

The MATLAB code to produce this experiment is:

```
% ----- Example 4.1 Collinearity experiment
n=100; k=4; u1 = randn(n,1); u2=u1*0.5; u3 = u1*0.1;
x1 = [ones(n,1) rand(n,k-1)]; % orthogonal x's
x2 = [x1(:,1:2) x1(:,4)+u1 x1(:,4)]; % collinear set 1
x3 = [x1(:,1:2) x1(:,4)+u2 x1(:,4)]; % collinear set 2
x4 = [x1(:,1:2) x1(:,4)+u3 x1(:,4)]; % collinear set 3
ndraws = 1000; beta = ones(k,1);
bsave1 = zeros(ndraws,k); bsave2 = zeros(ndraws,k);
bsave3 = zeros(ndraws,k); bsave3 = zeros(ndraws,k);
for i=1:ndraws; % do 1000 experiments
e = randn(n,1);
y = x1*beta + e; res = ols(y,x1); b1save(i,:) = res.beta';
y = x2*beta + e; res = ols(y,x2); b2save(i,:) = res.beta';
y = x3*beta + e; res = ols(y,x3); b3save(i,:) = res.beta';
y = x4*beta + e; res = ols(y,x4); b4save(i,:) = res.beta';
end;
% compute means and std deviations for betas
mtable = zeros(4,k);        stable = zeros(4,k);
mtable(1,:) = mean(b1save); mtable(2,:) = mean(b2save);
mtable(3,:) = mean(b3save); mtable(4,:) = mean(b4save);
stable(1,:) = std(b1save);  stable(2,:) = std(b2save);
stable(3,:) = std(b3save);  stable(4,:) = std(b4save);
% print tables
in.cnames = strvcat('alpha','beta','gamma','theta');
in.rnames = strvcat('beta means','bench','sigu=1.0','sigu=0.5','sigu=0.1');
in.fmt = '%10.4f';
mprint(mtable,in);
in.rnames = strvcat('stand dev','bench','sigu=1.0','sigu=0.5','sigu=0.1');
mprint(stable,in);
```

The results of the experiment showing both the means and standard deviations from the distribution of estimates are:

beta means	alpha	beta	gamma	theta
benchmark	1.0033	1.0027	1.0047	0.9903
sigu=1.0	1.0055	1.0027	1.0003	0.9899
sigu=0.5	1.0055	1.0027	1.0007	0.9896
sigu=0.1	1.0055	1.0027	1.0034	0.9868

standard dev	alpha	beta	gamma	theta
benchmark	0.3158	0.3285	0.3512	0.3512
sigu=1.0	0.2697	0.3286	0.1025	0.3753
sigu=0.5	0.2697	0.3286	0.2049	0.4225
sigu=0.1	0.2697	0.3286	1.0247	1.1115

A first point to note about the experimental outcomes is that the means of the estimates are unaffected by the collinearity problem. Collinearity creates problems with regard to the variance of the distribution of the estimates, not the mean. A second point is that the benchmark data set produced precise estimates, with standard deviations for the distribution of outcomes around 0.33. These standard deviations would result in t-statistics around 3, allowing us to infer that the true parameters are significantly different from zero.

Turning attention to the standard deviations from the three collinear data sets we see a clear illustration that increasing the severity of the near linear combination between X_2 and X_3 produces an increase in the standard deviation of the resulting distribution for the γ and θ estimates associated with X_2 and X_3. The increase is about three-fold for the worse case where $\sigma_u^2 = 0.1$ and the strength of the collinear relation between X_2 and X_3 is the greatest.

A diagnostic technique presented in Chapter 3 of *Regression Diagnostics* by Belsley, Kuh, and Welsch (1980) is implemented in the function **bkw**. The diagnostic is capable of determining the number of near linear dependencies in a given data matrix X, and the diagnostic identifies which variables are involved in each linear dependency. This diagnostic technique is based on the Singular Value Decomposition that decomposes a matrix $X = UDV'$, where U contains the eigenvectors of X and D is a diagonal matrix containing eigenvalues.

For diagnostic purposes the singular value decomposition is applied to the variance-covariance matrix of the least-squares estimates and rearranged to form a table of *variance-decomposition proportions.* The procedure for a k variable least-squares model is described in the following. The variance of the estimate $\hat{\beta}_k$ can be expressed as shown in (4.3).

$$\text{var}(\hat{\beta}_k) = \hat{\sigma}_\varepsilon^2 \sum_{j=1}^{k} (V_{kj}^2/\lambda_j^2) \tag{4.3}$$

The diagnostic value of this expression lies in the fact that it decomposes $\text{var}(\hat{\beta}_k)$ into a sum of components, each associated with one of the k singular values λ_j^2 that appear in the denominator. Expression (4.4) expands the summation in (4.3) to show this more clearly.

$$\text{var}(\hat{\beta}_k) = \hat{\sigma}_\varepsilon^2 \{(V_{11}^2/\lambda_1) + (V_{12}^2/\lambda_2) + (V_{13}^2/\lambda_3) + \ldots + (V_{1k}^2/\lambda_k)\} \tag{4.4}$$

Since small λ_j are associated with near linear dependencies, an unusually large proportion of the variance of the coefficients of variables involved in the linear dependency will be concentrated in the components associated with the small λ_j. The table is formed by defining the terms ϕ and π shown in (4.5) and (4.6).

$$\begin{aligned} \phi_{ij} &= (V_{ij}^2/\lambda_j^2) \\ \phi_i &= \sum_{j=1}^{k} \phi_{ij}, \quad i = 1, \ldots, k \end{aligned} \tag{4.5}$$

$$\pi_{ji} = (\phi_{ij}/\phi_i), \quad i, j = 1, \ldots, k \tag{4.6}$$

The term π_{ji} is called a *variance-decomposition proportion*. These magnitudes are placed in a table as shown in Table 3.1.

Table 4.1: Variance-decomposition proportions table

Condition index	$\text{var}(\hat{\beta}_1)$	$var(\hat{\beta}_2)$	$\ldots$	$var(\hat{\beta}_k)$
$\lambda_{max}/\lambda_{max}$	π_{11}	π_{12}	$\ldots$	π_{1k}
λ_{max}/λ_2	π_{21}	π_{22}	$\ldots$	π_{2k}
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$\lambda_{max}/\lambda_{min}$	π_{k1}	π_{k2}	$\ldots$	π_{kk}

It is shown in Belsley, Kuh and Welsch (1980) that a large value of the *condition index*, $\kappa(X) = \lambda_{max}/\lambda_i$ is associated with each near linear dependency, and the variates involved in the dependency are those with large proportions of their variance associated with large $\kappa(X)$ magnitudes. Empirical tests performed in Chapter 3 of Belsley, Kuh, and Welsch (1980) determined that variance-decomposition proportions in excess of 0.5 indicate the variates involved in specific linear dependencies. The joint condition of magnitudes for $\kappa(X) > 30$, and π_{ij} values > 0.5 diagnose the presence of strong collinear relations as well as determining the variates involved.

Table 3.2 shows an example of how the variance-decomposition proportions table might look for an actual data set. The example in the table would indicate that there exists one condition index, $\kappa(X)$, of 87 and another of 98. For these two condition indices we examine the variance proportions looking for those that exceed 0.5. We find that for $\kappa(X) = 87$, two variance-proportions exceed 0.5 pointing towards a near linear relationship between the x_1 and x_5 variable. The $\kappa(X) = 98$ also contains two variance-proportions that exceed 0.5 indicating the presence of a collinear relation involving x_2 and x_6. From Table 3.2 then we would conclude that two possible near linear relations existed in the data set.

Table 4.2: BKW collinearity diagnostics example

$\kappa(X)$	x_1	x_2	x_3	x_4	x_5	x_6
1	.00	.00	.01	.01	.00	.00
28	.02	.03	.07	.03	.04	.03
58	.04	.00	.78	.02	.28	.01
60	.01	.02	.03	.76	.02	.22
87	.58	.28	.04	.10	.55	.07
98	.36	.66	.07	.09	.11	.67

A function **bkw** implements these tests, with the documentation shown below:

```
 PURPOSE: computes and prints BKW collinearity diagnostics
          variance-decomposition proportions matrix
---------------------------------------------------
 USAGE: bkw(x,vnames,fmt)
 where:       x = independent variable matrix (from a regression model)
         vnames = (optional) variable name vector
            fmt = (optional) format string, e.g., '%12.6f' or '%12d'
                  default = %10.2f
---------------------------------------------------
 NOTE: you can use either x-variable names or an ols
       vnames argument containing y-variable + x-variable names
 e.g. vnames = strvcat('y','x1','x2') or
```

```
      vnames = strvcat('x1','x2')
 ---------------------------------------------------
 RETURNS:
        nothing, just prints the table out
 ---------------------------------------------------
 SEE ALSO: dfbeta, rdiag, diagnose
 ---------------------------------------------------
 REFERENCES: Belsley, Kuh, Welsch, 1980 Regression Diagnostics
 ----------------------------------------------------
```

The function allows a variable name vector and format as optional inputs. As a convenience, either a variable name vector with names for the variables in the data matrix X or one that includes a variable name for y as well as the variables in X can be used. This is because the **bkw** function is often called in the context of regression modeling, so we need only construct a single variable name string vector that can be used for printing regression results as well as labelling variables in the **bkw** output.

As an example of using the **bkw** function to carry out tests for collinearity, the program below generates a collinear data set and and uses the **bkw** function to test for near linear relationships.

```
% ----- Example 4.2 Using the bkw() function
n = 100; k = 5;
x = randn(n,k);
% generate collinear data
x(:,1) = x(:,2) + x(:,4) + randn(n,1)*0.1;
bkw(x);
```

The results of the program are shown below. They detect the near linear relationship between variables 1, 2 and 4 which we generated in the data matrix X.

```
Belsley, Kuh, Welsch Variance-decomposition
K(x)      var 1      var 2      var 3      var 4      var 5
 1         0.00       0.00       0.00       0.00       0.00
15         0.00       0.00       0.26       0.00       0.40
17         0.00       0.00       0.24       0.00       0.00
20         0.00       0.00       0.47       0.00       0.59
31         1.00       0.99       0.03       0.99       0.01
```

A common corrective procedure for this problem is ridge regression, which is implemented by the function **ridge**. Ridge regression attacks the problem of small eigenvalues in the $X'X$ matrix by augmenting or inflating the smallest values to create larger magnitudes. The increase in small eigenvalues is accomplished by adding a diagonal matrix γI_k to the $X'X$

matrix before inversion. The scalar term γ is called the 'ridge' parameter. The ridge regression formula is shown in (4.7).

$$\hat{\beta}_R = (X'X + \gamma I_k)^{-1}X'y \tag{4.7}$$

Recall that $X'X$ is of dimension (k x k), where k is the number of explanatory variables in the model. Of the k eigenvalues associated with $X'X$, the smallest are presumably quite small as a result of the collinear relationship between some of the explanatory variables. To see how addition of the diagonal matrix γI_k to the $X'X$ matrix increases the smallest eigenvalues, consider using the singular value decomposition of $X'X$. This allows us to rewrite (4.7) as:

$$\hat{\beta}_R = (V'DV + \gamma I_k)^{-1}X'y \tag{4.8}$$

Since γI_k is a diagonal matrix, containing zeros on the off-diagonal elements, adding this to the $V'DV$ matrices will only affect the elements of the diagonal matrix D. Noting this, we find that the ridge estimation formula can be written as in (4.9)

$$\hat{\beta}_R = (V'(D + \gamma I_k)V)^{-1}X'y \tag{4.9}$$

The diagonal matrix D from the Singular Value Decomposition contains the eigenvalues of the $X'X$ matrix, and equation (4.9) shows that the ridge parameter γ is added to each and every eigenvalue on the diagonal of the matrix D. An expansion of the matrix $(D + \gamma I_k)$ shown in (4.10) should make this clear.

$$(D + \gamma I_k) = \begin{pmatrix} \lambda_1 + \gamma & 0 & 0 & \ldots & 0 \\ 0 & \lambda_2 + \gamma & 0 & \ldots & 0 \\ \vdots & & & \ddots & \vdots \\ 0 & 0 & & & \lambda_k + \gamma \end{pmatrix} \tag{4.10}$$

To illustrate how addition of the γ parameter to the eigenvalues impacts the estimates, consider the following numerical example. The Monte Carlo experiment for the strongest collinear relationship (where $\sigma_U^2 = 0.1$) produced the eigenvalues shown in Table 3.3

There are two things to note concerning the impact of the addition of a $\gamma = 0.2$ value to the $X'X$ matrix. First, the large eigenvalues are not affected very much, since we are adding a small value to these large magnitudes. Second, the smallest eigenvalue will be increased dramatically, from 0.0456 to 0.2456. The impact of adding the small value to the eigenvalues

Table 4.3: Ridge Regression for our Monte Carlo example

Eigenvalues of $X'X$	Condition Index $X'X$	Eigenvalues of $(X'X + 0.2I_k)$	Condition Index $(X'X + 0.2I_k)$
201.2950	1.0	201.4950	1.0
10.0227	20.0	10.2227	19.7
5.6565	35.5	5.8565	34.4
0.0456	4414.3	0.2456	820.4

of the original data matrix X is to dramatically improve the conditioning of the estimation problem. In the example shown in Table 3.3, the condition index of the $X'X$ matrix falls dramatically to 820.4, producing a condition index for the X matrix equal to the square root of 820.4, which equals 28.6. (Note, the eigenvalues of the $X'X$ matrix are the square of those from the X matrix).

As an example, consider the following MATLAB program that generates a collinear data set and produces both least-squares and ridge regression estimates. The ridge estimates are produced using a value recommended by Hoerl and Kennard (1970), but the user has an option of entering a value for the ridge parameter as well.

```
% ----- Example 4.3 Using the ridge() function
n = 100; k = 5;
x = randn(n,k); e = randn(n,1); b = ones(k,1);
% generate collinear data
x(:,1) = x(:,2) + x(:,4) + randn(n,1)*0.1;
y = x*b + e;
% ols regression
res = ols(y,x);
prt(res);
% ridge regression
rres = ridge(y,x);
prt(rres);
```

The results from **ols** and **ridge** estimation are shown below. From these results we see that the near linear relationship between variables x_1, x_2 and x_4 lead to a decrease in precision for these estimates. The ridge estimates increase the precision as indicated by the larger $t-$statistics.

```
Ordinary Least-squares Estimates
R-squared      =    0.9055
```

```
Rbar-squared   =    0.9015
sigma^2        =    0.9237
Durbin-Watson  =    1.6826
Nobs, Nvars    =    100,     5
***************************************************************
Variable        Coefficient      t-statistic    t-probability
variable 1        -0.293563        -0.243685         0.808000
variable 2         2.258060         1.871842         0.064305
variable 3         1.192133        11.598932         0.000000
variable 4         2.220418         1.796384         0.075612
variable 5         0.922009         8.674158         0.000000

Ridge Regression Estimates
R-squared      =    0.9051
Rbar-squared   =    0.9011
sigma^2        =    0.9272
Durbin-Watson  =    1.6676
Ridge theta    =     0.0022705069
Nobs, Nvars    =    100,     5
***************************************************************
Variable        Coefficient      t-statistic    t-probability
variable 1         0.430781         0.552496         0.581906
variable 2         1.531597         1.956202         0.053378
variable 3         1.186535        11.541312         0.000000
variable 4         1.476339         1.841555         0.068661
variable 5         0.900636         8.680297         0.000000
```

A point to note about ridge regression is that it does not produce unbiased estimates. The amount of bias in the estimates is a function of how large the value of the ridge parameter γ is. Larger values of γ lead to improved precision in the estimates — at a cost of increased bias.

A function **rtrace** helps to assess the trade-off between bias and efficiency by plotting the ridge estimates for a range of alternative values of the ridge parameter. The documentation for **rtrace** is:

```
 PURPOSE: Plots ntheta ridge regression estimates
---------------------------------------------------
 USAGE: rtrace(y,x,thetamax,ntheta,vnames)
 where: y        = dependent variable vector
        x        = independent variables matrix
        thetamax = maximum ridge parameter to try
        ntheta   = number of values between 0 and thetamax to try
        vnames   = optional variable names vector
                   e.g. vnames = strvcat('y,'x1','x2');
---------------------------------------------------
 RETURNS:
        nothing, plots the parameter estimates as a function
```

```
       of the ntheta alternative ridge parameter values
 --------------------------------------------------
```

As an example of using this function, consider the following program where we recover the ridge parameter determined using the Hoerl and Kennard formula, double it and produce a trace plot using values between $\theta = 0$ and 2θ. A value of $\theta = 0$ represents least-squares estimates, and 2θ is twice the value we found using the Hoerl-Kennard formula.

```
% ----- Example 4.4 Using the rtrace() function
n = 100; k = 5;
x = randn(n,k); e = randn(n,1); b = ones(k,1);
% generate collinear data
x(:,1) = x(:,2) + x(:,4) + randn(n,1)*0.1;
y = x*b + e;
% ridge regression
res = ridge(y,x);
theta = res.theta;
tmax = 2*theta;
ntheta = 50;
vnames = strvcat('y','x1','x2','x3','x4','x5');
rtrace(y,x,tmax,ntheta,vnames);
```

To the extent that the parameter values vary greatly from those associated with values of $\theta = 0$, we can infer that a great deal of bias has been introduced by the ridge regression. A graph produced by **rtrace** is shown in Figure 4.1, indicating a fair amount of bias associated with 3 of the 5 parameters in this example.

Another solution for the problem of collinearity is to use a Bayesian model to introduce prior information. A function **theil** produces a set of estimates based on the 'mixed estimation' method set forth in Theil and Goldberger (1961). The documentation for this function is:

```
 PURPOSE: computes Theil-Goldberger mixed estimator
          y = X B + E, E = N(0,sige*IN)
          c = R B + U, U = N(0,v)
 ---------------------------------------------------
 USAGE: results = theil(y,x,c,R,v)
 where: y    = dependent variable vector
        x    = independent variables matrix of rank(k)
        c    = a vector of prior mean values, (c above)
        R    = a matrix of rank(r)           (R above)
        v    = prior variance-covariance      (var-cov(U) above)
 ---------------------------------------------------
 RETURNS: a structure:
          results.meth  = 'theil'
```

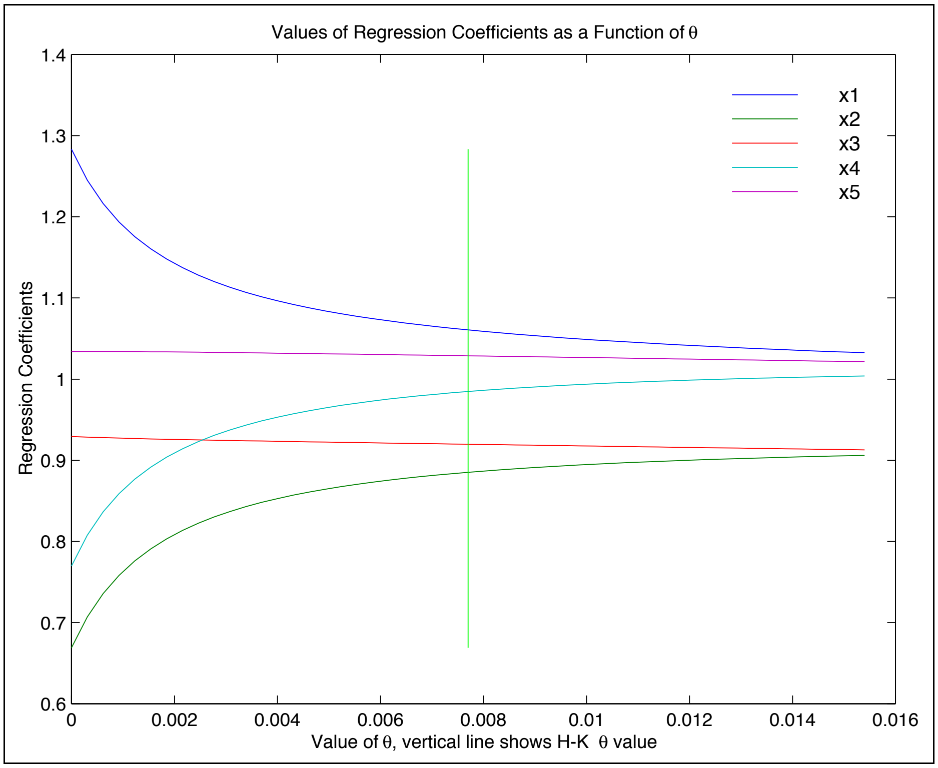

Figure 4.1: Ridge trace plot

```
         results.beta  = bhat estimates
         results.tstat = t-statistics
         results.pmean = prior means
         results.pstd  = prior std deviations
         results.yhat  = predicted values
         results.resid = residuals
         results.sige  = e'e/(n-k)
         results.rsqr  = r-squared
         results.rbar  = r-squared adjusted
         results.dw    = Durbin Watson
         results.nobs  = # of observations
         results.nvar  = # of variables
         results.y     = actual observations
 --------------------------------------------------
 SEE ALSO: prt, plt, ols_g
---------------------------------------------------
```

The user can specify subjective prior information in the form of a normal prior for the parameters β in the model. Theil and Goldberger showed that this prior information can be expressed as stochastic linear restrictions taking the form:

$$c = R\beta + u, \tag{4.11}$$

These matrices are used as additional *dummy or fake* data observations in the estimation process. The original least-squares model in matrix form can be rewritten as in (4.12) to show the role of the matrices defined above.

$$\begin{bmatrix} y \\ \ldots \\ c \end{bmatrix} = \begin{bmatrix} X \\ \ldots \\ R \end{bmatrix} \beta + \begin{bmatrix} \varepsilon \\ \ldots \\ u \end{bmatrix} \tag{4.12}$$

The partitioning symbol, $(\ldots)$, in the matrices and vectors of (4.12) designates that we are adding the matrix R and the vectors c and u to the original matrix X and vectors y and ε. These additional observations make it clear we are augmenting the *weak* sample data with our prior information. At this point we use an OLS estimation algorithm on the modified y vector and X matrix to arrive at the "mixed estimates". One minor complication arises here however, the theoretical disturbance vector no longer consists of the simple ε vector which obeys the Gauss-Markov assumptions, but has an additional u vector added to it. We need to consider the variance-covariance structure of this new disturbance vector which requires a minor modification of the OLS formula producing the resulting "mixed estimator" shown in (4.13).

$$\hat{\beta}_M = (\sigma_\varepsilon^{-2} X'X + R'\Sigma_u^{-1} R)^{-1} (\sigma_\varepsilon^{-2} X'y + R'\Sigma_u^{-1} c) \tag{4.13}$$

Where Σ_u in (4.13) represents a diagonal matrix containing the variances $\sigma_{ui}^2, i = 1, \ldots, k$ on the diagonal.

As an illustration, consider using a normal prior centered on the true parameter values of unity in our previous example. Prior variances of unity are also assigned, indicating a fairly large amount of uncertainty in our prior beliefs. The following program sets up the prior and calls **theil** to estimate the model.

```
% ----- Example 4.5 Using the theil() function
n = 100; k = 5;
x = randn(n,k); e = randn(n,1); b = ones(k,1);
% generate collinear data
x(:,1) = x(:,2) + x(:,4) + randn(n,1)*0.1;
y = x*b + e;
```

```
% set up prior information
c = ones(k,1); % prior means
sigu = eye(k); % prior variances
R = eye(k);
reso = ols(y,x); % ols
prt(reso);
res = theil(y,x,c,R,sigu); %theil
prt(res);
```

This program produced the following results indicating that use of prior information improved the precision of the estimates compared to the least-squares estimates.

```
Ordinary Least-squares Estimates
R-squared      =    0.9202
Rbar-squared   =    0.9168
sigma^2        =    1.0139
Durbin-Watson  =    1.9728
Nobs, Nvars    =    100,     5
***************************************************************
Variable        Coefficient      t-statistic    t-probability
variable 1         0.445405         0.486733         0.627569
variable 2         1.470548         1.575896         0.118374
variable 3         1.280441        12.802818         0.000000
variable 4         1.451183         1.569624         0.119828
variable 5         0.864678         7.908782         0.000000

Theil-Goldberger Regression Estimates
R-squared      =    0.9200
Rbar-squared   =    0.9167
sigma^2        =    1.0155
Durbin-Watson  =    1.9742
Nobs, Nvars    =    100,     5
***************************************************************
Variable         Prior Mean    Std Deviation
variable 1         1.000000         1.000000
variable 2         1.000000         1.000000
variable 3         1.000000         1.000000
variable 4         1.000000         1.000000
variable 5         1.000000         1.000000
***************************************************************
      Posterior Estimates
Variable        Coefficient      t-statistic    t-probability
variable 1         0.795158         1.619936         0.108561
variable 2         1.115213         2.215848         0.029090
variable 3         1.280282        12.789435         0.000000
variable 4         1.098660         2.202581         0.030043
variable 5         0.863902         7.899374         0.000000
```

4.2 Outlier diagnostics and procedures

Outlier observations are known to adversely impact least-squares estimates because the aberrant observations generate large errors. The least-squares criterion is such that observations associated with large errors receive more weight or exhibit more influence in determining the estimates of β.

A number of procedures have been proposed to diagnose the presence of outliers and numerous alternative estimation procedures exist that attempt to "robustify" or downweight aberrant or outlying observations. Function **dfbeta** produces a set of diagnostics discussed in Belsley, Kuh and Welsch (1980). They suggest examining the change in least-squares estimates $\hat{\beta}$ that arise when we omit each observation from the regression sample sequentially. The basic idea is that eliminating an influential observation will produce a large change in the $\hat{\beta}$ estimates, allowing us to detect these observations graphically.

The function **dfbeta** returns a structure that can be used to produce graphical output with **plt_dfb**, **plt_dff** or **plt**. The function documentation is:

```
 PURPOSE: computes BKW (influential observation diagnostics)
          dfbetas, dffits, hat-matrix, studentized residuals
---------------------------------------------------
 USAGE: result = dfbeta(y,x)
 where: y = dependent variable vector (from a regression model)
        x = independent variable matrix
---------------------------------------------------
 RETURNS: a structure
        results.meth   = 'dfbeta'
        results.dfbeta = df betas
        results.dffits = df fits
        results.hatdi  = hat-matrix diagonals
        results.stud   = studentized residuals
        results.nobs   = # of observations
        results.nvar   = # of variables in x-matrix
 --------------------------------------------------
 SEE ALSO: plt_dfb, plt_dff,  bkw
---------------------------------------------------
```

An example where we generate a data set and then artificially create two outliers at observations #50 and #70 is shown below. The graphical output from **plt_dfb** in Figure 4.2 shows a graph of the change in $\hat{\beta}$ associated with omitting each observation. We see evidence of the outliers at observations #50 and #70 in the plot.

```
% ----- Example 4.6 Using the dfbeta() function
n = 100; k = 4;
x = randn(n,k); e = randn(n,1);  b = ones(k,1);
y = x*b + e;
% now add a few outliers
y(50,1) = 10.0; y(70,1) = -10.0;
vnames = strvcat('y','x1','x2','x3','x4');
res = dfbeta(y,x);
plt_dfb(res,vnames);
pause;
plt_dff(res);
```

Figure 4.3 shows another diagnostic 'dffits' produced by the function **dfbeta** that indicates how the fitted values change when we sequentially eliminate each observation from the regression data set. A similar function **diagnose** computes many of the traditional statistics from the regression diagnostics literature and prints this information for candidate outlier observations.

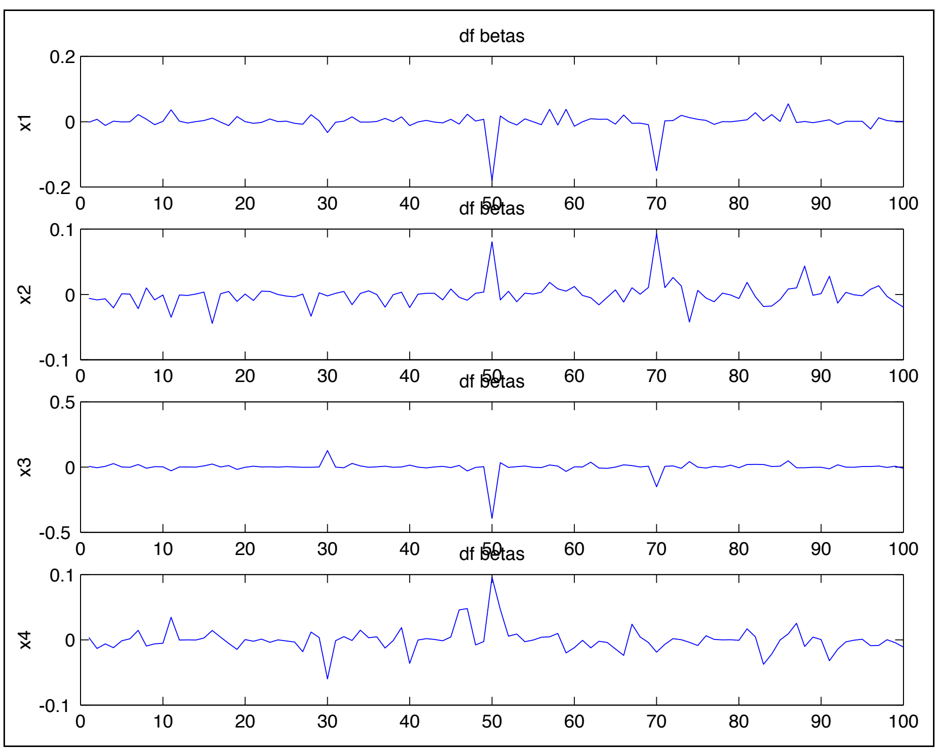

Figure 4.2: Dfbeta plots

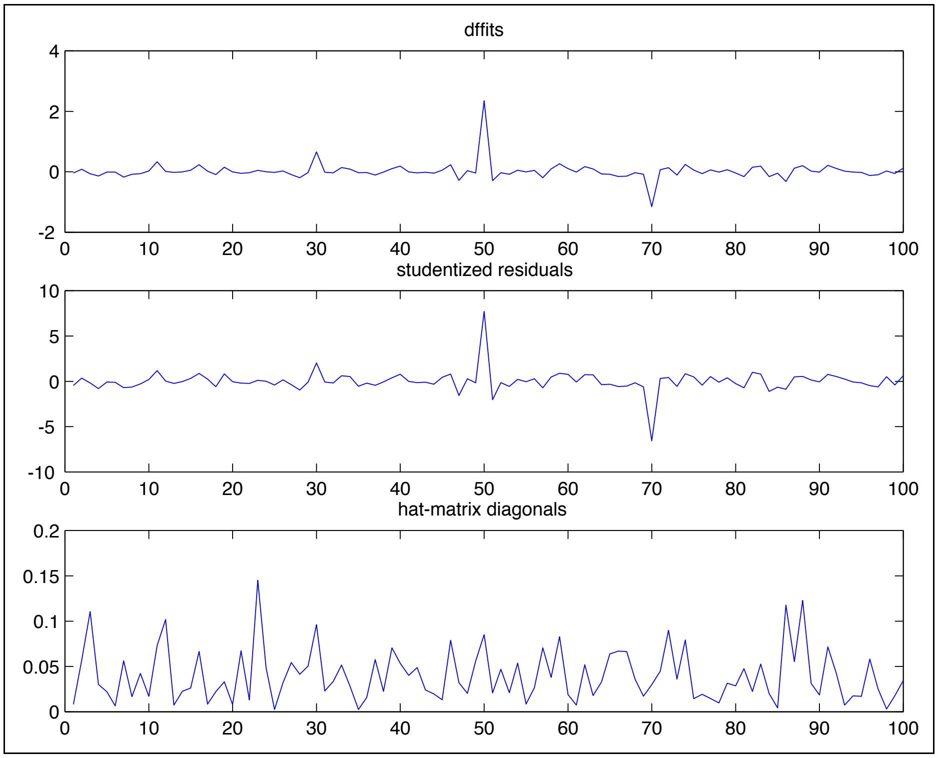

Figure 4.3: Dffits plots

Another routine, **rdiag**, produces graphical diagnostics for the residuals taking the form of 4 plots. A normal plot of the residuals, an I-chart of the residuals, a histogram of the residuals and the residuals versus the fitted values.

A number of alternative estimation methods exist that attempt to downweight outliers. The regression library contains a function **robust** and **olst** as well as **lad** that we developed in Chapter 3. The documentation for **robust** is:

```
 PURPOSE: robust regression using iteratively reweighted
          least-squares
---------------------------------------------------
 USAGE: results = robust(y,x,wfunc,wparm)
 where: y = dependent variable vector (nobs x 1)
        x = independent variables matrix (nobs x nvar)
    wfunc = 1 for Huber's t function
            2 for Ramsay's E function
```

```
            3 for Andrew's wave function
            4 for Tukey's biweight
    wparm = weighting function parameter
---------------------------------------------------
 RETURNS: a structure
        results.meth  = 'robust'
        results.beta  = bhat
        results.tstat = t-stats
        results.yhat  = yhat
        results.resid = residuals
        results.sige  = e'*e/(n-k)
        results.rsqr  = rsquared
        results.rbar  = rbar-squared
        results.dw    = Durbin-Watson Statistic
        results.iter  = # of iterations
        results.nobs  = nobs
        results.nvar  = nvars
        results.y     = y data vector
        results.wfunc = 'huber', 'ramsay', 'andrew', 'tukey'
        results.wparm = wparm
        results.weight = nobs - vector of weights
```

The function incorporates four alternative weighting schemes that have been proposed in the literature on iteratively re-weighted regression methods. As an illustration of all four methods, consider the following program that produces estimates using all methods.

```
% ----- Example 4.7 Using the robust() function
nobs = 100; nvar = 3; x = randn(nobs,nvar); x(:,1) = ones(nobs,1);
beta = ones(nvar,1);  evec = randn(nobs,1); y = x*beta + evec;
y(75,1) = 10.0; y(90,1) = -10.0; % put in 2 outliers
% parameter weighting from OLS (of course you're free to do differently)
reso = ols(y,x); sige = reso.sige;
bsave = zeros(nvar,5); bsave(:,1) = ones(nvar,1); % storage for results
for i=1:4; % loop over all methods producing estimates
 wfunc = i; wparm = 2*sige; % set weight to 2 sigma
 res = robust(y,x,wfunc,wparm);
 bsave(:,i+1) = res.beta;
end;
in.cnames = strvcat('Truth','Huber t','Ramsay','Andrews','Tukey');
in.rnames = strvcat('Coefficients','beta1','beta2','beta3');
in.fmt = '%10.4f';
mprint(bsave,in);
res = robust(y,x,4,2);
prt(res); plt(res); % demonstrate prt and plt functions
```

Note that we can use our function **mprint** from Chapter 3 to produce a formatted printout of the results that looks as follows:

```
Coefficients      Truth    Huber t     Ramsay    Andrews      Tukey
beta1            1.0000     1.1036     0.9529     1.0985     1.0843
beta2            1.0000     0.9996     1.1731     0.9987     0.9253
beta3            1.0000     1.1239     0.9924     1.1232     1.0650
```

The routine **olst** performs regression based on an assumption that the errors are $t-$distributed rather than normal, which allows for "fat-tailed" error distributions. The documentation is:

```
PURPOSE: ols with t-distributed errors
--------------------------------------------------
USAGE: results = olst(y,x,itmax,convg)
where:       y = dependent variable vector (nobs x 1)
             x = explanatory variables matrix (nobs x nvar)
         itmax = maximum # of iterations (default=500)
         convg = convergence criterion (default = 1e-8)
--------------------------------------------------
RETURNS: a structure
       results.meth = 'olst'
       results.beta  = bhat
       results.tstat = t-stats
       results.yhat  = yhat
       results.resid = residuals
       results.sige  = e'*e/(n-k)
       results.rsqr  = rsquared
       results.rbar  = rbar-squared
       results.dw    = Durbin-Watson Statistic
       results.nobs  = nobs
       results.nvar  = nvars
       results.y     = y data vector
       results.iter  = # of iterations
       results.conv  = convergence max(abs(bnew-bold))
--------------------------------------------------
NOTES: uses iterated re-weighted least-squares
       to find maximum likelihood estimates
--------------------------------------------------
SEE ALSO: prt, plt
---------------------------------------------------
REFERENCES: Section 22.3 Introduction to the Theory and Practice
            of Econometrics, Judge, Hill, Griffiths, Lutkepohl, Lee
```

Another graphical tool for regression diagnostics is the **pairs** function that produces pairwise scatterplots for a group of variables as well as histograms of the distribution of observations for each variable. This function is a modified version of a function by Anders Holtsberg's public domain statistics toolbox. The documentation was altered to conform to that for other functions in the *Econometrics Toolbox* and a variable names capability was added to the function. The following program illustrates use of this function and Figure 4.4 shows the resulting pairwise scatterplot.

```
% ----- Example 4.8 Using the pairs() function
n = 100;
y1 = randn(n,1);
y2 =  2*y1 + randn(n,1);
y3 = -2*y2 + randn(n,1);
y4 = randn(n,1);    % uncorrelated with y1,y2,y3
y5 = randn(n,1).^4; % leptokurtic variable
y = [y1 y2 y3 y4 y5];
vnames = strvcat('apples','oranges','pairs','ha ha','leptoku');
pairs(y,vnames);
```

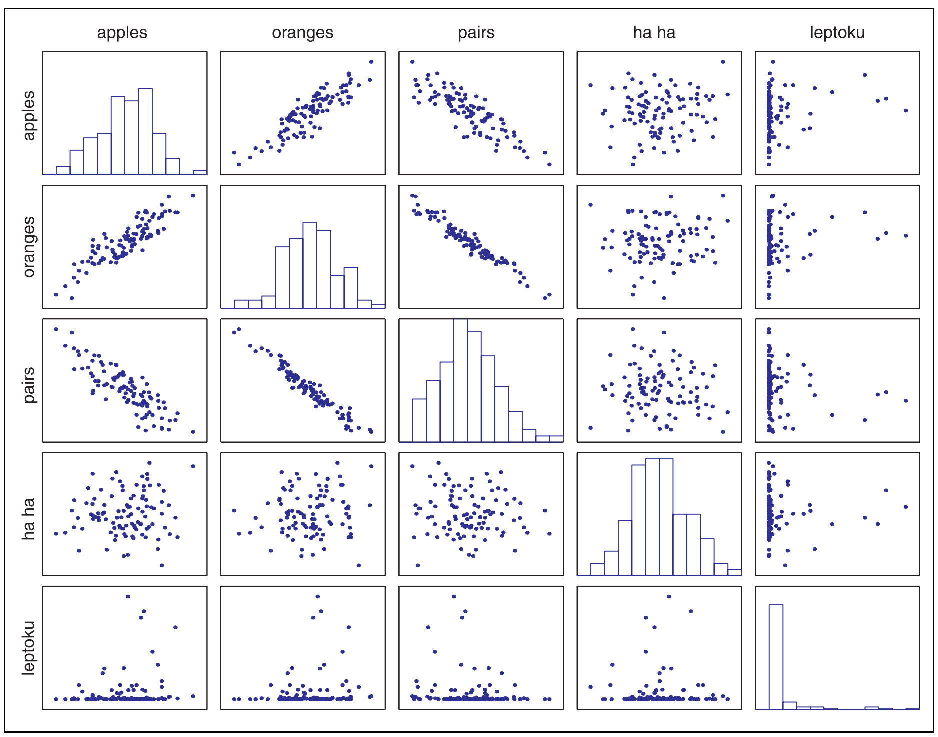

Figure 4.4: Pairwise scatter plots

4.3 Chapter summary

A large number of regression diagnostic tests and corrective estimation procedures have been proposed in the econometrics literature. MATLAB and Gauss code for implementing these methods can be found on many Internet sites. The *Econometrics Toolbox* design allows these routines to be implemented and documented in a way that provides a consistent user interface for printing and plotting the diagnostics.

Chapter 4 Appendix

A library of routines in the subdirectory **diagn** contain many of the functions discussed in this chapter.

```
regression diagnostics library

arch             - ARCH(p) test
bkw              - BKW collinearity diagnostics
bpagan           - Breusch-Pagan heteroscedasticity test
cusums           - Brown,Durbin,Evans cusum squares test
dfbeta           - BKW influential observation diagnostics
diagnose         - compute diagnostic statistics
plt_dfb          - plots dfbetas
plt_dff          - plots dffits
plt_cus          - plots cusums
recresid         - compute recursive residuals
rdiag            - graphical residuals diagnostics
studentize       - standarization transformation
unstudentize     - reverses studentize transformation
qstat2           - Box-Ljung Q-statistic

------- demonstration programs -----------

arch_d           - demonstrates arch
bkw_d            - demonstrates bkw
bpagan_d         - demonstrates bpagan
cusums_d         - demonstrates cusums
dfbeta_d         - demonstrates dfbeta, plt_dfb, plt_dff
diagnose_d       - demonstrates diagnose
recresid_d       - demonstrates recresid
rdiag_d          - demonstrates rdiag
unstudentize_d   - demonstrates studentize, unstudentize
qstat2_d         - demonstrates qstat2

------- support functions

ols              - least-squares regression
plt              - plots everything
```

```
plt_cus          - plots cusums test results
plt_dfb          - plots dfbetas
plt_dff          - plots dffits
```

Functions from the *regression library* discussed in this chapter were:

```
------- regression program functions -----------

olst            - regression with t-distributed errors
ridge           - ridge regression
robust          - iteratively reweighted least-squares
rtrace          - ridge trace plots
theil           - Theil-Goldberger mixed estimation

-------- Demo programs ---------------------------

olst_d          - olst demo
ridge_d         - ridge regression and rtrace demo
robust_d        - demonstrates robust regression
theil_d         - demonstrates theil-goldberger estimation
```

Chapter 5

VAR and Error Correction Models

This chapter describes the design and use of MATLAB functions to implement vector autoregressive (VAR) and error correction (EC) models. The MATLAB functions described here provide a consistent user-interface in terms of the MATLAB help information, and related routines to print and plot results from the various models. One of the primary uses of VAR and EC models is econometric forecasting, for which we provide a set of functions.

Section 5.1 describes the basic VAR model and our function to estimate and print results for this method. Section 5.2 turns attention to EC models while Section 5.3 discusses Bayesian variants on these models. Finally, we take up forecasting in Section 5.4. All of the functions implemented in our *vector autoregressive function library* are documented in the Appendix to this chapter.

5.1 VAR models

A VAR model is shown in (5.1) that contains n variables. The ε_{it} denote independent disturbances, C_i represent constants and $y_{it}, i = 1, \ldots, n$ denote the n variables in the model at time t. Model parameters $A_{ij}(\ell)$ take the form, $\sum_{k=1}^{m} a_{ijk}\ell^k$, where ℓ is the lag operator defined by $\ell^k y_t = y_{t-k}$, and m is the lag length specified by the modeler.

$$\begin{bmatrix} y_{1t} \\ y_{2t} \\ \vdots \\ y_{nt} \end{bmatrix} = \begin{bmatrix} A_{11}(\ell) & \dots & A_{1n}(\ell) \\ \vdots & \ddots & \vdots \\ A_{n1}(\ell) & \dots & A_{nn}(\ell) \end{bmatrix} \begin{bmatrix} y_{1t} \\ y_{2t} \\ \vdots \\ y_{nt} \end{bmatrix} + \begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_n \end{bmatrix} + \begin{bmatrix} \varepsilon_{1t} \\ \varepsilon_{2t} \\ \vdots \\ \varepsilon_{nt} \end{bmatrix} \tag{5.1}$$

The VAR model posits a set of relationships between past lagged values of all variables in the model and the current value of each variable in the model. For example, if the y_{it} represent employment in state i at time t, the VAR model structure allows employment variation in each state to be explained by past employment variation in the state itself, $y_{it-k}, k = 1, \dots, m$ as well as past employment variation in other states, $y_{jt-k}, k = 1, \dots, m,\ j \neq i$. This is attractive since regional or state differences in business cycle activity suggest lead/lag relationships in employment of the type set forth by the VAR model structure.

The model is estimated using ordinary least-squares, so we can draw on our **ols** routine from the regression library. A function **var** produces estimates for the coefficients in the VAR model as well as related regression statistics and Granger-causality test statistics.

The documentation for the **var** function is:

```
PURPOSE: performs vector autoregressive estimation
---------------------------------------------------
USAGE:  result = var(y,nlag,x)
where:    y    = an (nobs x neqs) matrix of y-vectors
          nlag = the lag length
          x    = optional matrix of variables (nobs x nx)
NOTE:     constant vector automatically included
---------------------------------------------------
RETURNS a structure
results.meth = 'var'
results.nobs = nobs, # of observations
results.neqs = neqs, # of equations
results.nlag = nlag, # of lags
results.nvar = nlag*neqs+nx+1, # of variables per equation
--- the following are referenced by equation # ---
results(eq).beta  = bhat for equation eq
results(eq).tstat = t-statistics
results(eq).tprob = t-probabilities
results(eq).resid = residuals
results(eq).yhat  = predicted values
results(eq).y     = actual values
results(eq).sige  = e'e/(n-k)
results(eq).rsqr  = r-squared
```

```
 results(eq).rbar  = r-squared adjusted
 results(eq).boxq  = Box Q-statistics
 results(eq).ftest = Granger F-tests
 results(eq).fprob = Granger marginal probabilities
---------------------------------------------------
 SEE ALSO: varf, prt_var, pgranger, pftests
---------------------------------------------------
```

This function utilizes a new aspect of MATLAB structure variables, arrays that can store information for each equation in the VAR model. Estimates of the $\hat{\beta}$ parameters for the first equation can be accessed from the results structure using: result(1).beta as can other results that are equation-specific.

In most applications, the user would simply pass the results structure on to the **prt_var** (or **prt**) function that will provide an organized printout of the regression results for each equation. Here is an example of a typical program to estimate a VAR model.

```
% ----- Example 5.1 Using the var() function
dates = cal(1982,1,12);
load test.dat;                % monthly mining employment in 8 states
y = growthr(test(:,1:2),12); % convert to growth-rates
yt = trimr(y,dates.freq,0);  % truncate
dates = cal(1983,1,1);       % redefine calendar for truncation
vnames   = strvcat('illinos','indiana');
nlag = 2;
result = var(yt,nlag);       % estimate 2-lag VAR model
prt(result,vnames);          % printout results
```

It would produce the following printout of the estimation results:

```
***** Vector Autoregressive Model *****
Dependent Variable =          illinos
R-squared     =    0.9044
Rbar-squared  =    0.9019
sige          =    3.3767
Q-statistic   =    0.2335
Nobs, Nvars   =    159,     5
******************************************************************
Variable            Coefficient      t-statistic    t-probability
illinos  lag1          1.042540        13.103752         0.000000
illinos  lag2         -0.132170        -1.694320         0.092226
indiana  lag1          0.228763         3.790802         0.000215
indiana  lag2         -0.213669        -3.538905         0.000531
constant              -0.333739        -1.750984         0.081940

 ****** Granger Causality Tests *******
```

```
Variable          F-value      Probability
illinos        363.613553         0.000000
indiana          7.422536         0.000837

Dependent Variable =          indiana
R-squared     =    0.8236
Rbar-squared  =    0.8191
sige          =    6.5582
Q-statistic   =    0.0392
Nobs, Nvars   =    159,     5
******************************************************************
Variable             Coefficient      t-statistic    t-probability
illinos  lag1           0.258853         2.334597         0.020856
illinos  lag2          -0.195181        -1.795376         0.074555
indiana  lag1           0.882544        10.493894         0.000000
indiana  lag2          -0.029384        -0.349217         0.727403
constant               -0.129405        -0.487170         0.626830

 ****** Granger Causality Tests *******
Variable          F-value      Probability
illinos          2.988892         0.053272
indiana        170.063761         0.000000
```

There are two utility functions that help analyze VAR model Granger-causality output. The first is **pgranger**, which prints a matrix of the marginal probabilities associated with the Granger-causality tests in a convenient format for the purpose of inference. The documentation is:

```
  PURPOSE: prints VAR model Granger-causality results
   --------------------------------------------------
  USAGE: pgranger(results,varargin);
         where: results = a structure returned by var(), ecm()
               varargin = a variable input list containing
                vnames  = an optional variable name vector
                cutoff  = probability cutoff used when printing
   usage example 1: pgranger(result,0.05);
         example 2: pgranger(result,vnames);
         example 3: pgranger(result,vnames,0.01);
         example 4: pgranger(result,0.05,vnames);
 ----------------------------------------------------
                  e.g. cutoff = 0.05 would only print
                       marginal probabilities < 0.05
 ---------------------------------------------------
  NOTES: constant term is added automatically to vnames list
        you need only enter VAR variable names plus deterministic
 ---------------------------------------------------
```

As example of using this function, consider our previous program to estimate the VAR model for monthly mining employment in eight states. Rather than print out the detailed VAR estimation results, we might be interested in drawing inferences regarding Granger-causality from the marginal probabilities. The following program would produce a printout of just these probabilities. It utilizes an option to suppress printing of probabilities greater than 0.1, so that our inferences would be drawn on the basis of a 90% confidence level.

```
% ----- Example 5.2 Using the pgranger() function
dates = cal(1982,1,12);     % monthly data starts in 82,1
load test.dat;              % monthly mining employment in 8 states
y = growthr(test,12);       % convert to growth-rates
yt = trimr(y,dates.freq,0); % truncate
dates = cal(1983,1,1);      % redefine the calendar for truncation
vname  = strvcat('il','in','ky','mi','oh','pa','tn','wv');
nlag = 12;
res = var(yt,nlag);         % estimate 12-lag VAR model
cutoff = 0.1;               % examine Granger-causality at 90% level
pgranger(res,vname,cutoff); % print Granger probabilities
```

We use the 'NaN' symbol to replace marginal probabilities above the cutoff point (0.1 in the example) so that patterns of causality are easier to spot. The results from this program would look as follows:

```
 ****** Granger Causality Probabilities *******
Variable      il      in      ky      mi      oh      pa      tn      wv
il          0.00    0.01    0.01     NaN     NaN    0.04    0.09    0.02
in          0.02    0.00     NaN     NaN     NaN     NaN     NaN     NaN
ky           NaN     NaN    0.00     NaN    0.10     NaN    0.07     NaN
mi           NaN    0.01     NaN    0.00     NaN     NaN     NaN     NaN
oh           NaN    0.05    0.08     NaN    0.00    0.01     NaN    0.01
pa          0.05     NaN     NaN     NaN     NaN    0.00     NaN    0.06
tn          0.02    0.05     NaN     NaN     NaN    0.09    0.00     NaN
wv          0.02    0.05    0.06    0.01     NaN    0.00     NaN    0.03
```

The format of the output is such that the columns reflect the Granger-causal impact of the column-variable on the row-variable. That is, Indiana, Kentucky, Pennsylvania, Tennessee and West Virginia exert a significant Granger-causal impact on Illinois employment whereas Michigan and Ohio do not. Indiana exerts the most impact, affecting Illinois, Michigan, Ohio, Tennessee, and West Virginia.

The second utility is a function **pftest** that prints just the Granger-causality joint F-tests from the VAR model. Use of this function is similar to **pgranger**, we simply call the function with the results structure returned by the **var** function, e.g., **pftest(result,vnames)**, where the 'vnames' argument is an optional string-vector of variable names. This function would produce the following output for each equation of a VAR model based on all eight states:

```
 ****** Granger Causality Tests *******
Equation  illinois                F-value    F-probability
illinois                         395.4534           0.0000
indiana                            3.3255           0.0386
kentucky                           0.4467           0.6406
michigan                           0.6740           0.5112
ohio                               2.9820           0.0536
pennsylvania                       6.6383           0.0017
tennessee                          0.9823           0.3768
west virginia                      3.0467           0.0504
```

A few points regarding how the **var** function was implemented. We rely on the **ols** function from the regression library to carry out the estimation of each equation in the model and transfer the 'results structure' returned by **ols** to a new **var** results structure array for each equation. Specifically the code looks as follows:

```
% pull out each y-vector and run regressions
for j=1:neqs;
yvec = y(nlag+1:nobs,j);
res = ols(yvec,xmat);
results(j).beta  = res.beta;      % bhats
results(j).tstat = res.tstat;     % t-stats
% compute t-probs
tstat = res.tstat;
tout = tdis_prb(tstat,nobse-nvar);
results(j).tprob = tout;          % t-probs
results(j).resid = res.resid;     % resids
results(j).yhat = res.yhat;       % yhats
results(j).y    = yvec;           % actual y
results(j).rsqr = res.rsqr;       % r-squared
results(j).rbar = res.rbar;       % r-adjusted
results(j).sige = res.sige;       % sig estimate
```

The explanatory variables matrix, 'xmat' is the same for all equations of the VAR model, so we form this matrix before entering the 'for-loop' over equations. Structure arrays can be formed by simply specifying:
struct(i).fieldname, where i is the array index. This makes them just as easy to use as the structure variables we presented in the discussion of the *regression function library* in Chapter 2.

The most complicated part of the **var** function is implementation of the Granger-causality tests which requires that we produce residuals based on models that sequentially omit each variable in the model from the explanatory variables matrix in each equation. These residuals reflect the restricted model in the joint F-test, whereas the VAR model residuals represent the unrestricted model. The Granger-causality tests represent a series of tests for the joint significance of each variable in each equation of the model. This sequence of calculations is illustrated in the code below:

```
% form x matrices for joint F-tests
% exclude each variable from the model sequentially
for r=1:neqs;
xtmp = [];
for s=1:neqs;
   if s ~= r
   xlag = mlag(y(:,s),nlag);
   xtmp = [xtmp trimr(xlag,nlag,0)];
   end;
end; % end of for s-loop
% we have an xtmp matrix that excludes 1 variable
% add deterministic variables (if any) and constant term
  if nx > 0
    xtmp = [xtmp x(1:nobse,:) ones(nobse,1)];
  else
    xtmp = [xtmp ones(nobse,1)];
  end;
% get ols residual vector
  b = xtmp\yvec; % using Cholesky solution
  etmp = yvec-xtmp*b;
  sigr = etmp'*etmp;
% joint F-test for variables r
  ftest(r,1) = ((sigr - sigu)/nlag)/(sigu/(nobse-k));
end; % end of for r-loop
results(j).ftest = ftest;
results(j).fprob = fdis_prb(ftest,nlag,nobse-k);
```

The loop over **r=1:neqs** and **s=1:neqs** builds up an explanatory variables matrix, **xtmp** by sequentially omitting each of the variables in the model. This programming construct is often useful. We start with a blank matrix **xtmp** = [] and then continue to add matrix columns to this blank matrix during the loop. Each time through the loop a set of columns representing lags for another variable in the model are added to the existing matrix **xtmp**, until we have 'built-up' the desired matrix. If **s == r** we skip adding lags for that variable to the matrix **xtmp**, to create the necessary exclusion of 1 variable at a time.

Next we need to add deterministic variables (if they exist) and a constant term to the explanatory variables matrix. Finally, we carry out least-squares using the matrix **xtmp** with a Cholesky solution provided by the MATLAB 'backslash' operator. We take the Cholesky approach rather than the qr matrix decomposition because a profile of the **var** function showed that over 50% of the time spent in the routine was devoted to this step. In contrast, only 12% of the time was spent determining the VAR regression information using the **ols** command. Finally, note that the code above is embedded in a loop over all equations in the model, so we store the 'ftest' and 'fprob' results in the structure for equation j.

Although the illustrations so far have not involved use of deterministic variables in the VAR model, the **var** function is capable of handling these variables. As an example, we could include a set of seasonal dummy variables in the VAR model using:

```
% ----- Example 5.3 VAR with deterministic variables
dates = cal(1982,1,12);         % monthly data starts in 82,1
load test.dat;
y = test;                       % use levels data
[nobs neqs] = size(test);
sdum = sdummy(nobs,dates.freq); % create seasonal dummies
sdum = trimc(sdum,1,0);         % omit 1 column because we have
                                % a constant included by var()
vnames  = strvcat('illinos','indiana','kentucky','michigan','ohio', ...
          'pennsylvania','tennessee','west virginia');
dnames = strvcat('dum1','dum2','dum3','dum4','dum5','dum6','dum7', ...
          'dum8','dum9','dum10','dum11');
vnames  = strvcat(vnames,dnames);
nlag = 12;
result = var(y,nlag,sdum);
prt(result,vnames);
```

A handy option on the **prt_var** (and **prt**) function is the ability to print the VAR model estimation results to an output file. Because these results are quite large, they can be difficult to examine in the MATLAB command window. Note that the wrapper function **prt** described in Chapter 3 also works to print results from VAR model estimation, as does **plt**.

```
 PURPOSE: Prints vector autoregression output
          from: var,bvar,rvar,ecm,becm,recm models
---------------------------------------------------
 USAGE: prt_var(result,vnames,fid)
        where: results = a structure returned by:
                         var,bvar,rvar,ecm,becm,recm
               vnames  = optional vector of variable names
                   fid = file-id for printing results to a file
                         (defaults to the MATLAB command window)
```

```
---------------------------------------------------
 NOTE: - constant term is added automatically to vnames list
         you need only enter VAR variable names plus deterministic
       - you may use prt_var(results,[],fid) to print
         output to a file with no vnames
---------------------------------------------------
```

In addition to the **prt** and **prt_var** functions, there are **plt** and **plt_var** functions that produce graphs of the actual versus predicted and residuals for these models.

One final issue associated with specifying a VAR model is the lag length to employ. A commonly used approach to determining the lag length is to perform statistical tests of models with longer lags versus shorter lag lengths. We view the longer lag models as an unrestricted model versus the restricted shorter lag version of the model, and construct a likelihood ratio statistic to test for the significance of imposing the restrictions. If the restrictions are associated with a statistically significant degradation in model fit, we conclude that the longer lag length model is more appropriate, rejecting the shorter lag model.

Specifically, the chi-squared distributed test statistic which has degrees of freedom equation to the number of restrictions imposed is:

$$LR = (T - c)(\log|\Sigma_r| - \log|\Sigma_u|) \tag{5.2}$$

where T is the number of observations, c is a degrees of freedom correction factor proposed by Sims (1980), and $|\Sigma_r|, |\Sigma_u|$ denote the determinant of the error covariance matrices from the restricted and unrestricted models respectively. The correction factor, c, recommended by Sims was the number of variables in each unrestricted equation of the VAR model.

A function **lrratio** implements a sequence of such tests beginning at a maximum lag (specified by the user) down to a minimum lag (also specified by the user). The function prints results to the MATLAB command window along with marginal probability levels. As an example, consider the following program to determine the 'statistically optimal' lag length to use for our VAR model involving the eight-state sample of monthly employment data for the mining industry.

```
% ----- Example 5.4 Using the lrratio() function
load test.dat;
y = test; % use all eight states
maxlag = 12;
minlag = 3;
% Turn on flag for Sim's correction factor
```

```
sims = 1;
disp('LR-ratio results with Sims correction');
lrratio(y,maxlag,minlag,sims);
```

The output from this program is:

```
LR-ratio results with Sims correction
nlag = 12 11, LR statistic =         75.6240, probability = 0.1517
nlag = 11 10, LR statistic =         89.9364, probability = 0.01798
nlag = 10  9, LR statistic =         91.7983, probability = 0.01294
nlag =  9  8, LR statistic =        108.8114, probability = 0.0004052
nlag =  8  7, LR statistic =        125.7240, probability = 6.573e-06
nlag =  7  6, LR statistic =        114.2624, probability = 0.0001146
nlag =  6  5, LR statistic =         81.3528, probability = 0.07059
nlag =  5  4, LR statistic =        118.5982, probability = 4.007e-05
nlag =  4  3, LR statistic =        127.1812, probability = 4.489e-06
```

There exists an option flag to use the degrees of freedom correction suggested by Sims, whereas the default behavior of **lrratio** is to set $c = 0$. Example 4.4 turns on the correction factor by setting a flag that we named 'sims' equal to 1. The results suggest that the lag length of 11 cannot be viewed as significantly degrading the fit of the model relative to a lag of 12. For the comparison of lags 11 and 10, we find that at the 0.05 level, we might reject lag 10 in favor of lag 11 as the optimal lag length. On the other hand, if we employ a 0.01 level of significance, we would conclude that the optimal lag length is 9, because the likelihood ratio tests reject lag 8 as significantly degrading the fit of the model at the 0.01 level of confidence.

Impulse response functions represent a method used to examine the interactions between variables in a VAR model. A shock to the system is imposed and the dynamic interactions of the variables based on the estimated coefficients are simulated for all variables in the system. The function **irf** contributed by Mike Cliff performs the calcuations and provides graphical output of the impulse responses.

```
% ----- Example 5.5 Using the irf() function
load test.dat; % a test data set containing
               % monthly mining employment for
               % il,in,ky,mi,oh,pa,tn,wv
% data covers 1982,1 to 1996,5
neqs = 8; y = test(:,1:neqs); % use neqs states
nlag = 6;  % number of lags in var-model
% estimate the model
results = var(y,nlag);
vnames =  strvcat('illinois','indiana','kentucky','michigan','ohio', ...
                  'pennsylvania','tennessee','west virginia');
```

```
nperiods = 48;
[m1 m2] = irf(results,nperiods,'o1',vnames);
```

The results are provided graphically as eight graphs for the example involving 8 equations, where a typical graph is shown in Figure 5.1. This function puts up multiple MATLAB graph windows with a varying number of the impulse response graphs in each window. This allows the user to examine all of the responses and compare them using the multiple MATLAB figure windows. The figure presents only a single graph window showing three of the eight graphs produced by the example program.

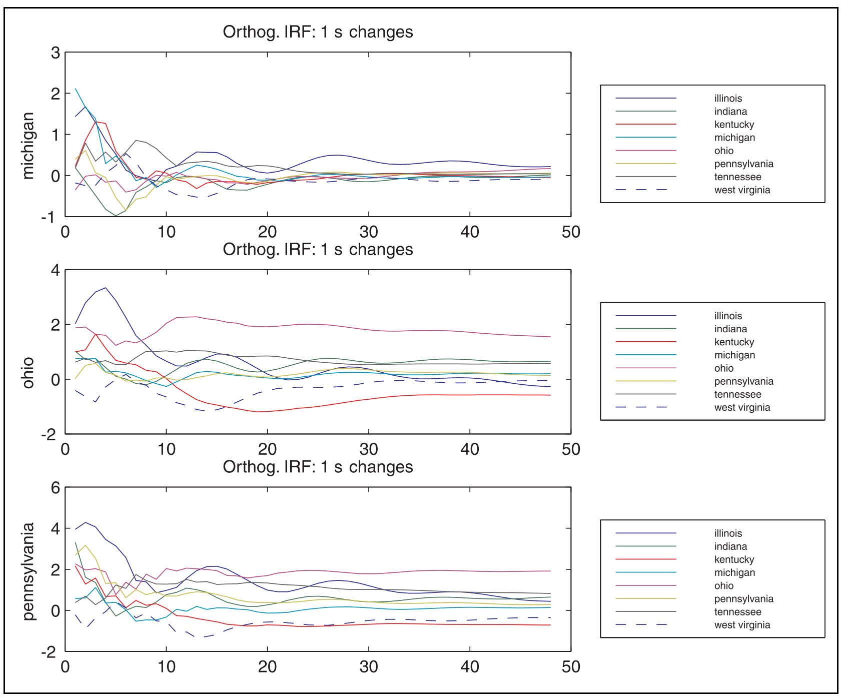

Figure 5.1: Impulse response graphs

5.2 Error correction models

We provide a cursory introduction to co-integration and error correction models and refer the reader to an excellent layman's introduction by Dickey, Jansen and Thornton (1991) as well as a more technical work by Johansen

(1995). LeSage (1990) and Shoesmith (1995) cover co-integration and EC models in the context of forecasting.

Focusing on the practical case of I(1), (integrated of order 1) series, let y_t be a vector of n time-series that are I(1). An I(1) series requires one difference to transform it to a zero mean, purely non-deterministic stationary process. The vector y_t is said to be co-integrated if there exists an n x r matrix α such that:

$$z_t = \alpha' y_t \tag{5.3}$$

Engle and Granger (1987) provide a Representation Theorem stating that if two or more series in y_t are co-integrated, there exists an error correction representation taking the following form:

$$\Delta y_t = A(\ell)\Delta y_t + \gamma z_{t-1} + \varepsilon_t \tag{5.4}$$

where γ is a matrix of coefficients of dimension n x r of rank r, z_{t-1} is of dimension r x 1 based on $r \leq n-1$ equilibrium error relationships, $z_t = \alpha' y_t$ from (5.3), and ε_t is a stationary multivariate disturbance. The error correction (EC) model in (5.4) is simply a VAR model in first-differences with r lagged error correction terms (z_{t-1}) included in each equation of the model. If we have deterministic components in y_t, we add these terms as well as the error correction variables to each equation of the model.

With the case of only two series y_t and x_t in the model, a two-step procedure proposed by Engle and Granger (1987) can be used to determine the co-integrating variable that we add to our VAR model in first-differences to make it an EC model. The first-step involves a regression: $y_t = \theta + \alpha x_t + z_t$ to determine estimates of α and z_t. The second step carries out tests on z_t to determine if it is stationary, I(0). If we find this to be the case, the condition $y_t = \theta + \alpha x_t$ is interpreted as the equilibrium relationship between the two series and the error correction model is estimated as:

$$\begin{aligned}
\Delta y_t &= -\gamma_1 z_{t-1} + \text{lagged}(\Delta x_t, \Delta y_t) + c_1 + \varepsilon_{1t} \\
\Delta x_t &= -\gamma_2 z_{t-1} + \text{lagged}(\Delta x_t, \Delta y_t) + c_2 + \varepsilon_{2t}
\end{aligned}$$

where: $z_{t-1} = y_{t-1} - \theta - \alpha x_{t-1}$, c_i are constant terms and ε_{it} denote disturbances in the model.

We provide a function **adf**, (augmented Dickey-Fuller) to test time-series for the I(1), I(0) property, and another routine **cadf** (co-integrating augmented Dickey-Fuller) to carry out the tests from step two above on z_t to determine if it is stationary, I(0). These routines as well as the function **johansen** that implements a multivariate extension of the two-step Engle and Granger procedure were designed to mimic a set of Gauss functions by Sam Quilaris named **coint**.

The **adf** function documentation is:

```
 PURPOSE: carry out DF tests on a time series vector
---------------------------------------------------
 USAGE: results = adf(x,p,nlag)
 where:      x = a time-series vector
             p = order of time polynomial in the null-hypothesis
                 p = -1, no deterministic part
                 p =  0, for constant term
                 p =  1, for constant plus time-trend
                 p >  1, for higher order polynomial
         nlags = # of lagged changes of x included
---------------------------------------------------
 RETURNS: a results structure
         results.meth  = 'adf'
         results.alpha = estimate of the autoregressive parameter
         results.adf   = ADF t-statistic
         results.crit  = (6 x 1) vector of critical values
                        [1% 5% 10% 90% 95% 99%] quintiles
         results.nlag = nlag
```

This would be used to test a time-series vector for I(1) or I(0) status. Allowance is made for polynomial time trends as well as constant terms in the function and a set of critical values are returned in a structure by the function. A function **prt_coint** (as well as **prt**) can be used to print output from **adf, cadf** and **johansen**, saving users the work of formatting and printing the result structure output.

The function **cadf** is used for the case of two variables, y_t, x_t, where we wish to test whether the condition $y_t = \alpha x_t$ can be interpreted as an equilibrium relationship between the two series. The function documentation is:

```
 PURPOSE: compute augmented Dickey-Fuller statistic for residuals
          from a cointegrating regression, allowing for deterministic
          polynomial trends
 ------------------------------------------------------------
 USAGE: results = cadf(y,x,p,nlag)
 where: y = dependent variable time-series vector
        x = explanatory variables matrix
             p = order of time polynomial in the null-hypothesis
                 p = -1, no deterministic part
                 p =  0, for constant term
```

```
                p =  1, for constant plus time-trend
                p >  1, for higher order polynomial
     nlag = # of lagged changes of the residuals to include in regression
 -----------------------------------------------------------
 RETURNS: results structure
          results.meth  = 'cadf'
          results.alpha = autoregressive parameter estimate
          results.adf   = ADF t-statistic
          results.crit  =  (6 x 1) vector of critical values
                        [1% 5% 10% 90% 95% 99%] quintiles
          results.nvar  = cols(x)
          results.nlag  = nlag
```

As an illustration of using these two functions, consider testing our two monthly time-series on mining employment in Illinois and Indiana for I(1) status and then carrying out a test of whether they exhibit an equilibrating relationship. The program would look as follows:

```
% ----- Example 5.6 Using the adf() and cadf() functions
dates = cal(1982,1,12);
load test.dat;
y = test(:,1:2); % use only two series
vnames = strvcat('illinois','indiana');
% test Illinois for I(1) status
nlags = 6;
  for i=1:nlags;
   res = adf(y(:,1),0,i);
   prt(res,vnames(1,:));
  end;
% test Indiana for I(1) status
nlags = 6;
  for i=1:nlags;
   res = adf(y(:,2),0,i);
   prt(res,vnames(2,:));
  end;
% test if Illinois and Indiana are co-integrated
  for i=1:nlags;
   res = cadf(y(:,1),y(:,2),0,i);
    prt(res,vnames);
  end;
```

The program sets a lag length of 6, and loops over lags of 1 to 6 to provide some feel for how the augmented Dickey-Fuller tests are affected by the number of lags used. We specify $p = 0$ because the employment time-series do not have zero mean, so we wish to include a constant term. The result structures returned by the **adf** and **cadf** functions are passed on to **prt** for printing. We present the output for only lag 6 to conserve on space, but all lags produced the same inferences. One point to note is that the **adf** and **cadf** functions return a set of 6 critical values for significance levels 1%,5%,10%,90%,95%,99% as indicated in the documentation for these functions. Only three are printed for purposes of clarity, but all are available in the results structure returned by the functions.

```
Augmented DF test for unit root variable:                 illinois
 ADF t-statistic       # of lags   AR(1) estimate
       -0.164599               6         0.998867
   1% Crit Value    5% Crit Value   10% Crit Value
          -3.464           -2.912           -2.588

Augmented DF test for unit root variable:                 indiana
 ADF t-statistic       # of lags   AR(1) estimate
       -0.978913               6         0.987766
   1% Crit Value    5% Crit Value   10% Crit Value
          -3.464           -2.912           -2.588

 Augmented DF test for co-integration variables:         illinois,indiana
CADF t-statistic        # of lags   AR(1) estimate
     -1.67691570                6        -0.062974
   1% Crit Value    5% Crit Value   10% Crit Value
          -4.025           -3.404           -3.089
```

We see from the **adf** function results that both Illinois and Indiana are I(1) variables. We reject the augmented Dickey-Fuller hypothesis of I(0) because our t-statistics for both Illinois and Indiana are less than (in absolute value terms) the critical value of -2.588 at the 90% level.

From the results of **cadf** we find that Illinois and Indiana mining employment are not co-integrated, again because the t-statistic of -1.67 does not exceed the 90% critical value of -3.08 (in absolute value terms). We would conclude that an EC model is not appropriate for these two time-series.

For most EC models, more than two variables are involved so the Engle and Granger two-step procedure needs to be generalized. Johansen (1988) provides this generalization which takes the form of a likelihood-ratio test. We implement this test in the function **johansen**. The Johansen procedure provides a test statistic for determining r, the number of co-integrating relationships between the n variables in y_t as well as a set of r co-integrating vectors that can be used to construct error correction variables for the EC model.

As a brief motivation for the work carried out by the **johansen** function, we start with a reparameterization of the EC model:

$$\Delta y_t = \Gamma_1 \Delta y_{t-1} + \ldots + \Gamma_{k-1} \Delta y_{t-k+1} - \Psi y_{t-k} + \varepsilon_t \qquad (5.5)$$

where $\Psi = (I_n - A_1 - A_2 - \ldots - A_k)$. If the matrix Ψ contains all zeros, (has rank=0), there are no co-integrating relationships between the variables in y_t. If Ψ is of full-rank, then we have n long-run equilibrium relationships, so all variables in the model are co-integrated. For cases where the matrix Ψ has rank $r < n$, we have r co-integrating relationships. The Johansen procedure provides two tests for the number of linearly independent co-integrating relationships among the series in y_t, which we have labeled r in our discussion. Both tests are based on an eigenvalue-eigenvector decomposition of the matrix Ψ, constructed from canonical correlations between Δy_t and y_{t-k} with adjustments for intervening lags, and taking into account that the test is based on estimates that are stochastic. The test statistics are labeled the 'trace statistic' and the 'maximal eigenvalue statistic'.

Given the value of r, (the number of co-integrating relationships), we can use the eigenvectors provided by the **johansen** function along with the levels of y_t lagged one period to form a set of error correction variables for our EC model. In practice, the function **ecm** does this for you, so you need not worry about the details.

The documentation for **johansen** is shown below. A few points to note. The published literature contains critical values for the trace statistic for VAR models with up to 12 variables in Johansen (1995), and for the maximal eigenvalue statistic, Johansen and Juselius (1988) present critical values for VAR models containing up to 5 variables. To extend the number of variables for which critical values are available, a procedure by MacKinnon (1996) was used to generate critical values for both the trace and maximal eigenvalue statistics for models with up to 12 variables. MacKinnon's method is an approximation, but it produces values close to those in Johansen (1995). The critical values for the trace statistic have been entered in a function **c_sjt** and those for the maximal eigenvalue statistic are in **c_sja**. The function **johansen** calls these two functions to obtain the necessary critical values. In cases where the VAR model has more than 12 variables, zeros are returned as critical values in the structure field 'result.cvt' for the trace statistic and the 'result.cvm' field for the maximal eigenvalue.

Another less serious limitation is that the critical values for these statistics are only available for trend transformations where $-1 \leq p \leq 1$. This should not present a problem in most applications where p will take on values of -1, 0 or 1.

```
PURPOSE: perform Johansen cointegration tests
-------------------------------------------------------
USAGE: result = johansen(x,p,k)
where:      x = input matrix of time-series in levels, (nobs x m)
```

```
             p = order of time polynomial in the null-hypothesis
                 p = -1, no deterministic part
                 p =  0, for constant term
                 p =  1, for constant plus time-trend
                 p >  1, for higher order polynomial
             k = number of lagged difference terms used when
                 computing the estimator
 ---------------------------------------------------
 RETURNS: a results structure:
          result.eig  = eigenvalues  (m x 1)
          result.evec = eigenvectors (m x m), where first
                        r columns are normalized coint vectors
          result.lr1  = likelihood ratio trace statistic for r=0 to m-1
                        (m x 1) vector
          result.lr2  = maximum eigenvalue statistic for r=0 to m-1
                        (m x 1) vector
          result.cvt  = critical values for trace statistic
                        (m x 3) vector [90% 95% 99%]
          result.cvm  = critical values for max eigen value statistic
                        (m x 3) vector [90% 95% 99%]
          result.ind  = index of co-integrating variables ordered by
                        size of the eigenvalues from large to small
 ---------------------------------------------------
 NOTE: c_sja(), c_sjt() provide critical values generated using
       a method of MacKinnon (1994, 1996).
       critical values are available for n<=12 and -1 <= p <= 1,
       zeros are returned for other cases.
```

As an illustration of the **johansen** function, consider the eight-state sample of monthly mining employment. We would test for the number of co-integrating relationships using the following code:

```
% ----- Example 5.7 Using the johansen() function
vnames  = strvcat('illinos','indiana','kentucky','michigan','ohio',  ...
           'pennsylvania','tennessee','west virginia');
y = load('test.dat'); % use all eight states
nlag = 9;
pterm = 0;
result = johansen(y,pterm,nlag);
prt(result,vnames);
```

The **johansen** function is called with the y matrix of time-series variables for the eight states, a value of $p = 0$ indicating we have a constant term in the model, and 9 lags. (We want $p = 0$ because the constant term is necessary where the levels of employment in the states differ.) The lag of 9 was determined to be optimal using the **lrratio** function in the previous section.

The **johansen** function will return results for a sequence of tests against alternative numbers of co-integrating relationships ranging from $r \leq 0$ up to $r \leq m-1$, where m is the number of variables in the matrix y.

The function **prt** provides a printout of the trace and maximal eigenvalue statistics as well as the critical values returned in the **johansen** results structure.

```
 Johansen MLE estimates
NULL:                  Trace Statistic  Crit 90%   Crit 95%   Crit 99%
r <= 0   illinos               307.689   153.634    159.529    171.090
r <= 1   indiana               205.384   120.367    125.618    135.982
r <= 2   kentucky              129.133    91.109     95.754    104.964
r <= 3   ohio                   83.310    65.820     69.819     77.820
r <= 4   pennsylvania           52.520    44.493     47.855     54.681
r <= 5   tennessee              30.200    27.067     29.796     35.463
r <= 6   west virginia          13.842    13.429     15.494     19.935
r <= 7   michigan                0.412     2.705      3.841      6.635

NULL:                  Eigen Statistic  Crit 90%   Crit 95%   Crit 99%
r <= 0   illinos               102.305    49.285     52.362     58.663
r <= 1   indiana                76.251    43.295     46.230     52.307
r <= 2   kentucky               45.823    37.279     40.076     45.866
r <= 3   ohio                   30.791    31.238     33.878     39.369
r <= 4   pennsylvania           22.319    25.124     27.586     32.717
r <= 5   tennessee              16.359    18.893     21.131     25.865
r <= 6   west virginia          13.430    12.297     14.264     18.520
r <= 7   michigan                0.412     2.705      3.841      6.635
```

The printout does not present the eigenvalues and eigenvectors, but they are available in the results structure returned by **johansen** as they are needed to form the co-integrating variables for the EC model. The focus of co-integration testing would be the trace and maximal eigenvalue statistics along with the critical values. For this example, we find: (using the 95% level of significance) the trace statistic rejects $r \leq 0$ because the statistic of 307.689 is greater than the critical value of 159.529; it also rejects $r \leq 1$, $r \leq 2$, $r \leq 3$, $r \leq 4$, and $r \leq 5$ because these trace statistics exceed the associated critical values; for $r \leq 6$ we cannot reject H0, so we conclude that $r = 6$. Note that using the 99% level, we would conclude $r = 4$ as the trace statistic of 52.520 associated with $r \leq 4$ does not exceed the 99% critical value of 54.681.

We find a different inference using the maximal eigenvalue statistic. This statistic allows us to reject $r \leq 0$ as well as $r \leq 1$ and $r \leq 2$ at the 95% level. We cannot reject $r \leq 3$, because the maximal eigenvalue statistic of 30.791 does not exceed the critical value of 33.878 associated with the 95% level.

This would lead to the inference that $r = 3$, in contrast to $r = 6$ indicated by the trace statistic. Using similar reasoning at the 99% level, we would infer $r = 2$ from the maximal eigenvalue statistics.

After the **johansen** test determines the number of co-integrating relationships, we can use these results along with the eigenvectors returned by the **johansen** function, to form a set of error correction variables. These are constructed using y_{t-1}, (the levels of y lagged one period) multiplied by the r eigenvectors associated with the co-integrating relationships to form r co-integrating variables. This is carried out by the **ecm** function, documented below.

```
 PURPOSE: performs error correction model estimation
---------------------------------------------------
 USAGE: result = ecm(y,nlag,r)
 where:    y    = an (nobs x neqs) matrix of y-vectors in levels
           nlag = the lag length
           r    = # of cointegrating relations to use
                  (optional: this will be determined using
                  Johansen's trace test at 95%-level if left blank)
 NOTES: constant vector automatically included
         x-matrix of exogenous variables not allowed
         error correction variables are automatically
         constructed using output from Johansen's ML-estimator
---------------------------------------------------
 RETURNS a structure
 results.meth = 'ecm'
 results.nobs = nobs, # of observations
 results.neqs = neqs, # of equations
 results.nlag = nlag, # of lags
 results.nvar = nlag*neqs+nx+1, # of variables per equation
 results.coint= # of co-integrating relations (or r if input)
 results.index= index of co-integrating variables ranked by
               size of eigenvalues large to small
 --- the following are referenced by equation # ---
 results(eq).beta   = bhat for equation eq (includes ec-bhats)
 results(eq).tstat  = t-statistics
 results(eq).tprob  = t-probabilities
 results(eq).resid  = residuals
 results(eq).yhat   = predicted values (levels) (nlag+2:nobs,1)
 results(eq).dyhat  = predicted values (differenced) (nlag+2:nobs,1)
 results(eq).y      = actual y-level values (nobs x 1)
 results(eq).dy     = actual y-differenced values (nlag+2:nobs,1)
 results(eq).sige   = e'e/(n-k)
 results(eq).rsqr   = r-squared
 results(eq).rbar   = r-squared adjusted
 results(eq).ftest  = Granger F-tests
 results(eq).fprob  = Granger marginal probabilities
```

```
---------------------------------------------------
```

The **ecm** function allows two options for implementing an EC model. One option is to specify the number of co-integrating relations to use, and the other is to let the **ecm** function determine this number using the **johansen** function and the trace statistics along with the critical values at the 95% level of significance. The motivation for using the trace statistic is that it seems better suited to the task of sequential hypotheses for our particular decision problem. If you find this decision problematical, you can specify r as an input to the **ecm** function, or change the code presented below to rely on the maximal eigenvalue statistic.

To illustrate how this is carried out by the **ecm** function consider the following code from the function.

```
if nargin == 3 % user is specifying the # of error correction terms to
               % include -- get them using johansen()
jres = johansen(y,0,nlag);
% recover error correction vectors
ecvectors = jres.evec;
    index = jres.ind; % recover index of ec variables
% construct r-error correction variables
x = mlag(y(:,index),1)*ecvectors(:,1:r);
   [nobs2 nx] = size(x);

elseif nargin == 2 % we need to find r
jres = johansen(y,0,nlag);
% find r = # significant co-integrating relations using
% the trace statistic output
trstat = jres.lr1; % max eigenvalue stats are in jres.lr2
tsignf = jres.cvt; % max eigenvalue crit values are in jres.cvm
r = 0;
for i=1:neqs;
if trstat(i,1) > tsignf(i,2)
r = i;
end;
end;
% recover error correction vectors
ecvectors = jres.evec;
    index = jres.ind; % recover index of ec variables
% construct r error correction variables
x = mlag(y(:,index),1)*ecvectors(:,1:r);
   [junk nx] = size(x);
else
error('Wrong # of arguments to ecm');
end;
```

Based on the number of input arguments, we know if the user is specifying the number of co-integrating relations (the case where **nargin == 3**) or wishes us to determine this, (the case where **nargin == 2**). The function calls **johansen** which computes eigenvalues and eigenvectors along with the trace and maximal eigenvalue statistics. In the case where the user specifies the number of co-integrating relations as an input argument r, we simply use this number of eigenvectors multiplied times the levels lagged 1 period to form the r error correction variables. One implementation detail is that the variables involved in the co-integrating relationships are those associated with the largest eigenvalues. The **johansen** function sorts the eigenvalues and eigenvectors by size and returns an index vector that points to the variables involved in the co-integrating relationships. We use this index to arrange the levels variables in y in a similar order as the sorted eigenvectors before carrying out the multiplication to produce error correction variables.

For the case where the user leaves determination of r to the **ecm** function, we loop over the trace statistics and find the value of r using the 95% significance level. This value is then used to form r error correction variables.

From this point, we need simply transform the y matrix to first differences and call our **var** estimation routine using y transformed to first differences along with the r error correction variables plus a constant entered as deterministic variables in the call to **var**.

An identical approach can be taken to implement a Bayesian variant of the EC model based on the Minnesota prior as well as a more recent Bayesian variant based on a "random-walk averaging prior". Both of these are discussed in the next section. To summarize, we can implement EC models using Bayesian priors as well as the usual case with no priors by simply calling **johansen** to determine the eigenvectors and number of co-integrating relationships if the user request this. We then transform the data to first-difference form and call the associated vector autoregressive estimation procedure using the differenced data and the error correction variables along with a constant term.

The **prt_var** function will produce a printout of the results structure returned by **ecm** showing the autoregressive plus error correction variable coefficients along with Granger-causality test results as well as the trace, maximal eigenvalue statistics, and critical values from the **johansen** procedure. As an example, we show a program to estimate an EC model based on our eight-state sample of monthly mining employment, where we have set the lag-length to 2 to conserve on the amount of printed output.

```
% ----- Example 5.8 Estimating error correction models
y = load('test.dat'); % monthly mining employment for
```

```
                      % il,in,ky,mi,oh,pa,tn,wv 1982,1 to 1996,5
vnames =  strvcat('il','in','ky','mi','oh','pa','tn','wv');
nlag = 2;  % number of lags in var-model
% estimate the model, letting ecm determine # of co-integrating vectors
result = ecm(y,nlag);
prt(result,vnames); % print results to the command window
```

The printed output is shown below for a single state indicating the presence of two co-integrating relationships involving the states of Illinois and Indiana. The estimates for the error correction variables are labeled as such in the printout. Granger causality tests are printed, and these would form the basis for valid causality inferences in the case where co-integrating relationships existed among the variables in the VAR model.

```
Dependent Variable =               wv
R-squared     =    0.1975
Rbar-squared  =    0.1018
sige          =  341.6896
Nobs, Nvars   =    170,    19
******************************************************************
Variable              Coefficient      t-statistic    t-probability
il  lag1                 0.141055         0.261353         0.794176
il  lag2                 0.234429         0.445400         0.656669
in  lag1                 1.630666         1.517740         0.131171
in  lag2                -1.647557        -1.455714         0.147548
ky  lag1                 0.378668         1.350430         0.178899
ky  lag2                 0.176312         0.631297         0.528801
mi  lag1                 0.053280         0.142198         0.887113
mi  lag2                 0.273078         0.725186         0.469460
oh  lag1                -0.810631        -1.449055         0.149396
oh  lag2                 0.464429         0.882730         0.378785
pa  lag1                -0.597630        -2.158357         0.032480
pa  lag2                -0.011435        -0.038014         0.969727
tn  lag1                -0.049296        -0.045237         0.963978
tn  lag2                 0.666889         0.618039         0.537480
wv  lag1                -0.004150        -0.033183         0.973572
wv  lag2                -0.112727        -0.921061         0.358488
ec term il              -2.158992        -1.522859         0.129886
ec term in              -2.311267        -1.630267         0.105129
constant                 8.312788         0.450423         0.653052
 ****** Granger Causality Tests *******
Variable          F-value      Probability
il               0.115699         0.890822
in               2.700028         0.070449
ky               0.725708         0.485662
mi               0.242540         0.784938
oh               1.436085         0.241087
pa               2.042959         0.133213
```

```
tn               0.584267         0.558769
wv               1.465858         0.234146
 Johansen MLE estimates
NULL:         Trace Statistic      Crit 90%       Crit 95%       Crit 99%
r <= 0   il           214.390       153.634        159.529        171.090
r <= 1   in           141.482       120.367        125.618        135.982
r <= 2   ky            90.363        91.109         95.754        104.964
r <= 3   oh            61.555        65.820         69.819         77.820
r <= 4   tn            37.103        44.493         47.855         54.681
r <= 5   wv            21.070        27.067         29.796         35.463
r <= 6   pa            10.605        13.429         15.494         19.935
r <= 7   mi             3.192         2.705          3.841          6.635
NULL:         Eigen Statistic      Crit 90%       Crit 95%       Crit 99%
r <= 0   il            72.908        49.285         52.362         58.663
r <= 1   in            51.118        43.295         46.230         52.307
r <= 2   ky            28.808        37.279         40.076         45.866
r <= 3   oh            24.452        31.238         33.878         39.369
r <= 4   tn            16.034        25.124         27.586         32.717
r <= 5   wv            10.465        18.893         21.131         25.865
r <= 6   pa             7.413        12.297         14.264         18.520
r <= 7   mi             3.192         2.705          3.841          6.635
```

The results indicate that given the two lag model, two co-integrating relationships were found leading to the inclusion of two error correction variables in the model. The co-integrating relationships are based on the trace statistics compared to the critical values at the 95% level. From the trace statistics in the printed output we see that, H0: $r \leq 2$ was rejected at the 95% level because the trace statistic of 90.363 is less than the associated critical value of 95.754. Keep in mind that the user has the option of specifying the number of co-integrating relations to be used in the **ecm** function as an optional argument. If you wish to work at the 90% level of significance, we would conclude from the **johansen** results that $r = 4$ co-integrating relationships exist. To estimate an **ecm** model based on $r = 4$ we need simply call the **ecm** function with:

```
% estimate the model, using 4 co-integrating vectors
result = ecm(y,nlag,4);
```

5.3 Bayesian variants

Despite the attractiveness of drawing on cross-sectional information from related economic variables, the VAR model has empirical limitations. For example, a model with eight variables and six lags produces 49 independent variables in each of the eight equations of the model for a total of 392 coefficients to estimate. Large samples of observations involving time series variables that cover many years are needed to estimate the VAR model, and these are not always available. In addition, the independent variables represent lagged values, e.g., $y_{1t-1}, y_{1t-2}, \ldots, y_{1t-6}$, which tend to produce high correlations that lead to degraded precision in the parameter estimates. To overcome these problems, Doan, Litterman and Sims (1984) proposed the use of Bayesian prior information. The Minnesota prior means and variances suggested take the following form:

$$\begin{array}{rcl} \beta_i & \sim & N(1, \sigma^2_{\beta_i}) \\ \beta_j & \sim & N(0, \sigma^2_{\beta_j}) \end{array} \tag{5.6}$$

where β_i denotes the coefficients associated with the lagged dependent variable in each equation of the VAR and β_j represents any other coefficient. The prior means for lagged dependent variables are set to unity in belief that these are important explanatory variables. On the other hand, a prior mean of zero is assigned to all other coefficients in the equation, β_j in (5.6), indicating that these variables are viewed as less important in the model.

The prior variances, $\sigma^2_{\beta_i}$, specify uncertainty about the prior means $\bar{\beta}_i = 1$, and $\sigma^2_{\beta_j}$ indicates uncertainty regarding the means $\bar{\beta}_j = 0$. Because the VAR model contains a large number of parameters, Doan, Litterman and Sims (1984) suggested a formula to generate the standard deviations as a function of a small number of hyperparameters: θ, ϕ and a weighting matrix $w(i,j)$. This approach allows a practitioner to specify individual prior variances for a large number of coefficients in the model using only a few parameters that are labeled hyperparameters. The specification of the standard deviation of the prior imposed on variable j in equation i at lag k is:

$$\sigma_{ijk} = \theta w(i,j) k^{-\phi} \left(\frac{\hat{\sigma}_{uj}}{\hat{\sigma}_{ui}} \right) \tag{5.7}$$

where $\hat{\sigma}_{ui}$ is the estimated standard error from a univariate autoregression involving variable i, so that $(\hat{\sigma}_{uj}/\hat{\sigma}_{ui})$ is a scaling factor that adjusts for varying magnitudes of the variables across equations i and j. Doan, Litterman and Sims (1984) labeled the parameter θ as 'overall tightness', reflecting the standard deviation of the prior on the first lag of the dependent variable. The term $k^{-\phi}$ is a lag decay function with $0 \leq \phi \leq 1$ reflecting the decay rate, a shrinkage of the standard deviation with increasing lag length. This has the effect of imposing the prior means of zero more tightly as the lag length increases, based on the belief that more distant lags represent less important variables in the model. The function $w(i,j)$ specifies the tightness of the prior for variable j in equation i relative to the tightness of the own-lags of variable i in equation i.

The overall tightness and lag decay hyperparameters used in the standard Minnesota prior have values $\theta = 0.1$, $\phi = 1.0$. The weighting matrix used is:

$$W = \begin{bmatrix} 1 & 0.5 & \dots & 0.5 \\ 0.5 & 1 & & 0.5 \\ \vdots & & \ddots & \vdots \\ 0.5 & 0.5 & \dots & 1 \end{bmatrix} \tag{5.8}$$

This weighting matrix imposes $\bar{\beta}_i = 1$ loosely, because the lagged dependent variable in each equation is felt to be an important variable. The weighting matrix also imposes the prior mean of zero for coefficients on other variables in each equation more tightly since the β_j coefficients are associated with variables considered less important in the model.

A function **bvar** will provide estimates for this model. The function documentation is:

```
PURPOSE: Performs a Bayesian vector autoregression of order n
---------------------------------------------------
USAGE:  result = bvar(y,nlag,tight,weight,decay,x)
where:    y    = an (nobs x neqs) matrix of y-vectors
          nlag = the lag length
         tight = Litterman's tightness hyperparameter
        weight = Litterman's weight (matrix or scalar)
         decay = Litterman's lag decay = lag^(-decay)
          x    = an optional (nobs x nx) matrix of variables
NOTE:  constant vector automatically included
---------------------------------------------------
RETURNS: a structure:
results.meth      = 'bvar'
results.nobs      = nobs, # of observations
results.neqs      = neqs, # of equations
results.nlag      = nlag, # of lags
results.nvar      = nlag*neqs+1+nx, # of variables per equation
results.tight     = overall tightness hyperparameter
results.weight    = weight scalar or matrix hyperparameter
results.decay     = lag decay hyperparameter
--- the following are referenced by equation # ---
results(eq).beta  = bhat for equation eq
results(eq).tstat = t-statistics
```

```
  results(eq).tprob = t-probabilities
  results(eq).resid = residuals
  results(eq).yhat  = predicted values
  results(eq).y     = actual values
  results(eq).sige  = e'e/(n-k)
  results(eq).rsqr  = r-squared
  results(eq).rbar  = r-squared adjusted
  ---------------------------------------------------
  SEE ALSO:  bvarf, var, ecm, rvar, plt_var, prt_var
  ---------------------------------------------------
```

The function **bvar** allows us to input a scalar weight value or a more general matrix. Scalar inputs will be used to form a symmetric prior, where the scalar is used on the off-diagonal elements of the matrix. A matrix will be used in the form submitted to the function.

As an example of using the **bvar** function, consider our case of monthly mining employment for eight states. A program to estimate a BVAR model based on the Minnesota prior is shown below:

```
% ----- Example 5.9 Estimating BVAR models
vnames =  strvcat('il','in','ky','mi','oh','pa','tn','wv');
y = load('test.dat'); % use all eight states
nlag = 2;
tight = 0.1;  % hyperparameter values
weight = 0.5;
decay = 1.0;
result = bvar(y,nlag,tight,weight,decay);
prt(result,vnames);
```

The printout shows the hyperparameter values associated with the prior. It does not provide Granger-causality test results as these are invalid given the Bayesian prior applied to the model. Results for a single equation of the mining employment example are shown below.

```
 ***** Bayesian Vector Autoregressive Model *****
 *****    Minnesota type Prior         *****
PRIOR hyperparameters
tightness =     0.10
decay     =     1.00
Symmetric weights based on     0.50

Dependent Variable =              il
R-squared     =    0.9942
Rbar-squared  =    0.9936
sige          =   12.8634
Nobs, Nvars   =    171,    17
```

```
****************************************************************
Variable              Coefficient      t-statistic    t-probability
il  lag1                 1.134855        11.535932         0.000000
il  lag2                -0.161258        -1.677089         0.095363
in  lag1                 0.390429         1.880834         0.061705
in  lag2                -0.503872        -2.596937         0.010230
ky  lag1                 0.049429         0.898347         0.370271
ky  lag2                -0.026436        -0.515639         0.606776
mi  lag1                -0.037327        -0.497504         0.619476
mi  lag2                -0.026391        -0.377058         0.706601
oh  lag1                -0.159669        -1.673863         0.095996
oh  lag2                 0.191425         2.063498         0.040585
pa  lag1                 0.179610         3.524719         0.000545
pa  lag2                -0.122678        -2.520538         0.012639
tn  lag1                 0.156344         0.773333         0.440399
tn  lag2                -0.288358        -1.437796         0.152330
wv  lag1                -0.046808        -2.072769         0.039703
wv  lag2                 0.014753         0.681126         0.496719
constant                 9.454700         2.275103         0.024149
```

There exists a number of attempts to alter the fact that the Minnesota prior treats all variables in the VAR model except the lagged dependent variable in an identical fashion. Some of the modifications suggested have focused entirely on alternative specifications for the prior variance. Usually, this involves a different (non-symmetric) weight matrix W and a larger value of 0.2 for the overall tightness hyperparameter θ in place of the value $\theta = 0.1$ used in the Minnesota prior. The larger overall tightness hyperparameter setting allows for more influence from other variables in the model. For example, LeSage and Pan (1995) constructed a weight matrix based on first-order spatial contiguity to emphasize variables from neighboring states in a multi-state agricultural output forecasting model. LeSage and Magura (1991) employed interindustry input-output weights to place more emphasis on related industries in a multi-industry employment forecasting model.

These approaches can be implemented using the **bvar** function by constructing an appropriate weight matrix. For example, the first order contiguity structure for the eight states in our mining employment example can be converted to a set of prior weights by placing values of unity on the main diagonal of the weight matrix, and in positions that represent contiguous entities. An example is shown in (5.9), where row 1 of the weight matrix is associated with the time-series for the state of Illinois. We place a value of unity on the main diagonal to indicate that autoregressive values from Illinois are considered important variables. We also place values of one in columns 2 and 3, reflecting the fact that Indiana (variable 2) and Kentucky (variable 3) are states that have borders touching Illinois. For other states that are not neighbors to Illinois, we use a weight of 0.1 to downweight their influence in the BVAR model equation for Illinois. A similar scheme is used to specify weights for the other seven states based on neighbors and non-neighbors.

$$W = \begin{bmatrix} 1.0 & 1.0 & 1.0 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 \\ 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.1 & 0.1 & 0.1 \\ 1.0 & 1.0 & 1.0 & 0.1 & 1.0 & 0.1 & 1.0 & 1.0 \\ 0.1 & 1.0 & 0.1 & 1.0 & 1.0 & 0.1 & 0.1 & 0.1 \\ 0.1 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.1 & 1.0 \\ 0.1 & 0.1 & 0.1 & 0.1 & 1.0 & 1.0 & 0.1 & 1.0 \\ 0.1 & 0.1 & 1.0 & 0.1 & 0.1 & 0.1 & 1.0 & 0.1 \\ 0.1 & 0.1 & 1.0 & 0.1 & 1.0 & 1.0 & 0.1 & 1.0 \end{bmatrix} \tag{5.9}$$

The intuition behind this set of weights is that we really don't believe the prior means of zero placed on the coefficients for mining employment in neighboring states. Rather, we believe these variables should exert an important influence. To express our lack of faith in these prior means, we assign a large prior variance to the zero prior means for these states by increasing the weight values. This allows the coefficients for these time-series variables to be determined by placing more emphasis on the sample data and less emphasis on the prior.

This could of course be implemented using **bvar** with a weight matrix specified, e.g.,

```
% ----- Example 5.10 Using bvar() with general weights
vnames =  strvcat('il','in','ky','mi','oh','pa','tn','wv');
dates = cal(1982,1,12);
y = load('test.dat'); % use all eight states
nlag = 2;
tight = 0.1;
decay = 1.0;

w = [1.0  1.0  1.0  0.1  0.1  0.1  0.1  0.1
     1.0  1.0  1.0  1.0  1.0  0.1  0.1  0.1
     1.0  1.0  1.0  0.1  1.0  0.1  1.0  1.0
     0.1  1.0  0.1  1.0  1.0  0.1  0.1  0.1
     0.1  1.0  1.0  1.0  1.0  1.0  0.1  1.0
     0.1  0.1  0.1  0.1  1.0  1.0  0.1  1.0
     0.1  0.1  1.0  0.1  0.1  0.1  1.0  0.1
     0.1  0.1  1.0  0.1  1.0  1.0  0.1  1.0];

result = bvar(y,nlag,tight,w,decay);
prt(result,vnames);
```

Another more recent approach to altering the equal treatment character of the Minnesota prior is a "random-walk averaging prior" suggested by LeSage and Krivelyova (1999).

As noted above, previous attempts to alter the fact that the Minnesota prior treats all variables in the VAR model except the first lag of the dependent variable in an identical fashion have focused entirely on alternative specifications for the prior variance. The prior proposed by LeSage and Krivelyova (1999) involves both prior means and variances motivated by the distinction between important and unimportant variables in each equation of the VAR model. To motivate the prior means, consider the weighting matrix for a five variable VAR model shown in (5.10). The weight matrix contains values of unity in positions associated with important variables in each equation of the VAR model and values of zero for unimportant variables. For example, the important variables in the first equation of the VAR model are variables 2 and 3 whereas the important variables in the fifth equation are variables 4 and 5.

Note that if we do not believe that autoregressive influences reflected by lagged values of the dependent variable are important, we have a zero on the main diagonal of the weight matrix. In fact, the weighting matrix shown in (5.10) classifies autoregressive influences as important in only two of the five equations in the VAR system, equations three and five. As an example of a case where autoregressive influences are totally ignored LeSage and Krivelyova (1999) constructed a VAR system based on spatial contiguity that relies entirely on the influence of neighboring states and ignores the autoregressive influence associated with past values of the variables from the states themselves.

$$W = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{5.10}$$

The weight matrix shown in (5.10) is standardized to produce row-sums of unity resulting in the matrix labeled C shown in (5.11).

$$C = \begin{bmatrix} 0 & 0.5 & 0.5 & 0 & 0 \\ 0.5 & 0 & 0.5 & 0 & 0 \\ 0.33 & 0.33 & 0.33 & 0 & 0 \\ 0 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & 0.5 & 0.5 \end{bmatrix} \tag{5.11}$$

Using the row-standardized matrix C, we consider the random-walk with drift that averages over the important variables in each equation i of the VAR model as shown in (5.12).

$$y_{it} = \alpha_i + \sum_{j=1}^{n} C_{ij} y_{jt-1} + u_{it} \tag{5.12}$$

Expanding expression (5.12) we see that multiplying $y_{jt-1}, j = 1, \ldots, 5$ containing 5 variables at time $t-1$ by the row-standardized weight matrix C shown in (5.11) produces a set of explanatory variables for each equation of the VAR system equal to the mean of observations from important variables in each equation at time $t-1$ as shown in (5.13).

$$\begin{bmatrix} y_{1t} \\ y_{2t} \\ y_{3t} \\ y_{4t} \\ y_{5t} \end{bmatrix} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \\ \alpha_5 \end{bmatrix} + \begin{bmatrix} 0.5y_{2t-1} + 0.5y_{3t-1} \\ 0.5y_{1t-1} + 0.5y_{3t-1} \\ 0.33y_{1t-1} + 0.33y_{2t-1} + 0.33y_{3t-1} \\ 0.5y_{3t-1} + 0.5y_{5t-1} \\ 0.5y_{4t-1} + 0.5y_{5t-1} \end{bmatrix} + \begin{bmatrix} u_{1t} \\ u_{2t} \\ u_{3t} \\ u_{4t} \\ u_{5t} \end{bmatrix} \tag{5.13}$$

This suggests a prior mean for the VAR model coefficients on variables associated with the first own-lag of important variables equal to $1/c_i$, where c_i is the number of important variables in each equation i of the model. In the example shown in (5.13), the prior means for the first own-lag of the important variables y_{2t-1} and y_{3t-1} in the y_{1t} equation of the VAR would equal 0.5. The prior means for unimportant variables, y_{1t-1}, y_{4t-1} and y_{5t-1} in this equation would be zero.

This prior is quite different from the Minnesota prior in that it may downweight the lagged dependent variable using a zero prior mean to discount the autoregressive influence of past values of this variable. In contrast, the Minnesota prior emphasizes a random-walk with drift model that relies on prior means centered on a model: $y_{it} = \alpha_i + y_{it-1} + u_{it}$, where the intercept term reflects drift in the random-walk model and is estimated using a diffuse prior. The random-walk averaging prior is centered on a random-walk model that averages over important variables in each equation of the model and allows for drift as well. As in the case of the Minnesota prior, the drift parameters α_i are estimated using a diffuse prior.

Consistent with the Minnesota prior, LeSage and Krivelyova use zero as a prior mean for coefficients on all lags other than first lags. Litterman (1986) motivates reliance on zero prior means for many of the parameters of the VAR model by appealing to ridge regression. Recall, ridge regression can be interpreted as a Bayesian model that specifies prior means of zero for all coefficients, and as we saw in Chapter 4 can be used to overcome collinearity problems in regression models.

One point to note about the random walk averaging approach to specifying prior means is that the time series for the variables in the model need to be scaled or transformed to have similar magnitudes. If this is not the case, it would make little sense to indicate that the value of a time series observation at time t was equal to the average of values from important related variables at time $t-1$. This should present no problem as time series data can always be expressed in percentage change form or annualized growth rates which meets our requirement that the time series have similar magnitudes.

The prior variances LeSage and Krivelyova specify for the parameters in the model differ according to whether the coefficients are associated with variables that are classified as important or unimportant as well as the lag length. Like the Minnesota prior, they impose lag decay to reflect a prior belief that time series observations from the more distant past exert a smaller influence on the current value of the time series we are modeling. Viewing variables in the model as important versus unimportant suggests that the prior variance (uncertainty) specification should reflect the following ideas:

1. Parameters associated with unimportant variables should be assigned a smaller prior variance, so the zero prior means are imposed more 'tightly' or with more certainty.

2. First own-lags of important variables are given a smaller prior variance, so the prior means force averaging over the first own-lags of important variables.

3. Parameters associated with unimportant variables at lags greater than one will be given a prior variance that becomes smaller as the lag length increases to reflect our belief that influence decays with time.

4. Parameters associated with lags other than first own-lag of important variables will have a larger prior variance, so the prior means of zero are imposed 'loosely'. This is motivated by the fact that we don't really have a great deal of confidence in the zero prior mean specification for longer lags of important variables. We think they should exert some influence, making the prior mean of zero somewhat inappropriate. We still impose lag decay on longer lags of important variables by decreasing our prior variance with increasing lag length. This reflects the idea that influence decays over time for important as well as unimportant variables.

It should be noted that the prior relies on inappropriate zero prior means for the important variables at lags greater than one for two reasons. First, it is difficult to specify a reasonable alternative prior mean for these variables that would have universal applicability in a large number of VAR model applications. The difficulty of assigning meaningful prior means that have universal appeal is most likely the reason that past studies relied on the Minnesota prior means while simply altering the prior variances. A prior mean that averages over previous period values of the important variables has universal appeal and widespread applicability in VAR modeling. The second motivation for relying on inappropriate zero prior means for longer lags of the important variables is that overparameterization and collinearity problems that plague the VAR model are best overcome by relying on a parsimonious representation. Zero prior means for the majority of the large number of coefficients in the VAR model are consistent with this goal of parsimony and have been demonstrated to produce improved forecast accuracy in a wide variety of applications of the Minnesota prior.

A flexible form with which to state prior standard deviations for variable j in equation i at lag length k is shown in (5.14).

$$\begin{array}{lllll} \pi(a_{ijk}) = N(1/c_i, \sigma_c), & j \in C, & k = 1, & i, j = 1, \ldots, n \\ \pi(a_{ijk}) = N(0, \tau\sigma_c/k), & j \in C, & k = 2, \ldots, m, & i, j = 1, \ldots, n \\ \pi(a_{ijk}) = N(0, \theta\sigma_c/k), & j \neg \in C, & k = 1, \ldots, m, & i, j = 1, \ldots, n \end{array} \tag{5.14}$$

where:

$$0 < \sigma_c < 1 \tag{5.15}$$

$$\tau > 1 \tag{5.16}$$

$$0 < \theta < 1 \tag{5.17}$$

For variables $j = 1, \ldots, m$ in equation i that are important in explaining variation in variable i, $(j \in C)$, the prior mean for lag length $k = 1$ is set to the average of the number of important variables in equation i and to zero for unimportant variables $(j \neg \in C)$. The prior standard deviation is set to σ_c for the first lag, and obeys the restriction set forth in (5.15), reflecting a tight imposition of the prior mean that forces averaging over important variables. To see this, consider that the prior means $1/c_i$ range between zero and unity so typical σ_c values might be in the range of 0.1 to 0.25. We use $\tau\sigma_c/k$ for lags greater than one which imposes a decrease in this variance as the lag length k increases. Equation (5.16) states the restriction necessary to ensure that the prior mean of zero is imposed on the parameters associated with lags greater than one for important variables loosely, relative to a tight imposition of the prior mean of $1/c_i$ on first own-lags of important variables. We use $\theta\sigma_c/k$ for lags on unimportant variables whose prior means are zero, imposing a decrease in the variance as the lag length increases. The restriction in (5.17) would impose the zero means for unimportant variables with more confidence than the zero prior means for important variables.

This mathematical formulation adequately captures all aspects of the intuitive motivation for the prior variance specification enumerated above. A quick way to see this is to examine a graphical depiction of the prior mean and standard deviation for an important versus unimportant variable. An artificial example was constructed for an important variable in Figure 5.1 and an unimportant variable in Figure 5.2. Figure 5.1 shows the prior mean along with five upper and lower limits derived from the prior standard deviations in (5.14). The five standard deviation limits shown in the figure reflect $\pm$ 2 standard deviation limits resulting from alternative settings for the prior hyperparameter τ ranging from 5 to 9 and a value of $\sigma_c = 0.25$. Larger values of τ generated the wider upper and lower limits.

The solid line in Figure 5.1 reflects a prior mean of 0.2 for lag 1 indicating five important variables, and a prior mean of zero for all other lags. The prior standard deviation at lag 1 is relatively tight producing a small band around the averaging prior mean for this lag. This imposes the 'averaging' prior belief with a fair amount of certainty. Beginning at lag 2, the prior standard deviation is increased to reflect relative uncertainty about the new prior mean of zero for lags greater than unity. Recall, we believe that important variables at lags greater than unity will exert some influence, making the prior mean of zero not quite appropriate. Hence, we implement this prior mean with greater uncertainty.

Figure 5.2 shows an example of the prior means and standard deviations for an unimportant variable based on $\sigma_c = 0.25$ and five values of θ ranging from .35 to .75. Again, the larger θ values produce wider upper and lower limits. The prior for unimportant variables is motivated by the Minnesota prior that also uses zero prior means and rapid decay with increasing lag length.

A function **rvar** implements the random-walk averaging prior and a related function **recm** carries out estimation for an EC model based on this prior. The documentation for the **rvar** function is shown below, where we have eliminated information regarding the results structure variable returned by the function to save space.

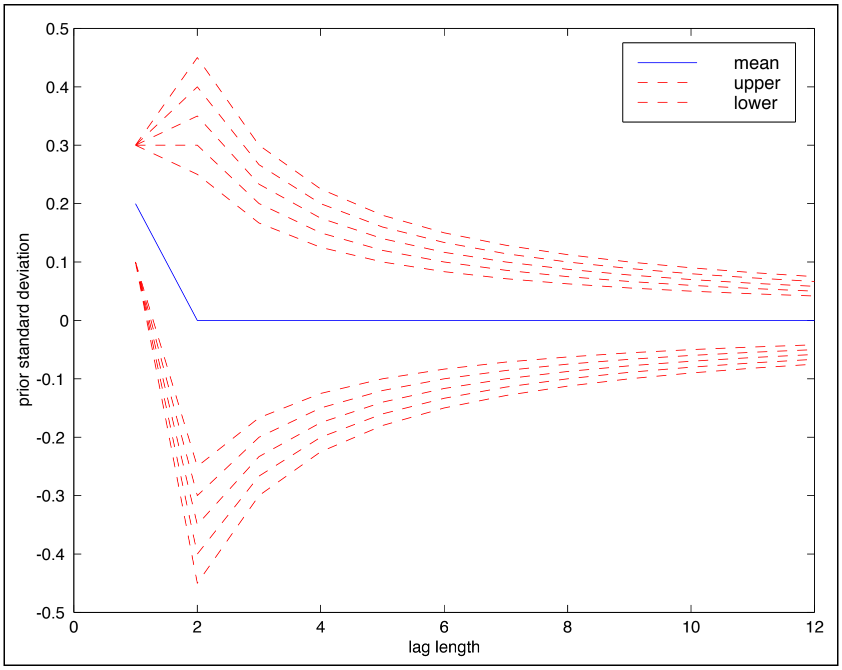

Figure 5.2: Prior means and precision for important variables

```
 PURPOSE: Estimates a Bayesian vector autoregressive model
          using the random-walk averaging prior
---------------------------------------------------
 USAGE:  result = rvar(y,nlag,w,freq,sig,tau,theta,x)
 where:    y    = an (nobs x neqs) matrix of y-vectors (in levels)
           nlag = the lag length
           w    = an (neqs x neqs) matrix containing prior means
                  (rows should sum to unity, see below)
           freq = 1 for annual, 4 for quarterly, 12 for monthly
           sig  = prior variance hyperparameter (see below)
           tau  = prior variance hyperparameter (see below)
          theta = prior variance hyperparameter (see below)
           x    = an (nobs x nx) matrix of deterministic variables
                  (in any form, they are not altered during estimation)
                  (constant term automatically included)
 priors important variables:   N(w(i,j),sig) for 1st own lag
                               N(  0 ,tau*sig/k) for lag k=2,...,nlag
 priors unimportant variables: N(w(i,j),theta*sig/k) for lag 1
                               N(  0 ,theta*sig/k)   for lag k=2,...,nlag
 e.g., if y1, y3, y4 are important variables in eq#1, y2 unimportant
  w(1,1) = 1/3, w(1,3) = 1/3, w(1,4) = 1/3, w(1,2) = 0
```

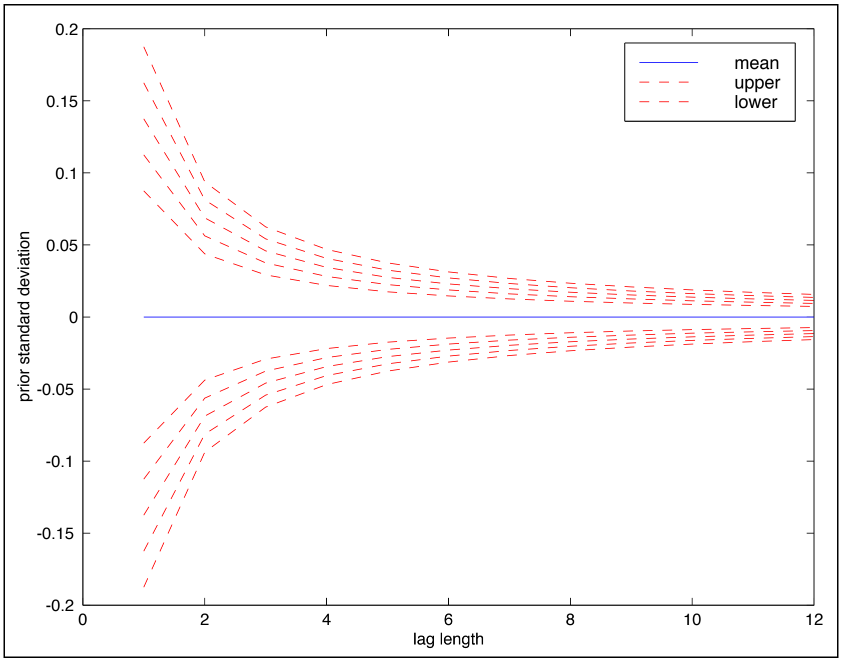

Figure 5.3: Prior means and precision for unimportant variables

```
 typical values would be: sig = .1-.3, tau = 4-8, theta = .5-1
---------------------------------------------------
 NOTES: - estimation is carried out in annualized growth terms because
          the prior means rely on common (growth-rate) scaling of variables
          hence the need for a freq argument input.
        - constant term included automatically
---------------------------------------------------
```

Because this model is estimated in growth-rates form, an input argument for the data frequency is required. As an illustration of using both the **rvar** and **recm** functions, consider the following example based on the eight-state mining industry data. We specify a weight matrix for the prior means using first-order contiguity of the states.

```
% ----- Example 5.11 Estimating RECM models
y = load('test.dat'); % a test data set
vnames =  strvcat('il','in','ky','mi','oh','pa','tn','wv');
nlag = 6;  % number of lags in var-model
sig = 0.1;
tau = 6;
theta = 0.5;
```

```
freq = 12;   % monthly data
% this is an example of using 1st-order contiguity
% of the states as weights to produce prior means
W=[0      0.5    0.5    0      0      0     0     0
   0.25   0      0.25   0.25   0.25   0     0     0
   0.20   0.20   0      0      0.20   0     0.20  0.20
   0      0.50   0      0      0.50   0     0     0
   0      0.20   0.20   0.20   0      0.20  0.20  0.20
   0      0      0      0      0.50   0     0     0.50
   0      0      1      0      0      0     0     0
   0      0      0.33   0      0.33   0.33  0     0];
% estimate the rvar model
results = rvar(y,nlag,W,freq,sig,tau,theta);
% print results to a file
fid = fopen('rvar.out','wr');
prt(results,vnames,fid);
% estimate the recm model letting the function
% determine the # of co-integrating relationships
results = recm(y,nlag,W,freq,sig,tau,theta);
% print results to a file
fid = fopen('recm.out','wr');
prt(results,vnames,fid);
```

5.3.1 Theil-Goldberger estimation of these models

VAR models based on both the Minnesota prior and the random-walk averaging prior can be estimated using the 'mixed estimation' method set forth in Theil and Goldberger (1961). Letting the matrix X represent the lagged values of $y_{it}, i = 1, \ldots, n$ and the vector A denote the coefficients $a_{ijk}(\ell)$ from (5.1), we can express a single equation of the model as:

$$y_1 = XA + \varepsilon_1 \tag{5.18}$$

where it is assumed, $\mathrm{var}(\varepsilon_1) = \sigma^2 I$. The stochastic prior restrictions for this single equation can be written as:

$$\begin{bmatrix} m_{111} \\ m_{112} \\ \vdots \\ m_{nnk} \end{bmatrix} = \begin{bmatrix} \sigma/\sigma_{111} & 0 & \ldots & 0 \\ 0 & \sigma/\sigma_{112} & 0 & 0 \\ 0 & & \ddots & 0 \\ 0 & 0 & 0 & \sigma/\sigma_{nnk} \end{bmatrix} \begin{bmatrix} a_{111} \\ a_{112} \\ \vdots \\ a_{nnk} \end{bmatrix} + \begin{bmatrix} u_{111} \\ u_{112} \\ \vdots \\ u_{nnk} \end{bmatrix} \tag{5.19}$$

where we assume, $\mathrm{var}(u) = \sigma^2 I$ and the σ_{ijk} take the form shown in (5.7) for the Minnesota prior, and that set forth in (5.14) for the random-walk averaging prior model. Similarly, the prior means m_{ijk} take the form described for the Minnesota and averaging priors. Noting that (5.19) can be written in the form suggested by Theil-Goldberger:

$$r = RA + u, \tag{5.20}$$

the estimates for a typical equation are derived using (5.21).

$$\hat{A} = (X'X + R'R)^{-1}(X'y_1 + R'r) \tag{5.21}$$

The difference in prior means specified by the Minnesota prior and the random-walk averaging prior resides in the m_{ijk} terms found on the left-hand-side of the equality in (5.20). The Minnesota prior indicates values: $(\sigma/\sigma_{111}, 0, \ldots, 0)'$, where the non-zero value occurs in the position representing the lagged dependent variable. The averaging prior would have non-zero values in locations associated with important variables and zeros elsewhere.

5.4 Forecasting the models

A set of forecasting functions are available that follow the format of the **var, bvar, rvar, ecm, becm, recm** functions named **varf, bvarf, rvarf, ecmf, becmf, recmf**. These functions all produce forecasts of the time-series levels to simplify accuracy analysis and forecast comparison from the alternative models. They all take time-series levels arguments as inputs and carry out the necessary transformations. As an example, the **varf** documentation is:

```
PURPOSE: estimates a vector autoregression of order n
          and produces f-step-ahead forecasts
-------------------------------------------------------------
 USAGE:yfor = varf(y,nlag,nfor,begf,x,transf)
 where:    y    = an (nobs * neqs) matrix of y-vectors in levels
           nlag = the lag length
           nfor = the forecast horizon
           begf = the beginning date of the forecast
                  (defaults to length(x) + 1)
           x    = an optional vector or matrix of deterministic
                  variables (not affected by data transformation)
         transf = 0, no data transformation
                = 1, 1st differences used to estimate the model
                = freq, seasonal differences used to estimate
                = cal-structure, growth rates used to estimate
                  e.g., cal(1982,1,12) [see cal() function]
-------------------------------------------------------------
```

```
 NOTE: constant term included automatically
---------------------------------------------------------------
 RETURNS:
  yfor = an nfor x neqs matrix of level forecasts for each equation
---------------------------------------------------------------
```

Note that you input the variables y in levels form, indicate any of four data transformations that will be used when estimating the VAR model, and the function **varf** will carry out this transformation, produce estimates and forecasts that are converted back to levels form. This greatly simplifies the task of producing and comparing forecasts based on alternative data transformations.

Of course, if you desire a transformation other than the four provided, such as logs, you can transform the variables y prior to calling the function and specify 'transf=0'. In this case, the function does not provide levels forecasts, but rather forecasts of the logged-levels will be returned. Setting 'transf=0', produces estimates based on the data input and returns forecasts based on this data.

As an example of comparing alternative VAR model forecasts based on two of the four alternative transformations, consider the program in example 5.12.

```
% ----- Example 5.12 Forecasting VAR models
y = load('test.dat'); % a test data set containing
                      % monthly mining employment for
                      % il,in,ky,mi,oh,pa,tn,wv
dates = cal(1982,1,12); % data covers 1982,1 to 1996,5
nfor = 12; % number of forecast periods
nlag = 6;  % number of lags in var-model
begf = ical(1995,1,dates);  % beginning forecast period
endf = ical(1995,12,dates); % ending forecast period
% no data transformation example
fcast1 = varf(y,nlag,nfor,begf);
% seasonal differences data transformation example
freq = 12; % set frequency of the data to monthly
fcast2 = varf(y,nlag,nfor,begf,[],freq);
% compute percentage forecast errors
actual = y(begf:endf,:);
error1 = (actual-fcast1)./actual;
error2 = (actual-fcast2)./actual;
vnames =  strvcat('il','in','ky','mi','oh','pa','tn','wv');
fdates = cal(1995,1,12);
fprintf(1,'VAR model in levels percentage errors \n');
tsprint(error1*100,fdates,vnames,'%7.2f');
fprintf(1,'VAR - seasonally differenced data percentage errors \n');
tsprint(error2*100,fdates,vnames,'%7.2f');
```

The results from the program are:

```
VAR model in levels percentage errors
 Date       il      in      ky      mi      oh      pa      tn      wv
 Jan95   -3.95   -2.86   -1.15   -6.37   -5.33   -7.83   -0.19   -0.65
 Feb95   -5.63   -2.63   -3.57   -7.77   -7.56   -8.28   -0.99    0.38
 Mar95   -3.62   -1.75   -4.66   -5.49   -5.67   -6.69    2.26    2.30
 Apr95   -3.81   -4.23   -7.11   -4.27   -5.18   -5.41    2.14    0.17
 May95   -4.05   -5.60   -8.14   -0.92   -5.88   -3.93    2.77   -1.11
 Jun95   -4.10   -3.64   -8.87    0.10   -4.65   -4.15    2.90   -2.44
 Jul95   -4.76   -3.76  -10.06    1.99   -1.23   -5.06    3.44   -3.67
 Aug95   -8.69   -3.89   -9.86    4.85   -2.49   -5.41    3.63   -3.59
 Sep95   -8.73   -3.63  -12.24    0.70   -4.33   -6.28    3.38   -4.04
 Oct95  -11.11   -3.23  -12.10   -7.38   -4.74   -8.34    3.21   -5.57
 Nov95  -11.79   -4.30  -11.53   -8.93   -4.90   -7.27    3.60   -5.69
 Dec95  -12.10   -5.56  -11.12  -13.11   -5.57   -8.78    2.13   -9.38

VAR - seasonally differenced data percentage errors
 Date       il      in      ky      mi      oh      pa      tn      wv
 Jan95   -6.53   -0.52   -3.75    3.41   -1.49   -0.06    3.86    0.05
 Feb95   -4.35    1.75   -6.29    0.35   -3.53   -2.76    4.46    2.56
 Mar95   -1.12    2.61   -6.83    1.53   -2.72    2.24    2.96    3.97
 Apr95   -0.38   -2.36   -7.03   -4.30   -1.28    0.70    5.55    2.73
 May95    0.98   -5.05   -3.90   -4.65   -1.18    2.02    6.49   -0.43
 Jun95   -0.73   -2.55   -2.04   -0.30    2.30    0.81    3.96   -1.44
 Jul95   -1.41   -0.36   -1.69    0.79    4.83   -0.06    7.68   -4.24
 Aug95   -3.36    2.36   -1.78    7.99    4.86   -1.07    8.75   -3.38
 Sep95   -3.19    3.47   -3.26    6.91    2.31   -1.44    8.30   -3.02
 Oct95   -2.74    3.27   -2.88   -2.14    2.92   -0.73    9.00    0.08
 Nov95   -2.47    1.54   -2.63   -5.23    4.33    0.36    9.02    0.64
 Dec95   -1.35    0.48   -3.53   -7.89    4.38    1.33    7.03   -3.92
```

It is also possible to build models that produce forecasts that "feed-in" to another model as deterministic variables. For example, suppose we wished to use national employment in the primary metal industry (SIC 33) as a deterministic variable in our model for primary metal employment in the eight states. The following program shows how to accomplish this.

```
% ----- Example 5.13 Forecasting multiple related models
dates = cal(1982,1,12); % data starts in 1982,1
y=load('sic33.states'); % industry sic33 employment for 8 states
[nobs neqs] = size(y);
load sic33.national;    % industry sic33 national employment
ndates = cal(1947,1,12);% national data starts in 1947,1

begs = ical(1982,1,ndates); % find 1982,1 for national data
ends = ical(1996,5,ndates); % find 1996,5 for national data
```

```
x = sic33(begs:ends,1);  % pull out national employment in sic33
                         % for the time-period corresponding to
                         % our 8-state sample
begf = ical(1990,1,dates);  % begin forecasting date
endf = ical(1994,12,dates); % end forecasting date
nfor = 12;                  % forecast 12-months-ahead
nlag = 6;
xerror = zeros(nfor,1);
yerror = zeros(nfor,neqs);
cnt = 0; % counter for the # of forecasts we produce
for i=begf:endf % loop over dates producing forecasts
xactual = x(i:i+nfor-1,1);  % actual national employment
yactual = y(i:i+nfor-1,:);  % actual state employment
% first forecast national employment in sic33
xfor = varf(x,nlag,nfor,i); % an ar(6) model
xdet = [x(1:i-1,1)          % actual national data up to forecast period
        xfor      ];        % forecasted national data
% do state forecast using national data and forecast as input
yfor = varf(y,nlag,nfor,i,xdet);
% compute forecast percentage errors
xerror = xerror + abs((xactual-xfor)./xactual);
yerror = yerror + abs((yactual-yfor)./yactual);
cnt = cnt+1;
end; % end loop over forecasting experiment dates
% compute mean absolute percentage errors
xmape = xerror*100/cnt; ymape = yerror*100/cnt;
% printout results
in.cnames =  strvcat('national','il','in','ky','mi','oh','pa','tn','wv');
rnames = 'Horizon';
for i=1:12; rnames = strvcat(rnames,[num2str(i),'-step']); end;
in.rnames = rnames;
in.fmt = '%6.2f';
fprintf(1,'national and state MAPE percentage forecast errors \n');
fprintf(1,'based on %d 12-step-ahead forecasts \n',cnt);
mprint([xmape ymape],in);
```

Our model for national employment in SIC33 is simply an autoregressive model with 6 lags, but the same approach would work for a matrix X of deterministic variables used in place of the vector in the example. We can also provide for a number of deterministic variables coming from a variety of models that are input into other models, not unlike traditional structural econometric models. The program produced the following output.

```
national and state MAPE percentage forecast errors
based on 60 12-step-ahead forecasts
Horizon national     il     in     ky     mi     oh     pa     tn     wv
1-step      0.27   0.70   0.78   1.00   1.73   0.78   0.56   0.88   1.08
2-step      0.46   1.02   1.10   1.15   1.95   1.01   0.78   1.06   1.58
```

```
3-step      0.68   1.22   1.26   1.39   2.34   1.17   1.00   1.16   1.91
4-step      0.93   1.53   1.45   1.46   2.81   1.39   1.25   1.35   2.02
5-step      1.24   1.84   1.63   1.74   3.27   1.55   1.57   1.53   2.10
6-step      1.55   2.22   1.70   2.05   3.41   1.53   1.81   1.64   2.15
7-step      1.84   2.62   1.59   2.24   3.93   1.68   1.99   1.76   2.49
8-step      2.21   3.00   1.56   2.34   4.45   1.82   2.10   1.89   2.87
9-step      2.55   3.30   1.59   2.58   4.69   1.93   2.33   1.99   3.15
10-step     2.89   3.64   1.74   2.65   5.15   2.08   2.51   2.12   3.39
11-step     3.25   3.98   1.86   2.75   5.75   2.29   2.70   2.27   3.70
12-step     3.60   4.36   1.94   2.86   6.01   2.40   2.94   2.23   3.96
```

Consider that it would be quite easy to assess the contribution of using national employment as a deterministic variable in the model by running another model that excludes this deterministic variable.

As a final example, consider an experiment where we wish to examine the impact of using different numbers of error correction variables on the forecast accuracy of the EC model. Shoesmith (1995) suggests that one should employ the number of error correction variables associated with the Johansen likelihood ratio statistics, but he provides only limited evidence regarding this contention.

The experiment uses time-series on national monthly employment from 12 manufacturing industries covering the period 1947,1 to 1996,12. Forecasts are carried out over the period from 1970,1 to 1995,12 using the number of error correction terms suggested by the Johansen likelihood ratio trace statistics, as well as models based on +/-1 and +/-2 error correction terms relative to the value suggested by the trace statistic.

We then compare the relative forecast accuracy of these models by examining the ratio of the MAPE forecast error from the models with +/-1 and +/-2 terms to the errors from the model based on r relationships suggested by the trace statistic.

Here is the program code:

```
% ----- Example 5.14 comparison of forecast accuracy as a function of
%                    the # of co-integrating vectors used
load level.mat;             % 20 industries national employment
y = level(:,1:12);          % use only 12 industries
[nobs neqs] = size(y);      dates = cal(1947,1,12);
begf = ical(1970,1,dates);  % beginning forecast date
endf = ical(1995,12,dates); % ending forecast date
nfor = 12;                  % forecast horizon
nlag = 10; cnt = 1;         % nlag based on lrratio() results
for i=begf:endf;
jres = johansen(y,0,nlag);  trstat = jres.lr1; tsignf = jres.cvt;
 r = 0;
```

```
 for j=1:neqs; % find r indicated by trace statistic
   if trstat(j,1) > tsignf(j,2), r = j; end;
 end;
% set up r-1,r-2 and r+1,r+2 forecasts in addition to forecasts based on r
if (r >= 3 & r <=10)
 frm2 = ecmf(y,nlag,nfor,i,r-2); frm1 = ecmf(y,nlag,nfor,i,r-1);
 fr   = ecmf(y,nlag,nfor,i,r);   frp1 = ecmf(y,nlag,nfor,i,r+1);
 frp2 = ecmf(y,nlag,nfor,i,r+2); act  = y(i:i+nfor-1,1:12);
 % compute forecast MAPE
 err(cnt).rm2 = abs((act-frm2)./act); err(cnt).rm1 = abs((act-frm1)./act);
 err(cnt).r   = abs((act-fr)./act);   err(cnt).rp1 = abs((act-frp1)./act);
 err(cnt).rp2 = abs((act-frp2)./act); cnt = cnt+1;
else
 fprintf(1,'time %d had %d co-integrating relations \n',i,r);
end; % end if-else;  end; % end of loop over time
rm2 = zeros(12,12); rm1 = rm2; rm0 = rm2; rp1 = rm2; rp2 = rm2;
for i=1:cnt-1;
rm2 = rm2 + err(i).rm2; rm1 = rm1 + err(i).rm1;
rm0 = rm0 + err(i).r;   rp1 = rp1 + err(i).rp1;
rp2 = rp2 + err(i).rp2;
end;
rm2 = rm2/(cnt-1);      rm1 = rm1/(cnt-1);
rm0 = rm0/(cnt-1);      rp1 = rp1/(cnt-1);
rp2 = rp2/(cnt-1);
rnames = 'Horizon'; cnames = [];
for i=1:12;
rnames = strvcat(rnames,[num2str(i),'-step']);
cnames = strvcat(cnames,['IND',num2str(i)]);
end;
in.rnames = rnames; in.cnames = cnames; in.fmt = '%6.2f';
fprintf(1,'forecast errors relative to error by ecm(r) model \n');
fprintf(1,'r-2 relative to r \n');
mprint(rm2./rm0,in);
fprintf(1,'r-1 relative to r \n');
mprint(rm2./rm0,in);
fprintf(1,'r+1 relative to r \n');
mprint(rp1./rm0,in);
fprintf(1,'r+2 relative to r \n');
mprint(rp2./rm0,in);
```

The program code stores the individual MAPE forecast errors in a structure variable using: **err(cnt).rm2 = abs((actual-frm2)./actual);**, which will have fields for the errors from all five models. These fields are matrices of dimension 12 x 12, containing MAPE errors for each of the 12-step-ahead forecasts for time **cnt** and for each of the 12 industries. We are not really interested in these individual results, but present this as an illustration. As part of the illustration, we show how to access the individual results to compute the average MAPE errors for each horizon and industry. If you wished to access industry number 2's forecast errors based on the model using r co-integrating relations, for the first experimental forecast period you would use: **err(1).rm(:,2)**. The results from our experiment are shown below. These results represent an average over a total of 312 twelve-step-ahead forecasts. Our simple MATLAB program produced a total of 224,640 forecasts, based on 312 twelve-step-ahead forecasts, for 12 industries, times 5 models!

Our experiment indicates that using more than the r co-integrating relationships determined by the Johansen likelihood trace statistic degrades the forecast accuracy. This is clear from the large number of forecast error ratios greater than unity for the two models based on $r + 1$ and $r + 2$ versus those from the model based on r. On the other hand, using a smaller number of co-integrating relationships than indicated by the Johansen trace statistic seems to improve forecast accuracy. In a large number of industries at many of the twelve forecast horizons, we see comparison ratios less than unity. Further, the forecast errors associated with $r-2$ are superior to those from $r - 1$, producing smaller comparison ratios in 9 of the 12 industries.

5.5 Chapter summary

We found that a library of functions can be constructed to produce estimates and forecasts for a host of alternative vector autoregressive and error correction models. An advantage of MATLAB over a specialized program like RATS is that we have more control and flexibility to implement specialized priors. The prior for the **rvar** model cannot be implemented in RATS software as the vector autoregressive function in that program does not allow you to specify prior means for variables other than the first own-lagged variables in the model.

Another advantage is the ability to write auxiliary functions that process the structures returned by our estimation functions and present output in a format that we find helpful. As an example of this, the function **pgranger** produced a formatted table of Granger-causality probabilities making it easy to draw inferences.

forecast errors relative to error by ecm(r) model

r-2 relative to r

Horizon	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10	I11	I12
1-step	1.01	0.99	1.00	1.01	1.00	1.00	1.01	0.99	1.00	0.98	0.97	0.99
2-step	0.92	1.01	0.99	0.96	1.03	1.00	1.02	0.99	1.01	1.03	0.99	0.94
3-step	0.89	1.04	1.00	0.94	1.03	1.02	1.01	0.98	0.99	1.03	1.00	0.93
4-step	0.85	1.03	0.99	0.94	1.05	1.03	1.02	1.00	0.97	1.01	1.00	0.91
5-step	0.82	1.03	0.98	0.94	1.03	1.03	1.04	1.00	0.97	0.98	1.02	0.92
6-step	0.81	1.05	0.97	0.94	1.01	1.04	1.04	0.99	0.97	0.96	1.03	0.92
7-step	0.79	1.07	0.96	0.93	0.99	1.03	1.05	0.98	0.97	0.94	1.03	0.92
8-step	0.78	1.04	0.95	0.93	0.98	1.02	1.04	0.96	0.96	0.93	1.03	0.93
9-step	0.76	1.03	0.93	0.92	0.97	1.01	1.02	0.95	0.95	0.91	1.01	0.94
10-step	0.76	1.01	0.92	0.91	0.96	0.99	1.01	0.94	0.94	0.90	0.99	0.94
11-step	0.75	1.00	0.91	0.91	0.95	0.98	1.01	0.95	0.94	0.90	0.99	0.95
12-step	0.74	0.99	0.90	0.91	0.94	0.98	0.99	0.94	0.93	0.89	0.98	0.95

r-1 relative to r

Horizon	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10	I11	I12
1-step	1.01	0.99	1.00	1.01	1.00	1.00	1.01	0.99	1.00	0.98	0.97	0.99
2-step	0.92	1.01	0.99	0.96	1.03	1.00	1.02	0.99	1.01	1.03	0.99	0.94
3-step	0.89	1.04	1.00	0.94	1.03	1.02	1.01	0.98	0.99	1.03	1.00	0.93
4-step	0.85	1.03	0.99	0.94	1.05	1.03	1.02	1.00	0.97	1.01	1.00	0.91
5-step	0.82	1.03	0.98	0.94	1.03	1.03	1.04	1.00	0.97	0.98	1.02	0.92
6-step	0.81	1.05	0.97	0.94	1.01	1.04	1.04	0.99	0.97	0.96	1.03	0.92
7-step	0.79	1.07	0.96	0.93	0.99	1.03	1.05	0.98	0.97	0.94	1.03	0.92
8-step	0.78	1.04	0.95	0.93	0.98	1.02	1.04	0.96	0.96	0.93	1.03	0.93
9-step	0.76	1.03	0.93	0.92	0.97	1.01	1.02	0.95	0.95	0.91	1.01	0.94
10-step	0.76	1.01	0.92	0.91	0.96	0.99	1.01	0.94	0.94	0.90	0.99	0.94
11-step	0.75	1.00	0.91	0.91	0.95	0.98	1.01	0.95	0.94	0.90	0.99	0.95
12-step	0.74	0.99	0.90	0.91	0.94	0.98	0.99	0.94	0.93	0.89	0.98	0.95

r+1 relative to r

Horizon	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10	I11	I12
1-step	1.01	1.00	1.02	1.00	1.00	1.01	1.01	0.99	1.00	1.01	1.02	1.01
2-step	0.99	1.02	1.01	0.99	0.99	1.03	1.00	0.99	0.99	1.05	1.03	1.04
3-step	0.99	1.01	1.01	0.99	1.00	1.04	1.00	0.99	0.98	1.07	1.03	1.04
4-step	0.99	0.99	1.01	0.98	1.01	1.05	1.01	1.01	0.97	1.08	1.04	1.03
5-step	0.98	0.98	1.03	0.99	1.01	1.05	1.01	1.03	0.97	1.08	1.04	1.04
6-step	0.98	0.98	1.03	0.99	1.01	1.06	1.00	1.03	0.97	1.07	1.04	1.04
7-step	0.98	0.98	1.04	1.00	1.01	1.06	1.00	1.04	0.97	1.08	1.04	1.04
8-step	0.98	0.96	1.05	1.00	1.02	1.06	0.99	1.05	0.97	1.06	1.04	1.04
9-step	0.97	0.95	1.05	1.01	1.02	1.07	0.99	1.05	0.96	1.05	1.04	1.04
10-step	0.97	0.96	1.05	1.01	1.02	1.07	0.98	1.05	0.96	1.04	1.04	1.03
11-step	0.97	0.97	1.05	1.01	1.02	1.07	0.98	1.06	0.95	1.05	1.04	1.03
12-step	0.97	0.97	1.05	1.01	1.02	1.07	0.98	1.07	0.95	1.05	1.04	1.03

r+2 relative to r

Horizon	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10	I11	I12
1-step	1.00	1.01	1.02	1.01	0.99	1.01	1.01	0.99	0.99	1.05	1.03	1.01
2-step	1.00	1.05	1.00	0.97	1.00	1.03	1.01	1.00	0.99	1.11	1.03	1.06
3-step	1.00	1.02	1.01	0.96	1.01	1.06	1.02	1.02	0.98	1.13	1.04	1.06
4-step	1.00	0.99	1.01	0.97	1.02	1.07	1.02	1.04	0.97	1.14	1.05	1.05
5-step	1.01	0.97	1.03	0.98	1.04	1.08	1.02	1.07	0.97	1.15	1.05	1.04
6-step	1.01	0.95	1.04	0.99	1.04	1.10	1.02	1.08	0.97	1.15	1.06	1.04
7-step	1.01	0.96	1.06	1.00	1.05	1.10	1.01	1.09	0.96	1.15	1.06	1.03
8-step	1.00	0.93	1.08	0.99	1.05	1.10	1.00	1.10	0.95	1.15	1.07	1.02
9-step	1.01	0.92	1.09	0.99	1.06	1.11	0.99	1.11	0.95	1.14	1.08	1.02
10-step	1.01	0.92	1.09	0.99	1.05	1.11	0.98	1.11	0.94	1.13	1.08	1.01
11-step	1.01	0.93	1.09	0.99	1.05	1.12	0.98	1.13	0.94	1.14	1.09	1.00
12-step	1.00	0.93	1.09	0.99	1.05	1.12	0.97	1.13	0.94	1.15	1.09	0.99

Finally, many of the problems encountered in carrying out forecast experiments involve transformation of the data for estimation purposes and reverse transformations needed to compute forecast errors on the basis of the levels of the time-series. Our functions can perform these transformations for the user, making the code necessary to carry out forecast experiments quite simple. In fact, one could write auxiliary functions that compute alternative forecast accuracy measures given matrices of forecast and actual values.

We also demonstrated how the use of structure array variables can facilitate storage of individual forecasts or forecast errors for a large number of time periods, horizons and variables. This would allow a detailed examination of the accuracy and characteristics associated with individual forecast errors for particular variables and time periods. As noted above, auxiliary functions could be constructed to carry out this type of analysis.

Chapter 5 Appendix

The *vector autoregressive library functions* are in a subdirectory **var_bvar**.

```
 vector autoregressive function library

------- VAR/BVAR program functions -----------

 becm_g     - Gibbs sampling BECM estimates
 becmf      - Bayesian ECM model forecasts
 becmf_g    - Gibbs sampling BECM forecasts
 bvar       - BVAR model
 bvar_g     - Gibbs sampling BVAR estimates
 bvarf      - BVAR model forecasts
 bvarf_g    - Gibbs sampling BVAR forecasts
 ecm        - ECM (error correction) model estimates
 ecmf       - ECM model forecasts
 irf        - impulse response functions
 lrratio    - likelihood ratio lag length tests
 pftest     - prints Granger F-tests
 pgranger   - prints Granger causality probabilities
 recm       - ecm version of rvar
 recm_g     - Gibbs sampling random-walk averaging estimates
 recmf      - random-walk averaging ECM forecasts
 recmf_g    - Gibbs sampling random-walk averaging forecasts
 rvar       - Bayesian random-walk averaging prior model
 rvar_g     - Gibbs sampling RVAR estimates
 rvarf      - Bayesian RVAR model forecasts
 rvarf_g    - Gibbs sampling RVAR forecasts
 var        - VAR model
 varf       - VAR model forecasts

------ demonstration programs  -----------

 becm_d     - BECM model demonstration
 becm_gd    - Gibbs sampling BECM estimates demo
 becmf_d    - becmf demonstration
 becmf_gd   - Gibbs sampling BECM forecast demo
 bvar_d     - BVAR model demonstration
```

```
 bvar_gd    - Gibbs sampling BVAR demonstration
 bvarf_d    - bvarf demonstration
 bvarf_gd   - Gibbs sampling BVAR forecasts demo
 ecm_d      - ECM model demonstration
 ecmf_d     - ecmf demonstration
 irf_d      - impulse response function demo
 irf_d2     - another irf demo
 lrratio_d  - demonstrates lrratio
 pftest_d   - demo of pftest function
 recm_d     - RECM model demonstration
 recm_gd    - Gibbs sampling RECM model demo
 recmf_d    - recmf demonstration
 recmf_gd   - Gibbs sampling RECM forecast demo
 rvar_d     - RVAR model demonstration
 rvar_g     - Gibbs sampling rvar model demo
 rvarf_d    - rvarf demonstration
 rvarf_gd   - Gibbs sampling rvar forecast demo
 var_d      - VAR model demonstration
 varf_d     - varf demonstration

------- support functions  -----------

 johansen   - used by ecm,ecmf,becm,becmf,recm,recmf
 lag        - does ordinary lags
 mlag       - does var-type lags
 nclag      - does contiguous lags (used by rvar,rvarf,recm,recmf)
 ols        - used for VAR estimation
 prt        - prints results from all functions
 prt_coint  - used by prt_var for ecm,becm,recm
 prt_var    - prints results of all var/bvar models
 prt_varg   - prints results of all Gibbs var/bvar models
 rvarb      - used for RVARF forecasts
 scstd      - does univariate AR for BVAR
 theil_g    - used for Gibbs sampling estimates and forecasts
 theilbf    - used for BVAR forecasts
 theilbv    - used for BVAR estimation
 trimr      - used by VARF,BVARF, johansen
 vare       - used by lrratio
```

The *co-integration library functions* are in a subdirectory **coint**.

```
 co-integration library

 ------ co-integration testing routines --------

 adf        - carries out Augmented Dickey-Fuller unit root tests
 cadf       - carries out ADF tests for co-integration
 johansen   - carries out Johansen's co-integration tests
 phillips   - Phillips-Peron unit root test
```

```
------ demonstration programs ----------

adf_d      - demonstrates adf
cadf_d     - demonstrates cadf
johansen_d - demonstrates johansen
phillips_d - demonstrates phillips

------ support functions -------------

c_sja      - returns critical values for SJ maximal eigenvalue test
c_sjt      - returns critical values for SJ trace test
cols       - (like Gauss cols)
detrend    - used by johansen to detrend data series
prt_coint  - prints results from adf,cadf,johansen
ptrend     - used by adf to create time polynomials
rows       - (like Gauss rows)
rztcrit    - returns critical values for cadf test
tdiff      - time-series differences
trimr      - (like Gauss trimr)
ztcrit     - returns critical values for adf test
```

Chapter 6

Markov Chain Monte Carlo Models

A *gibbs sampling library* of utility functions along with various estimation functions are described in this chapter. Additional examples that demonstrate gibbs library functions for limited dependent variable logit, probit, and tobit models are presented in the next chapter that discusses limited dependent variable estimation.

A recent methodology known as Markov Chain Monte Carlo is based on the idea that rather than compute a probability density, say $p(\theta|y)$, we would be just as happy to have a large random sample from $p(\theta|y)$ as to know the precise form of the density. Intuitively, if the sample were large enough, we could approximate the form of the probability density using kernel density estimators or histograms. In addition, we could compute accurate measures of central tendency and dispersion for the density, using the mean and standard deviation of the large sample. This insight leads to the question of how to efficiently simulate a large number of random samples from $p(\theta|y)$.

Metropolis, et al. (1953) showed that one could construct a Markov chain stochastic process for $(\theta_t, t \geq 0)$ that unfolds over time such that: 1) it has the same state space (set of possible values) as θ, 2) it is easy to simulate, and 3) the equilibrium or stationary distribution which we use to draw samples is $p(\theta|y)$ after the Markov chain has been run for a long enough time. Given this result, we can construct and run a Markov chain for a very large number of iterations to produce a sample of $(\theta_t, t = 1, \ldots)$ from the posterior distribution and use simple descriptive statistics to examine any features of the posterior in which we are interested.

This approach, known as Markov Chain Monte Carlo, (MCMC) to determining posterior densities has greatly reduced the computational problems that previously plagued application of the Bayesian methodology. Gelfand and Smith (1990), as well as a host of others, have popularized this methodology by demonstrating its use in a wide variety of statistical applications where intractable posterior distributions previously hindered Bayesian analysis. A simple introduction to the method can be found in Casella and George (1990) and an expository article dealing specifically with the normal linear model is Gelfand, Hills, Racine-Poon and Smith (1990). Two recent books that deal in detail with all facets of these methods are: Gelman, Carlin, Stern and Rubin (1995) and Gilks, Richardson and Spiegelhalter (1996).

The most widely used approach to MCMC is due to Hastings (1970) which generalizes a method of Metropolis et al. (1953). A second approach (that we focus on) is known as Gibbs sampling due to Geman and Geman (1984). Hastings approach suggests that given an initial value θ_0 we can construct a chain by recognizing that any Markov chain that has found its way to a state θ_t can be completely characterized by the probability distribution for time $t+1$. His algorithm relies on a proposal or candidate distribution, $f(\theta|\theta_t)$ for time $t+1$, given that we have θ_t. A candidate point $\theta^\star$ is sampled from the proposal distribution and:

1. This point is accepted as $\theta_{t+1} = \theta^\star$ with probability:

$$\alpha_H(\theta_t, \theta^\star) = \min\left[1, \frac{p(\theta^\star|y)f(\theta_t|\theta^\star)}{p(\theta_t|y)f(\theta^\star|\theta_t)}\right] \tag{6.1}$$

2. otherwise, $\theta_{t+1} = \theta_t$, that is we stay with the current value of θ.

In other words, we can view the Hastings algorithm as indicating that we should toss a Bernoulli coin with probability α_H of heads and make a move to $\theta_{t+1} = \theta^\star$ if we see a heads, otherwise set $\theta_{t+1} = \theta_t$. Hastings demonstrates that this approach to sampling represents a Markov chain with the correct equilibrium distribution capable of producing samples from the posterior $p(\theta|y)$ we are interested in.

Gibbs sampling dates to the work of Geman and Geman (1984) in image analysis and is related to the EM algorithm (Dempster, Laird and Rubin, 1977) which dealt with maximum likelihood and Bayesian estimation with missing information. Assume a parameter vector $\theta = (\theta_1, \theta_2)$, a prior $p(\theta)$, and likelihood $l(\theta|y)$, that produces a posterior distribution $p(\theta|y) = cp(\theta)l(\theta|y)$, with c a normalizing constant. It is often the case that the posterior distribution over all parameters is difficult to work with. On the other had, if we partition our parameters into two sets θ_1, θ_2 and had initial estimates for θ_1 (treated like missing information in the EM algorithm), we could estimate θ_2 conditional on θ_1 using $p(\theta_2|y, \hat{\theta}_1)$. (Presumably, this estimate is much easier to derive. We will provide details illustrating this case in section 6.2.) Denote the estimate, $\hat{\theta}_2$ derived by using the posterior mean or mode of $p(\theta_2|y, \hat{\theta}_1)$, and consider than we are now able to construct a new estimate of θ_1 based on the conditional distribution $p(\theta_1|y, \hat{\theta}_2)$, which can be used to construct another value for θ_2, and so on.

In general, for the case of k parameters, the algorithm can be summarized as:

Initialize θ_0

Repeat {

Sample $\theta_1^{t+1} \sim p[\theta_1|y, (\theta_2^t, \ldots, \theta_k^t)]$
Sample $\theta_2^{t+1} \sim p[\theta_2|y, (\theta_1^{t+1}, \theta_3^t, \ldots, \theta_k^t)]$
$\vdots$
Sample $\theta_k^{t+1} \sim p[\theta_k|y, (\theta_1^{t+1}, \theta_2^{t+1}, \ldots, \theta_{k-1}^{t+1})]$
$t = t + 1$

}

Geman and Geman (1984) demonstrated that the stochastic process θ^t from this approach to sampling the complete sequence of conditional distributions represents a Markov chain with the correct equilibrium distribution. Gibbs sampling is in fact closely related to Hastings and Metropolis MCMC methods. This chapter only deals with Gibbs sampling, providing a simple introduction that draws heavily on work by Geweke (1993), who introduced the approach presented here in Section 6.3 for dealing with heteroscedastic regression models.

Section 6.1 introduces a simple Bayesian regression example, and the following section takes the reader through an application and computational code necessary to carry out Gibbs sampling estimation. MATLAB functions for diagnosing convergence of the sampling draws are the subject of Section 6.2. The Bayesian regression model from Section 6.1 is extended in Section 6.3 to the case of $t-$distributed errors that can accommodate outliers and non-constant variance. A Markov Chain Monte Carlo model for estimating autoregressive models with stability restrictions imposed on the parameters is used in Section 6.4 to illustrate construction of MATLAB functions that carry out Gibbs sampling estimation of econometric models.

In Section 6.5 we illustrate a 'Metropolis step within Gibbs sampling' algorithm that is useful in many Gibbs sampling models. Metropolis sampling represents a special case of the Hastings method where the proposal distribution is restricted to be symmetric. The Metropolis (or Hastings) within Gibbs sampling approach is frequently used when one encounters a conditional distribution that is not readily identified. In this case, we cannot draw samples using a conventional distribution and must rely on the Metropolis (or Hastings) algorithm to carry out these draws within the sequence of Gibbs sampling.

A concluding section briefly describes alternative econometric estimation methods that have been implemented in the *gibbs sampling library*. Gibbs sampling for probit and tobit model estimation is deferred until Chapter 7 where limited dependent variable models are discussed.

6.1 The Bayesian Regression Model

As an introduction to Gibbs sampling for Bayesian regression models, we consider the case of a linear regression model with an informative prior. The multiple normal linear regression model can be written as in (6.2).

$$\begin{aligned} y &= X\beta + \varepsilon \\ \varepsilon &\sim N(0, \sigma^2 I_n) \end{aligned} \tag{6.2}$$

Where y is an nx1 vector of dependent variables and X represents the nxk matrix of explanatory variables. We assume that ε is an nx1 vector of independent identically distributed normal random variates.

The parameters to be estimated in (6.2) are (β, σ), for which we assign a prior density of the form $\pi(\beta, \sigma) = \pi_1(\beta)\pi_2(\sigma)$. That is, we assume our prior for the parameters β is independent from that for the parameter σ. The normal prior density we assign to β requires that we specify a mean and variance for these parameters which can be expressed in terms of linear combinations of β as shown in (6.3).

$$R\beta \sim N(r, T) \Leftrightarrow \pi_1(\beta) \propto \exp\{-(1/2)(R\beta - r)'T^{-1}(R\beta - r)\} \tag{6.3}$$

Where the number of linear combinations of β can be equal to $m \leq k$, so that R is an mxk matrix that establishes the m linear relations, r is an mx1 vector containing the prior means and T is an mxm matrix containing the prior variances and covariances.

As is well-known, when $m < k$, the prior in (6.3) is improper, but can be justified as the limiting case of a set of proper priors. For our purposes it is convenient to express (6.3) in an alternative (equivalent) form based on a factorization of T^{-1} into $Q'Q = T^{-1}$, and $q = Qr$ leading to (6.4).

$$Q\beta \sim N(q, I_m) \Leftrightarrow \pi_1(\beta) \propto \exp\{-(1/2)(Q\beta - q)'(Q\beta - q)\} \tag{6.4}$$

For simplicity, we assume the diffuse prior for σ, $\pi_2(\sigma) \propto (1/\sigma)$, but all of our results would follow for the case of an informative conjugate gamma prior for this parameter. Following the usual Bayesian methodology, we combine the likelihood function for our simple model:

$$L(\beta, \sigma) \propto (1/\sigma^n)\exp[(y - X\beta)'(y - X\beta)/2\sigma^2] \tag{6.5}$$

with the priors $\pi_1(\beta)$ and $\pi_2(\sigma)$ to produce the posterior density for (β, σ) shown in (6.6).

$$\begin{aligned} p(\beta, \sigma) &\propto (1/\sigma^{n+1})\exp[\beta - \hat{\beta}(\sigma)]'[V(\sigma)]^{-1}[\beta - \hat{\beta}(\sigma)] \\ \hat{\beta}(\sigma) &= (X'X + \sigma^2 Q'Q)^{-1}(X'y + \sigma^2 Q'q) \\ V(\sigma) &= \sigma^2(X'X + \sigma^2 Q'Q)^{-1} \end{aligned} \tag{6.6}$$

In (6.6), we have used the notation $\hat{\beta}(\sigma)$ to convey that the mean of the posterior, $\hat{\beta}$, is conditional on the parameter σ, as is the variance, denoted by $V(\sigma)$. This single parameter prevents analytical solution of the Bayesian regression problem. In order to overcome this problem, Theil and Goldberger (1961) observed that conditional on σ, the posterior density for β is multivariate normal. They proposed that σ^2 be replaced by an estimated value, $\hat{\sigma}^2 = (y - X\hat{\beta})'(y - X\hat{\beta})/(n - k)$, based on least-squares estimates $\hat{\beta}$. The advantage of this solution is that the estimation problem can be solved using existing least-squares regression software. Their solution produces a point estimate which we label $\hat{\beta}_{TG}$ and an associated variance-covariance estimate, both of which are shown in (6.7). This estimation procedure is implemented by the function **theil** discussed in Chapter 4.

$$\begin{aligned} \hat{\beta}_{TG} &= (X'X + \hat{\sigma}^2 Q'Q)^{-1}(X'y + \hat{\sigma}^2 Q'q) \\ \text{var}(\hat{\beta}_{TG}) &= \hat{\sigma}^2(X'X + \hat{\sigma}^2 Q'Q)^{-1} \end{aligned} \tag{6.7}$$

6.2 The Gibbs Sampler

The Gibbs sampler provides a way to sample from a multivariate probability density based only on the densities of subsets of vectors conditional on all others. The attractiveness of this approach is that it provides a solution to the Bayesian multiple integration problem when the conditional densities are simple and easy to obtain.

The Gibbs sampling approach set forth here specifies the complete conditional distributions for all parameters in the model and proceeds to carry out random draws from these distributions to collect a large sample of parameter draws. Gelfand and Smith (1990) demonstrate that Gibbs sampling from the sequence of complete conditional distributions for all parameters in the model, we achieve a set of draws that converge in the limit to the true (joint) posterior distribution of the parameters. That is, despite the use of conditional distributions in our sampling scheme, a large sample of the draws can be used to produce valid posterior inferences about the mean and moments of the parameters (or any function of the parameters) one is interested in.

The method is most easily described by developing and implementing a two-step Gibbs sampler for the posterior distribution (6.6) of our Bayesian regression model based on the distribution of β conditional on σ and the distribution of σ conditional on β. For our regression problem, the posterior density for β conditional on σ, $p(\beta|\sigma)$, is multivariate normal with mean equal to (6.8) and variance as indicated in (6.9).

$$\hat{\beta}(\sigma) = (X'X + \sigma^2 Q'Q)^{-1}(X'y + \sigma^2 Q'q) \tag{6.8}$$
$$V(\sigma) = \sigma^2 (X'X + \sigma^2 Q'Q)^{-1} \tag{6.9}$$

The posterior density for σ conditional on β, $p(\sigma|\beta)$ is:

$$p(\sigma|\beta) \propto (1/\sigma^{(n+1)})\mathrm{exp}[-(y - X\beta)'(y - X\beta)] \tag{6.10}$$

Which can be manipulated to obtain:

$$[(y - X\beta)'(y - X\beta)/\sigma^2] \mid \beta \sim \chi^2(n) \tag{6.11}$$

The Gibbs sampler suggested by these two conditional posterior distributions involves the following computations.

1. Begin with arbitrary values for the parameters β^0 and σ^0, which we designate with the superscript 0.

2. Compute the mean and variance of β using (6.8) and (6.9) conditional on the initial value σ^0.

3. Use the computed mean and variance of β to draw a multivariate normal random vector, which we label β^1.

4. Use the value β^1 along with a random $\chi^2(n)$ draw to determine σ^1 using (6.11).

The above four steps are known as a 'single pass' through our (two-step) Gibbs sampler, where we have replaced the initial arbitrary values of β^0 and σ^0 with new values labeled β^1 and σ^1. We now return to step 1 using the new values β^1 and σ^1 in place of the initial values β^0 and σ^0, and make another 'pass' through the sampler. This produces a new set of values, β^2 and σ^2.

Gelfand and Smith (1990) outline fairly weak conditions under which continued passes through our Gibbs sampler will produce a distribution of (β^i, σ^i) values that converges to the joint posterior density in which we are interested, $p(\beta, \sigma)$. Given independent realizations of β^i, σ^i, the strong law of large numbers suggests we can approximate the expected value of the β, σ parameters using averages of these sampled values.

To illustrate the Gibbs sampler for our Bayesian regression model, we generate a regression model data set containing 100 observations and 3 explanatory variables; an intercept term, and two uncorrelated explanatory variables generated from a standard normal distribution. The true values of β_0 for the intercept term, and the two slope parameters β_1 and β_2, were set to unity. A standard normal error term (mean zero, variance equal to unity) was used in generating the data.

The prior means for the β parameters were set to unity and the prior variance used was also unity, indicating a fair amount of uncertainty. The following MATLAB program implements the Gibbs sampler for this model.

```
% ----- Example 6.1 A simple Gibbs sampler
n=100; k=3;                     % set nobs and nvars
x = randn(n,k); b = ones(k,1); % generate data set
y = x*b + randn(n,1);
r = [1.0 1.0 1.0]';             % prior means
R = eye(k); T = eye(k);         % prior variance
nu = 0; d0 = 0;                 % diffuse prior for sige
Q = chol(inv(T)); Qpr = Q*r;
b0 = (x'*x)\(x'*y);             % use ols as initial values
sige = (y-x*b0)'*(y-x*b0)/(n-k);
ndraw = 1100; nomit = 100;      % set the number of draws
```

```
bsave = zeros(ndraw,k);        % allocate storage for results
ssave = zeros(ndraw,1);
tic;
for i=1:ndraw;                 % Start the sampling
xpxi = inv(x'*x + sige*Q);  xpy = (x'*y + sige*Qpr);
% update b
b = xpxi*xpy;
a = chol(xpxi);
b = sqrt(sige)*a'*randn(k,1) + b;
bsave(i,:) = b';
% update sige
nu1 = n + nu;
e = y - x*b;
d1 = d0 + e'*e;
chi = chis_rnd(1,nu1);
sige = d1/chi;
ssave(i,1) = sige;
end;                           % End the sampling
toc;
bhat = mean(bsave(nomit+1:ndraw,:));  % calculate means and std deviations
bstd = std(bsave(nomit+1:ndraw,:)); tstat = bhat./bstd;
sighat = mean(ssave(nomit+1:ndraw,1));
tout = tdis_prb(tstat',n);     % compute t-stat significance levels
% set up for printing results
in.cnames = strvcat('Coefficient','t-statistic','t-probability');
in.rnames = strvcat('Variable','variable 1','variable 2','variable 3');
in.fmt = '%16.6f';
tmp = [bhat' tstat' tout];
fprintf(1,'Gibbs estimates \n'); % print results
mprint(tmp,in);
result = theil(y,x,r,R,T); % compare to Theil-Goldberger estimates
prt(result);
```

We rely on MATLAB functions **norm_rnd** and **chis_rnd** to provide the multivariate normal and chi-squared random draws which are part of the *distributions function library* discussed in Chapter 9. Note also, we omit the first 100 draws at start-up to allow the Gibbs sampler to achieve a steady state before we begin sampling for the parameter distributions.

The results are shown below, where we find that it took only 1.75 seconds to carry out the 1100 draws and produce a sample of 1000 draws on which we can base our posterior inferences regarding the parameters β and σ. For comparison purposes, we produced estimates using the **theil** function from the *regression function library*.

```
elapsed_time =
    1.7516
Gibbs estimates
```

```
Variable        Coefficient      t-statistic    t-probability
variable 1         0.725839         5.670182         0.000000
variable 2         0.896861         7.476760         0.000000
variable 3         1.183147         9.962655         0.000000

Theil-Goldberger Regression Estimates
R-squared      =    0.6598
Rbar-squared   =    0.6527
sigma^2        =    1.4518
Durbin-Watson  =      2.12336
Nobs, Nvars    =    100,     3
***************************************************************
Variable         Prior Mean    Std Deviation
variable 1         1.000000         1.000000
variable 2         1.000000         1.000000
variable 3         1.000000         1.000000
***************************************************************
      Posterior Estimates
Variable        Coefficient      t-statistic    t-probability
variable 1         0.724510         4.653860         0.000010
variable 2         0.898450         6.231785         0.000000
variable 3         1.188448         8.840263         0.000000
```

6.2.1 Monitoring convergence of the sampler

An important issue in using Gibbs sampling is convergence of the sampler to the posterior distribution. We know from theory that the sampler converges in the limit as $n \rightarrow \infty$, but in any applied problem one must determine how many draws to make with the sampler. Ad-hoc convergence tests have been used in applied work and found to work well in simple regression models of the type considered here. For example, Smith and Roberts (1992) proposed a test they label the 'felt-tip pen test', that compares smoothed histograms or distributions from earlier draws in the sequence of passes through the sampler to those from later draws. If the two distributions are similar, within the tolerance of the felt-tip pen, convergence is assumed to have taken place.

There is some evidence that linear regression models exhibit rapid convergence, confirmed in our example by the Raftery and Lewis (1992a) procedure which we demonstrate below. It should be noted that once convergence occurs, we need to continue making passes to build up a sample from the posterior distribution which we use to make inferences about the parameters. As one might expect, convergence is a function of how complicated the set of conditional distributions are. For example, Geweke (1992) found that Gibbs sampling the Tobit censured regression model produced poor results with 400 passes and much better results with 10,000 passes. We will illustrate Tobit censured regression in Chapter 7.

Best et al., 1995 provide a set of Splus functions that implement six different MCMC convergence diagnostics, some of which have been implemented in a MATLAB function **coda**. This function provides: autocorrelation estimates, Rafterty-Lewis (1995) MCMC diagnostics, Geweke (1992) NSE, (numerical standard errors) RNE (relative numerical efficiency) estimates, Geweke Chi-squared test on the means from the first 20% of the sample versus the last 50%. We describe the role of each of these diagnostic measures using an applied example.

First, we have implemented the Gibbs sampler for the Bayesian regression model as a MATLAB function **ols_g** that returns a structure variable containing the draws from the sampler along with other information. Details regarding this function are presented in Section 6.4.

Some options pertain to using the function to estimate heteroscedastic linear models, a subject covered in the next section. For our purposes we can use the function to produce Gibbs samples for the Bayesian homoscedastic linear model by using a large value of the hyperparameter r. Note that this function utilizes a structure variable named 'prior' to input information to the function. Here is an example that uses the function to produce Gibbs draws that should be similar to those illustrated in the previous section (because we use a large hyperparameter value of $r = 100$).

```
% ----- Example 6.2 Using the coda() function
n=100; k=3; % set number of observations and variables
randn('seed',10101);
x = randn(n,k); b = ones(k,1); % generate data set
randn('seed',20201); y = x*b + randn(n,1);
ndraw = 1100; nomit = 100;     % set the number of draws
r = [1.0 1.0 1.0]'; % prior b means
T = eye(k);         % prior b variance
rval = 100;         % homoscedastic prior for r-value
prior.beta = r;
prior.bcov = T;
prior.rval = rval;
result = ols_g(y,x,prior,ndraw,nomit);
vnames = strvcat('beta1','beta2','beta3');
coda(result.bdraw,vnames);
```

The sample Gibbs draws for the parameters β are in the results structure variable, result.bdraw which we send down to the **coda** function to produce convergence diagnostics. This function uses a MATLAB variable 'nargout' to determine if the user has called the function with an output argument. If so, the function returns a result structure variable that can be printed later using the **prt** (or **prt_gibbs**) functions. In the case where the user supplies no output argument, (as in the example code above) the convergence diagnostics will be printed to the MATLAB command window.

Note that if we wished to analyze convergence for the estimates of the σ parameters in the model, we could call the function with these as arguments in addition to the draws for the β parameters, using:

```
coda([result.bdraw result.sdraw]);
```

A partial listing of the documentation for the function **coda** is shown below, where the form of the structure variable returned by **coda** when called with an output argument has been eliminated to save space.

```
PURPOSE: MCMC convergence diagnostics, modeled after Splus coda
------------------------------------------------------
USAGE:              coda(draws,vnames,info)
       or: result = coda(draws)
where: draws = a matrix of MCMC draws (ndraws x nvars)
      vnames = (optional) string vector of variable names (nvar x 1)
        info = (optional) structure setting input values
      info.q = Raftery quantile           (default = 0.025)
      info.r = Raftery level of precision (default = 0.005)
      info.s = Rafterty probability for r  (default = 0.950)
      info.p1 = 1st % of sample for Geweke chi-sqr test (default = 0.2)
      info.p2 = 2nd % of sample for Geweke chi-sqr test (default = 0.5)
------------------------------------------------------
NOTES: you may supply only some of the info-structure arguments
       the remaining ones will take on default values
------------------------------------------------------
RETURNS: output to command window if nargout = 0
         autocorrelation estimates
         Rafterty-Lewis MCMC diagnostics
         Geweke NSE, RNE estimates
         Geweke chi-sqr prob on means from info.p1 vs info.p2
         a results structure if nargout = 1
```

The results from executing the example program as are follows:

```
MCMC CONVERGENCE diagnostics
Based on sample size =       1000
Autocorrelations within each parameter chain
Variable        Lag 1        Lag 5        Lag 10        Lag 50
beta1           0.031        0.009        -0.071        -0.058
beta2           0.024       -0.041        -0.004         0.072
beta3          -0.023        0.019         0.029         0.006
```

```
Raftery-Lewis Diagnostics for each parameter chain
(q=0.0250, r=0.005000, s=0.950000)
Variable       Thin       Burn   Total(N)     (Nmin)     I-stat
beta1             1          2       3869       3746      1.033
beta2             1          2       3869       3746      1.033
beta3             1          2       3869       3746      1.033

Geweke Diagnostics for each parameter chain
Variable      Mean    std dev    NSE iid    RNE iid
beta1     1.041488   0.111201   0.003516   1.000000
beta2     0.980489   0.104348   0.003300   1.000000
beta3     0.938445   0.113434   0.003587   1.000000

Variable    NSE 4%     RNE 4%     NSE 8%     RNE 8%    NSE 15%    RNE 15%
beta1     0.003529   0.992811   0.003092   1.293611   0.002862   1.509356
beta2     0.003653   0.816099   0.003851   0.734157   0.003594   0.842994
beta3     0.003135   1.309284   0.002790   1.652990   0.003150   1.296704

Geweke Chi-squared test for each parameter chain
First 20% versus Last 50% of the sample
Variable            beta1
NSE estimate         Mean       N.S.E.  Chi-sq Prob
i.i.d.           1.040547     0.004257     0.842837
4% taper         1.040811     0.004374     0.829737
8% taper         1.040860     0.003910     0.808189
15% taper        1.040747     0.003354     0.782689

Variable            beta2
NSE estimate         Mean       N.S.E.  Chi-sq Prob
i.i.d.           0.980194     0.004015     0.078412
4% taper         0.980483     0.004075     0.090039
8% taper         0.980873     0.003915     0.088567
15% taper        0.982157     0.003365     0.095039

Variable            beta3
NSE estimate         Mean       N.S.E.  Chi-sq Prob
i.i.d.           0.940191     0.004233     0.961599
4% taper         0.940155     0.003972     0.957111
8% taper         0.940150     0.003735     0.954198
15% taper        0.940160     0.003178     0.946683
```

The role of each convergence diagnostic measure is described in the following sub-sections.

6.2.2 Autocorrelation estimates

The role of the time-series autocorrelation estimates is to provide an indication of how much independence exists in the sequence of each β parameter draws. From time-series analysis we know that if $\beta, i = 1, \ldots, n$ is a stationary correlated process, then $\bar{\beta} = (1/n)\sum_{i=1}^{n} \beta_i$ is a consistent estimate of $E(\beta)$ as $n \rightarrow \infty$, so it is permissible to simulate correlated draws from the posterior distribution in order to summarize the features of the posterior. This is provided that we produce a large enough sample of draws, and the amount of correlation plays a role in determining the number of draws necessary. For example, if β follows an AR(1) process with serial correlation parameter ρ, mean $\bar{\beta}$ and standard deviation σ, we know that $\bar{\beta}$ has a standard deviation given by:

$$\sigma_{\beta} = \sigma/(n)^{1/2}\sqrt{(1+\rho)/(1-\rho)} \tag{6.12}$$

If $\rho = 0$, so that the draws represent an independent identically distributed (iid) process, the standard deviation has the usual form: $\sigma/(n)^{1/2}$, but if $\rho = 0.9$, the standard error becomes quite large. In this case, we would have to run the sampler $\sqrt{1.9/0.1} = 4.35$ times longer than if the Gibbs draws represented iid draws. A high degree of autocorrelation indicates that we may need to carry out more draws to achieve a sample of sufficient size to draw accurate posterior estimates.

The **coda** results indicate that our draws for the parameters β exhibit small autocorrelations at lags 1, 5, 10 and 50, so we need not be concerned about this problem in our particular application.

6.2.3 Raftery-Lewis diagnostics

Raftery and Lewis (1992a, 1992b, 1995) proposed a set of diagnostics that they implemented in a FORTRAN program named Gibbsit, which were converted to a MATLAB function **raftery**. This function is called by **coda**, but can also be used independently of **coda**. Given some output from a Gibbs (or MCMC) sampler, Raftery and Lewis provide an answer regarding how long to monitor the chain of draws that is based on the accuracy of the posterior summaries desired by the user. They require that the user specify three pieces of information that are set to default values by **coda**, or can be user-defined using the 'info' structure variable as an input argument to the function.

1. Which quantiles of the marginal posteriors are you interested in? Usually the answer is the 2.5% and 97.5% points, because these are the

basis for a 95% interval estimate. This information is set using 'info.q', which has a default value of 0.025.

2. What is the minimum probability needed to archive the accuracy goals. A default value of 95% is used, or can be set by the user with 'info.s', which has a default value of 0.95.

3. How much accuracy is desired in the estimated quantiles? Raftery and Lewis specify this using the area to the left (in the case of info.q = 0.025) or right (in the case of info.q=0.975) of the reported quantile in the CDF. By default info.r=0.005, so that nominal reporting based on a 95% interval using the 0.025 and 0.975 quantile points should result in actual posterior values that lie between 0.95 and 0.96.

Given our draws for β, **raftery** dichotomizes the draws using a binary time-series that is unity if $\beta_i \leq$ 'info.q' and zero otherwise. This binary chain should be approximately Markovian so standard results for two-state Markov chains can be used to estimate how long the chain should be run to achieve the desired accuracy for the chosen quantile 'info.q'.

The function **coda** prints out three different estimates from the **raftery** function. A thinning ratio which is a function of the amount of autocorrelation in the draws, the number of draws to use for 'burn-in' before beginning to sample the draws for purposes of posterior inference, and the total number of draws needed to achieve the accuracy goals.

Some terminology that will help to understand the **raftery** output. It is always a good idea to discard a number of initial draws referred to as "burn-in" draws for the sampler. Starting from arbitrary parameter values makes it unlikely that initial draws come from the stationary distribution needed to construct posterior estimates. Another practice followed by researchers involves saving only every third, fifth, tenth, etc. draw since the draws from a Markov chain are not independent. This practice is labeled "thinning" the chain. Neither thinning or burn-in are mandatory in Gibbs sampling and they tend to reduce the effective number of draws on which posterior estimates are based.

From the **coda** output, we see that the thinning estimate provided by **raftery** in the second column is 1, which is consistent with the lack of autocorrelation in the sequence of draws. The third column reports that only 2 draws are required for burn-in, which is quite small. Of course, we started our sampler using the default least-squares estimates provided by **ols_g** which should be close to the true values of unity used to generate the regression data set. In the fourth column, we find the total number of draws needed to achieve the desired accuracy for each parameter. This is given as 3869 for β_1, β_2 and β_3, which exceeds the 1,000 draw we used, so it would be advisable to run the sampler again using this larger number of draws.

On the other hand, a call to the function **raftery** with the desired accuracy ('info.r') set to 0.01, so that nominal reporting based on a 95% interval using the 0.025 and 0.975 quantile points should result in actual posterior values that lie between 0.95 and 0.97, produces the results shown below. These indicate that our 1,000 draws would be adequate to produce this desired level of accuracy for the posterior.

```
% ----- Example 6.3 Using the raftery() function
q = 0.025;
r = 0.01;
s = 0.95;
res = raftery(result.bdraw,q,r,s);
prt(res,vnames);
Raftery-Lewis Diagnostics for each parameter chain
(q=0.0250, r=0.010000, s=0.950000)
Variable       Thin       Burn   Total(N)     (Nmin)     I-stat
beta1             1          2        893        937      0.953
beta2             1          2        893        937      0.953
beta3             1          2        893        937      0.953
```

The Nmin reported in the fifth column represents the number of draws that would be needed if the draws represented an iid chain, which is virtually true in our case. Finally, the $i-$statistic is the ratio of the fourth to the fifth column. Raftery and Lewis indicate that values exceeding 5 for this statistic are indicative of convergence problems with the sampler.

6.2.4 Geweke diagnostics

The function **coda** also produces estimates of the numerical standard errors (NSE) and relative numerical efficiency (RNE) proposed by Geweke (1992). Using spectral analysis of time-series methods, we can produce an estimate of the variance of the β parameters we are interested in based on the sampled values using:

$$\mathrm{var}(\hat{\beta}_i) = S(0)/k \tag{6.13}$$

where $S(0)$ is the spectral density of β_i evaluated at $\omega = 0$. Issues arise in how one approximates $S(\omega)$, so alternative tapering of the spectral window is used. The **coda** function reports estimates of the NSE and RNE based on 4%, 8% and 15% tapering or truncation of the periodgram window. The

MATLAB functions that implement these calculations are adaptions of routines provided by Geweke and Chib, that can be found on the internet at http://www.econ.umn.edu/bacc.

The first set of NSE and RNE estimates reported are based on the assumption that the draws come from an iid process. These are reported along with the means and standard deviations of the chain of draws. A second set of NSE and RNE estimates are reported based on alternative tapering of the spectral window, where the non-iid nature of the draws is taken into account by the NSE and RNE estimates. Dramatic differences between these estimates would lead one to rely on the latter set of estimates, as these differences would reflect autocorrelation in the draws.

The RNE estimates provide an indication of the number of draws that would be required to produce the same numerical accuracy if the draws represented had been made from an iid sample drawn directly from the posterior distribution. In our example, the RNE's are close to unity, indicative of the iid nature of our sample. RNE estimates greater than unity, say around 3, would indicate that only 33% of the number of draws would be required to achieve the same accuracy from an iid set of draws.

These results are produced by a call to the MATLAB function **momentg**, which is called by **coda**. As with the function **raftery**, this function can be called by itself without invoking **coda**. As an example:

```
% ----- Example 6.4 Geweke's convergence diagnostics
result = ols_g(y,x,prior,ndraw,nomit);
vnames = strvcat('beta1','beta2','beta3');
geweke = momentg(result.bdraw);
prt(geweke,vnames);
Geweke Diagnostics for each parameter chain
Variable       Mean    std dev    NSE iid    RNE iid
beta1      1.040659   0.111505   0.003526   1.000000
beta2      0.980891   0.104660   0.003310   1.000000
beta3      0.938394   0.113805   0.003599   1.000000

Variable     NSE 4%     RNE 4%     NSE 8%     RNE 8%    NSE 15%    RNE 15%
beta1      0.003509   1.009612   0.003309   1.135733   0.003229   1.192740
beta2      0.003423   0.934811   0.003453   0.918728   0.003194   1.073586
beta3      0.003235   1.237219   0.002876   1.565849   0.003250   1.225889
```

A second set of diagnostics suggested by Geweke are printed by **coda** after the NSE and RNE estimates. These diagnostics represent a test of whether the sample of draws has attained an equilibrium state based on the means of the first 20% of the sample of draws versus the last 50% of the sample. If the Markov chain of draws from the Gibbs sampler has reached an equilibrium state, we would expect the means from these two splits of the sample to be roughly equal. A $Z-$test of the hypothesis of equality of these two means is carried out and the chi-squared marginal significance is reported. For our illustrative example, the second β parameter does not fair well on these tests. We cannot reject the hypothesis of equal means at the 95% level of significance, but we can at the 90% level. Increasing the number of draws to 4,100 (suggested by the Rafterty and Lewis diagnostics) and discarding the first 100 for burn-in produced the following results for the chi-squared test of equality of the means from the first 20% versus the last 50% of the 4,000 draws.

```
Geweke Chi-squared test for each parameter chain
First 20% versus Last 50% of the sample
Variable            beta1
NSE estimate         Mean       N.S.E.  Chi-sq Prob
i.i.d.           1.042974     0.002132     0.453998
4% taper         1.042788     0.002274     0.461874
8% taper         1.042987     0.002008     0.428475
15% taper        1.043294     0.001643     0.394748

Variable            beta2
NSE estimate         Mean       N.S.E.  Chi-sq Prob
i.i.d.           0.982636     0.001962     0.596623
4% taper         0.982807     0.001856     0.612251
8% taper         0.982956     0.001668     0.623582
15% taper        0.982959     0.001695     0.630431

Variable            beta3
NSE estimate         Mean       N.S.E.  Chi-sq Prob
i.i.d.           0.942324     0.002137     0.907969
4% taper         0.942323     0.001813     0.891450
8% taper         0.942284     0.001823     0.885642
15% taper        0.942288     0.001936     0.892751
```

Here we see that the means are equal, indicating no problems with convergence. The **coda** function allows the user to specify the proportions of the sample used to carry out this test as 'info.p1' and 'info.p2' in the structure variable used to input user-options to **coda**. The default values based on the first 20% of the sample versus the last 50% are values used by the Splus version of CODA.

The chi-squared tests are implemented by a call inside **coda** to a MATLAB function **apm**. This function allows one to produce posterior moment estimates that represent an average over two sets of draws. This function can be used without invoking **coda** and would be useful in cases where one wished to combine smaller samples from multiple MCMC or Gibbs sampling runs and examine the impact of the additional draws on NSE or test for equality of the means from the two samples. The documentation for **apm** is:

```
PURPOSE: computes Geweke's chi-squared test for
two sets of MCMC sample draws
------------------------------------------------
USAGE: result = apm(results1,results2)
where: results1 = a structure returned by gmoment
       results2 = a structure returned by gmoment
------------------------------------------------
RETURNS: a structure:
     results(i).pmean(k) = posterior mean for variable i
                           for k = nse, nse1,nse2,nse3
     results(i).nse(k)   = nse for variable i
                           for k = nse, nse1,nse2,nse3
     results(i).prob(k)  = chi-sq test prob for variable i
                           for k = nse, nse1,nse2,nse3
------------------------------------------------
```

As an illustration, suppose we started our **ols_g** sampler at different starting values. A structure variable can be used as input to **ols_g** to set starting values for the sampler as illustrated in Example 6.5 below. Each call to the function will also use different seeds for the random number generators that produce the normal and chi-squared random deviate draws. To test convergence, we call the **apm** function with results structures returned by **momentg** based on the two sets of draws. We then use **prt** to print the means, NSE and chi-squared test results.

```
% ----- Example 6.5 Using the momentg() function
n=100; k=3; % set number of observations and variables
randn('seed',10101);
x = randn(n,k); b = ones(k,1); % generate data set
randn('seed',20201);
y = x*b + randn(n,1);
ndraw1 = 600; ndraw2 = 1100; nomit = 100;
r = [1.0 1.0 1.0]';            % prior b means
R = eye(k); T = eye(k);        % prior b variance
rval = 100;                    % homoscedastic prior for r-value
prior.beta = r; prior.bcov = T;
prior.rmat = R; prior.rval = rval;
start1.b = zeros(k,1); start1.sig = 1.0; start1.V = ones(n,1);
result1 = ols_g(y,x,prior,ndraw1,nomit,start1);
gres1 = momentg(result1.bdraw);
start2.b = ones(k,1); start2.sig = 10.0; start2.V = start1.V;
```

```
randn('seed',30301);
result2 = ols_g(y,x,prior,ndraw2,nomit,start2);
gres2 = momentg(result2.bdraw);
result = apm(gres1,gres2);
prt(result)
Geweke Chi-squared test for each parameter chain
First 33 % versus Last 67 % of the sample
Variable       variable 1
NSE estimate           Mean           N.S.E.          Equality chi sq
 i.i.d.              1.04442600       0.00285852           0.6886650
 4% taper            1.04414401       0.00303774           0.6864577
 8% taper            1.04428137       0.00303411           0.6934830
15% taper            1.04455890       0.00267794           0.6867602
Variable       variable 2
NSE estimate           Mean           N.S.E.          Equality chi sq
 i.i.d.              0.97690447       0.00271544           0.9589017
 4% taper            0.97686937       0.00233080           0.9581417
 8% taper            0.97684701       0.00199957           0.9586626
15% taper            0.97683040       0.00172392           0.9614142
Variable       variable 3
NSE estimate           Mean           N.S.E.          Equality chi sq
 i.i.d.              0.93683408       0.00298336           0.7947394
 4% taper            0.93733319       0.00260842           0.7662486
 8% taper            0.93728792       0.00243342           0.7458718
15% taper            0.93728776       0.00227986           0.7293916
```

Another useful function for examining MCMC output is **pltdens**, a function from the *graphing library* that produces density plots. This function is part of a public domain statistics toolbox written by Anders Holtsberg with the documentation altered to conform to that used by all *Econometrics Toolbox* functions. Samples of MCMC draws can be used to produce posterior density plots with a simple call such as:

```
pltdens(result.bdraw(:,1));
% demo of pltdens options
bandwidth = 0.2; % a kernel density smoothing parameter option
positive = 1;    % a flag for densities with zero mass at negative values
kerneltype = 1;  % a Gaussian kernel type
pltdens(result.sdraw,bandwidth,positive,kerneltype);
```

6.3 A heteroscedastic linear model

There is a history of Bayesian literature that deals with heteroscedastic and leptokurtic disturbances. A non-Bayesian regression methodology was introduced by Lange, Little and Taylor (1989), which assumes an independent Student-t distribution for the regression disturbances. Geweke (1993) provides an equivalent Bayesian approach which he labels a heteroscedastic normal linear regression model. We adopt the approach of Geweke (1993) to extend our basic model from Section 6.2.

The regression model we wish to implement is shown in (6.14).

$$
\begin{aligned}
y &= X\beta + \varepsilon \\
\varepsilon &\sim \mathrm{N}(0, \sigma^2 V), \qquad V = \mathrm{diag}(v_1, v_2, \ldots, v_n) \\
\beta &\sim N(c, T) \\
\sigma &\sim (1/\sigma) \\
r/v_i &\sim \mathrm{ID}\ \chi^2(r)/r \\
r &\sim \Gamma(m, k)
\end{aligned}
\tag{6.14}
$$

Where y is an nx1 vector of dependent variable observations and X is an nxk matrix of explanatory variables. As before, we place a multivariate normal prior on β and a diffuse prior on σ. The parameters to be estimated are β, σ and the relative variance terms $(v_1, v_2, \ldots, v_n)$, which are assumed fixed but unknown. The thought of estimating n parameters, $v_1, v_2, \ldots, v_n$, in addition to the k+1 parameters, β, σ using n data observations may seem quite problematical. However, a Bayesian approach is taken which assigns an independent $\chi^2(r)/r$ prior distribution to the v_i terms that depends on a single hyperparameter r. This allows us to estimate these additional n model parameters by adding the single parameter r to our model estimation procedure.

This type of prior has been used by Lindley (1971) for cell variances in an analysis of variance problem, and Geweke (1993) in modeling heteroscedasticity and outliers. The specifics regarding the prior assigned to the V_i terms can be motivated by considering that the mean of prior, which we designate E_p equals unity, that is: $E_p(1/v_{ij}) = 1$, and the prior variance also designated with the subscript p is: $\mathrm{var}_p(1/v_{ij}) = 2/r$. This implies that as r becomes very large, the prior reflects the special case where $\varepsilon_i \sim N(0, \sigma^2 I_n)$. We will see that the role of $V_i \neq I_n$ is to robustify against outliers and observations containing large variances by downweighting these observations. Large r values are associated with a prior belief that outliers and non-constant variances do not exist.

We choose to control the values assigned to the hyperparameter r by assigning a $\Gamma(m, k)$ prior distribution to this parameter. This distribution has a mean of m/k and variance m/k^2, so using $m = 8, k = 2$ would assign a prior to r centered on a small $r = 4$ with variance of r equal to 2. For small values of r, we can see the impact of the prior distribution assumed for v_{ij} by considering that, the mean of the prior is $r/(r-2)$ and the mode of the prior equals $r/(r+2)$. Small values of the hyperparameter r allow the v_{ij} to take on a skewed form where the mean and mode are quite different. This is illustrated in Figure 6.1 where distributions for v_i associated with various values of the parameter r are presented.

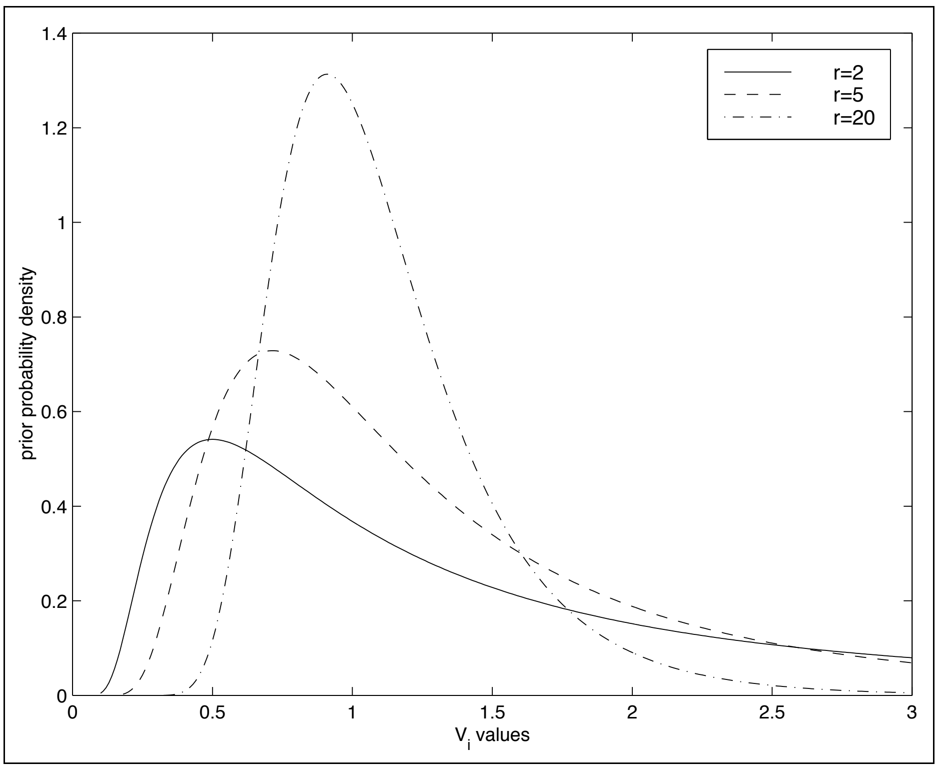

Figure 6.1: Prior V_i distributions for various values of r

As an intuitive motivation for the conditional distributions in this model, note that, given values for $V = \text{diag}(v_1, \ldots, v_n)$, we could proceed to construct estimates for β and σ using a version of the Theil-Goldberger mixed estimator based on generalized least-squares. It is often the case with the Gibbs sampling methodology that complex estimation problems are simplified considerably by conditioning on unknown parameters, that is, assuming these values are known.

The conditional posterior density for β is:

$$\begin{aligned} \beta|(\sigma, V) &\sim N[H(X'V^{-1}y + \sigma^2 R'T^{-1}c)\ ,\ \sigma^2 H]. \\ H &= (X'V^{-1}X + R'T^{-1}R)^{-1} \end{aligned} \tag{6.15}$$

If we let $e_i = y_i - x_i'\beta$, the conditional posterior density for σ is

$$[\sum_{i=1}^{n}(e_i^2/v_i)/\sigma^2]|(\beta, V) \sim \chi^2(n) \tag{6.16}$$

This result parallels our simple case from section 6.2 where we adjust the e_i using the relative variance terms v_i.

Geweke (1993) shows that the posterior distribution of V conditional on (β, σ) is proportional to:

$$[(\sigma^{-2}e_i^2 + r)/v_i]|(\beta, \sigma) \sim \chi^2(r + 1) \tag{6.17}$$

Given the three conditional posterior densities in (6.15), through (6.17), we can formulate a Gibbs sampler for this model using the following steps:

1. Begin with arbitrary values for the parameters β^0, σ^0, v_i^0 and r^0, which we designate with the superscript 0.
2. Compute the mean and variance of β using (6.15) conditional on the initial values σ^0, v_i^0 and r^0.
3. Use the computed mean and variance of β to draw a multivariate normal random vector, which we label β^1.
4. Calculate expression (6.16) using β^1 determined in step 3 and use this value along with a random $\chi^2(n)$ draw to determine σ^1.
5. Using β^1 and σ^1, calculate expression (6.17) and use the value along with an $n-$vector of random $\chi^2(r + 1)$ draws to determine $v_i, i = 1, \ldots, n$.
6. Draw a $\Gamma(m, k)$ value to update r^0.

These steps constitute a single pass of the Gibbs sampler. As before, we wish to make a large number of passes to build up a sample $(\beta^j, \sigma^j, v_i^j, r^j)$ of j values from which we can approximate the posterior distributions for our parameters.

The implementation of the heteroscedastic linear model is similar to our previous example from Section 6.2 with the addition of an updating step to handle draws for the v_i parameters and the $\Gamma(m,k)$ draw to update the hyperparameter r. To demonstrate MATLAB code for this model, we generate a heteroscedastic data set by multiplying normally distributed constant variance disturbances for the last 50 observations in a sample of 100 by the square root of a time trend variable. Prior means for β were set equal to the true values of unity and prior variances equal to unity as well. A diffuse prior for σ was employed and the $\Gamma(m,k)$ prior for r was set to $m = 8, k = 2$ indicating a prior belief in heteroscedasticity.

```
% ----- Example 6.6 Heteroscedastic Gibbs sampler
n=100; k=3; % set number of observations and variables
randn('seed',202112); % set seed for random number generator
x = randn(n,k); b = ones(k,1);  % generate data set
tt = ones(n,1); tt(51:100,1) = [1:50]';
y = x*b + randn(n,1).*sqrt(tt); % heteroscedastic disturbances
ndraw = 2100; nomit = 100;      % set the number of draws
bsave = zeros(ndraw,k);         % allocate storage for results
ssave = zeros(ndraw,1);  rsave = zeros(ndraw,1);
vsave = zeros(ndraw,n);
r = [1.0 1.0 1.0]';             % prior b means
R = eye(k); T = eye(k);         % prior b variance
Q = inv(T); Qpr = Q*r;
b0 = (x'*x)\(x'*y);             % use ols starting values
sige = (y-x*b0)'*(y-x*b0)/(n-k);
V = ones(n,1); in = ones(n,1);  % initial value for V
rval = 4;                       % initial value for rval
mm=8; kk=2;                     % prior for r-value
                                % mean(rvalue) = 4, var(rvalue) = 2
tic;                            % start timing
for i=1:ndraw;                  % Start the sampling
  ys = y.*sqrt(V); xs = matmul(x,sqrt(V));
  xpxi = inv(xs'*xs + sige*Q);
  b = xpxi*(xs'*ys + sige*Qpr); % update b
  a = chol(xpxi);
  b = sqrt(sige)*a'*randn(k,1) + b; % draw MV normal mean(b), var(b)
  bsave(i,:) = b';              % save b draws
  e = y - x*b; ssr = e'*e;      % update sige
  chi = chis_rnd(1,n);          % do chisquared(n) draw
  sige = ssr/chi; ssave(i,1) = sige; % save sige draws
  chiv = chis_rnd(n,rval+1);    % update vi
  vi = ((e.*e./sige) + in*rval)./chiv;
  V = in./vi; vsave(i,:) = vi'; % save the draw
  rval = gamm_rnd(1,1,mm,kk);   % update rval
  rsave(i,1) = rval;            % save the draw
end;                            % End the sampling
toc;                            % stop timing
bhat = mean(bsave(nomit+1:ndraw,:));  % calculate means and std deviations
```

```
bstd = std(bsave(nomit+1:ndraw,:));   tstat = bhat./bstd;
smean = mean(ssave(nomit+1:ndraw,1)); vmean = mean(vsave(nomit+1:ndraw,:));
rmean = mean(rsave(nomit+1:ndraw,1));
tout = tdis_prb(tstat',n); % compute t-stat significance levels
% set up for printing results
in.cnames = strvcat('Coefficient','t-statistic','t-probability');
in.rnames = strvcat('Variable','variable 1','variable 2','variable 3');
in.fmt = '%16.6f'; tmp = [bhat' tstat' tout];
fprintf(1,'Gibbs estimates \n'); % print results
mprint(tmp,in);
fprintf(1,'Sigma estimate = %16.8f \n',smean);
fprintf(1,'rvalue estimate = %16.8f \n',rmean);
result = theil(y,x,r,R,T); % compare to Theil-Goldberger estimates
prt(result);  plot(vmean); % plot vi-estimates
title('mean of vi-estimates');
```

The program makes use of the fact that V^{-1} is a diagonal matrix, so we transform the vector y by multiplying it by $\sqrt{(}V^{-1})$ and we use the Gauss function **matmul** to carry out the same transformation on the matrix X. Using this transformation saves space in RAM memory and speeds up the Gibbs sampler. The function **gamm_rnd** from the *distributions function library* (see Chapter 9) is used to produce the random $\Gamma(m, k)$ draws and **chis_rnd** is capable of generating a vector of random χ^2 draws.

The results based on 1,100, 2,100 and 10,100 draws with the first 100 omitted for burn-in are shown below along with the Theil-Goldberger estimates. We can see that convergence is not a problem as the means and standard deviations from the sample of 2,100 and 10,100 draws are quite close. The time required to carry out the draws on a MacIntosh G3, 266 Mhz. computer are also reported for each set of draws. The heteroscedastic linear model estimates are slightly more accurate than the Theil-Goldberger estimates, presumably because the latter do not take the heteroscedastic nature of the disturbances into account. Note that both estimation procedures are based on the same prior information.

```
elapsed_time = 13.2920 seconds (1,100 draws)
Gibbs estimates
Variable        Coefficient      t-statistic    t-probability
variable 1         1.510766         4.382439         0.000029
variable 2         1.056020         3.026450         0.003146
variable 3         0.975192         2.967255         0.003759
Sigma estimate =      11.70012398
rvalue estimate =       4.05923847

elapsed_time = 23.9866 seconds (2,100 draws)
Gibbs estimates
```

```
Variable        Coefficient      t-statistic    t-probability
variable 1         1.500910         4.437038         0.000023
variable 2         1.064960         2.977550         0.003645
variable 3         0.995109         2.956360         0.003883
Sigma estimate =      11.66566125
rvalue estimate =       4.02780204

elapsed_time = 109.46 seconds (10,100 draws)
Gibbs estimates
Variable        Coefficient      t-statistic    t-probability
variable 1         1.506098         4.505296         0.000018
variable 2         1.060568         3.027138         0.003140
variable 3         0.992387         2.952396         0.003930
Sigma estimate =      11.70178479
rvalue estimate =       4.01026191

Theil-Goldberger Regression Estimates
R-squared      =    0.2935
Rbar-squared   =    0.2790
sigma^2        =   11.4540
Durbin-Watson  =      2.05308
Nobs, Nvars    =    100,     3
***************************************************************
Variable         Prior Mean    Std Deviation
variable 1         1.000000         0.707107
variable 2         1.000000         0.707107
variable 3         1.000000         0.707107

***************************************************************
      Posterior Estimates
Variable        Coefficient      t-statistic    t-probability
variable 1         1.543398         1.429473         0.156081
variable 2         1.104141         0.996164         0.321649
variable 3         1.025292         0.953436         0.342739
```

A plot of the mean over the 2000 samples of v_i estimates is shown in Figure 6.2. These estimates appear to capture the nature of the heteroscedastic disturbances introduced in the model for observations 51 to 100.

6.4 Gibbs sampling functions

To make implementation of these models more convenient, we can construct a function that implements the Markov Chain Monte Carlo sampling for the Bayesian heteroscedastic linear model. The documentation for this function **ols_g** is shown below.

```
PURPOSE: Gibbs estimates for the Bayesian heteroscedastic linear model
```

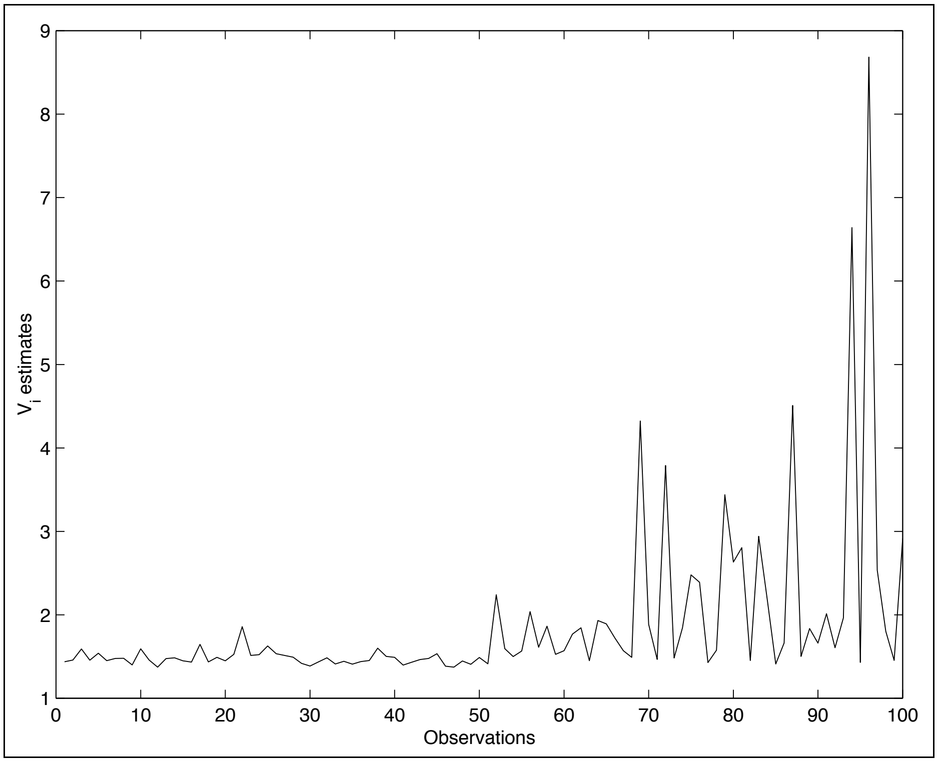

Figure 6.2: Mean of V_i draws

```
          y = X B + E, E = N(0,sige*V),
          V = diag(v1,v2,...vn), r/vi = ID chi(r)/r, r = Gamma(m,k)
          B = N(c,T),  sige = gamma(nu,d0)
---------------------------------------------------
 USAGE: results = ols_g(y,x,,ndraw,nomit,prior,start)
 where: y    = dependent variable vector
        x    = independent variables matrix of rank(k)
       ndraw = # of draws
       nomit = # of initial draws omitted for burn-in
       prior = a structure for prior information input:
               prior.beta, prior means for beta,   c above
               prior.bcov, prior beta covariance , T above
               prior.rval, r prior hyperparameter, default=4
               prior.m,    informative Gamma(m,k) prior on r
               prior.k,    informative Gamma(m,k) prior on r
               prior.nu,   informative Gamma(nu,d0) prior on sige
               prior.d0    informative Gamma(nu,d0) prior on sige
                           default for above: nu=0,d0=0 (diffuse prior)
```

```
      start = (optional) structure containing starting values:
              defaults: OLS beta,sige, V= ones(n,1)
              start.b   = beta starting values (nvar x 1)
              start.sig = sige starting value  (scalar)
              start.V   = V starting values (n x 1)
 ---------------------------------------------------
 RETURNS: a structure:
         results.meth  = 'ols_g'
         results.bdraw = bhat draws (ndraw-nomit x nvar)
         results.vmean = mean of vi draws (nobs x 1)
         results.sdraw = sige draws (ndraw-nomit x 1)
         results.yhat  = mean of draws from posterior for y-predicted
         results.rdraw = r draws (ndraw-nomit x 1), if Gamma(m,k) prior
         results.pmean = b prior means (prior.beta from input)
         results.pstd  = b prior std deviation, sqrt(prior.bcov)
         results.m     = prior m-value for r hyperparameter (if input)
         results.k     = prior k-value for r hyperparameter (if input)
         results.r     = value of hyperparameter r (if input)
         results.nu    = prior nu-value for sige prior
         results.d0    = prior d0-value for sige prior
         results.nobs  = # of observations
         results.nvar  = # of variables
         results.ndraw = # of draws
         results.nomit = # of initial draws omitted
         results.y     = actual observations
         results.x     = x-matrix
         results.time  = time taken for sampling
 --------------------------------------------------
 NOTE: use either improper prior.rval
       or informative Gamma prior.m, prior.k, not both of them
---------------------------------------------------
 SEE ALSO: coda, gmoment, prt_gibbs(results)
---------------------------------------------------
 REFERENCES: Geweke (1993)  'Bayesian Treatment of the
 Independent Student-t Linear Model', Journal of Applied
 Econometrics, 8, s19-s40.
 ----------------------------------------------------
```

The function allows an improper prior for the hyperparameter r to be entered by simply setting a value, or parameter values m, k for a proper $\Gamma(m, k)$ prior can be input. We use a structure variable 'prior' to input the prior parameters and then rely on a series of 'strcmp' comparisons to parse the prior information input by the user. An informative prior for σ^2 can be set using the 'prior.nu' and 'prior.d0' structure variable fields, or a diffuse prior (the default) can be input by setting these fields to zero values.

A design decision was made to return the draws (excluding the first 'nomit' values) so the user can construct posterior densities and graphically examine the sampling process. Note that the format of the draws returned in the structure is such that **mean(results.bdraw)** or **std(results.bdraw)** will produce posterior means and standard deviations.

A corresponding function **prt_gibbs** is used to process the results structure returned by **ols_g** and print output in the form of a regression as shown below:

```
Bayesian Heteroscedastic Linear Model Gibbs Estimates
R-squared      =    0.1556
Rbar-squared   =    0.1382
sigma^2        =    4.6415
Nobs, Nvars    =    100,     3
ndraws,nomit   =   1100,   100
time in secs   =   12.1118
r-value        =      4
***************************************************************
Variable           Prior Mean    Std Deviation
variable   1        0.000000       31.622777
variable   2        0.000000       31.622777
variable   3        0.000000       31.622777
***************************************************************
      Posterior Estimates
Variable          Coefficient      t-statistic    t-probability
variable   1        1.124002         3.917272         0.000164
variable   2        1.045441         3.664259         0.000399
variable   3        0.812168         2.882300         0.004833
```

As an example of constructing such a function, consider the case of an AR(m) autoregressive model:

$$(1 - \phi_1 L - \phi_2 L^2 - \ldots - \phi_m L^m) y_t = c + \varepsilon_t \tag{6.18}$$

where we wish to impose the restriction that the mth order difference equation is stable. This requires that the roots of

$$(1 - \phi_1 z - \phi_2 z^2 - \ldots - \phi_m z^m) = 0 \tag{6.19}$$

lie outside the unit circle. Restrictions such as this, as well as non-linear restrictions, can be imposed on the parameters during Gibbs sampling by simply rejecting values that do not meet the restrictions (see Gelfand, Hills, Racine-Poon and Smith, 1990). Below is a function **ar_g** that implements a Gibbs sampler for this model and imposes the stability restrictions using rejection sampling. Information regarding the results structure is not shown to save space, but the function returns a structure variable containing draws and other information in a format similar to the **ols_g** function.

```
 PURPOSE: estimates Bayesian heteroscedastic AR(m) model imposing
          stability using Gibbs sampling
          y = A(L)y B + E, E = N(0,sige*V),
          V = diag(v1,v2,...vn), r/vi = ID chi(r)/r, r = Gamma(m,k)
          B = N(c,T),  sige = gamma(nu,d0)
---------------------------------------------------
 USAGE:    results = ar_g(y,nlag,ndraw,nomit,prior,start)
 where: y    = dependent variable vector
        nlag = # of lagged values
       ndraw = # of draws
       nomit = # of initial draws omitted for burn-in
       prior = a structure for prior information input:
               prior.beta, prior means for beta,   c above
               prior.bcov, prior beta covariance , T above
               prior.rval, r prior hyperparameter, default=4
               prior.m,    informative Gamma(m,k) prior on r
               prior.k,    informative Gamma(m,k) prior on r
               prior.const, a switch for constant term,
                            default = 1 (a constant included)
               prior.nu,    a prior parameter for sige
               prior.d0,    a prior parameter for sige
                            (default = diffuse prior for sige)
       start = (optional) structure containing starting values:
               defaults: OLS beta,sige, V= ones(n,1)
               start.b   = beta starting values (nvar x 1)
               start.sig = sige starting value  (scalar)
               start.V   = V starting values (n x 1)
 --------------------------------------------------
 NOTES: a constant term is automatically included in the model
        unless you set prior.const = 0;
 --------------------------------------------------
 SEE ALSO: prt, prt_gibbs(results), coda
 ----------------------------------------------------
  REFERENCES: Chib (1993) `Bayes regression with autoregressive
 errors: A Gibbs sampling approach,'  Journal of Econometrics, pp. 275-294.
 ----------------------------------------------------
[n junk] = size(y); results.y = y;
if ~isstruct(prior)  % error checking on input
    error('ar_g: must supply the prior as a structure variable');
elseif nargin == 6   % user-supplied starting values
    if ~isstruct(start)
        error('ar_g: must supply starting values in a structure');
    end;
b0 = start.b; sige = start.sig; V = start.V; sflag = 1;
elseif  nargin == 5, sflag = 0;  % we supply ols starting values
else
error('Wrong # of arguments to ar_g');
end;
fields = fieldnames(prior); nf = length(fields); % parse prior info
```

```
mm = 0; rval = 4; const = 1; nu = 0; d0 = 0;     % set defaults
for i=1:nf
    if strcmp(fields{i},'rval'),      rval = prior.rval;
    elseif strcmp(fields{i},'m'),     mm = prior.m; kk = prior.k;
        rval = gamm_rnd(1,1,mm,kk);   % initial value for rval
    elseif strcmp(fields{i},'const'), const = prior.const;
    elseif strcmp(fields{i},'nu'),    nu = prior.nu; d0 = prior.d0;
    end;
end;
if sflag == 0 % we supply ols starting values
 if const == 1, x = [ones(n,1) mlag(y,nlag)];
 else, x = mlag(y,nlag);
 end;
x = trimr(x,nlag,0); y = trimr(y,nlag,0); % feed the lags
nadj = length(y);
b0 = x\y;  % Find ols values as initial starting values
k = nlag+const; sige = (y-x*b0)'*(y-x*b0)/(nadj-k);
V = ones(nadj,1); in = ones(nadj,1);      % initial value for V
else
 if const == 1, x = [ones(n,1) mlag(y,nlag)];
 else, x = mlag(y,nlag);
 end;
x = trimr(x,nlag,0); y = trimr(y,nlag,0); % feed the lags
nadj = length(y); in = ones(nadj,1);      % initial value for V
end;
c = prior.beta; [checkk,junk] = size(c);
if checkk ~= k,   error('ar_g: prior means are wrong');
elseif junk ~= 1, error('ar_g: prior means are wrong');
end;
T = prior.bcov; [checkk junk] = size(T);
if checkk ~= k,   error('ar_g: prior bcov is wrong');
elseif junk ~= k, error('ar_g: prior bcov is wrong');
end;
Q = inv(T); Qpc = Q*c;
% storage for draws
ssave = zeros(ndraw-nomit,1); rsave = zeros(ndraw-nomit,1);
bsave = zeros(ndraw-nomit,k); vmean = zeros(nadj,1); yhat = zeros(nadj,1);
hwait = waitbar(0,'Gibbs sampling ...');
t0 = clock; iter = 1; counter = 0;
while iter <= ndraw; % Start sampling
% generate beta conditional on sige
ys = y.*sqrt(V); xs = matmul(x,sqrt(V));
xpx = inv(xs'*xs + sige*Q); beta1 = xpx*(xs'*ys + sige*Qpc);
c = chol(sige*xpx); accept = 0; % rejection sampling
 while accept == 0;
 beta = beta1 + c'*randn(k,1);    betap = beta';
 coef = [-fliplr(betap(2:k)) 1];  root = roots(coef);
 rootmod = abs(root);
 if min(rootmod) >= 1.0001; accept = 1;
```

```
 else, counter = counter+1; % counts acceptance rate; accept = 0;
 end;
end; % end of while loop
% generate sige conditional on beta
nu1 = nadj + nu;  e = ys - xs*beta; d1 = d0 + e'*e;
chi = chis_rnd(1,nu1); t2 = chi/d1; sige = 1/t2;
  chiv = chis_rnd(nadj,rval+1);   % update vi
  vi = ((e.*e./sige) + in*rval)./chiv; V = in./vi;
  if mm ~= 0, rval = gamm_rnd(1,1,mm,kk);  % update rval; end;
if iter > nomit; % save draws
vmean = vmean + vi;   ssave(iter-nomit,1) = sige;
yhat = yhat + randn(nadj,1).*sqrt(sige*vi) + x*beta;
bsave(iter-nomit,:) = beta';
    if mm~= 0, rsave(i-nomit,1) = rval; end;
end; % end of if
iter = iter+1;
waitbar(iter/ndraw);
end; % end of while iter < ndraw
gtime = etime(clock,t0);
results.accept = 1 - counter/(iter+counter); % find acceptance rate
results.meth  = 'ar_g';  results.bdraw = bsave;
results.sdraw = ssave;   results.vdraw = vsave;
results.pmean = prior.beta; results.pstd  = sqrt(diag(T));
if mm~= 0, results.rdraw = rsave; results.m = mm; results.k = kk;
else,      results.r = rval; results.rdraw = rsave;
end;
results.nobs  = n;          results.nadj  = nadj;
results.nvar  = nlag+const; results.ndraw = ndraw;
results.nomit = nomit;      results.time = gtime;
results.x = x;              results.nu = nu; results.d0 = d0;
```

The function allows for informative or diffuse priors on the noise variance σ^2_ε and allows for a homoscedastic or heteroscedastic implementation using the chi-squared prior described for the Bayesian heteroscedastic linear model. Processing this input information from the user is accomplished using the MATLAB 'fieldnames' command and 'strcmp' statements, with defaults provided by the function for cases where the user inputs no information for a particular field.

One point to note is the rejection sampling code, where we use the MATLAB function **roots** to examine the roots of the polynomial for stability. We also rely on a MATLAB function **fliplr** that 'flips' a vector or matrix from 'left to right'. This is needed because of the format assumed for the polynomial coefficients by the **roots** function. If the stability condition is not met, we simply carry out another multivariate random draw to obtain a new vector of coefficients and check for stability again. This process continues until we obtain a coefficient draw that meets the stability conditions. One could view this as discarding (or rejecting) draws where the coefficients are inconsistent with stability.

Use of rejection sampling can produce problems if the acceptance rate becomes excessively low. Consider that if we need a sample of 1000 draws to obtain posterior inferences and the acceptance rate is 1 of 100 draws, we would need to make 100,000 multivariate random draws for the autoregressive parameters to obtain a usable sample of 1000 draws on which to based our posterior inferences. To help the user monitor problems that might arise due to low acceptance rates, we calculate the acceptance rate and return this information in the results structure. Of course, the function **prt_gibbs** was modified to produce a printout of the results structure from the **ar_g** estimation procedure.

In addition to returning the time required in seconds, the function places a 'waitbar' on the screen that shows graphically the progress of the Gibbs sampling. This is accomplished with the MATLAB function **waitbar()**.

The following program generates data based on an AR(2) model which is on the edge of the stability conditions. The AR(1)+AR(2) coefficients equal unity, where the restriction for stability of this model requires that these two coefficients sum to less than unity. The demonstration program compares estimates from **ols**, **ar_g** and **theil**.

```
% ----- Example 6.7 Using the ar_g() function
n = 200; k = 3; e = randn(n,1)*2; y = zeros(n,1);
for i=3:n
    y(i,1) = 1 + y(i-1,1)*0.25 + y(i-2,1)*0.75 + e(i,1);
end;
x = [ones(n,1) mlag(y,2)];
yt = trimr(y,100,0); xt = trimr(x,100,0); % omit first 100 for startup
vnames = strvcat('y-variable','constant','ylag1','ylag2');
res1 = ols(yt,xt);
prt(res1,vnames);
ndraw = 1100; nomit = 100;
bmean = zeros(k,1); bcov = eye(k)*100;
prior.bcov = bcov;  % diffuse prior variance
prior.beta = bmean; % prior means of zero
res2 = ar_g(yt,2,ndraw,nomit,prior);
prt(res2,'y-variable');
res3 = theil(yt,xt,bmean,eye(k),bcov);
prt(res3,vnames);
```

The results from running the program are shown below. Running the program 100 times produced results where the stability conditions were violated 10 percent of the time by the least-squares and Theil-Goldberger estimates. Figure 6.3 shows a histogram of the distribution of $\phi_1 + \phi_2$ for the 100 sets of estimates, where the 10 violations of stability for least-squares and Theil-Goldberger appear as the two bars farthest to the right. The mean acceptance rate for the Gibbs sampler over the 100 runs was 0.78, and the median was 0.86.

```
Ordinary Least-squares Estimates
Dependent Variable =       y-variable
R-squared      =    0.9790
Rbar-squared   =    0.9786
sigma^2        =    3.5352
Durbin-Watson  =    2.0156
Nobs, Nvars    =    100,     3
***************************************************************
Variable        Coefficient      t-statistic    t-probability
constant          -0.682783        -0.561070         0.576044
ylag1              0.350730         4.429152         0.000025
ylag2              0.666715         8.256773         0.000000

Bayesian Autoregressive Model Gibbs Estimates
Dependent Variable =       y-variable
R-squared      =    0.9786
Rbar-squared   =    0.9782
sigma^2        =    3.6092
nu,d0          =      0,     0
Nobs, Nvars    =    100,     3
ndraws,nomit   =   1100,   100
accept rate    =    0.2942
time in secs   =   14.8978
rvalue         =    4.0000
***************************************************************
Variable                Prior Mean    Std Deviation
constant                  0.000000        10.000000
y-variable lag  1         0.000000        10.000000
y-variable lag  2         0.000000        10.000000
***************************************************************
      Posterior Estimates
Variable               Coefficient      t-statistic    t-probability
constant                  1.565553         2.085717         0.039550
y-variable lag  1         0.325273         3.903220         0.000172
y-variable lag  2         0.663394         7.935940         0.000000

Theil-Goldberger Regression Estimates
Dependent Variable =       y-variable
R-squared      =    0.9790
Rbar-squared   =    0.9786
sigma^2        =    3.5353
Durbin-Watson  =    2.0156
Nobs, Nvars    =    100,     3
```

```
***************************************************************
Variable          Prior Mean     Std Deviation
constant            0.000000         10.000000
ylag1               0.000000         10.000000
ylag2               0.000000         10.000000
***************************************************************
      Posterior Estimates
Variable         Coefficient       t-statistic    t-probability
constant           -0.672706         -0.296170         0.767733
ylag1               0.350736          2.355845         0.020493
ylag2               0.666585          4.391659         0.000029
```

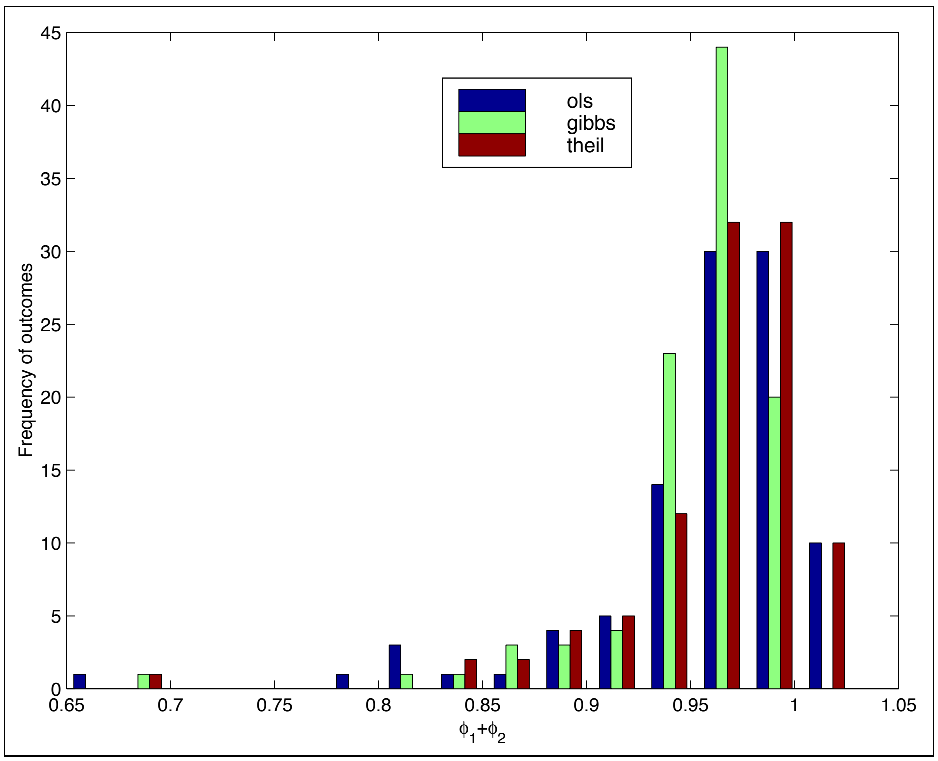

Figure 6.3: Distribution of $\phi_1 + \phi_2$

6.5 Metropolis sampling

Econometric estimation problems amenable to Gibbs sampling can take one of two forms. The simplest case is where all of the conditional distributions are from well-known distributions allowing us to sample random deviates using standard computational algorithms. This circumstance is sometimes referred to as a 'standard conjugate prior-to-posterior updating', evoking the sense of conjugate priors from Bayesian analysis. This was the situation with the Gibbs samplers described in the previous sections.

A second more complicated case that one sometimes encounters in Gibbs sampling is where one or more of the conditional distributions can be expressed mathematically, but they take an unknown form. It is still possible to implement a Gibbs sampler for these models using a host of alternative methods that are available to produce draws from distributions taking a non-standard form.

One of the more commonly used ways to deal with this situation is known as the 'Metropolis algorithm'. To illustrate this situation, we draw on a first-order spatial autoregressive (FAR) model which takes the following form:

$$\begin{aligned} y &= \rho W y + \varepsilon \\ \varepsilon &\sim N(0, \sigma^2 I_n) \end{aligned} \tag{6.20}$$

where y contains an nx1 vector of dependent variables collected from points in space, e.g., counties, states, or neighborhoods. W is an nxn known spatial weight matrix, usually containing first-order contiguity relations or functions of distance. A first-order contiguity matrix has zeros on the main diagonal, rows that contain zeros in positions associated with non-contiguous observational units and ones in positions reflecting neighboring units that are (first-order) contiguous. An example is shown in (6.21) for a sample of five areas.

$$C = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{6.21}$$

Information regarding first-order contiguity is recorded for each observation as ones for areas that are neighbors (e.g., observations 2 and 3 are neighbors to 1) and zeros for those that are not (e.g., observations 4 and 5 are not neighbors to 1). By convention, zeros are placed on the main diagonal of the spatial weight matrix. Standardization to produce row-sums of unity results in the matrix W shown in (6.22) that is used in the model.

$$W = \begin{bmatrix} 0 & 0.5 & 0.5 & 0 & 0 \\ 0.5 & 0 & 0.5 & 0 & 0 \\ 0.33 & 0.33 & 0 & 0.33 & 0 \\ 0 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{6.22}$$

The parameter ρ is a coefficient on the spatially lagged dependent variable Wy that reflects the influence of this explanatory variable on variation in the dependent variable y. The model is called a first order spatial autoregression because its represents a spatial analogy to the first order autoregressive model from time series analysis, $y_t = \rho y_{t-1} + \varepsilon_t$. Multiplying a vector y containing 5 areas cross-sectional data observations by the standardized spatial contiguity matrix W produces an explanatory variable equal to the mean of observations from contiguous states.

Using diffuse priors, $\pi(\rho)$ and $\pi(\sigma)$ for the parameters (ρ, σ) shown in (6.23),

$$\begin{aligned} \pi(\rho) &\propto \text{constant} \\ \pi(\sigma) &\propto (1/\sigma), \quad 0 < \sigma < +\infty \end{aligned} \tag{6.23}$$

which can be combined with the likelihood for this model, we arrive at a joint posterior distribution for the parameters, $p(\rho, \sigma|y)$.

$$p(\rho, \sigma|y) \propto |I_n - \rho W|\sigma^{-(n+1)}\exp\{-\frac{1}{2\sigma^2}(y - \rho W y)'(y - \rho W y)\} \tag{6.24}$$

If we treat ρ as known, the kernel for the conditional posterior for σ given ρ takes the form:

$$p(\sigma|\rho, y) \propto \sigma^{-(n+1)}\exp\{-\frac{1}{2\sigma^2}\varepsilon'\varepsilon\} \tag{6.25}$$

where $\varepsilon = y - \rho W y$. It is important to note that by conditioning on ρ (treating it as known) we can subsume the determinant, $|I_n - \rho W|$, as part of the constant of proportionality, leaving us with one of the standard distributional forms. From (6.25) we conclude that $\sigma^2 \sim \chi^2(n)$.

Unfortunately, the conditional distribution of ρ given σ takes the following non-standard form:

$$p(\rho|\sigma, y) \propto |I_n - \rho W|\{(y - \rho W y)'(y - \rho W y)\}^{-n/2} \tag{6.26}$$

To sample from (6.26) we can rely on a method called 'Metropolis sampling', within the Gibbs sampling sequence, hence it is often labeled 'Metropolis-within-Gibbs'.

Metroplis sampling is described here for the case of a symmetric normal candidate generating density. This should work well for the conditional distribution of ρ because, as Figure 6.4 shows, the function $|I_n - \rho W|$ is everywhere positive and exhibits a form similar to the normal distribution.

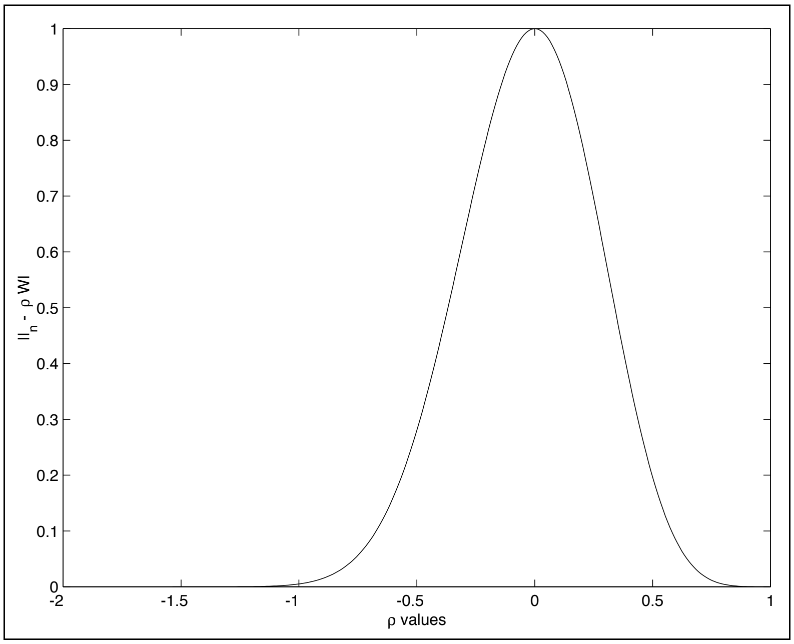

Figure 6.4: $|I_n - \rho W|$ as a function of ρ

To describe Metropolis sampling in general, suppose we are interested in sampling from a density $f()$ and x_0 denotes the current draw from f. Let the candidate value be generated by $y = x_0 + cZ$, where Z is a draw from a standard normal distribution and c is a known constant. An acceptance probability is computed using: $p = min\{1, f(y)/f(x_0\}$. We then draw a uniform random deviate we label U, and if $U < p$, the next draw from f is given by $x_1 = y$. If on the other hand, $U \geq p$, the draw is taken to be the current value, $x_1 = x_0$.

A MATLAB program to implement this approach for the case of the first-order spatial autoregressive model is shown in Example 6.8. The function **c_rho** evaluates the conditional distribution for ρ at any value of ρ. A further complication that arises in producing estimates for this model is that Anselin (1988) shows that:

$$1/\lambda_{min} < \rho < 1/\lambda_{max}$$

where λ_{min} and λ_{max} are the minimum and maximum eigenvalues of the standardized spatial weight matrix W. We impose this restriction using rejection sampling as demonstrated in the previous section.

```
% ----- Example 6.8 Metropolis within Gibbs sampling
n=49; ndraw = 1100; nomit = 100; nadj = ndraw-nomit;
% generate data based on a given W-matrix
load wmat.dat; W = wmat; IN = eye(n); in = ones(n,1); weig = eig(W);
lmin = 1/min(weig); lmax = 1/max(weig); % bounds on rho
rho = 0.7;    % true value of rho
y = inv(IN-rho*W)*randn(n,1); Wy = W*y;
              % set starting values
rho = 0.5;    % starting value for the sampler
sige = 10.0;  % starting value for the sampler
c = 0.5;      % for the Metropolis step (adjusted during sampling)
rsave = zeros(nadj,1); % storage for results
ssave = zeros(nadj,1);  rtmp = zeros(nomit,1);
iter = 1; cnt = 0;
while (iter <= ndraw);        % start sampling;
e = y - rho*Wy; ssr = (e'*e); % update sige;
chi = chis_rnd(1,n); sige = (ssr/chi);
% metropolis step to get rho update
rhox = c_rho(rho,n,y,W);      % c_rho evaluates conditional
rho2 = rho + c*randn(1); accept = 0;
 while accept == 0;           % rejection bounds on rho
  if ((rho2 > lmin) & (rho2 < lmax)); accept = 1;  end;
 rho2 = rho + c*randn(1); cnt = cnt+1;
 end;                         % end of rejection for rho
rhoy = c_rho(rho2,n,y,W);     % c_rho evaluates conditional
ru = unif_rnd(1,0,1); ratio = rhoy/rhox; p = min(1,ratio);
if (ru < p)
 rho = rho2; rtmp(iter,1) = rho; iter = iter+1;
 end;
  if (iter >= nomit);
    if iter == nomit          % update c based on initial draws
    c = 2*std(rtmp(1:nomit,1));
    end;
   ssave(iter-nomit+1,1) = sige; rsave(iter-nomit+1,1) = rho;
  end; % end of if iter > nomit
end; % end of sampling loop
% printout results
fprintf(1,'hit rate =  %6.4f \n',ndraw/cnt);
fprintf(1,'mean and std of rho %6.3f %6.3f \n',mean(rsave),std(rsave));
fprintf(1,'mean and std of sig %6.3f %6.3f \n',mean(ssave),std(ssave));
```

Some points to note about the code are:

1. We generate a model using a first-order spatial contiguity matrix W from 49 neighborhoods in Columbus, Ohio presented in Anselin (1988).

2. We adjust the parameter c used in the Metropolis step after the initial 'nomit' passes through the sampler based on a two standard deviation measure of the ρ values sampled up to this point.

3. Since 'rho2' is the candidate value that might become are updated value of ρ depending on the outcome of the Metropolis step, we carry out rejection sampling on this draw to ensure that sampled values will meet the restriction. In the event that 'rho2' does not meet the restriction, we discard it and draw another value. This process continues until we obtain a value for 'rho2' that meets the restriction.

4. The function **c_rho** evaluates (6.26) for a given value of ρ.

```
function yout = c_rho(rho,n,y,W)
% evaluates conditional distribution of rho
% for the spatial autoregressive model
IN = eye(n); B = IN - rho*W;
term0 = log(det(B));  yt = y - rho*W*y;
term3 = (n/2)*log(yt'*yt);
tmp = term0-term3;
yout = exp(tmp);
```

This estimator has been incorporated in a function **far_g** that allows the user to input an informative prior for the spatial autocorrelation parameter ρ.

We carried out an experiment to illustrate Metropolis sampling on the conditional distribution of ρ in the first-order spatial autoregressive model. A series of ten models were generated using values of ρ ranging from -0.9 to 0.9 in increments of 0.2. (The bounds on ρ for this standardized spatial weight matrix were -1.54 and 1.0, so these values of ρ used to generate the data were within the bounds.) Estimates for all ten models were derived using a sample of 2,100 draws with the first 100 omitted for "burn-in". In addition to producing Gibbs estimates, based on a diffuse prior centered on the true value of ρ having a variance of 10. We also produced maximum likelihood estimates for the ten models using the maximum likelihood methods presented in Anselin (1988). Timing results as well as acceptance rates for the Gibbs sampler are reported in Table 5.1.

Table 6.1: A Comparison of FAR Estimators

True ρ	Metropolis $\hat{\rho}$	Time (Seconds)	Accept Rate	Max Likel $\hat{\rho}$
-0.900	-0.899	12.954	0.915	-1.094
-0.700	-0.701	13.069	0.975	-1.028
-0.500	-0.500	13.373	0.999	-0.307
-0.300	-0.314	13.970	0.999	-0.797
-0.100	-0.100	13.729	0.999	-0.020
0.100	0.100	13.163	1.000	0.195
0.300	0.300	13.118	0.994	0.426
0.500	0.500	13.029	0.992	0.523
0.700	0.700	13.008	0.700	0.857
0.900	0.900	13.283	0.586	0.852

From the table we see that the Metropolis algorithm produced estimates close to the true value of ρ used to generate the data vector y, as did the maximum likelihood method. The acceptance rates are lower for the value of $\rho = 0.9$ because this magnitude is near the upper bound of unity on ρ for this particular weight matrix.

6.6 Functions in the Gibbs sampling library

This section describes some of the Gibbs sampling functions available in addition to **ar_g**, **ols_g** and **far_g** already described in the chapter.

The use of Gibbs sampling to estimate vector autoregressive models should be intuitively clear given the description of the Bayesian heteroscedastic linear model and its comparison to the Theil-Goldberger estimates in Section 6.3. A series of functions **bvar_g, rvar_g, becm_g, recm_g** implement Gibbs sampling estimation of the **bvar, rvar, becm, recm** models described in Chapter 5. These functions should be useful in cases where outliers or non-constant variance is present. In cases where the disturbances obey the Gauss-Markov assumptions, the estimates produced by Gibbs sampling should be very similar to those from Theil-Goldberger estimation of these models.

It makes little sense to implement these models with a homoscedastic prior represented by a large value for the hyperparameter r, so a default value of $r = 4$ is set indicating heteroscedasticity or outliers.

The input format for these functions rely on a structure variable 'prior' for inputting prior hyperparameters. For example, the documentation for the function **bvar_g** is shown below, where information regarding the results structure returned by the function was eliminated to save space.

```
 PURPOSE: Gibbs sampling estimates for Bayesian vector
          autoregressive model using Minnesota-type prior
          y = A(L) Y + X B + E, E = N(0,sige*V),
          V = diag(v1,v2,...vn), r/vi = ID chi(r)/r, r = Gamma(m,k)
          c = R A(L) + U, U = N(0,Z), Minnesota prior
          a diffuse prior is used for B associated with deterministic
          variables
---------------------------------------------------
 USAGE:  result = bvar_g(y,nlag,prior,ndraw,nomit,x)
 where:    y    = an (nobs x neqs) matrix of y-vectors
           nlag = the lag length
          ndraw = # of draws
          nomit = # of initial draws omitted for burn-in
          prior = a structure variable
               prior.tight,  Litterman's tightness hyperparameter
               prior.weight, Litterman's weight (matrix or scalar)
               prior.decay,  Litterman's lag decay = lag^(-decay)
               prior.rval, r prior hyperparameter, default=4
               prior.m,    informative Gamma(m,k) prior on r
               prior.k,    informative Gamma(m,k) prior on r
          x     = an optional (nobs x nx) matrix of variables
 NOTE:  constant vector automatically included
---------------------------------------------------
```

The output from these functions can be printed using the wrapper function **prt** which calls a function **prt_varg** that does the actual working of printing output results.

In addition, Gibbs forecasting functions **bvarf_g, rvarf_g, becmf_g, recmf_g** analogous to the forecasting functions with names corresponding to those described in Chapter 5 are in the *vector autoregressive function library*.

Probit and tobit models can be estimated using Gibbs sampling, a subject covered in the next chapter. Functions **probit_g** and **tobit_g** implement this method for estimating these models and allow for heteroscedastic errors using the chi-squared prior described in Section 6.3.

A Bayesian model averaging procedure set forth in Raftery, Madigan and Hoeting (1997) is implemented in the function **bma_g**. Leamer (1983) suggested that a Bayesian solution to model specification uncertainty is to average over all possible models using the posterior probabilities of the models as weights. Raftery, Madigan and Hoeting (1997) devise a way to implement this approach using a Metropolis sampling scheme to systematically move through model space, producing an MCMC sample from the space of all possible models.

A key insight they exploit is that the Bayes factor or posterior odds ratios from two alternative model specifications can be used to construct a Markov chain that traverses model neighborhoods. They argue that this systematic movement of the Markov Chain via a Hastings step generates a stochastic process that moves through model space. Under certain regularity conditions the average of the parameters from the sampled models converges to the model uncertainty solution proposed by Leamer (1983).

As an illustration, consider the following example program that generates a dependent variable y based on a constant term and variables x_1, x_2 and two dummy variables x_5, x_6. Six variables are input to the **bma_g** function and 5,000 draws are carried out. The example program is:

```
% ----- Example 6.9 Bayesian model averaging with the bma_g() function
nobs = 200;        % true model based on variables iota,x1,x2,x5,x6
vin = [1 2];       % variables in the model
vout = [3 4];      % variables not in the model
nv1 = length(vin); nv2 = length(vout);
nvar = nv1+nv2;    % total # of variables
x1 = randn(nobs,nvar);
xdum1 = zeros(nobs,1); xdum1(150:nobs,1) = 1.0;
xdum2 = zeros(nobs,1); xdum2(125:150,1) = 1.0;
x2 = [xdum1 xdum2];
x1t = x1(:,vin);  x2t = x2(:,1:2);
xtrue = [x1t x2t]; % true model based on variables iota,x1,x2,x5,x6
[junk nvart] = size(xtrue);
btrue = ones(nvart,1); btrue(3,1) = 5.0;
y = 10*ones(nobs,1) + xtrue*btrue + 0.5*randn(nobs,1);
ndraw = 5000;
result = bma_g(ndraw,y,x1,x2);
prt(result);
```

The printed output presents models with posterior probabilities greater than 1 percent and the explanatory variables associated with these models in the format shown below. In addition, estimates and $t-$statistics based on a posterior probability weighted average over all unique models found during the MCMC sampling are produced and printed.

```
Bayesian Model Averaging Estimates
R-squared      =  0.961
sigma^2        =  0.248
Nobs           =    200
Nvars          =      6
```

```
# of draws     =   5000
nu,lam,phi     =  4.000, 0.250, 3.000
# of models    =     20
time(seconds)  =   30.8
***************************************************************
Model averaging information
Model      v1    v2    v3    v4    v5    v6   Prob Visit
model 1     1     1     0     1     1     1  1.624   479
model 2     1     1     1     0     1     1 33.485   425
model 3     1     1     0     0     1     1 64.823   212

***************************************************************
Variable      Coefficient      t-statistic    t-probability
const           10.069325       222.741085         0.000000
v1               0.947768        25.995923         0.000000
v2               0.996021        26.755817         0.000000
v3              -0.026374        -0.734773         0.463338
v4               0.000018         0.000462         0.999632
v5               4.785116        58.007254         0.000000
v6               1.131978        10.349354         0.000000
```

From the example, we see that the correct model is found and assigned a posterior probability of 64.8%. The printout also shows how many times the MCMC sampler 'visited' each particular model. The remaining 4,000 MCMC draws spent time visiting other models with posterior probabilities less than one percent. All unique models visited are returned in the results structure variable by the **bma_g** function so you can examine them. It is often the case that even when the procedure does not find the true model, the estimates averaged over all models are still quite close to truth. The true model is frequently among the high posterior probability models.

This function was created from Splus code provided by Raftery, Madigan and Hoeting available on the internet. A few points to note about using the function **bma_g**. The function assigns a diffuse prior based on data vectors y and explanatory matrices X that are standardized inside the function so the default prior should work well in most applied settings. Example 6.8 above relies on these default prior settings, but they can be changed as an input option. Priors for dichotomous or polychotomous variables are treated differently from continuous variables, requiring that these two types of variables be entered separately as in Example 6.9. The documentation for the function is:

```
 PURPOSE: Bayes model averaging estimates of Raftery, Madigan and Hoeting
-----------------------------------------------------------------
 USAGE:    result = bma_g(ndraw,y,x1,x2,prior)
       or: result = bma_g(ndraw,y,x1)
```

```
 where:  ndraw = # of draws to carry out
             y = dependent variable vector (nobs x 1)
            x1 = continuous explanatory variables (nobs x k1)
            x2 = (optional) dummy variables (nobs x k2)
         prior = (optional) structure variable with prior information
     prior.nu  = nu hyperparameter  (see NOTES)
     prior.lam = lam hyperparameter (see NOTES)
     prior.phi = phi hyperparameter (see NOTES)
-----------------------------------------------------------------
 RETURNS: a structure results:
     results.meth  = 'bma'
     results.nmod  = # of models visited during sampling
     results.beta  = bhat averaged over all models
     results.tstat = t-statistics averaged over all models
     results.prob  = posterior prob of each model (nmod x 1)
     results.model = indicator variables for each model (nmod x k1+k2)
     results.yhat  = yhat averaged over all models
     results.resid = residuals based on yhat averaged over models
     results.sige  = averaged over all models
     results.rsqr  = rsquared based on yhat averaged over models
     results.nobs  = nobs
     results.nvar  = nvars = k1+k2
     results.k1    = k1, # of continuous explanatory variables
     results.k2    = k2, # of dichotomous variables
     results.y     = y data vector
     results.visit = visits to each model during sampling (nmod x 1)
     results.time  = time taken for MCMC sampling
     results.ndraw = # of MCMC sampling draws
     results.nu    = prior hyperparameter
     results.phi   = prior hyperparameter
     results.lam   = prior hyperparameter
--------------------------------------------------
 NOTES: prior is: B = N(m,sig*V), nu*lam/sig = chi(nu)
        m = (b0,0,...,0), b0 = ols intercept estimate
        V = diag[var(y), phi^2/var(x1), phi^2/var(x2) ...]
        defaults: nu=4, lam = 0.25, phi=3
--------------------------------------------------
 SEE ALSO: prt(results), plt(results)
---------------------------------------------------
 REFERENCES: Raftery, Madigan and Hoeting (1997) 'Bayesian model averaging
 for linear regression models', JASA 92, pp. 179-191
-----------------------------------------------------------------
```

As an illustration of how this function might be useful, consider a data set on ninth grade proficiency scores for 610 Ohio school districts. A host of 15 potential explanatory variables were explored in an attempt to explain variation in proficiency scores across the school districts. The **bma_g** function explored 503 distinct models in 1,000 MCMC draws. From the results printed below, we see that 30 of the 503 distinct models exhibited posterior probabilities exceeding 1 percent.

Certain variables such as student attendance rates (attend), median income (medinc), teachers average salary (tpay), a large city dummy variable (bcity) and a dummy variable for the northwest region of the state (northw) appeared in all of the 30 high posterior probability models. Other variables such as small city dummy variable (scity) as well as regional dummy variables (southe, northe, northc) never appeared in the high posterior probability models.

Perhaps most interesting are the increased posterior probabilities associated with the class size variable (csize) entering models 21 to 30 and the welfare (welf) variable entering models 23 to 30. Also of interest is that expenditures per pupil (expnd) enters and exits the 30 highest posterior probability models. The may explain why economists disagree about the impact of resources on student performance. Different model specifications will find this variable to be significant or insignificant depending on the other variables entered in the model.

```
Bayesian Model Averaging Estimates
Dependent Variable =         proficiency scores
R-squared      =  0.554
sigma^2        = 107.795
Nobs           =    610
Nvars          =     15
# of draws     =   1000
nu,lam,phi     =  4.000, 0.250, 3.000
# of models    =    503
time(seconds)  = 10004.5
***************************************************************
Model averaging information
Model      unemp  nwhite  medinc    welf   expnd   csize    tpay  attend
model 1        1       1       1       0       1       0       1       1
model 2        1       1       1       1       1       0       1       1
model 3        1       1       1       1       1       0       1       1
model 4        0       1       1       1       1       0       1       1
model 5        0       1       1       0       1       0       1       1
model 6        1       1       1       0       1       0       1       1
model 7        1       1       1       1       0       0       1       1
model 8        0       1       1       1       1       0       1       1
model 9        1       1       1       1       0       0       1       1
model 10       0       1       1       0       0       0       1       1
model 11       1       1       1       0       0       0       1       1
model 12       0       1       1       1       0       0       1       1
model 13       0       1       1       0       1       0       1       1
model 14       1       1       1       0       1       1       1       1
```

model 15	0	1	1	0	1	1	1	1
model 16	0	1	1	0	0	1	1	1
model 17	0	1	1	1	0	0	1	1
model 18	1	1	1	0	1	1	1	1
model 19	1	1	1	0	0	1	1	1
model 20	0	1	1	0	0	0	1	1
model 21	0	1	1	0	1	1	1	1
model 22	0	1	1	0	0	1	1	1
model 23	1	1	1	1	1	1	1	1
model 24	1	1	1	1	0	1	1	1
model 25	0	1	1	1	1	1	1	1
model 26	1	1	1	1	1	1	1	1
model 27	1	1	1	1	0	1	1	1
model 28	0	1	1	1	0	1	1	1
model 29	0	1	1	1	1	1	1	1
model 30	0	1	1	1	0	1	1	1

Model	bcity	scity	subur	southe	northc	northe	northw	Prob	Visit
model 1	1	0	1	0	0	0	1	1.415	2
model 2	1	0	1	0	0	0	1	1.466	3
model 3	1	0	1	0	0	0	1	1.570	2
model 4	1	0	1	0	0	0	1	1.620	3
model 5	1	0	1	0	0	0	1	1.622	1
model 6	1	0	1	0	0	0	1	1.643	4
model 7	1	0	1	0	0	0	1	1.794	2
model 8	1	0	1	0	0	0	1	1.900	2
model 9	1	0	1	0	0	0	1	1.945	1
model 10	1	0	1	0	0	0	1	1.971	1
model 11	1	0	1	0	0	0	1	1.997	2
model 12	1	0	1	0	0	0	1	2.061	3
model 13	1	0	1	0	0	0	1	2.201	2
model 14	1	0	1	0	0	0	1	2.338	1
model 15	1	0	1	0	0	0	1	2.358	1
model 16	1	0	1	0	0	0	1	2.519	2
model 17	1	0	1	0	0	0	1	2.606	2
model 18	1	0	1	0	0	0	1	2.757	5
model 19	1	0	1	0	0	0	1	2.938	5
model 20	1	0	1	0	0	0	1	3.119	4
model 21	1	0	1	0	0	0	1	3.179	3
model 22	1	0	1	0	0	0	1	3.329	4
model 23	1	0	1	0	0	0	1	3.781	2
model 24	1	0	1	0	0	0	1	4.247	6
model 25	1	0	1	0	0	0	1	4.571	4
model 26	1	0	1	0	0	0	1	4.612	7
model 27	1	0	1	0	0	0	1	5.083	4
model 28	1	0	1	0	0	0	1	5.147	5
model 29	1	0	1	0	0	0	1	7.420	7
model 30	1	0	1	0	0	0	1	8.172	5

```
***************************************************************
Variable      Coefficient      t-statistic    t-probability
const         -374.606723        -7.995893         0.000000
unemp            0.029118         0.088849         0.929231
nwhite          -0.287751        -6.413557         0.000000
medinc           0.000748         5.498948         0.000000
welf            -0.113650        -1.286391         0.198795
expnd           -0.000138        -0.352608         0.724504
csize           -0.287077        -1.475582         0.140572
tpay             0.000478         3.661464         0.000273
attend           4.358154         9.260027         0.000000
bcity           17.116312         3.692185         0.000242
scity           -0.000026        -0.000009         0.999993
subur            1.935253         1.543629         0.123197
southe           0.002544         0.001778         0.998582
northc           0.000012         0.000008         0.999994
northe           0.000123         0.000109         0.999913
northw           5.651431         4.328001         0.000018
```

A set of functions that implement Gibbs sampling estimation of Bayesian spatial autoregressive models are also included in the *spatial econometrics function library*. The function **sar_g** estimates a heteroscedastic spatial autoregressive model, **sart_g** estimates a tobit version of the spatial autoregressive model and **sarp_g** implements the probit version of the spatial autoregressive model. LeSage (1997, 1998) describes these models and their Gibbs sampling estimation.

6.7 Chapter summary

An introduction to Bayesian methods for exploring posterior distributions via MCMC and Gibbs sampling was provided along with illustrations of how these estimation methods can be encapsulated in MATLAB functions. Although the functions presented here make these methods easy to implement, diagnostics for convergence are important when using this approach to estimation. A set of MATLAB functions that address convergence by providing a battery of diagnostic tests for convergence was also described. These functions can be used to process the output from the MCMC estimation functions contained in the results structure variables returned by the estimation functions.

One drawback to the design approach taken here is that a large amount of RAM memory is needed to handle problems where say 10,000 draws are carried out for an estimation problem involving 10 or 20 parameters. Keep in mind that you always have the option of carrying out a smaller number of draws and writing the output to a file for storage using the **mprint** function. Additional sequences of draws can then be generated and placed in files for storage until a sufficient sample of draws has been accumulated. Most of the MCMC estimation functions allow the user to input starting values which can also be useful in this circumstance. A series of draws based on alternative starting values is another approach that is often used to diagnose convergence of the sampler.

Chapter 6 Appendix

The Gibbs convergence diagnostic functions are in a subdirectory **gibbs**.

```
Gibbs sampling convergence diagnostics library functions

--------------- convergence testing functions -----------------

apm       - Geweke's chi-squared test
coda      - convergence diagnostics
momentg   - Geweke's NSE, RNE
raftery   - Raftery and Lewis program Gibbsit for convergence

--------------- demonstration programs -----------------

apm_d     - demonstrates apm
coda_d    - demonstrates coda
momentg_d - demonstrates momentg
raftery_d - demonstrates raftery

--------------- support functions -----------------

prt_coda  - prints coda, raftery, momentg, apm output  (use prt)
empquant  - These were converted from:
indtest   - Rafferty and Lewis FORTRAN program.
mcest     - These function names follow the FORTRAN subroutines
mctest    -
ppnd      -
thin      -
```

The Gibbs regression estimation functions discussed are in the **regress** subdirectory.

```
 Gibbs sampling regression functions

------- Gibbs sampling regression functions -----------

 ar_g.m      - Bayesian autoregressive model  (homo/heteroscedastic)
```

```
bma_g.m     - Bayesian model averaging
ols_g.m     - Bayesian linear model (homo/heteroscedastic)
probit_g.m  - Bayesian probit model
tobit_g.m   - Bayesian tobit model

------- demonstration programs -----------

ar_gd.m     - demonstration of ar_g
bma_gd.m    - demonstrates Bayesian model averaging
ols_gd.m    - demonstration of ols_g
probit_gd.m - demonstrates Bayesian probit model
tobit_gd.m  - demonstrates Bayesian tobit model

------- support functions -----------

bmapost.m   - used by bma_g
find_new.m  - used by bma_g
prt_gibbs   - prints results for bma_g,ar_g,ols_g,tobit_g,probit_g
sample.m    - used by bma_g
```

The Gibbs sampling spatial econometric estimation functions discussed are in a subdirectory **spatial**.

```
spatial econometrics function library

------- spatial regression program functions -----------

casetti    - Casetti's spatial expansion model
darp       - Casetti's darp model
far        - 1st order spatial AR model    - y = pWy + e
far_g      - Gibbs sampling Bayesian far model
gwr        - geographically weighted regression
gwr_logit  - logit version of gwr
gwr_probit - probit version of gwr
bgwr       - Bayesian geographically weighted regression
bgwrv      - robust geographically weighted regression
lmerror    - LM error statistic for regression model
lmsar      - LM error statistic for sar model
lratios    - Likelihood ratio statistic for regression models
moran      - Moran's I-statistic
normxy     - isotropic normalization of x-y coordinates
normw      - normalizes a spatial weight matrix
sac        - spatial model  - y = p*W1*y + X*b + u, u = c*W2*u + e
sac_g      - Gibbs sampling Bayesian sac model
sacp_g     - Gibbs sampling Bayesian sac probit model
sact_g     - Gibbs sampling Bayesian sac tobit model
sar        - spatial autoregressive model  - y = p*W*y + X*b + e
sar_g      - Gibbs sampling Bayesian sar model
sarp_g     - Gibbs sampling Bayesian sar probit model
```

```
sart_g     - Gibbs sampling Bayesian sar tobit model
sem        - spatial error model  - y = X*b +u, u=c*W + e
sem_g      - Gibbs sampling Bayesian spatial error model
semp_g     - Gibbs sampling Bayesian spatial error probit model
semt_g     - Gibbs sampling Bayesian spatial error tobit model
semo       - spatial error model (optimization solution)
sdm        - spatial Durbin model  y = a + X*b1 + W*X*b2 + e
sdm_g      - Gibbs sampling Bayesian sdm model
sdmp_g     - Gibbs sampling Bayesian sdm probit model
sdmt_g     - Gibbs sampling Bayesian sdm tobit model
slag       - creates spatial lags
walds      - Wald test for regression models
xy2cont    - constructs a contiguity matrix from x-y coordinates

------- demonstration programs  -----------

casetti_d  - Casetti model  demo
darp_d     - Casetti darp demo
darp_d2    - darp for all data observations
far_d      - demonstrates far using a small data set
far_d2     - demonstrates far using a large data set
far_gd     - far Gibbs sampling with small data set
far_gd2    - far Gibbs sampling with large data set
gwr_d      - geographically weighted regression demo
gwr_d2     - GWR demo with Harrison-Rubinfeld Boston data
bgwr_d     - demo of Bayesian GWR
bgwr_d2    - BGWR demo with Harrison-Rubinfeld Boston data
lmerror_d  - lmerror demonstration
lmsar_d    - lmsar demonstration
lratios_d  - likelihood ratio demonstration
moran_d    - moran demonstration
sac_d      - sac model demo
sac_d2     - sac model demonstration large data set
sac_gd     - sac Gibbs sampling demo
sac_gd2    - sac Gibbs demo with large data set
sacp_gd    - sac Gibbs probit demo
sact_gd    - sac Gibbs tobit demo
sact_gd2   - sac tobit right-censoring demo
sar_d      - sar model demonstration
sar_d2     - sar model demonstration large data set
sar_gd     - sar Gibbs sampling demo
sar_gd2    - sar Gibbs demo with large data set
sarp_gd    - sar probit Gibbs sampling demo
sart_gd    - sar tobit model Gibbs sampling demo
sart_gd2   - sar tobit right-censoring demo
sdm_d      - sdm model demonstration
sdm_d2     - sdm model demonstration large data set
sdm_gd     - sdm Gibbs sampling demo
sdm_gd2    - sdm Gibbs demo with large data set
```

```
sdmp_g     - sdm Gibbs probit demo
sdmt_g     - sdm Gibbs tobit demo
sem_d      - sem model demonstration
sem_d2     - sem model demonstration large data set
sem_gd     - sem Gibbs sampling demo
sem_gd2    - sem Gibbs demo with large data set
semo_d     - semo function demonstration
semo_d2    - semo demo with large data set
semp_gd    - sem Gibbs probit demo
semt_gd    - sem Gibbs tobit demo
semt_gd2   - sem tobit right-censoring demo
slag_d     - demo of slag function
walds_d    - Wald test demonstration
xy2cont_d  - xy2cont demo

------- support functions  -----------

anselin.dat- Anselin (1988) Columbus crime data
boston.dat - Harrison-Rubinfeld Boston data set
latit.dat  - latittude for HR data
longi.dat  - longitude for HR data
c_far      - used by far_g
c_sem      - used by sem_g
c_sar      - used by sar_g
c_sdm      - used by sdm_g
c_sac      - used by sac_g
darp_lik1  - used by darp
darp_lik2  - used by darp
elect.dat  - Pace and Barry 3,107 obs data set
ford.dat   - Pace and Barry 1st order contiguity matrix
f_far      - far model likelihood (concentrated)
f_sac      - sac model likelihood (concentrated)
f_sar      - sar model likelihood (concentrated)
f_sem      - sem model likelihood (concentrated)
f_sdm      - sdm model likelihood (concentrated)
f2_far     - far model likelihood
f2_sac     - sac model likelihood
f2_sar     - sar model likelihood
f2_sem     - sem model likelihood
f3_sem     - semo model likelihood
f2_sdm     - sdm model likelihood
prt_gwr    - prints gwr_reg results structure
prt_spat   - prints results from spatial models
scoref     - used by gwr
wmat.dat   - Anselin (1988) 1st order contiguity matrix
```

The Gibbs sampling VAR/BVAR estimation functions discussed are in the **var_bvar** subdirectory.

```
------- VAR/BVAR Gibbs sampling program functions  -----------

 becm_g      - Gibbs sampling BECM estimates
 becmf_g     - Gibbs sampling BECM forecasts
 bvar_g      - Gibbs sampling BVAR estimates
 bvarf_g     - Gibbs sampling BVAR forecasts
 recm_g      - Gibbs sampling random-walk averaging estimates
 recmf_g     - Gibbs sampling random-walk averaging forecasts
 rvar_g      - Gibbs sampling RVAR estimates
 rvarf_g     - Gibbs sampling RVAR forecasts

------- Gibbs sampling VAR/BVAR demonstrations  -----------

 becm_g      - Gibbs sampling BECM estimates demo
 becmf_gd    - Gibbs sampling BECM forecast demo
 bvar_gd     - Gibbs sampling BVAR demonstration
 bvarf_gd    - Gibbs sampling BVAR forecasts demo
 recm_gd     - Gibbs sampling RECM model demo
 recmf_gd    - Gibbs sampling RECM forecast demo
 rvar_gd     - Gibbs sampling rvar model demo
 rvarf_gd    - Gibbs sampling rvar forecast demo

------- support functions  -----------

 prt_varg    - prints results of all Gibbs var/bvar models
 theil_g     - used for Gibbs sampling estimates and forecasts
```

Chapter 7

Limited Dependent Variable Models

The *regression function library* contains routines for estimating limited dependent variable logit, probit, and tobit models. In addition, there are Gibbs sampling functions to implement recently proposed MCMC methods for estimating these models proposed by Chib (1992) and Albert and Chib (1993).

These models arise when the dependent variable y in our regression model takes values 0, 1, 2, ... representing counts of some event or a coding system for qualitative outcomes. For example, $y = 0$ might represent a coding scheme indicating a lack of labor force participation by an individual in our sample, and $y = 1$ denotes participation in the labor force. As another example where the values taken on by y represent counts, we might have $y = 0, 1, 2, \ldots$ denoting the number of foreign direct investment projects in a given year for a sample of states in the U.S.

Regression analysis of these data usually are interpreted in the framework of a probability model that attempts to describe the $Prob(\text{event } i \text{ occurs}) = F(X\text{: parameters})$. If the outcomes represent two possibilities, $y = 0, 1$, the model is said to be binary, whereas models with more than two outcomes are referred to as multinomial or polychotomous.

Ordinary least-squares can be used to carry out a regression with a binary response variable $y = 0, 1$, but two problems arise. First, the errors are by construction heteroscedastic. This is because the actual $y = 0, 1$ minus the value $X\beta = -X\beta$ or $\iota - X\beta$. Note also that the heteroscedastic errors are a function of the parameter vector β. The second problem with least-squares is that the predicted values can take on values outside the (0,1) interval, which is problematic given a probability model framework. In this setting we would like to see:

$$lim_{X\beta \to +\infty} Prob(y = 1) = 1 \tag{7.1}$$

$$lim_{X\beta \to -\infty} Prob(y = 1) = 0 \tag{7.2}$$

Two distributions that have been traditionally used to produce this type of outcome (that ensures predicted values between zero and one) are the logisitic and normal distributions resulting in the logit model shown in (7.3) and probit model shown in (7.4), where Φ denotes the cumulative normal probability function.

$$Prob(y = 1) = e^{X\beta}/(1 + e^{X\beta}) \tag{7.3}$$

$$Prob(y = 1) = \Phi(X\beta) \tag{7.4}$$

The logistic distribution is similar to the normal except in the tails where it is fatter resembling a Student $t-$distribution. Green (1997) and others indicate that the logistic distribution resembles a $t-$distribution with seven degrees of freedom. Graphs of the cumulative logistic, normal and two $t-$distributions are shown in Figure 7.1. From the figure, it appears that the logistic distribution is somewhat different from the two $t-$distributions, one based on 7 degrees of freedom and the other based on 2 degrees of freedom.

Albert and Chib (1993) propose a $t-$distribution in a Bayesian setting where the degrees of freedom parameter becomes part of the estimation problem. Data sets that produce a posterior estimate indicating a small degrees of freedom parameter suggest the logit distribution is more appropriate, whereas estimates leading to a large degrees of freedom parameter indicate a probit distribution. (Keep in mind that the $t-$distribution with a large degrees of freedom parameter approaches the normal.) Given the lack of total similarity between the $t-$distributions and the logistic illustrated in Figure 7.1, we shouldn't take this reasoning too formally.

Geweke and Keane (1997) propose a mixture of normal distributions and demonstrate how to draw a posterior inference regarding the number of components in the mixture using posterior odds ratios. They demonstrate that in an applied multinomial setting, their mixture of normal distributions is often favored over the traditional logit, probit and Student t distributions.

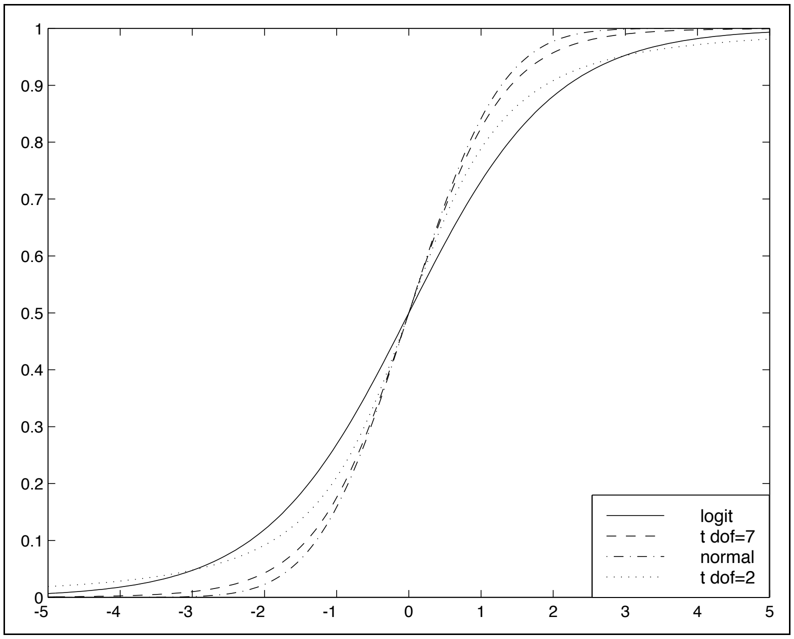

Figure 7.1: Cumulative distribution functions compared

Section 7.1 presents the logit and probit regression functions from the *regression function library* and Section 7.2 takes up Gibbs sampling estimation of these models. In Section 7.3 tobit regression is discussed and Section 7.4 turns attention to Gibbs sampling this model.

7.1 Logit and probit regressions

To illustrate the **logit** and **probit** functions we use a data set from Spector and Mazzeo (1980) that contains a binary dependent variable indicating improvement in students grades after exposure to a new teaching method for economics. The explanatory variables in the regression are: an intercept term, grade point average, a pre-test for understanding of college-level economics (TUCE) and a binary variable indicator whether the student was exposed to the new teaching method.

The example program simply loads the data and calls the functions **logit** and **probit** in addition to carrying out a least-squares regression for contrast.

```
% ----- Example 7.1 Logit and probit regression functions
```

```
load spector.dat; % data from Spector and Mazzeo, 1980
y = spector(:,1);
x = spector(:,2:5);
vnames = strvcat('grade','const','psi','tuce','gpa');
reso = ols(y,x);
prt(reso,vnames);
resl = logit(y,x);
prt(resl,vnames);
resp = probit(y,x);
prt(resp,vnames);
```

The resulting printouts are shown below. The **logit** function displays the usual coefficient estimates, $t-$statistics and marginal probabilities as well as measures of fit proposed by McFadden (1984) and Estrella (1998). The maximized value of the log-likelihood function is reported along with a log-likelihood ratio test statistic and marginal probability that all slopes in the model are zero. This is based on a comparison of the maximized log-likelihood (Lu) versus the log-likelihood for a restricted model (Lr) with only a constant term.

```
Ordinary Least-squares Estimates
Dependent Variable =           grade
R-squared      =    0.4159
Rbar-squared   =    0.3533
sigma^2        =    0.1506
Durbin-Watson  =    2.3464
Nobs, Nvars    =     32,     4
***************************************************************
Variable      Coefficient      t-statistic    t-probability
const           -1.498017        -2.859419         0.007929
psi              0.378555         2.720035         0.011088
tuce             0.010495         0.538685         0.594361
gpa              0.463852         2.864054         0.007841

Logit Maximum Likelihood Estimates
Dependent Variable =           grade
McFadden R-squared     =    0.3740
Estrella R-squared     =    0.4528
LR-ratio, 2*(Lu-Lr)    =   15.4042
LR p-value             =    0.0015
Log-Likelihood         =  -12.8896
# of iterations        =      7
Convergence criterion  =     6.976422e-08
Nobs, Nvars            =     32,     4
# of 0's, # of 1's     =     21,    11
***************************************************************
Variable      Coefficient      t-statistic    t-probability
```

```
const          -13.021347       -2.640538         0.013382
psi              2.378688        2.234424         0.033617
tuce             0.095158        0.672235         0.506944
gpa              2.826113        2.237723         0.033376

Probit Maximum Likelihood Estimates
Dependent Variable =           grade
McFadden R-squared     =    0.3775
Estrella R-squared     =    0.4566
LR-ratio, 2*(Lu-Lr)    =   15.5459
LR p-value             =    0.0014
Log-Likelihood         =  -12.8188
# of iterations        =      7
Convergence criterion  =    2.1719814e-10
Nobs, Nvars            =     32,     4
# of 0's, # of 1's     =     21,    11
***************************************************************
Variable      Coefficient      t-statistic    t-probability
const           -7.452320        -2.931131         0.006656
psi              1.426332         2.397045         0.023445
tuce             0.051729         0.616626         0.542463
gpa              1.625810         2.343063         0.026459
```

Solving the logit model for parameter estimates involves non-linear optimization of the likelihood function. Fortunately, the likelihood function takes a form that is globally concave allowing us to use Newton's method. This approach would usually converge in just a few iterations unless we encounter an ill-conditioned data set. (We don't actually use Newton's method, an issue discussed below.)

Since this is our first encounter with optimization, an examination of the **logit** function is instructive. The iterative 'while-loop' that optimizes the likelihood function is shown below:

```
iter = 1;
while (iter < maxit) & (crit > tol)
tmp = (i+exp(-x*b)); pdf = exp(-x*b)./(tmp.*tmp); cdf = i./(i+exp(-x*b));
tmp = find(cdf <=0); [n1 n2] = size(tmp);
if n1 ~= 0; cdf(tmp) = 0.00001; end;
tmp = find(cdf >= 1); [n1 n2] = size(tmp);
if n1 ~= 0; cdf(tmp) = 0.99999; end;
% gradient vector for logit, see page 883 Green, 1997
term1 = y.*(pdf./cdf); term2 = (i-y).*(pdf./(i-cdf));
for kk=1:k;
tmp1(:,kk) = term1.*x(:,kk); tmp2(:,kk) = term2.*x(:,kk);
end;
g = tmp1-tmp2; gs = (sum(g))'; delta = exp(x*b)./(i+exp(x*b));
% Hessian for logit, see page 883 Green, 1997
```

```
H = zeros(k,k);
for ii=1:t;
xp = x(ii,:)'; H = H - delta(ii,1)*(1-delta(ii,1))*(xp*x(ii,:));
end;
db = -inv(H)*gs;
% stepsize determination
s = 2; term1 = 0; term2 = 1;
while term2 > term1
s = s/2;
term1 = lo_like(b+s*db,y,x);
term2 = lo_like(b+s*db/2,y,x);
end;
bn = b + s*db; crit = abs(max(max(db)));
b = bn; iter = iter + 1;
end; % end of while
```

The MATLAB variables 'maxit' and 'tol' can be set by the user as an input option, but there are default values supplied by the function. The documentation for the function shown below makes this clear.

```
 PURPOSE: computes logistic regression estimates
---------------------------------------------------
 USAGE: results = logit(y,x,maxit,tol)
 where: y = dependent variable vector (nobs x 1)
        x = independent variables matrix (nobs x nvar)
    maxit = optional (default=100)
      tol = optional convergence (default=1e-6)
---------------------------------------------------
 RETURNS: a structure
        result.meth   = 'logit'
        result.beta   = bhat
        result.tstat  = t-stats
        result.yhat   = yhat
        result.resid  = residuals
        result.sige   = e'*e/(n-k)
        result.r2mf   = McFadden pseudo-R^2
        result.rsqr   = Estrella R^2
        result.lratio = LR-ratio test against intercept model
        result.lik    = unrestricted Likelihood
        result.cnvg   = convergence criterion, max(max(-inv(H)*g))
        result.iter   = # of iterations
        result.nobs   = nobs
        result.nvar   = nvars
        result.zip    = # of 0's
        result.one    = # of 1's
        result.y      = y data vector
 --------------------------------------------------
 SEE ALSO: prt(results), probit(), tobit()
---------------------------------------------------
```

The 'while-loop' first evaluates the logistic probability density and cumulative density function at the initial least-squares values set for the parameter vector β, returned by the **ols** function. Values for the cdf outside the (0,1) bounds are set to 0.00001 and 0.99999 to avoid overflow and underflow computational problems by the following code.

```
tmp = find(cdf <=0); [n1 n2] = size(tmp);
if n1 ~= 0; cdf(tmp) = 0.00001; end;
tmp = find(cdf >= 1); [n1 n2] = size(tmp);
if n1 ~= 0; cdf(tmp) = 0.99999; end;
```

Next, the analytical gradient and hessian functions presented in Green (1997) are computed and used to determine a new vector of β values. Nonlinear optimization problems can be solved by iterative methods that begin from an initial value β_0. If β_0 is not the optimal value for β, a direction vector Δ_0, and step size λ_0 are computed to find a new value β_1 for the next iteration using:

$$\beta_1 = \beta_0 + \lambda_0 \Delta_0 \tag{7.5}$$

Gradient methods of optimization rely on a direction vector $\Delta = Wg$, where W is a positive definite matrix and g is the gradient of the function evaluated at the current value β_0, $\partial F(\beta_0)/\partial\beta_0$. Newton's method is based on a linear Taylor series expansion of the first order conditions: $\partial F(\beta_0)/\partial\beta_0 = 0$, which leads to $W = -H^{-1}$ and $\Delta = -H^{-1}g$. Note that Newton's method implicitly sets λ_0 to unity.

Our algorithm determines a stepsize variable for λ, which need not be unity, using the following code.

```
db = -inv(H)*gs;
% stepsize determination
s = 2;
term1 = 0; term2 = 1;
while term2 > term1
s = s/2;
term1 = lo_like(b+s*db,y,x);
term2 = lo_like(b+s*db/2,y,x);
end;
bn = b + s*db; crit = abs(max(max(db)));
b = bn; iter = iter + 1;
end; % end of while
```

After convergence, the **logit** function evaluates the analytical hessian at the maximum likelihood parameter values for the purpose of computing asymptotic variances, $t-$statistics, measures of fit, as well as the log-likelihood ratio tests reported in the printout.

In addition to these functions, a function **mlogit** implements maximum likelihood estimation for the multinomial logit model.

7.2 Gibbs sampling logit/probit models

Albert and Chib (1993) propose augmenting the data set in logit/probit models with a set of variables z that are drawn from an underlying continuous distribution such that $y_i = 1$ when $z_i > 0$ and $y_i = 0$ otherwise. They then point out that the conditional distribution of β given z takes the form of a simple normal distribution with a mean that can be easily computed. Furthermore, the conditional distribution of the augmenting variables z given β take the form of truncated normal distributions that are also easy to compute. This leads to a Gibbs sampling approach to estimation of logit/probit models.

Consider the case of logit/probit models where the observed y can be viewed as associated with a latent variable $z_i < 0$ which produces an observed variable $y_i = 0$ and $z_i \geq 0$ associated with $y_i = 1$. Albert and Chib (1993) show that the posterior distribution of z_i conditional on all other parameters in the model β, σ takes the form of a truncated normal distribution. This truncated normal distribution is constructed by truncating a $N[\tilde{y}_i, \sigma_i^2]$ distribution from the right by zero. If we let $\tilde{y}$ denote the predicted value for the ith row of z_i, and let σ_i^2 denote the variance of the prediction, the pdf of the latent variables z_i is:

$$f(z_i|\rho, \beta, \sigma) \sim \left\{ \begin{array}{ll} N(\tilde{y}_i, \sigma_i^2), & \text{truncated at the left by 0 if } y_i = 1 \\ N(\tilde{y}_i, \sigma_i^2), & \text{truncated at the right by 0 if } y_i = 0 \end{array} \right. \tag{7.6}$$

Because the probit model is unable to identify both β and σ_ε^2, we scale our problem to make σ_ε^2 equal to unity.

These expressions simply indicate that we can replace values of $y_i = 1$ with the sampled normals truncated at the left by 0 and values of $y_i = 0$ with sampled normals truncated at the right by 0.

As an intuitive view of why this works, consider the following MATLAB program that generates a vector of latent values z and then converts these values to zero or one, depending on whether they take on values greater or less than zero. The program carries out Gibbs sampling using the function **probit_g**. During the sampling, generated values for the latent variable z from each pass through the sampler are saved and returned in a structure variable results.ydraw. The program then plots the mean of these predictions versus the actual vector of latent z values, which is shown in Figure 7.2.

```
% ----- Example 7.2 Demonstrate how Albert-Chib latent variable works
nobs=100; nvar=4; % generate a probit data set
randn('seed',1010);
x = rand(nobs,nvar);
beta = ones(nvar,1);
evec = randn(nobs,1);
y = x*beta + evec;
ysave = y;   % save the generate y-values
 for i=1:nobs % convert to 0,1 values
        if y(i,1) >= 0, y(i,1) = 1;
        else,           y(i,1) = 0;
        end;
 end;
prior.beta = zeros(4,1);      % diffuse prior for beta
prior.bcov = eye(4)*10000;
ndraw = 1100; nomit = 100;
prior.rval = 100;             % probit prior for r-value
results = probit_g(y,x,ndraw,nomit,prior);
ymean = results.ymean; % recover the mean of draws
tt=1:nobs;
plot(tt,ysave,tt,ymean,'--');
xlabel('observations');
ylabel('actual y-values vs mean of y values drawn');
legend('acutal','draws');
```

Of course, in an applied problem we would not have access to the latent variables z_i generated in example 7.2, but this provides an illustration of the insight behind Albert and Chib's Gibbs sampling approach to estimating these models.

The documentation for the function **probit_g** which implements the Gibbs sampler for the logit/probit model is shown below with information regarding the results structure returned omitted to save space.

```
PURPOSE: Gibbs sampler for the Bayesian Probit model
         y = X B + E, E = N(0,V),
         V = diag(v1,v2,...vn), r/vi = ID chi(r)/r, r = Gamma(m,k)
         B = N(c,T)
---------------------------------------------------------------
USAGE: results =  probit_g(y,x,ndraw,nomit,prior,start)
where: y = nobs x 1 independent variable vector
        x = nobs x nvar explanatory variables matrix
      ndraw = # of draws
      nomit = # of initial draws omitted for burn-in
      prior = a structure for prior information input
```

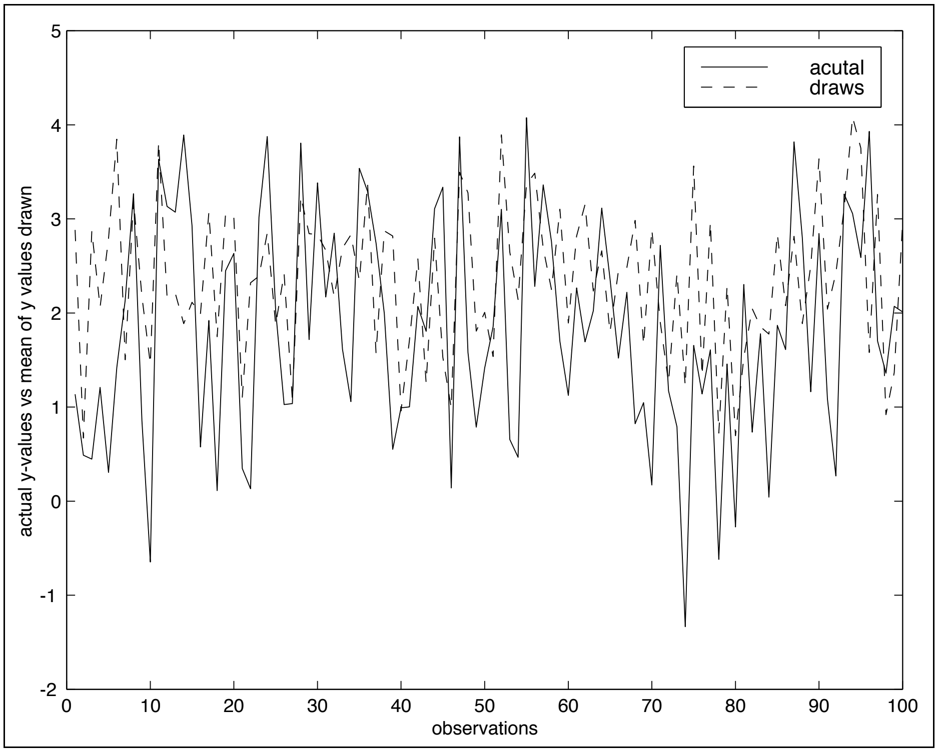

Figure 7.2: Actual y vs. mean of latent y-draws

```
            prior.beta, prior means for beta,   c above
            prior.bcov, prior beta covariance , T above
            prior.rval, r prior hyperparameter, default=4
            prior.m,    informative Gamma(m,k) prior on r
            prior.k,    informative Gamma(m,k) prior on r
     start = (optional) structure containing starting values:
            defaults: max likelihood beta, V= ones(n,1)
            start.b   = beta starting values (nvar x 1)
            start.V   = V starting values (n x 1)
---------------------------------------------------------------
 NOTE: use either improper prior.rval
       or informative Gamma prior.m, prior.k, not both of them
---------------------------------------------------------------
```

The first point to note is that the function allows a prior for the parameters β and implements the heteroscedastic model based on $t-$distributed errors discussed in Chapter 6. Albert and Chib (1993) point out that taking this approach allows one to produce a family of models that encompass both traditional logit and probit models. Recall that the logistic distribution is somewhat near the $t-$distribution with seven degrees of freedom and the normal distribution assumed by probit models can be represented as a $t-$distribution with a very large degrees of freedom parameter.

An implication of this is that the user can rely on a large prior hyperparameter value for $r = 100$ say, and a diffuse prior for β to produce estimates close to a traditional probit model. On the other hand, setting $r = 7$ or even $r = 3$ and relying on a diffuse prior for β should produce estimates close to those from a traditional logit regression. Green (1997) states that the issue of which distributional form should be used on applied econometric problems is unresolved. He further indicates that inferences from either logit or probit models are often the same.

Example 7.3 illustrates these ideas by comparing maximum likelihood logit estimates to those from Gibbs sampling with a hyperparameter value for $r = 7$.

```
% ----- Example 7.3 Gibbs vs. maximum likelihood logit
nobs=100; nvar=4; % generate data
x = randn(nobs,nvar)*2;  beta = ones(nvar,1);
beta(1,1) = -1; beta(2,1) = -1; evec = randn(nobs,1);
xb = x*beta + evec;
y = exp(xb)./(ones(nobs,1) + exp(xb)); % logit model generation
ysave = y;
for i=1:nobs;  % convert to 0,1 values
        if ysave(i,1) >= 0.5, y(i,1) = 1;
        else,                 y(i,1) = 0;
        end;
end;
prt(logit(y,x)); % print out maximum likelihood estimates
ndraw = 2100; nomit = 100;
prior.beta = zeros(4,1);  % diffuse prior means for beta
prior.bcov = eye(4)*10000;% diffuse prior variance for beta
prior.rval = 7;           % logit prior r-value
resp = probit_g(y,x,ndraw,nomit,prior); % Gibbs sampling
prt(resp);                % print Gibbs probit results
```

The results shown below indicate that the estimates, $t-$statistics, and fit from maximum likelihood logit and Gibbs sampling are quite similar as they should be.

```
Logit Maximum Likelihood Estimates
McFadden R-squared     =    0.7367
Estrella R-squared     =    0.8427
LR-ratio, 2*(Lu-Lr)    =  102.1035
LR p-value             =    0.0000
```

```
Log-Likelihood         =  -18.2430
# of iterations        =     10
Convergence criterion  =    6.5006301e-11
Nobs, Nvars            =    100,     4
# of 0's, # of 1's     =     51,    49
***************************************************************
Variable        Coefficient      t-statistic    t-probability
variable 1        -1.852287       -3.582471         0.000537
variable 2        -1.759636       -3.304998         0.001336
variable 3         2.024351        3.784205         0.000268
variable 4         1.849547        3.825844         0.000232

Bayesian Heteroscedastic Probit Model Gibbs Estimates
McFadden R^2    =    0.7256
Estrella R^2    =    0.8334
Nobs, Nvars     =    100,     4
# 0, 1 y-values =     51,    49
ndraws,nomit    =   2100,   100
time in secs    =   80.0993
r-value         =      7
***************************************************************
Variable           Prior Mean    Std Deviation
variable   1         0.000000       100.000000
variable   2         0.000000       100.000000
variable   3         0.000000       100.000000
variable   4         0.000000       100.000000
***************************************************************
      Posterior Estimates
Variable          Coefficient      t-statistic    t-probability
variable   1        -1.495815        -3.657663         0.000408
variable   2        -1.411028        -3.430553         0.000877
variable   3         1.640488         3.766352         0.000280
variable   4         1.482100         3.893146         0.000179
```

Example 7.4 illustrates that we can rely on the same function **probit_g** to produce probit estimates using a setting for the hyperparameter r of 100, which results in the normal probability model.

```
% ----- Example 7.4 Gibbs vs. maximum likelihood probit
nobs=100; nvar=4; % generate data
x = randn(nobs,nvar);  beta = ones(nvar,1);
beta(1,1) = -1; beta(2,1) = -1; evec = randn(nobs,1);
xb = x*beta + evec; y = norm_cdf(xb); % generate probit model
ysave = y;
for i=1:nobs
    if ysave(i,1) >= 0.5, y(i,1) = 1;
    else,                 y(i,1) = 0;
    end;
```

```
end;
prt(probit(y,x)); % print out maximum likelihood estimates
ndraw = 2100; nomit = 100;
prior.beta = zeros(4,1);  % diffuse prior means for beta
prior.bcov = eye(4)*10000;% diffuse prior variance for beta
prior.rval = 100;         % probit prior r-value
resp = probit_g(y,x,ndraw,nomit,prior); % Gibbs sampling
prt(resp);                % print Gibbs probit results
```

Again, the results shown below indicate that Gibbs sampling can be used to produce estimates similar to those from maximum likelihood estimation.

```
Probit Maximum Likelihood Estimates
McFadden R-squared     =    0.5786
Estrella R-squared     =    0.6981
LR-ratio, 2*(Lu-Lr)    =   80.1866
LR p-value             =    0.0000
Log-Likelihood         =  -29.2014
# of iterations        =      8
Convergence criterion  =     1.027844e-07
Nobs, Nvars            =    100,     4
# of 0's, # of 1's     =     51,    49
***************************************************************
Variable        Coefficient      t-statistic    t-probability
variable 1        -1.148952        -3.862861         0.000203
variable 2        -1.393497        -4.186982         0.000063
variable 3         1.062447         3.167247         0.002064
variable 4         1.573209         4.353392         0.000034

Bayesian Heteroscedastic Probit Model Gibbs Estimates
McFadden R^2    =    0.5766
Estrella R^2    =    0.6961
Nobs, Nvars     =    100,     4
# 0, 1 y-values =     51,    49
ndraws,nomit    =   2100,   100
time in secs    =  112.2768
r-value         =    100
***************************************************************
Variable           Prior Mean    Std Deviation
variable   1         0.000000       100.000000
variable   2         0.000000       100.000000
variable   3         0.000000       100.000000
variable   4         0.000000       100.000000
***************************************************************
      Posterior Estimates
Variable           Coefficient      t-statistic    t-probability
variable   1         -1.265604        -4.503577         0.000018
variable   2         -1.532119        -4.838973         0.000005
variable   3          1.191029         4.040334         0.000105
```

```
variable   4         1.763315         4.991809         0.000003
```

In addition to the ability to replicate maximum likelihood estimation results, the Gibbs sampling approach can produce robust estimates in the face of outliers and provide a set of variance estimates for every sample observation. The function **probit_g** returns the draws made for the v_i terms in the results structure variable and these can be averaged to produce a mean of the posterior distribution of these terms. Inferences about the presence of outliers at certain observations can be based on a graphical presentation of the posterior v_i estimates.

Example 7.5 illustrates these ideas by generating a data set that contains outliers at observations 50 and 75. Note that we need to use a prior $r-$value that is small, indicating our prior belief in outliers, consistent with a logit model.

```
% ----- Example 7.5 Heteroscedastic probit model
nobs=100; nvar=4; % generate data
x = randn(nobs,nvar);
x(50,:) = 20*abs(x(50,:)); % insert outliers
x(75,:) = 20*abs(x(75,:)); % at observations 50,75
beta = ones(nvar,1);  beta(1,1) = -1; beta(2,1) = -1;
evec = randn(nobs,1); xb = x*beta + evec;
y = exp(xb)./(ones(nobs,1) + exp(xb)); % logit model generation
ysave = y;
for i=1:nobs;  % convert to 0,1 values
    if ysave(i,1) >= 0.5,y(i,1) = 1;
    else,                y(i,1) = 0;
    end;
end;
prt(logit(y,x)); % print out maximum likelihood estimates
ndraw = 2100; nomit = 100;
prior.beta = zeros(4,1);  % diffuse prior means for beta
prior.bcov = eye(4)*10000;% diffuse prior variance for beta
prior.rval = 5;           % prior r-value for outliers
resp = probit_g(y,x,ndraw,nomit,prior); % Gibbs sampling
prt(resp);                % print Gibbs probit results
vmean = mean(resp.vdraw);
tt=1:nobs;
plot(tt,vmean);
ylabel('v_i estimates');
xlabel('observations');
```

The maximum likelihood as well as Gibbs estimates and a graph of the v_i estimates are shown below the code for example 7.5. The estimates for the variance scalars v_i clearly point to the outliers at observations 50 and

75. The Gibbs estimates that robustify for the outliers are closer to the true values of the parameters used to generate the data, as we would expect.

```
Logit Maximum Likelihood Estimates
McFadden R-squared     =    0.6299
Estrella R-squared     =    0.7443
LR-ratio, 2*(Lu-Lr)    =   86.4123
LR p-value             =    0.0000
Log-Likelihood         =  -25.3868
# of iterations        =      9
Convergence criterion  =    2.9283713e-07
Nobs, Nvars            =    100,     4
# of 0's, # of 1's   =     44,    56
***************************************************************
Variable        Coefficient      t-statistic    t-probability
variable 1        -1.579212        -3.587537         0.000528
variable 2        -2.488294        -3.743234         0.000310
variable 3         2.633014         4.038189         0.000108
variable 4         1.958486         3.632929         0.000452

Bayesian Heteroscedastic Probit Model Gibbs Estimates
McFadden R^2    =    0.6278
Estrella R^2    =    0.7423
Nobs, Nvars     =    100,     4
# 0, 1 y-values =     44,    56
ndraws,nomit    =   2100,   100
time in secs    =   71.3973
r-value         =      5
***************************************************************
Variable           Prior Mean    Std Deviation
variable   1         0.000000       100.000000
variable   2         0.000000       100.000000
variable   3         0.000000       100.000000
variable   4         0.000000       100.000000
***************************************************************
      Posterior Estimates
Variable          Coefficient      t-statistic    t-probability
variable   1        -0.953621        -3.256567         0.001540
variable   2        -1.437799        -3.896968         0.000176
variable   3         1.715255         3.906306         0.000170
variable   4         1.304000         4.119908         0.000078
```

7.2.1 The probit_g function

The function that implements Gibbs sampling estimation of the probit model relies on two functions, **nmrt_rnd** and **nrlt_rnd** from the *distributions function library* discussed in Chapter 9. These carry out the draws from the

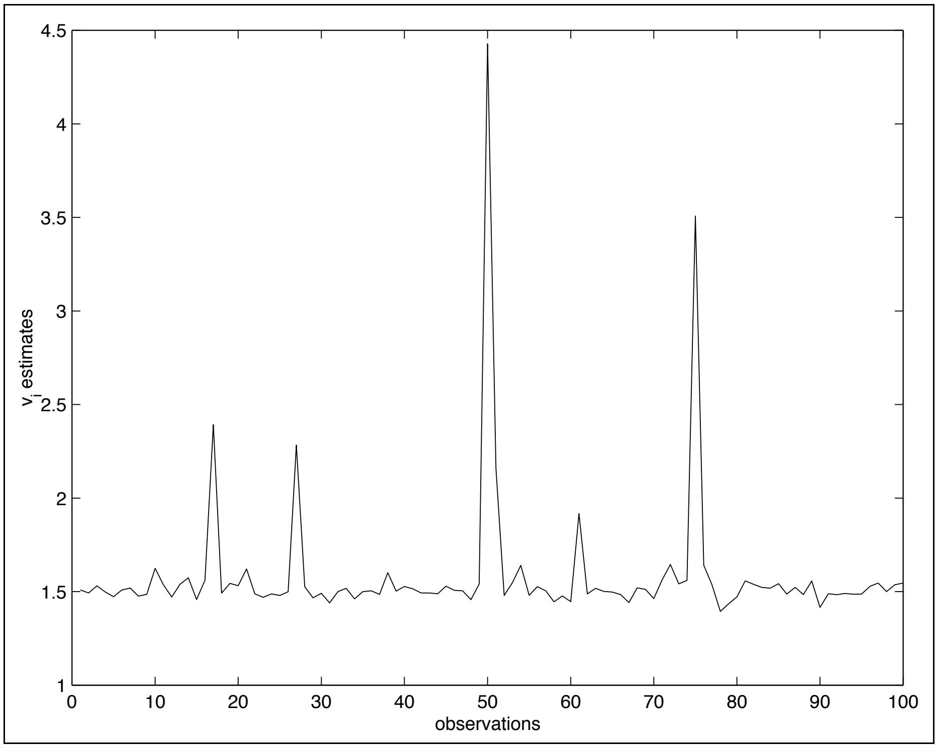

Figure 7.3: Posterior mean of v_i draws with outliers

right- and left-truncated normal distributions and are based on an approach set forth in Geweke (1991). The code for the sampling loop is shown below:

```
hwait = waitbar(0,'Gibbs sampling ...');
t0 = clock;
for i=1:ndraw; %  Start the sampling
         % update z
      lp=xstar*bhat;
          bb=.5*(1+erf(-lp/sqrt(2)));
      tt=(bb.*(1-yin)+(1-bb).*yin).*rand(n,1)+bb.*yin;
          y=sqrt(2)*erfinv(2*tt-1) + lp;
          % update beta
          xpxi = inv(xstar'*xstar + Q);
          xpy = xstar'*ystar + Qpc;
          bhat = xpxi*xpy;
          bhat = norm_rnd(xpxi) + bhat;
          % update V
          e = y - x*bhat;
          chiv = chis_rnd(n,rval+1);
```

```
          vi = ((e.*e) + in*rval)./chiv;
          V = in./vi;
          if mm ~= 0
           rval = gamm_rnd(1,1,mm,kk);  % update rval
          end;
if i > nomit % if we are past burn-in, save the draws
    bsave(i-nomit,:) = bhat';
    ymean = ymean + lp;
    vmean = vmean + vi;
    yhat = yhat + 1-bb;
    if mm~= 0
        rsave(i-nomit,1) = rval;
    end;
end; % end of if i > nomit
waitbar(i/ndraw);
end;     % End the sampling
gtime = etime(clock,t0);
close(hwait);
```

A few points to note about the sampling routine. First, we allow the user to rely on an improper prior for the hyperparameter r, or an informative Gamma(m,k) prior. In the event of an informative Gamma prior, the MATLAB variable $mm \neq 0$ and we update the hyperparameter r using a random gamma draw produced by the function **gamm_rnd** based on the informative prior parameters stored in mm, kk. Note also that we save the mean of latent variable draws for y and the mean of the non-constant variance parameters v_i and return these in the result structure variable.

The code that carries out the left-truncated normal random draw is:

```
          % update z
      lp=xstar*bhat;
          bb=.5*(1+erf(-lp/sqrt(2)));
      tt=(bb.*(1-yin)+(1-bb).*yin).*rand(n,1)+bb.*yin;
          y=sqrt(2)*erfinv(2*tt-1) + lp;
```

An inefficient alternative procedure for producing the truncated normal draws is that suggested by Albert and Chib (1993). They propose making random normal draws with rejection of draws that do not meet the truncation restriction.

7.3 Tobit models

The censored regression or tobit model involves situations where truncation occurs because the sample data are drawn from a subset of a larger population of interest. For example, a study of income that draws a sample based on incomes above or below a poverty level of income would provide invalid inferences regarding the population as a whole.

The general formulation of a tobit regression model relies on an index function $y_i^\star$ so that:

$$y_i^\star = x_i\beta + \varepsilon_i \quad (7.7)$$

$$y_i = 0 \quad \text{if} \quad y_i^\star \leq 0 \quad (7.8)$$

$$y_i = y_i^\star \quad \text{if} \quad y_i^\star > 0 \quad (7.9)$$

where the index function produces a set of limit and non-limit observations. This results in a likelihood function that consists of two parts, one part corresponding to a classical regression for the non-limit observations and the other part involving a discrete distribution of the relevant probabilities for the limiting cases.

The log-likelihood function shown in (7.10) was placed in the MATLAB function **to_like** which we maximize using the function **maxlik** from the *optimization function library* discussed in Chapter 10.

$$lnL = \sum_{y_i>0} -1/2[ln(2\pi) + ln\sigma^2 + (y_i - x_i\beta)^2/\sigma^2] + \sum_{y_i=0} ln[1 - \Phi(x_i\beta)/\sigma] \quad (7.10)$$

The documentation for the **tobit** function that produces maximum likelihood estimates for this model is shown below.

```
% PURPOSE: computes Tobit Regression
%---------------------------------------------------
% USAGE: results = tobit(y,x,info)
% where: y = censored dependent variable vector (nobs x 1)
%        x = independent variables matrix (nobs x nvar)
%     info = a structure with options:
%        info.trunc = 'left' or 'right' for censoring (default=left)
%        info.limit = value for censoring (default=0)
%        info.meth  = 'bfgs','dfp' (optional) default = bfgs
%        info.btol  =  tolerance for b convergence (default = 1e-8)
%        info.ftol  = tolerance for FUN convergence (default = 1e-8)
%        info.maxit = maximum # of iterations (default = 500)
%        info.b     = starting values for parameters (default = ols)
% RETURNS: a structure
%        results.meth  = 'tobit'
%        results.beta  = bhat
%        results.tstat = t-stats
%        results.yhat  = yhat
```

```
%        results.resid = residuals
%        results.sige  = e'*e/(n-k)
%        results.rsqr  = rsquared
%        results.rbar  = rbar-squared
%        results.lik   = Log-Likelihood value
%        results.iter  = # of iterations taken
%        results.grad  = gradient at solution
%        results.opt   = method of optimization used
%        results.nobs  = nobs
%        results.nobsc = # of censored observations
%        results.nvar  = nvars
%        results.y     = y data vector
```

Because the likelihood function represents a nonstandard mixture of discrete and continuous distributions, the **tobit** function provides user input options for: different types of hessian updating during optimization, different starting values, and alternative convergence criterion. Note also that the function provides input options to allow for 'left' or 'right' censoring of values in the model, (with the default being the case of 'left' censoring), and a user-specified 'limit value'. Example 7.6 illustrates use of the default convergence criterion, but implements all three alternative hessian updating schemes.

```
% ----- example 7.6 Tobit regression function
% generate uncensored data
n=100; k=2;
x = randn(n,k); x(:,1) = ones(n,1);
beta = ones(k,1)*2.0;
beta(1,1) = -2.0; beta(2,1) = -2.0;
y = x*beta + randn(n,1);
% now censor the data
for i=1:n
if y(i,1) < 0, y(i,1) = 0.0; end;
end;
vnames = strvcat('y','x1','x2');
info.meth = 'bfgs';
tic; resp = tobit(y,x,info); toc;
prt(resp,vnames);
info.meth = 'bhhh';
tic; resp = tobit(y,x,info); toc;
prt(resp,vnames);
info.meth = 'dfp';
tic; resp = tobit(y,x,info); toc;
prt(resp,vnames);
```

The resulting estimates and the CPU time taken are shown below. We see that the estimates produced by all three optimization methods are identical to three decimal places. All optimization algorithms took about the same time. In addition to reporting estimates, t-statistics and marginal probabilities, the printing routine for **tobit** estimation reports the gradient at the solution, which might provide useful information if the user wishes to change convergence tolerances.

```
elapsed_time =   1.1433
Tobit Regression Estimates
Dependent Variable =              y
R-squared      =    0.8028
Rbar-squared   =    0.8008
sigma^2        =    0.8144
Log-Likelihood =        -33.80814
# iterations   =     13
optimization   =   bfgs
Nobs, Nvars    =    100,     2
# of censored  =     81
***************************************************************
gradient at solution
Variable         Gradient
x1             0.00214371
x2            -0.00233626
sigma          0.00148077
Variable      Coefficient      t-statistic    t-probability
x1              -2.240217        -4.913982         0.000004
x2              -2.194446        -6.957942         0.000000

elapsed_time =    0.6400
Tobit Regression Estimates
Dependent Variable =              y
R-squared      =    0.8028
Rbar-squared   =    0.8008
sigma^2        =    0.8144
Log-Likelihood =        -33.80814
# iterations   =     13
optimization   =   bhhh
Nobs, Nvars    =    100,     2
# of censored  =     81
***************************************************************
gradient at solution
Variable         Gradient
x1             0.00214371
x2            -0.00233626
sigma          0.00148077
Variable      Coefficient      t-statistic    t-probability
x1              -2.240217        -4.913982         0.000004
x2              -2.194446        -6.957942         0.000000
```

```
elapsed_time =    0.6229
Tobit Regression Estimates
Dependent Variable =              y
R-squared      =    0.8028
Rbar-squared   =    0.8008
sigma^2        =    0.8144
Log-Likelihood =        -33.80814
# iterations   =     13
optimization   =    dfp
Nobs, Nvars    =    100,     2
# of censored  =     81
***************************************************************
gradient at solution
Variable         Gradient
x1             0.00214371
x2            -0.00233626
sigma          0.00148077
Variable      Coefficient      t-statistic    t-probability
x1              -2.240217        -4.913982         0.000004
x2              -2.194446        -6.957942         0.000000
```

7.4 Gibbs sampling Tobit models

Chib (1992) examines the tobit model from a Bayesian perspective and illustrates that Gibbs sampling can be used to estimate this model. Intuitively, we simply sample the latent variable from a right-truncated normal distribution for limit observations. Given the sample of y_i observations that are non-limit plus the sampled limit observations, we can proceed to sample conditionally for β and σ as in the usual Bayesian regression model.

The function **tobit_g** implements a Gibbs sampling solution for the tobit regression model and allows for heteroscedastic disturbances as in the case of the **probit_g** function. The documentation for the function is shown below, but the results structure returned by the function is omitted.

```
          y = X B + E, E = N(0,sige*V),
          V = diag(v1,v2,...vn), r/vi = ID chi(r)/r, r = Gamma(m,k)
          B = N(c,T),  sige = gamma(nu,d0)
----------------------------------------------------------------
 USAGE: result =  tobit_g(y,x,ndraw,nomit,prior,start)
 where: y = nobs x 1 independent variable vector
        x = nobs x nvar explanatory variables matrix
    ndraw = # of draws
    nomit = # of initial draws omitted for burn-in
    prior = a structure variable for prior information input
            prior.beta, prior means for beta,  c above (default=0)
```

```
          priov.bcov, prior beta covariance, T above (default=1e+12)
          prior.rval, r prior hyperparameter, default=4
          prior.m,    informative Gamma(m,k) prior on r
          prior.k,    informative Gamma(m,k) prior on r
                      default for above: not used, rval=4 is used
          prior.nu,   informative Gamma(nu,d0) prior on sige
          prior.d0    informative Gamma(nu,d0) prior on sige
                      default for above: nu=0,d0=0 (diffuse prior)
          prior.trunc = 'left' or 'right' (default = 'left')
          prior.limit = value for censoring (default = 0)
    start = (optional) structure containing starting values:
          defaults: max likelihood beta, sige, V= ones(n,1)
          start.b    = beta starting values (nvar x 1)
          start.sige = sige starting value (1x1)
          start.V    = V starting values (n x 1)
---------------------------------------------------------------
 NOTE: use either improper prior.rval
       or informative Gamma prior.m, prior.k, not both of them
----------------------------------------------------------------
```

Implementing this model with a diffuse prior for β, σ and assuming a homoscedastic prior for the disturbances should produce estimates similar to those from maximum likelihood estimation. Example 7.7 illustrates this type of application.

```
% ----- Example 7.7 Gibbs sampling tobit estimation
n=100; k = 4; sige = 10; evec = randn(n,1)*sqrt(sige);
x = randn(n,k); b = ones(k,1); b(1,1) = -1; b(2,1) = -1;
y = x*b + evec; yc = zeros(n,1);
% now censor the data
for i=1:n
    if y(i,1) >= 0, yc(i,1) = y(i,1); end;
end;
Vnames = strvcat('y','x1','x2','x3','x4');
prt(tobit(yc,x),Vnames);
prior.bcov = eye(k)*1000; % diffuse prior var-cov for b
prior.beta = zeros(k,1);  % diffuse prior means for b
prior.rval = 100;         % homoscedastic prior
ndraw = 1100; nomit = 100;
result = tobit_g(yc,x,ndraw,nomit,prior);
prt(result,Vnames);
```

The maximum likelihood and Gibbs sampling estimates are very similar, as they should be, and would lead to similar inferences. As in the case of the Gibbs sampling probit model, the value of the Bayesian tobit model lies in its ability to incorporate subjective prior information and to deal with cases where outliers or non-constant variance exist in the model.

```
Tobit Regression Estimates
Dependent Variable =              y
R-squared      =    0.9874
Rbar-squared   =    0.9870
sigma^2        =    6.7321
Log-Likelihood =       -156.76619
# iterations   =      7
optimization   =   bfgs
Nobs, Nvars    =    100,     4
# of censored  =     45
***************************************************************
gradient at solution
Variable         Gradient
x1            -0.00073896
x2            -0.00084981
x3             0.00075886
x4             0.00066791
sigma          0.00029843
Variable      Coefficient      t-statistic    t-probability
x1              -1.442589        -4.994263         0.000003
x2              -0.849097        -3.461847         0.000803
x3               0.989221         3.159387         0.002115
x4               0.842224         3.111376         0.002453

Bayesian Heteroscedastic Tobit Model Gibbs Estimates
Dependent Variable =              y
R-squared      =    0.9874
sigma^2        =    7.2537
nu,d0          =      0,     0
Nobs, Nvars    =    100,     4
# of censored  =     45
ndraws,nomit   =   1100,   100
time in secs   =   55.0607
r-value        =    100
***************************************************************
Variable       Prior Mean    Std Deviation
x1               0.000000        31.622777
x2               0.000000        31.622777
x3               0.000000        31.622777
x4               0.000000        31.622777
***************************************************************
      Posterior Estimates
Variable      Coefficient      t-statistic    t-probability
x1              -1.454368        -4.965893         0.000003
x2              -0.834533        -3.352712         0.001131
x3               1.027901         3.242982         0.001608
x4               0.817880         2.993917         0.003471
```

Green (1997) discusses the issue of heteroscedasticity in tobit models and points out that maximum likelihood estimates are problematical in this circumstance. Studies relying on a single dummy variable or one with groupwise heteroscedasticity indicate that heteroscedasticity presents a serious problem for maximum likelihood estimation. An approach to solving this problem is to replace the constant variance term σ with σ_i, where specification of a particular model for σ_i needs to be made by the investigator. This of course complicates the task of maximizing the likelihood function.

The Bayesian approach introduced by Geweke (1993) implemented in the function **tobit_g** eliminates the need to specify the form of the non-constant variance and accommodates the case of outliers as well as non-constant variance. In addition, the estimated parameters v_i based on the mean of Gibbs draws (representing the mean of posterior distribution for these parameters) can be used to draw inferences regarding the nature of the non-constant variance.

7.5 Chapter summary

This chapter demonstrated the *regression function library* programs for maximum likelihood estimation of logit, probit, and tobit models. In addition to these maximum likelihood estimation functions, two powerful functions, **probit_g** and **tobit_g** that implement a Gibbs sampling approach to estimating these models were discussed and illustrated. These functions have the advantage of dealing with non-constant variance and outliers that pose problems for maximum likelihood estimation of these models.

In addition to the functions described in this chapter, there are functions in the *spatial econometrics function library* that implement Gibbs sampling estimation of heteroscedastic spatial autoregressive probit and tobit models. LeSage (1997) describes these models and methods and compares them to an alternative approach set forth in McMillen (1992) based on the EM algorithm.

Although we did not utilize the **coda** convergence diagnostic functions (discussed in detail in Chapter 6) in conjunction with **probit_g** and **tobit_g**, these convergence diagnostics should of course be used in any application of these functions to diagnose convergence of the Gibbs sampler.

Chapter 7 Appendix

The maximum likelihood logit, mlogit, probit and tobit estimation functions as well as the Gibbs sampling functions probit_g and tobit_g discussed in this chapter are part of the *regression function library* in subdirectory **regress**.

```
 Regression functions discussed in this chapter

------- logit, probit, tobit and Gibbs regression functions -----------

 logit       - logit regression
 mlogit      - multinomial logit regression
 probit      - probit regression
 probit_g    - Gibbs sampling Bayesian probit model
 tobit       - tobit regression
 tobit_g     - Gibbs sampling Bayesian tobit model

 ------- demonstration programs -----------

 logit_d     - demonstrates logit regression
 mlogit_d    - demonstrates mlogit regression
 probit_d    - probit regression demo
 probit_gd   - demonstrates Bayesian probit model
 tobit_d     - tobit regression demo
 tobit_d2    - right-censored tobit regression demo
 tobit_gd    - demonstrates Bayesian tobit model
 tobit_gd2   - demonstrates right-censored Bayesian tobit model


------- support functions -----------

 dmult       - used by mlogit
 lo_like     - used by logit  (likelihood)
 maxlik      - used by tobit
 mderivs     - used by mlogit
 mlogit_lik  - used by mlogit
 nmlt_rnd    - used by probit_g
 nmrt_rnd    - used by probit_g, tobit_g
```

```
norm_cdf    - used by probit, pr_like
norm_pdf    - used by  probit
pr_like     - used by probit (likelihood)
prt_gibbs   - prints results for tobit_g,probit_g
stdn_cdf    - used by norm_cdf
stdn_pdf    - used by norm_pdf
to_rlike    - used by right-censored tobit  (likelihood)
to_llike    - used by left-censored tobit  (likelihood)
```

The spatial autoregressive versions of the probit and tobit models mentioned in the chapter summary are in the *spatial econometrics function library* in subdirectory **spatial**.

```
------- spatial econometrics program functions -----------

sarp_g    - Gibbs sampling Bayesian sar Probit model
sart_g    - Gibbs sampling Bayesian sar Tobit model
sacp_g    - Gibbs sampling Bayesian sac Probit model
sact_g    - Gibbs sampling Bayesian sac tobit model
semp_g    - Gibbs sampling Bayesian sem Probit model
semt_g    - Gibbs sampling Bayesian sem tobit model

------- demonstrations  -----------

sarp_gd   - sar Probit Gibbs sampling demo
sart_gd   - sar Tobit model Gibbs sampling demo
sacp_gd   - sac Probit Gibbs sampling demo
sact_gd   - sac Tobit model Gibbs sampling demo
semp_gd   - sem Probit Gibbs sampling demo
semt_gd   - sem Tobit model Gibbs sampling demo

------- support functions  -----------

anselin   - Anselin (1988) Columbus crime data
g_rho     - used by sar_g,sart_g,sarp_g
prt_spat  - prints results from spatial models
wmat.dat  - Anselin (1988) 1st order contiguity
c_sacl    - conditional distribution for sacp_g, sact_g
c_sacp    - conditional distribution for sacp_g, sact_g
c_sar     - conditional distribution for sarp_g, sart_g
c_sem     - conditional distribution for semp_g, semt_g
```

Chapter 8

Simultaneous Equation Models

The *regression function library* contains routines for two-stage least-squares, three-stage least-squares and seemingly unrelated regression models. This chapter discusses these functions and provides examples that use these estimation routines. Section 8.1 covers two-stage least-squares with three-stage least-squares demonstrated in Section 8.2. Seemingly unrelated regression equations are discussed and illustrated in Section 8.3.

8.1 Two-stage least-squares models

The problem of simultaneity in regression models arises when the explanatory variables matrix cannot be viewed as fixed during the dependent variable generation process. An example would be a simple Keynesian consumption function model:

$$C_t = \alpha + \beta Y_t + \varepsilon \tag{8.1}$$

Where C_t represents aggregate consumption at time t and Y_t is disposable income at time t.

Because of the national income accounting identity in this simple model: $Y_t \equiv C_t + I_t$, we cannot plausibly argue that the variable Y_t is fixed in repeated samples. The accounting identity implies that every time C_t changes, so will the variable Y_t.

A solution to the 'simultaneity problem' is to rely on two-stage least-squares estimation rather than ordinary least-squares. The function **tsls** will implement this estimation procedure. The documentation for the function is:

```
 PURPOSE: computes Two-Stage Least-squares Regression
---------------------------------------------------
 USAGE: results = tsls(y,yendog,xexog,xall)
 where: y      = dependent variable vector (nobs x 1)
        yendog = endogenous variables matrix (nobs x g)
        xexog  = exogenous variables matrix for this equation
        xall   = all exogenous and lagged endogenous variables
                 in the system
---------------------------------------------------
 RETURNS: a structure
        results.meth  = 'tsls'
        results.bhat  = bhat estimates
        results.tstat = t-statistics
        results.yhat  = yhat predicted values
        results.resid = residuals
        results.sige  = e'*e/(n-k)
        results.rsqr  = rsquared
        results.rbar  = rbar-squared
        results.dw    = Durbin-Watson Statistic
        results.nobs  = nobs,
        results.nendog = # of endogenous
        results.nexog  = # of exogenous
        results.nvar   = results.nendog + results.nexog
        results.y      = y data vector
 --------------------------------------------------
 NOTE: you need to put a constant term in the x1 and xall matrices
 --------------------------------------------------
 SEE ALSO: prt(results), prt_reg(), plt(), thsls()
---------------------------------------------------
```

As an example of using this function, consider example 8.1, where a dependent variable **y2** is generated that depends on two variables **y1** and **x2**. The variable **y1** is generated using the same error vector **evec** as in the generation process for **y2** so this variable is correlated with the error term. This violates the Gauss-Markov assumption that the explanatory variables in the regression model are fixed in repeated sampling, leading to biased and inconsistent estimates from least-squares.

```
% ----- Example 8.1 Two-stage least-squares
nobs = 200;
x1 = randn(nobs,1);  x2 = randn(nobs,1);
b1 = 1.0; b2 = 1.0;  iota = ones(nobs,1);
y1 = zeros(nobs,1);  y2 = zeros(nobs,1);
evec = randn(nobs,1);
```

```
% create simultaneously determined variables y1,y2
for i=1:nobs;
y1(i,1) = iota(i,1) + x1(i,1)*b1 + evec(i,1);
y2(i,1) = iota(i,1) + y1(i,1) + x2(i,1)*b2 + evec(i,1);
end;
vname1 = strvcat('y1-eqn','y2 variable','constant','x1 variable');
vname2 = strvcat('y2-eqn','y1 variable','constant','x2 variable');
% use all exogenous in the system as instruments
xall = [iota x1 x2];
% do ols regression for comparison with two-stage estimates
result1 = ols(y2,[y1 iota x2]);
prt(result1,vname2);
% do tsls regression
result2 = tsls(y2,y1,[iota x2],xall);
prt(result2,vname2);
```

Note that the MATLAB function **tsls** requires that the user supply a matrix of variables that will be used as instruments during the first-stage regression. In example 8.1 all exogenous variables in the system of equations are used as instrumental variables, a necessary condition for two-stage least-squares to produce consistent estimates.

Least-squares and two-stage least-squares estimates from example 8.1 are presented below. The least-squares estimates exhibit the simultaneity bias by producing estimates that are significantly different from the true values of $\beta = 1$, used to generate the dependent variable **y2**. In contrast, the two-stage least-squares estimates are much closer to the true values of unity. The results from the example also illustrate that the simultaneity bias tends to center on the parameter estimates associated with the constant term and the right-hand-side endogenous variable **y1**. The coefficient estimates for **x2** from least-squares and two-stage least-squares are remarkably similar.

```
Ordinary Least-squares Estimates
Dependent Variable =      y2-eqn
R-squared      =    0.9144
Rbar-squared   =    0.9136
sigma^2        =    0.5186
Durbin-Watson  =    2.1629
Nobs, Nvars    =    200,     3
***************************************************************
Variable         Coefficient      t-statistic    t-probability
y1 variable         1.554894        42.144421         0.000000
constant            0.462866         7.494077         0.000000
x2 variable         0.937300        18.154273         0.000000

Two Stage Least-squares Regression Estimates
Dependent Variable =      y2-eqn
```

```
R-squared      =    0.7986
Rbar-squared   =    0.7966
sigma^2        =    1.2205
Durbin-Watson  =    2.0078
Nobs, Nvars    =    200,     3
***************************************************************
Variable        Coefficient      t-statistic    t-probability
y1 variable        0.952439        10.968835         0.000000
constant           1.031016         9.100701         0.000000
x2 variable        0.937037        11.830380         0.000000
```

Implementation of the MATLAB function **tsls** is relatively straightforward as illustrated by the following code:

```
if (nargin ~= 4); error('Wrong # of arguments to tsls'); end;
results.meth = 'tsls';
[nobs1 g] = size(y1); [nobs2 k] = size(x1); [nobs3 l] = size(xall);
results.nendog = g; results.nexog = k; results.nvar = k+g;
if nobs1 == nobs2;
  if nobs2 == nobs3, nobs = nobs1; end;
else
error('tsls: # of observations in yendog, xexog, xall not the same');
end;
results.y = y; results.nobs = nobs;
% xall contains all explanatory variables
% x1 contains exogenous; % y1 contains endogenous
xapxa = inv(xall'*xall);
% form xpx and xpy
xpx = [y1'*xall*xapxa*xall'*y1     y1'*x1
       x1'*y1                      x1'*x1];
xpy = [y1'*xall*xapxa*xall'*y
       x1'*y                  ];
xpxi = inv(xpx);
results.beta  = xpxi*xpy;             % bhat
results.yhat  = [y1 x1]*results.beta; % yhat
results.resid = y - results.yhat;     % residuals
sigu = results.resid'*results.resid;
results.sige = sigu/(nobs-k-g);       % sige
tmp = results.sige*(diag(xpxi));
results.tstat = results.beta./(sqrt(tmp));
ym = y - ones(nobs,1)*mean(y);
rsqr1 = sigu; rsqr2 = ym'*ym;
results.rsqr = 1.0 - rsqr1/rsqr2; % r-squared
rsqr1 = rsqr1/(nobs-k-g);
rsqr2 = rsqr2/(nobs-1.0);
results.rbar = 1 - (rsqr1/rsqr2); % rbar-squared
ediff = results.resid(2:nobs) - results.resid(1:nobs-1);
results.dw = (ediff'*ediff)/sigu; % durbin-watson
```

After error checking on the input arguments, the function simply forms the matrices needed to carry out two-stage least-squares estimation of the parameters (e.g., Green, 1997). Given parameter estimates, the usual summary statistics measuring fit and dispersion of the parameters are calculated and added to the results structure variable that is returned by the function. Of course, a corresponding set of code to carry out printing the results was added to the **prt_reg** function, which is called by the wrapper function **prt**.

As a final example that clearly demonstrates the nature of inconsistent estimates, consider example 8.2 where a Monte Carlo experiment is carried out to compare least-squares and two-stage least-squares over 100 runs. In the program code for example 8.2, we rely on the utility function **mprint** described in Chapter 3, to produce a tabular print-out of our results with row and column labels. Another point to note regarding the code is that we dimension our matrices to store the coefficient estimates as (100,3), where we have 3 parameter estimates and 100 Monte Carlo runs. This facilitates using the MATLAB functions **mean** and **std** to compute means and standard deviations of the resulting estimates. Recall these functions work down the columns of matrices to compute averages and standard deviations of each column in the input matrix.

```
% ----- Example 8.2 Monte Carlo study of ols() vs. tsls()
nobs = 200;
x1 = randn(nobs,1); x2 = randn(nobs,1);
b1 = 1.0; b2 = 1.0; iota = ones(nobs,1);
y1 = zeros(nobs,1); y2 = zeros(nobs,1);
evec = randn(nobs,1);
% create simultaneously determined variables y1,y2
for i=1:nobs;
y1(i,1) = iota(i,1) + x1(i,1)*b1 + evec(i,1);
y2(i,1) = iota(i,1) + y1(i,1) + x2(i,1)*b2 + evec(i,1);
end;
% use all exogenous in the system as instruments
xall = [iota x1 x2];
niter = 100;            % number of Monte Carlo loops
bols  = zeros(niter,3); % storage for ols results
b2sls = zeros(niter,3); % storage for 2sls results
disp('patience -- doing 100 2sls regressions');
for iter=1:niter;       % do Monte Carlo looping
y1 = zeros(nobs,1);  y2 = zeros(nobs,1); evec = randn(nobs,1);
% create simultaneously determined variables y1,y2
for i=1:nobs;
y1(i,1) = iota(i,1)*1.0 + x1(i,1)*b1 + evec(i,1);
y2(i,1) = iota(i,1)*1.0 + y1(i,1)*1.0 + x2(i,1)*b2 + evec(i,1);
end;
result1 = ols(y2,[y1 iota x2]);       % do ols regression
```

```
result2 = tsls(y2,y1,[iota x2],xall); % do tsls regression
bols(iter,:)  = result1.beta';
b2sls(iter,:) = result2.beta';
end;                    % end Monte Carlo looping
% find means and std deviations over the niter runs
bolsm = mean(bols); b2slsm = mean(b2sls);
bolss = std(bols);  b2slss = std(b2sls);
% print results
fprintf(['OLS results over ',num2str(niter),' runs\n']);
in.rnames = strvcat('Coefficients','b1','b2','b3');
in.cnames = strvcat('Mean','std dev');
mprint([bolsm' bolss'],in);
fprintf(['TSLS results over ',num2str(niter),' runs\n']);
mprint([b2slsm' b2slss'],in);
```

The biased and inconsistent estimates produced by least-squares in the face of simultaneity should exhibit estimates that deviate from the true $\beta = 1$ values used in the experiment. The fact that this bias remains even when averaged over 100 Monte Carlo runs, illustrates an important point about consistent versus inconsistent estimates.

```
OLS results over 100 runs
Coefficients       Mean    std dev
b1               1.4769     0.0233
b2               0.4820     0.0460
b3               1.0581     0.0418

TSLS results over 100 runs
Coefficients       Mean    std dev
b1               0.9986     0.0694
b2               1.0162     0.1105
b3               0.9972     0.0791
```

8.2 Three-stage least-squares models

When the disturbances from one equation in a system of simultaneous equations are contemporaneously correlated with those from other equations, we can rely on three-stage least-squares as a method of incorporating this correlation structure in the resulting parameter estimates. This will tend to increase the precision of the estimates. A potential problem with this estimation procedure is that model misspecification in one equation can work through the correlation structure in the disturbances to contaminate the specification of other equations in the system.

We face a challenge in constructing a three-stage least-squares routine because the user must provide input to our MATLAB function **thsls** that indicates the equation structure of the entire system as well as all of the associated variable vectors. This problem was solved using MATLAB structure variables. An additional challenge is that the results structure must return results for all equations in the system, which we also solve using a MATLAB structure variable, as in the case of vector autoregressive models discussed in Chapter 5.

The documentation for the **thsls** function is:

```
 PURPOSE: computes Three-Stage Least-squares Regression
          for a model with neqs-equations
---------------------------------------------------
 USAGE: results = thsls(neqs,y,Y,X)
 where:
      neqs  = # of equations
        y   = an 'eq' structure containing dependent variables
              e.g. y(1).eq = y1; y(2).eq = y2; y(3).eq = y3;
        Y   = an 'eq' structure containing RHS endogenous
              e.g. Y(1).eq = []; Y(2).eq = [y1 y3]; Y(3).eq = y2;
        X   = an 'eq' structure containing exogenous/lagged endogenous
              e.g. X(1).eq = [iota x1 x2];
                   X(2).eq = [iota x1];
                   X(3).eq = [iota x1 x2 x3];
---------------------------------------------------
        NOTE:  X(i), i=1,...,G should include a constant vector
                    if you want one in the equation
---------------------------------------------------
 RETURNS a structure:
   result.meth      = 'thsls'
   result(eq).beta  = bhat for each equation
   result(eq).tstat = tstat for each equation
   result(eq).tprob = tprobs for each equation
   result(eq).resid = residuals for each equation
   result(eq).yhat  = yhats for each equation
   result(eq).y     = y for each equation
   result(eq).rsqr  = r-squared for each equation
   result(eq).rbar  = r-squared adj for each equation
   result(eq).nvar  = nvar in each equation
   result(eq).sige  = e'e/nobs for each equation
   result(eq).dw    = Durbin-Watson
   result.nobs      = nobs
   result.neqs      = neqs
   result.sigma     = sig(i,j) across equations
   result.ccor      = correlation of residuals across equations
 --------------------------------------------------
 SEE ALSO: prt, prt_eqs, plt
---------------------------------------------------
```

As an example of using the function, consider example 8.3 that generates a three equation system with cross-equation correlation between the disturbances in the second and third equations. Variable **y1** appears as a right-hand-side endogenous variable in the **y2** equation and variable **y2** appears in a similar way in the **y3** equation.

The function **thsls** requires that we set up structure variables that take the form of var_name(eq#).eq, where eq# is the number of the equation. We need to input: left-hand-side or dependent variables for each equation in the first structure variable input argument to the function, right-hand-side endogenous variables in the second structure variable argument to the function, and exogenous variables for each equation as the third input argument to the function. In the example, the first structure variable was named **y**, the second is **Y** and the third is **X**, following the usual econometrics textbook naming conventions.

Some things to note about the input arguments are that:

1. We enter an empty matrix for equations where no right-hand-side variables exist, for example: **Y(1).eq = [];**

2. We enter constant term vectors in the structure variable containing the exogenous variables in the model for all equations where we desire their presence.

3. The function does not check for identification, that is the user's responsibility.

4. The structure variable must have a name ending with **.eq**.

5. The number of observations in all equations for all variable vectors input to the function must be equal.

We construct three separate string vectors using the MATLAB **strvcat** function that contain the variable names for each equation. This function is used again to concatenate these three string vectors into a single long string vector for the printing function. One could of course simply construct a single long string vector of names, but the approach taken here allows us to carry out two-stage least-squares estimates or ordinary least-squares estimates on each equation and print results with the corresponding variable names.

The variable name strings must of course reflect the order of the variables in each equation and they need to follow the equation order from first to last.

```
% ----- Example 8.3 Three-stage least-squares
nobs = 100; neqs = 3;
x1 = randn(nobs,1); x2 = randn(nobs,1); x3 = randn(nobs,1);
b1 = 1.0; b2 = 1.0; b3 = 1.0; iota = ones(nobs,1);
y1 = zeros(nobs,1); y2 = zeros(nobs,1); y3 = zeros(nobs,1);
e = randn(nobs,3);
e(:,2) = e(:,3) + randn(nobs,1); % create cross-eqs corr
% create simultaneously determined variables y1,y2
for i=1:nobs;
y1(i,1) = iota(i,1)*10.0 + x1(i,1)*b1 + e(i,1);
y2(i,1) = iota(i,1)*10.0 + y1(i,1)*1.0 + x2(i,1)*b2 + e(i,2);
y3(i,1) = iota(i,1)*10.0 + y2(i,1)*1.0 + x2(i,1)*b2 + x3(i,1)*b3 + e(i,3);
end;
vname1 = strvcat('y1-LHS','constant','x1 var');
vname2 = strvcat('y2-LHS','y1-RHS','constant','x2 var');
vname3 = strvcat('y3-LHS','y2-RHS','constant','x2 var','x3 var');
% set up a structure for y containing y's for each eqn
y(1).eq = y1; y(2).eq = y2; y(3).eq = y3;
% set up a structure for Y (RHS endogenous) for each eqn
Y(1).eq = []; Y(2).eq = [y1]; Y(3).eq = [y2];
% set up a structure for X (exogenous) in each eqn
X(1).eq = [iota x1]; X(2).eq = [iota x2]; X(3).eq = [iota x2 x3];
result = thsls(neqs,y,Y,X); % do thsls regression
vname = strvcat(vname1,vname2,vname3);
prt(result,vname);
```

A printout produced by running example 8.3 is shown below. A separate function **prt_eqs** was developed to carry out the printing, but as with printing results from other functions in the *Econometrics Toolbox*, the wrapper function **prt** will work to produce printed output by calling the appropriate function.

The printed results are presented in order for each equation in the system and then the cross-equation error covariance and correlation estimates are presented. From the results shown for example 8.3 we see that all of the constant term estimates are near the true value of 10, and all other estimates are near the true values of unity. We see no evidence of cross-equation correlation between the disturbances of equations 1 and 2, but in equations 2 and 3 where we generated a correlation, the estimated coefficient is 0.58.

```
Three Stage Least-squares Estimates -- Equation   1
Dependent Variable =         y1-LHS
R-squared      =    0.4808
Rbar-squared   =    0.4755
sigma^2        =    3.3261
Durbin-Watson  =    1.8904
Nobs, Nvars    =    100,     2
```

```
***************************************************************
Variable      Coefficient      t-statistic    t-probability
constant         9.985283       100.821086         0.000000
x1 var           1.101060         9.621838         0.000000

Three Stage Least-squares Estimates -- Equation   2
Dependent Variable =         y2-LHS
R-squared      =    0.6650
Rbar-squared   =    0.6581
sigma^2        =    3.3261
Durbin-Watson  =    1.8810
Nobs, Nvars    =    100,     3
***************************************************************
Variable      Coefficient      t-statistic    t-probability
y1-RHS           0.959248         7.615010         0.000000
constant        10.389051         8.177007         0.000000
x2 var           0.925196         7.326562         0.000000

Three Stage Least-squares Estimates -- Equation   3
Dependent Variable =         y3-LHS
R-squared      =    0.9156
Rbar-squared   =    0.9129
sigma^2        =    3.3261
Durbin-Watson  =    1.6829
Nobs, Nvars    =    100,     4
***************************************************************
Variable      Coefficient      t-statistic    t-probability
y2-RHS           1.005874         9.667727         0.000000
constant         9.819170         4.705299         0.000008
x2 var           0.969571         6.898767         0.000000
x3 var           1.097547        14.974708         0.000000

Cross-equation sig(i,j) estimates
equation    y1-LHS    y2-LHS    y3-LHS
y1-LHS      0.9779    0.1707   -0.1108
y2-LHS      0.1707    1.4383    0.7660
y3-LHS     -0.1108    0.7660    0.9005

Cross-equation correlations
equation    y1-LHS    y2-LHS    y3-LHS
y1-LHS      1.0000    0.1439   -0.1181
y2-LHS      0.1439    1.0000    0.6731
y3-LHS     -0.1181    0.6731    1.0000
```

There is also a function **plt_eqs** that is called by the wrapper function **plt** to produce graphs of the actual versus predicted and residuals for each equation of the system. The graphs are produced in a 'for loop' with a pause between the graphs for each equation in the system. A virtue of the multiple equation result structure is that printing and plotting functions written previously for single equation regression methods can be adapted. For the case of systems of equations we can embed the previous code in 'for loops' that range over the equations in the results structure variable.

As an example of this, consider the following code fragment from the function **plt_eqs** that loops over the structure variable **results** accessing the actual and predicted values for each equation as well as the residuals.

```
tt=1:nobs;
clf;
cnt = 1;
for j=1:neqs;
nvar = results(j).nvar;
subplot(2,1,1), plot(tt,results(j).y,'-',tt,results(j).yhat,'--');
 if nflag == 1
  title([upper(results(1).meth), ' Act vs. Predicted  ',vnames(cnt,:)]);
 else
  title([upper(results(1).meth), ' Act vs. Predicted eq ',num2str(j)]);
 end;
subplot(2,1,2), plot(tt,results(j).resid)
cnt = cnt+nvar+1;
pause;
end;
```

8.3 Seemingly unrelated regression models

A function **sur** exists for estimating seemingly unrelated regression models, where a series of regression relations exhibit covariation in the disturbances across equations. An initial least-squares regression is carried out for all equations in the model and the residuals from this series of regressions are used to form the cross-equation error covariance matrix estimates. One design consideration of interest here is that we rely on a special function **olse** to carry out the initial least-squares regressions rather than **ols**. This function avoids many of the computations performed in **ols** because our only concern is a residual vector from least-squares squares estimation.

The function **olse** is shown below, where we rely on the Cholesky decomposition approach to solving the least-squares problem, again for speed.

```
function resid=olse(y,x)
% PURPOSE: OLS regression returning only residual vector
%---------------------------------------------------
% USAGE: residual = olse(y,x)
% where: y = dependent variable vector (nobs x 1)
%        x = independent variables matrix (nobs x nvar)
```

```
%---------------------------------------------------
% RETURNS: the residual vector
%---------------------------------------------------
if (nargin ~= 2); error('Wrong # of arguments to olse'); end;
beta = x\y;
resid = y - x*beta;
```

The documentation for the **sur** function is shown below with the information regarding the results structure returned by the function omitted to save space.

```
 PURPOSE: computes seemingly unrelated regression estimates
          for a model with neqs-equations
---------------------------------------------------
 USAGE:     results = sur(neqs,y,x,iflag,info)
        or, results = sur(neqs,y,x) (for no iteration)
 where:
 neqs  = # of equations
   y   = an 'eq' structure with dependent variables
         e.g. y(1).eq = y1; y(2).eq = y2; y(3).eq = y3;
   x   = an 'eq' structure with explanatory variables
            e.g. x(1).eq = [iota x1 x4];
                 x(2).eq = [iota x1];
                 x(3).eq = [iota x1 x2 x5];
 iflag = 1 for iteration on error covariance matrix,
         0 for no iteration (0 = default)
  info = a structure for iteration options:
         info.itmax = maximum # of iterations (default = 100)
         info.crit  = criterion for absolute bhat change
              (default = 0.001)
---------------------------------------------------
 NOTE:  x(i), i=1,...,G should include a constant vector
        if you want one in the equation
---------------------------------------------------
 SEE ALSO: prt(results), prt_reg(), plt(results)
---------------------------------------------------
```

As noted in the documentation for **sur**, there is an option to iterate on the cross-equation error covariance matrix estimates. Iteration involves using the residuals from the initial seemingly unrelated regression system (based on the least-squares residuals) to form another estimate of the error covariance matrix. This process of using the sur residuals to form an updated error covariance matrix estimate continues until there is a small change in the $\hat{\beta}$ estimates from iteration to iteration. A structure variable with fields 'itmax' and 'crit' allows the user to input these iteration options. The 'itmax' field allows the user to specify the maximum number of iterations and the field 'crit' allows input of the convergence criterion. The latter scalar input represents the change in the sum of the absolute value of the $\hat{\beta}$ estimates from iteration to iteration. When the estimates change by less than the 'crit' value from one iteration to the next, convergence has been achieved and iteration stops.

As an example, consider the Grunfeld investment model, where we have annual investment time-series covering the period 1935-54 for five firms. In addition, market value of the firm and desired capital stock serve as explanatory variables in the model. The explanatory variables represent market value lagged one year and a constructed desired capital stock variable that is also lagged one year. See Theil (1971) for an explanation of this model.

Example 8.4 estimates the Grunfeld model using the **sur** function with and without iteration.

```
% ----- Example 8.4  Using the sur() function
load grun.dat; % grunfeld investment data (page 650, Green 1997)
y1  = grun(:,1); x11 = grun(:,2); x12 = grun(:,3); % general electric
y2  = grun(:,4); x21 = grun(:,5); x22 = grun(:,6); % westinghouse
y3  = grun(:,7); x31 = grun(:,8); x32 = grun(:,9); % general motors
y4  = grun(:,10);x41 = grun(:,11);x42 = grun(:,12);% chrysler
y5  = grun(:,13);x51 = grun(:,14);x52 = grun(:,15);% us steel
nobs = length(y1); iota = ones(nobs,1);
vname1 = strvcat('I gen motors','cons','fgm','cgm');
vname2 = strvcat('I chrysler','const','fcry','ccry');
vname3 = strvcat('I gen electric','const','fge','cge');
vname4 = strvcat('I westinghouse','const','fwest','cwest');
vname5 = strvcat('I us steel','const','fuss','cuss');
% set up a structure for y in each eqn
% (order follows that in Green, 1997)
y(1).eq = y3; % gm
y(2).eq = y4; % chrysler
y(3).eq = y1; % general electric
y(4).eq = y2; % westinghouse
y(5).eq = y5; % us steel
% set up a structure for X in each eqn
X(1).eq = [iota x31 x32];
X(2).eq = [iota x41 x42];
X(3).eq = [iota x11 x12];
X(4).eq = [iota x21 x22];
X(5).eq = [iota x51 x52];
% do sur regression with iteration
neqs = 5; iflag = 1; % rely on default itmax, crit values
result = sur(neqs,y,X,iflag);
% do sur regression with no iteration
result2 = sur(neqs,y,X);
```

```
vname = strvcat(vname1,vname2,vname3,vname4,vname5);
prt(result,vname); % print results for iteration
prt(result2,vname);% print results for no iteration
```

The results indicate that iteration does not make a great deal of different in the resulting estimates. To conserve on space, we present comparative results for a two equation model with General Electric and Westinghouse for which results are reported in Theil (1971).

```
% RESULTS with iteration
Seemingly Unrelated Regression -- Equation   1
Dependent Variable =   I gen electr
System R-sqr   =    0.6124
R-squared      =    0.6868
Rbar-squared   =    0.6500
sigma^2        = 11961.0900
Durbin-Watson  =    0.0236
Nobs, Nvars    =     20,     3
***************************************************************
Variable           Coefficient      t-statistic    t-probability
constant            -30.748395        -1.124423         0.276460
fge                   0.040511         3.021327         0.007696
cge                   0.135931         5.772698         0.000023

Seemingly Unrelated Regression -- Equation   2
Dependent Variable =   I westinghouse
System R-sqr   =    0.6124
R-squared      =    0.7378
Rbar-squared   =    0.7070
sigma^2        = 2081.1840
Durbin-Watson  =    0.0355
Nobs, Nvars    =     20,     3
***************************************************************
Variable           Coefficient      t-statistic    t-probability
constant             -1.701600        -0.245598         0.808934
fwest                 0.059352         4.464549         0.000341
cwest                 0.055736         1.143147         0.268821

Cross-equation sig(i,j) estimates
equation         I gen electr I westinghouse
I gen electr         702.2337       195.3519
I westinghouse       195.3519        90.9531

Cross-equation correlations
equation         I gen electr I westinghouse
I gen electr           1.0000         0.7730
I westinghouse         0.7730         1.0000
```

```
% RESULTS without iteration
Seemingly Unrelated Regression -- Equation   1
Dependent Variable =   I gen electr
System R-sqr   =    0.6151
R-squared      =    0.6926
Rbar-squared   =    0.6564
sigma^2        = 11979.6253
Durbin-Watson  =    0.0216
Nobs, Nvars    =     20,     3
***************************************************************
Variable            Coefficient      t-statistic    t-probability
constant             -27.719317        -1.019084         0.322448
fge                    0.038310         2.870938         0.010595
cge                    0.139036         5.948821         0.000016

Seemingly Unrelated Regression -- Equation   2
Dependent Variable =   I westinghouse
System R-sqr   =    0.6151
R-squared      =    0.7404
Rbar-squared   =    0.7099
sigma^2        = 2082.7474
Durbin-Watson  =    0.0338
Nobs, Nvars    =     20,     3
***************************************************************
Variable            Coefficient      t-statistic    t-probability
constant              -1.251988        -0.180970         0.858531
fwest                  0.057630         4.337133         0.000448
cwest                  0.063978         1.314408         0.206168

Cross-equation sig(i,j) estimates
equation         I gen electr I westinghouse
I gen electr         689.4188       190.6363
I westinghouse       190.6363        90.0650

Cross-equation correlations
equation         I gen electr I westinghouse
I gen electr           1.0000         0.7650
I westinghouse         0.7650         1.0000
```

8.4 Chapter summary

Estimation functions for systems of equations can be constructed by using MATLAB structure variables to input data vectors and matrices for each equation in the model. Structure variables can also be used to return estimation results for each equation. This allows us to use many of the same approaches developed for single equation regression models for printing and plotting results by embedding the previous code in a for-loop over the equations in the model.

Chapter 8 Appendix

The tsls, thsls and sur estimation functions discussed in this chapter are part of the *regression function library* in the subdirectory **regress**.

```
 Regression functions discussed in this chapter

------- tsls, thsls, sur regression functions -----------

 sur        - seemingly unrelated regressions
 tsls       - two-stage least-squares
 thsls      - three-stage least-squares

 ------- demonstration programs -----------

 sur_d      - demonstrates sur using Grunfeld's data
 tsls_d     - two-stage least-squares demo
 thsls_d    - three-stage least-squares demo

------- support functions -----------

 olse       - ols returning only residuals (used by sur)
 prt_eqs    - prints equation systems
 plt_eqs    - plots equation systems
 grun.dat   - Grunfeld's data used by sur_d
 grun.doc   - documents Grunfeld's data set
```

Chapter 9

Distribution functions library

The *distributions function library* contains routines to carry out calculations based on a host of alternative statistical distributions. For each statistical distribution, functions exist to: calculate the probability density function (pdf), cumulative density function (cdf), inverse or quantile function (inv), as well as generate random numbers. A naming convention is used for these functions that takes the form of: a four-letter abbreviation for the statistical distribution name followed by an underscore character and the three letter codes, **cdf, pdf, inv** or **rnd**. As an example, the beta statistical distribution has four functions named: **beta_cdf, beta_pdf, beta_inv, beta_rnd**.

The library contains these four functions for the following 12 distributions with the indicated four-letter abbreviations.

1. beta distribution (beta)
2. binomial distribution (bino)
3. chi-square distribution (chis)
4. F-distribution (fdis)
5. gamma distribution (gamm)
6. hypergeometric distribution (hypg)
7. log-normal distribution (logn)
8. logistic distribution (logt)
9. multivariate normal distribution (norm)

10. poisson distribution (pois)
11. standard normal distribution (stdn)
12. Student t-distribution (tdis)

There are also some related more special purpose functions. These are shown in the list below:

chis_prb - computes marginal probabilities for chi-squared distributed statistics

fdis_prb - computes marginal probabilities for F-distributed statistics

tdis_prb - computes marginal probabilities for t-distributed statistics

norm_crnd - generates normally distributed random variates from a contaminated normal distribution, $(1-\gamma)N(0,1)+\gamma N(0,\sigma)$.

nmlt_rnd - generates left-truncated normal draws.

nmrt_rnd - generates right-truncated normal draws.

unif_rnd - generates uniform draws between upper and lower limits.

wish_rnd - generates random draws from a Wishart distribution.

Section 9.1 demonstrates use of the **pdf, cdf, inv, rnd** statistical distribution functions and the specialized functions are taken up in Section 9.2.

9.1 The pdf, cdf, inv and rnd functions

Each of the 12 distributions in the library has a demonstration program that illustrates use of the four associated functions, **pdf,cdf,inv,rnd**. For example, the demonstration function **beta_d** carries out random draws from the beta distribution, based on parameters (a, b), where the mean of the distribution is: $a/(a+b)$ and the variance is: $ab/((a+b)^2(a+b+1))$.

Given a sample of 1000 random beta(a,b) draws, the mean and variance are computed and compared to the theoretical values. Next, a grid of 1000 uniform random values is generated and sorted from low to high. The pdf, cdf and quantiles are generated for this grid of values. Plotting the pdf, cdf and quantile values against the input grid should produce smooth densities that conform to the beta distribution shape. Example 9.1 shows the demonstration program for the beta distribution.

```
% ----- Example 9.1 Beta distribution function example
n = 1000; a = 10; b = 5;
tst = beta_rnd(n,a,b); % generate random draws
 % mean should equal a/(a+b)
fprintf('mean should       = %16.8f \n',a/(a+b));
fprintf('mean of draws     = %16.8f \n',mean(tst));
% variance should equal a*b/((a+b)*(a+b)*(a+b+1))
fprintf('variance should   = %16.8f \n',(a*b)/((a+b)*(a+b)*(a+b+1)));
fprintf('variance of draws = %16.8f \n',std(tst)*std(tst));
tst = rand(n,1); tsort = sort(tst); % generate a grid of values
pdf = beta_pdf(tsort,a,b); % calculate pdf over the grid
cdf = beta_cdf(tsort,a,b); % calculate cdf over the grid
x = beta_inv(tsort,a,b);   % calculate quantiles for the grid
subplot(3,1,1),
plot(tsort,pdf); xlabel(['pdf of beta(',num2str(a),',',num2str(b),')']);
subplot(3,1,2),
plot(tsort,cdf); xlabel(['cdf of beta(',num2str(a),',',num2str(b),')']);
subplot(3,1,3),
plot(tsort,x);   xlabel(['inv of beta(',num2str(a),',',num2str(b),')']);
```

The results from running the example 9.1 program are displayed below and the plots produced by the program are shown in Figure 9.1.

```
mean should       =       0.66666667
mean of draws     =       0.66721498
variance should   =       0.01388889
variance of draws =       0.01426892
```

These demonstration programs also provide a way to check the validity of the 12 distribution functions. Many of the functions in the distributions library were found on the internet and have been crafted by others. In all cases, the function documentation was modified to be consistent with other functions in the *Econometrics Toolbox* and in some cases, the functions were converted to MATLAB from other languages. In cases where the MATLAB functions were written by others, the names of the authors have been left in the function source code.

9.2 The specialized functions

Some of the specialized functions need no explanation. For example, the ability to calculate probabilities associated with $t-$distributed, $F-$distributed and χ^2-distributed statistics provided by **tdis_prb, fdis_prb, chis_prb** seems obviously useful.

As we have seen in Chapters 6 and 7, the ability to carry out random draws from multivariate normal, chi-squared, gamma distributions and

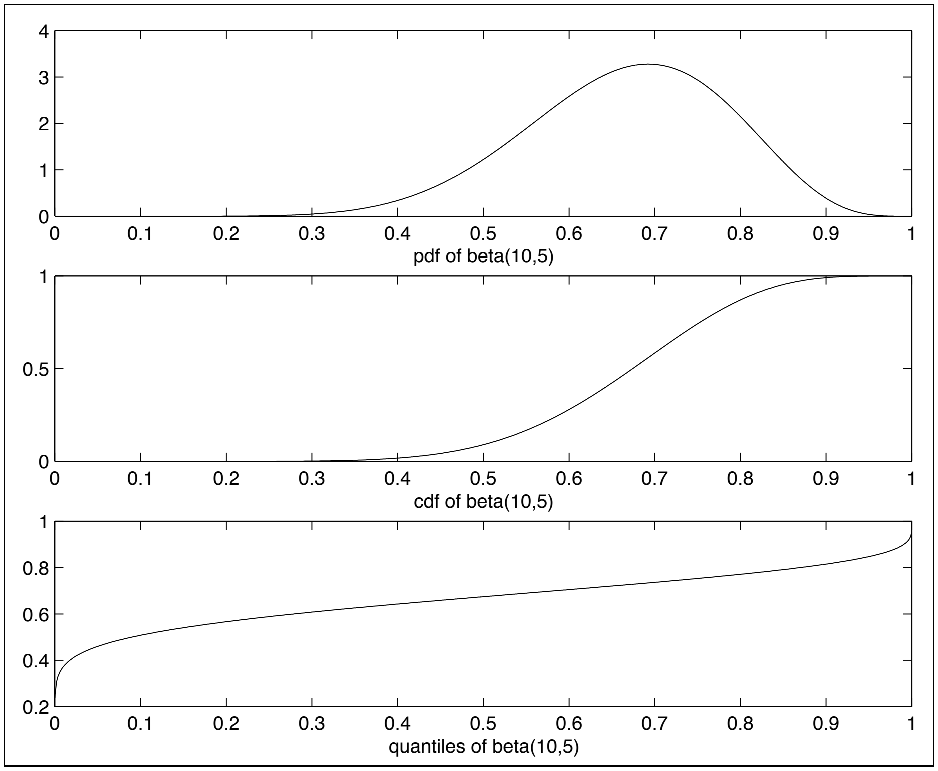

Figure 9.1: Beta distribution demonstration program plots

left or right-truncated normal distributions is needed to implement Gibbs sampling estimation.

Another random number generation function that is useful in this type of estimation is the Wishart random draw function **wish_rnd**. Consider the case of a general multivariate normal model where m responses $y_{ij}, j = 1, \ldots, m$ exist from measurements taken at m different time intervals or during multiple experimental runs. This situation, where we have a set of n m-variate observations, leads to errors in the model $\varepsilon_{ij}, i = 1, \ldots, n, j = 1, \ldots, m$, that are often assumed to take the form of an $m-$variate Normal $N_m(0, \Sigma)$, that depend on a set of parameters θ.

Consider a Bayesian analysis where we assume: an independence in the priors for θ and Σ, a locally uniform prior for θ, and the non-informative reference prior $p(\Sigma) \propto |\Sigma|^{-1/2(m+1)}$. This produces a posterior distribution for Σ conditional on θ that takes the form of a Wishart distribution. Specifically, $\Sigma^{-1} \sim W_m(S^{-1}(\theta), n-m+1)$, where $S(\theta) = \sum_{i=1}^{n} \varepsilon_{ij}\varepsilon_{ij}$, (Box and Tiao, 1992).

An example from Tanner (1991) provides an applied illustration of how this distribution arises in the context of Gibbs sampling. Data on 30 young rats growth measured over a span of five weeks were analyzed in Gelfand, Hills, Racine-Poon and Smith (1990). Assuming a linear model for individual rats growth curves:

$$y_{ij} \sim N(\alpha_i, \beta_i x_{ij}, \sigma^2) \tag{9.1}$$

Where, $i = 1, \ldots, 30$ representing individual rats and $j = 1, \ldots, 5$ denoting the five weeks in the measurement period. The explanatory variable in the model, x_{ij} measures the age of the ith rat at measurement period j. The prior for the intercept and slope parameters in the model takes the form of a multivariate normal, where all rats α_i, β_i are centered on α_0, β_0. The prior variance-covariance structure (Σ) for these parameters takes the form of a Wishart (W) distribution.

This prior information can be written:

$$\begin{pmatrix} \alpha_i \\ \beta_i \end{pmatrix} = N \begin{pmatrix} \begin{matrix} \alpha_0 \\ \beta_0 \end{matrix} , W \end{pmatrix} \tag{9.2}$$

The priors for:

$$\alpha_0, \beta_0 = N(\gamma, C)$$

$$\Sigma^{-1} = W(\rho R)^{-1}, \rho)$$

$$\sigma^2 = IG(\nu_0/2, \nu_0\tau_0^2/2)$$

Where IG denotes the inverted gamma distribution. Gelfand et al. (1990) use $C^{-1} = 0, \nu_0 = 0, \rho = 2$ and $R = diag(100, 0.1)$ to reflect a vague prior. Letting $V = (30\Sigma^{-1} + C^{-1})^{-1}$, the following sequence of conditional distributions for the parameters in the model form the Gibbs sampler.

$$p(\alpha_i, \beta_i | \ldots) = N\{(G(\sigma^{-2}X_i'Y_i + \Sigma^{-1}\mu), G\},$$

where: $G = (\sigma^{-2}X_i'X_i + \Sigma^{-1})^{-1}$

$$p(\mu | \ldots) = N\{V(30\Sigma^{-1}\bar{\theta} + C^{-1}\nu), V\}$$

where: $\bar{\theta} = (1/30)\sum_{i=1}^{30}(\alpha_i, \beta_i)'$

$$p(\Sigma^{-1} | \ldots) = W\{[\sum_i(\theta_i - \mu)(\theta_i - \mu)' + \rho R]^{-1}, 30 + \rho\}$$

$$p(\sigma^2|\ldots) = IG\{(n+\nu_0)/2, (1/2)[\sum_i (Y_i - X_i\theta_i)'(Y_i - X_i\theta_i) + \nu_0\tau_0]\}$$

As a demonstration of random draws from the Wishart distribution consider example 9.2, where we generate a symmetric matrix for input and assign $v = 10$ degrees of freedom. The mean of the Wishart should equal v times the input matrix. In the example, we carry out 1000 draws and average over the random matrix draws to find the mean which is compared to the theoretical mean.

```
% ----- Example 9.2 Random Wishart draws
ndraws = 1000;
n = 100; k=5;
x = randn(n,5); xpx = x'*x; xpxi = inv(xpx); v = 10;
w = zeros(k,k);
for i=1:ndraws;
w = w + wish_rnd(xpx,v);
end;
fprintf('mean of wishart should = \n');
mprint(v*xpx);
fprintf('mean of wishart draws = \n');
mprint(w/ndraws);
```

The output from example 9.2 is:

```
mean of wishart should =
  749.0517  -20.5071    6.8651   75.1039  -57.3764
  -20.5071  899.6492   13.0913   30.3349 -121.7186
    6.8651   13.0913  912.6154   -3.2621 -186.6620
   75.1039   30.3349   -3.2621  985.6231   27.4726
  -57.3764 -121.7186 -186.6620   27.4726  945.3465

mean of wishart draws =
  765.3774  -13.8467   26.8851   92.3882  -61.7533
  -13.8467  946.6986   43.2879   26.6926 -129.1496
   26.8851   43.2879  941.1585    0.2385 -185.0185
   92.3882   26.6926    0.2385  997.2087   36.5898
  -61.7533 -129.1496 -185.0185   36.5898  886.5470
```

Another set of specialized functions, **nmlt_rnd** and **nmrt_rnd** were used to produce left- and right-truncated normal draws when Gibbs sampling estimates for the probit and tobit models. Example 9.3 shows the use of these functions and produces a series of three histograms based on draws from the truncated normal distributions. It should be noted that one can implement these function by simply drawing from a normal distribution and rejecting draws that don't meet the truncation restrictions. This however is not very efficient and tends to produce a very slow routine. The functions **nmlt_rnd** and **nmrt_rnd** are based on FORTRAN code that implements an efficient method described in Geweke (1991).

```
% ----- Example 9.3 Left- and right-truncated normal draws
n = 1000; x = zeros(n,1);
% generate from -infinity < 0
for i=1:n; x(i,1) = nmrt_rnd(0); end;
subplot(3,1,1), hist(x,20); xlabel('right-truncated at zero normal');
% generate from 1 < infinity
for i=1:n; x(i,1) = nmlt_rnd(1); end;
subplot(3,1,2), hist(x,20); xlabel('left-truncated at +1 normal');
% generate from -1 < +infinity
for i=1:n; x(i,1) = nmlt_rnd(-1); end;
subplot(3,1,3), hist(x,20); xlabel('left-truncated at -1 normal');
```

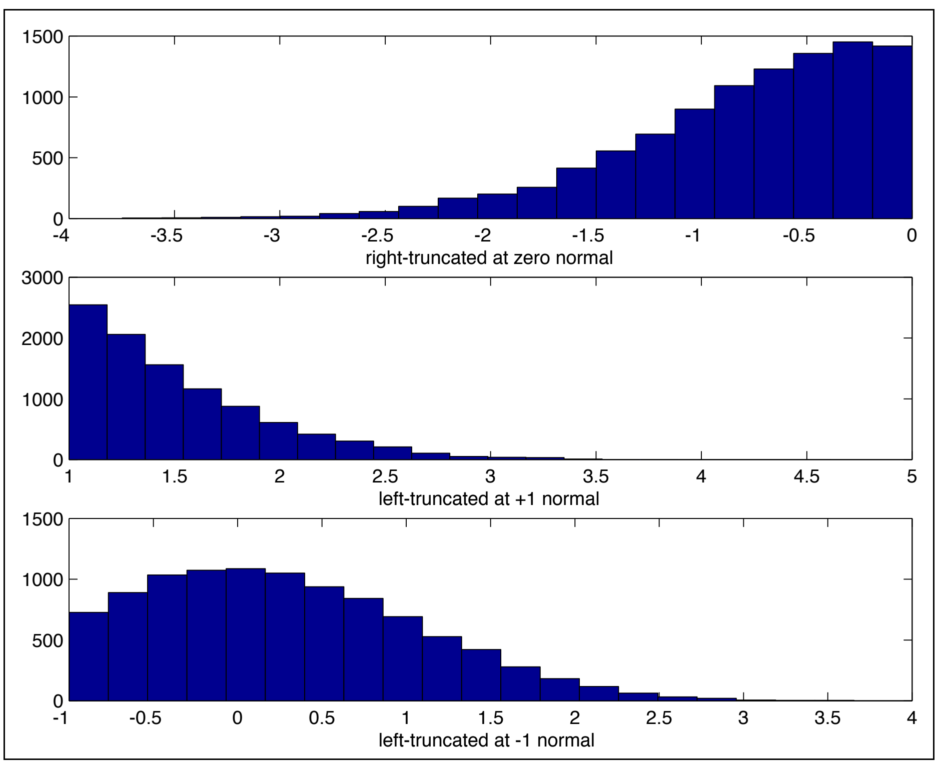

Figure 9.2: Histograms of truncated normal distributions

Example 9.4 constructs a program to explore the improvement in speed of **nmrt_rnd** over the inefficient approach based on rejection sampling of draws from a normal. The program produces a set of 10 draws using 7 different truncation limits ranging from -3 to +3. In addition to using the **nmrt_rnd** function, we include a 'while loop' that simply draws from the random normal centered on zero until it collects a sample of 10 draws that meet the truncation restriction.

Consider that simple rejection will encounter very few normal draws that take on values less than -3, requiring a large amount of time. Further, for cases where the truncation limit is -3.5 or -4, we would have an even longer wait.

```
% ----- Example 9.4 Rejection sampling of truncated normal draws
tt=-3:3; m = length(tt);n = 10;
x = zeros(n,m); x2 = zeros(n,m); time = zeros(m,2);
for j=1:m;
% generate from -infinity to tt(j)
% using nmrt_rnd function and keep track of time
t0 = clock;
for i=1:n; x(i,j) = nmrt_rnd(tt(j)); end;
time(j,1) = etime(clock,t0);
% generate from -infinity to tt(j) using rejection
%  of draws that don't meet the truncation constraint
cnt=1;
t0 = clock;
while cnt <= n
tst = randn(1,1);
 if tst < tt(j)
 x2(cnt,j) = tst;
 cnt = cnt+1;
 end;
end;
time(j,2) = etime(clock,t0);
end;
time
```

The timing results from running the program are shown below, indicating that for the case of truncation at -3, the simple rejection approach took over 80 times as long as the **nmrt_rnd** function to produce 10 draws. Similarly, for the truncation limit at -2, rejection sampling took 7 times as long as **nmrt_rnd**. For truncation limits ranging from -1 to 3 rejection sampling was equal or up to 3 times faster than the **nmrt_rnd** function.

```
    Rejection sampling versus nmrt_rnd function
    (time in seconds)
truncation    nmrt_rnd  rejection
   at
   -3         0.0763    6.2548
```

```
 -2          0.0326    0.2305
 -1          0.0340    0.0301
  0          0.0321    0.0125
  1          0.0244    0.0091
  2          0.0248    0.0081
  3          0.0294    0.0092
```

Another special purpose distribution function is **norm_crnd** that provides random draws from a contaminated normal distribution. This distribution takes the form of a mixture of two normals: $(1-\gamma)N(0,1)+\gamma N(0,\sigma)$. Typical values for the parameters in the mixture would be: $\gamma = 0.05$ and a large variance $\sigma^2 = 10$. This produces a distribution with outliers exhibiting fatter tails than those of the standard normal distribution.

A graphical illustration is provided in Figure 9.3, where draws from the contaminated normal distribution are plotted alongside those from a standard normal. The fatter tails associated with the contaminated normal are clearly visible in the figure.

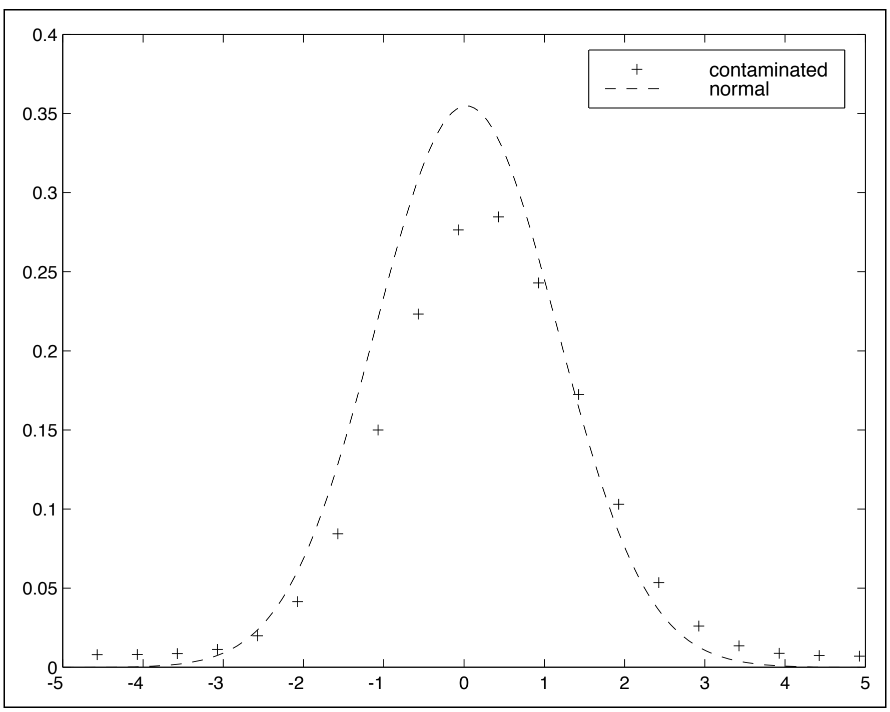

Figure 9.3: Contaminated normal versus a standard normal distribution

9.3 Chapter summary

A large number of algorithms for generating random deviates, evaluating the probability density, cumulative density and quantile functions exits for a host of statistical distributions. These have been collected in a library of functions that employ a common naming convention. The Gibbs sampling estimation methods discussed in Chapters 6 and 7 rely heavily on this library of distribution functions.

In addition to the library of 12 statistical distributions, some specialized functions exist that are useful in computing probabilities for hypothesis testing and generating various results related to statistical distributions.

Chapter 9 Appendix

Distribution functions discussed in this chapter are in the subdirectory **distrib**.

```
Distribution functions library

------- pdf, cdf, inverse functions -----------

beta_cdf   - beta(a,b) cdf
beta_inv   - beta inverse (quantile)
beta_pdf   - beta(a,b) pdf
bino_cdf   - binomial(n,p) cdf
bino_inv   - binomial inverse (quantile)
bino_pdf   - binomial pdf
chis_cdf   - chisquared(a,b) cdf
chis_inv   - chi-inverse (quantile)
chis_pdf   - chisquared(a,b) pdf
chis_prb   - probability for chi-squared statistics
fdis_cdf   - F(a,b) cdf
fdis_inv   - F inverse (quantile)
fdis_pdf   - F(a,b) pdf
fdis_prb   - probability for F-statistics
gamm_cdf   - gamma(a,b) cdf
gamm_inv   - gamma inverse (quantile)
gamm_pdf   - gamma(a,b) pdf
hypg_cdf   - hypergeometric cdf
hypg_inv   - hypergeometric inverse
hypg_pdf   - hypergeometric pdf
logn_cdf   - lognormal(m,v) cdf
logn_inv   - lognormal inverse (quantile)
logn_pdf   - lognormal(m,v) pdf
logt_cdf   - logistic cdf
logt_inv   - logistic inverse (quantile)
logt_pdf   - logistic pdf
norm_cdf   - normal(mean,var) cdf
norm_inv   - normal inverse (quantile)
norm_pdf   - normal(mean,var) pdf
norm_prb   - normal probability
```

```
normt_inv  - truncated normal inverse
normrt_inv - right-truncated normal inverse
normlt_inv - left-truncated normal inverse
pois_cdf   - poisson cdf
pois_inv   - poisson inverse
pois_pdf   - poisson pdf
stdn_cdf   - std normal cdf
stdn_inv   - std normal inverse
stdn_pdf   - std normal pdf
tdis_cdf   - student t-distribution cdf
tdis_inv   - student t inverse (quantile)
tdis_pdf   - student t-distribution pdf
tdis_prb   - probability for t-statistics

------- random samples -----------

beta_rnd  - random beta(a,b) draws
bino_rnd  - random binomial draws
chis_rnd  - random chi-squared(n) draws
fdis_rnd  - random F(a,b) draws
gamm_rnd  - random gamma(a,b) draws
hypg_rnd  - random hypergeometric draws
logn_rnd  - random log-normal draws
logt_rnd  - random logistic draws
nmlt_rnd  - left-truncated normal draw
nmrt_rnd  - right-truncated normal draw
normt_rnd - truncated normal random draws
norm_crnd - contaminated normal random draws
norm_rnd  - multivariate normal draws
pois_rnd  - poisson random draws
tdis_rnd  - random student t-distribution draws
unif_rnd  - random uniform draws (lr,rt) interval
wish_rnd  - random Wishart draws

-------- demonstration and test programs --------

beta_d    - demo of beta distribution functions
bino_d    - demo of binomial distribution functions
chis_d    - demo of chi-squared distribution functions
fdis_d    - demo of F-distribution functions
gamm_d    - demo of gamma distribution functions
hypg_d    - demo of hypergeometric distribution functions
logn_d    - demo of lognormal distribution functions
logt_d    - demo of logistic distribution functions
normt_d  - demo of normt_rnd
normlt_d - demo of normlt_rnd
normrt_d - demo of normrt_rnd
pois_d    - demo of poisson distribution functions
stdn_d    - demo of std normal distribution functions
```

```
 tdis_d    - demo of student-t distribution functions
 trunc_d   - demo of truncated normal distribution function
 unif_d    - demo of uniform random distribution function

-------- support functions --------

 betacfj   - used by fdis_prb
 betai     - used by fdis_prb
 bincoef   - binomial coefficients
 com_size  - test and converts to common size
 gammalnj  - used by fdis_prb
 is_scalar - test for scalar argument
```

Chapter 10

Optimization functions library

The **optimization function library** contains routines for maximum likelihood estimation of econometric models. Given a likelihood function which depends on the data as well as the parameters of the estimation problem, these routines will find a set of parameter values that maximize the likelihood function. In addition to discussing and illustrating use of the functions in the **optimization function library**, we also show how to use the MATLAB simplex optimization algorithms **fmin** and **fmins** as well as Em algorithms to solve for maximum likelihood estimates in econometric models. An example that demonstrates how to incorporate a new optimization algorithm obtained from the Internet into the library is also provided.

Unlike Gauss software, MATLAB does not contain a single 'maxlik' function that serves as an 'all purpose' routine for solving econometric maximum likelihood optimization problems. In an attempt to remedy this, a function **maxlik** was devised for MATLAB. Unfortunately, this function is nowhere near as robust or professionally crafted as the Gauss function by the same name. This function as well as univariate and multivariate simplex routines, algorithms from *Numerical Recipes* that have been re-coded for MATLAB, some Internet optimization functions written for MATLAB and a general purpose function **minz** contributed by Mike Cliff form the basis of the optimization library,.

The MathWorks sells an optimization toolbox that contains a host of algorithms for solving constrained and unconstrained optimization problems. These functions are not oriented toward econometric models, but rather solution of very general optimization problems. To use these routines in solving econometric optimization problems, an interface similar to that described in Section 10.3 would need to be developed to adapt the algorithms for use in an econometric likelihood function setting.

Section 10.1 illustrates the use of a univariate simplex algorithm **fmin** from the MATLAB toolbox. This type of algorithm is useful for cases where the optimization problem involves a single parameter, a case often encountered with concentrated likelihood functions. It also demonstrates the use of a multivariate simplex optimization function **fmins** from the MATLAB toolbox.

Section 10.2 demonstrates use of the EM-algorithm approach to solving maximum likelihood estimation problems. Finally, Section 10.3 deals with the most general case where the likelihood function is optimized with respect to a multivariate vector of parameters using numerical gradient methods.

10.1 Simplex optimization

Simplex optimization has an advantage in that it does not require computation of the derivatives of the function being optimized. It also has a disadvantage in that the usual estimates of dispersion for the parameters in the model obtained from the numerical hessian matrix in gradient methods are not available. The MATLAB toolbox contains both a univariate simplex algorithm **fmin** and a multivariate function **fmins**. The use of the univariate function is taken up in Section 10.1.1 and **fmins** is illustrated in Section 10.1.2.

10.1.1 Univariate simplex optimization

As an example of using the MATLAB **fmin** univariate simplex optimization function to solve an econometric maximum likelihood estimation problem, consider the Box-Cox model. This model relies on a transformation of the sample data vector $y^{(\lambda)} = (y^{\lambda} - 1)/\lambda$ for the case where $\lambda \neq 0$ and $y^{(\lambda)} = ln(y)$ when $\lambda = 0$. The model may also involve an identical transformation of the explanatory variables matrix, which we denote $X^{(\lambda)}$. A more general model involves different values, λ_1, λ_2 associated with the $y^{(\lambda_1)}$-transformation and $X^{(\lambda_2)}$-transformations, and a very general model allows for individual $\lambda_{xi}, i = 1, \ldots, k$ for each explanatory variable vector in the sample data matrix X.

We focus initially on the simple model involving a single parameter λ. This model produces a log-linear model when $\lambda_1 = \lambda_2 = 0$, a semi-log model when $\lambda_1 = 1, \lambda_2 = 0$, as well as more general flexible functional forms associated with values like, $\lambda = 1/2$, or $\lambda = \sqrt{2}$. Some of the functional forms associated with values for λ_1, λ_2 are:

$$\begin{aligned}
ln\ y_i &= \beta_0 + ln\ X_i\beta + \varepsilon_i, \quad \lambda_1 = 0, \lambda_2 = 0 & (10.1)\\
y_i &= \beta_0 + ln\ X_i\beta + \varepsilon_i, \quad \lambda_1 = 1, \lambda_2 = 0 & (10.2)\\
y_i &= \beta_0 + X_i\beta + \varepsilon_i, \quad \lambda_1 = 1, \lambda_2 = 1 & (10.3)
\end{aligned}$$

The log likelihood function is concentrated with respect to $\beta(\lambda), \sigma(\lambda)$, producing a function of only a single parameter λ:

$$L(\lambda|y, X) = \text{const} + (\lambda - 1)\sum_{i=1}^{n} ln\ y_i - (n/2)\ ln\ \hat{\sigma}^2(\lambda) \tag{10.4}$$

where:

$$\begin{aligned}
\hat{\sigma}^2(\lambda) &= [y^{(\lambda)} - X^{(\lambda)}\hat{\beta}(\lambda)'(y^{(\lambda)} - X^{(\lambda)}\hat{\beta}(\lambda)]/n & (10.5)\\
\hat{\beta}(\lambda) &= (X^{(\lambda)\prime}X^{(\lambda)})^{-1}X^{(\lambda)\prime}y^{(\lambda)}
\end{aligned}$$

To minimize the log of this likelihood function with respect to the parameter λ we can use the MATLAB function **fmin** that implements a simplex optimization procedure. This function also allows us to set lower and upper limits on the parameter λ to which the simplex search will be constrained. It is often the case that, values of $-2 \leq \lambda \leq 2$ are thought to represent a reasonable range of feasible values for the parameter λ in the Box-Cox model.

Our first task in using **fmin** is to write a log-likelihood function that evaluates the concentrated log-likelihood for any value of the parameter λ and returns a scalar value equal to the negative of the log-likelihood function. (Minimizing the negative log-likelihood is equivalent to maximizing the log-likelihood.) This function is shown below:

```
function like = box_lik(lam,y,x,model);
% PURPOSE: evaluate Box-Cox model concentrated likelihood function
%----------------------------------------------------
% USAGE: like = box_lik(lam,y,x,model)
% where:  lam = box-cox parameter (scalar)
%           y = dependent variable vector (un-transformed)
%           x = explanatory variables matrix (un-transformed)
%       model = 0 for y-transform only, 1 for y,x both transformed
```

```
% NOTE: x should contain intercept vector in 1st column (if desired)
%---------------------------------------------------
% RETURNS: like = (a scalar) = -log likelihood function
%---------------------------------------------------
[n k] = size(x); ys = boxc_trans(y,lam);
if model == 1 % user wants to transform both y,x
   % see if an intercept term exists in the model
   iota = x(:,1); ifind = find(iota == 1);
   if isempty(ifind), xs = boxc_trans(x,lam); % no intercept
   else, if length(ifind) == n,               % an intercept
          xtrans = boxc_trans(x(:,2:k),lam);
          xs = [ones(n,1) xtrans];
         else, xs = boxc_trans(x,lam);        % no intercept
         end;
   end;
elseif model == 0, xs = x; % transform only y-vector
end;
bhat = inv(xs'*xs)*xs'*ys; e = ys - xs*bhat; sige = (e'*e)/n;
like = (lam - 1)*sum(log(y)) -(n/2)*log(sige); like = -like;
```

The function relies on another function **boxc_trans** to carry out the Box-Cox data transformation. It also contains an argument 'model' that allows for a case ('model=0') where the y variable alone is transformed and another case ('model=1') where both y and the X variables are transformed.

The function **boxc_trans** is:

```
function z = boxc_trans(x,lam)
% PURPOSE: compute box-cox transformation
%---------------------------------------------------
% USAGE: bdata = boxc_trans(data,lam)
% where:    lam  = scalar transformation parameter
%           data = matrix nobs x k
%---------------------------------------------------
% RETURNS: bdata = data matrix box-cox transformed
[n k] = size(x); z = zeros(n,k); iota = ones(n,1);
 for i=1:k;
  if lam ~= 0, z(:,i) = (x(:,i).^lam - iota)/lam;
  else, z(:,i) = log(x(:,i)); end;
 end;
```

Now we can turn attention to use of the MATLAB simplex optimization function **fmin**, which can be called with: a string containing the function name 'box_lik', an upper and lower limit for the simplex optimization search over values of λ ('lamlo','lamup'), a 4x1 vector of optimization options ('foptions'), and arguments that will be passed along to our likelihood function 'box_lik'. In this example, we wish to pass 'y,x,model' as arguments to the 'box_lik' function. An example of the call is:

```
[lam options] = fmin('box_lik',lamlo,lamup,foptions,y,x,model);
```

This call is made by a Box-Cox regression function **box_cox** which allows the user to input a sample data vector y and matrix X as well as lower and upper limits on the parameter λ, a flag to indicate whether the transformation should be applied to just y or both the y and X data, and optional optimization options. The function documentation is:

```
PURPOSE: box-cox regression using a single scalar transformation
         parameter for both y and (optionally) x
---------------------------------------
USAGE: results = box_cox(y,x,lam,lam_lo,lam_up,model,foptions)
     where: y = dependent variable vector
            x = explanatory variables matrix
                (intercept vector in 1st column --- if desired)
       lam_lo = scalar, lower limit for simplex search
       lam_up = scalar, upper limit for simplex search
       model  = 0 for y-transform only
              = 1 for both y, and x-transform
       foptions  = (optional) a 4x1 vector of optimization information
       foptions(1) = flag to display intermediate results while working
       foptions(2) = convergence for simplex (default = 1e-4)
       foptions(3) = convergence for function value (default = 1e-4)
       foptions(4) = maximum number of iterations (default = 500)
---------------------------------------
RETURNS: a structure:
       results.meth  = 'boxcox'
       results.beta  = bhat estimates
       results.lam   = lamda estimate
       results.tstat = t-stats for bhat
       results.yhat  = yhat (box-cox transformed)
       results.resid = residuals
       results.sige  = e'*e/(n-k)
       results.rsqr  = rsquared
       results.rbar  = rbar-squared
       results.nobs  = nobs
       results.nvar  = nvars
       results.y     = y data vector (box-cox transformed)
       results.iter  = # of iterations
       results.like  = -log likelihood function value
--------------------------------------------------
NOTE: uses MATLAB simplex function fmin
--------------------------------------------------
SEE ALSO: prt(results), plt(results)
---------------------------------------------------
```

The function **box_cox** is responsible for checking that the input data are positive (as required by the Box-Cox regression procedure) and providing default optimization options for the user if they are not input. The function also constructs estimates of $\hat{\beta}, \hat{\sigma}^2$, $t-$statistics, $R-$squared, predicted values, etc., based on the maximum likelihood estimate of the parameter λ. These values are returned in a structure variable suitable for use with the **prt_reg** function for printing results and **plt_reg** function for plotting actual vs. predicted values and residuals.

Example 10.1 demonstrates use of the function **box_cox**. In the example, two models are generated. One model involves a y-transformation only such that the estimated value of λ should equal zero. This is accomplished by transforming a typical regression generated y-variable using the exponential function, and carrying out the Box-Cox regression on the transformed $y = \exp(y)$. The second model performs no transformation on y or X, so the estimated value of λ should equal unity.

```
% --- Example 10.1 Simplex max likelihood estimation for Box-Cox model
% generate box-cox model data
n = 100; k = 2; kp1 = k+1;
x = abs(randn(n,k)) + ones(n,k)*10;
btrue = ones(k,1); epsil = 0.2*randn(n,1); x = [ones(n,1) x];
y = 10*x(:,1) + x(:,2:k+1)*btrue + epsil;
ycheck = find(y > 0); % ensure y-positive
if length(ycheck) ~= n,error('all y-values must be positive'); end;
yt = exp(y); % should produce lambda = 0 estimate
model = 0;   % transform only y-variable
result = box_cox(yt,x,-2,2,model); prt(result);
model = 1;   % transform both y,x variables
xcheck = find(x > 0);
if length(xcheck) ~= n*kp1, error('all x-values must be positive'); end;
yt = y; xt = x; % should produce lambda=1 estimate
result = box_cox(yt,xt,-2,2,model); prt(result); plt(result);
```

The printed output from the **box_cox** function indicates λ estimates near zero for the first regression and unity for the second. The parameter estimates are also reasonably close to the values used to generate the data.

```
Box-Cox 1-parameter model Estimates
R-squared      =    0.9419
Rbar-squared   =    0.9407
sigma^2        =    0.0909
Lambda         =    0.0155
Nobs, Nvars    =    100,     3
# iterations   =     13
-log like      =        2977.9633
***************************************************************
Variable        Coefficient      t-statistic    t-probability
```

```
variable 1         4.506392         5.029223         0.000002
variable 2         1.765483        29.788761         0.000000
variable 3         1.592008        28.310745         0.000000

Box-Cox 1-parameter model Estimates
R-squared      =    0.9415
Rbar-squared   =    0.9403
sigma^2        =    0.0185
Lambda         =    0.9103
Nobs, Nvars    =    100,     3
# iterations   =      8
-log like      =       -168.62035
***************************************************************
Variable        Coefficient      t-statistic    t-probability
variable 1         8.500698        21.621546         0.000000
variable 2         0.983223        29.701359         0.000000
variable 3         0.886072        28.204172         0.000000
```

One problem that arises with univariate simplex optimization algorithm is that the numerical hessian matrix is not available to provide estimates of dispersion for the parameters. This presents a problem with respect to the parameter λ in the Box-Cox model, but asymptotic likelihood ratio statistics based on -2 ln L are traditionally used to test hypotheses regarding λ

One way to overcome this limitation is to rely on the theoretical information matrix to produce estimates of dispersion. An example of this is the function **sar** that implements maximum likelihood estimation for the spatial autoregressive model (already discussed in Chapter 6 in the context of Gibbs sampling):

$$y = \rho W y + X\beta + \varepsilon \tag{10.6}$$

This model involves a cross-section of observations made at various points in space. The parameter ρ is a coefficient on the spatially lagged dependent variable Wy, that reflects the influence of neighboring observations on variation in individual elements of the dependent variable y. The model is called a first order spatial autoregression because its represents a spatial analogy to the first order autoregressive model from time series analysis, $y_t = \rho y_{t-1} + \varepsilon_t$. The matrix W represents a standardized spatial contiguity matrix such that the row-sums are unity. The product Wy produces an explanatory variable equal to the mean of observations from contiguous areas.

Anselin (1988) provides the theoretical information matrix for this model along with a concentrated likelihood function that depends on the single parameter ρ. Using the theoretical information matrix, we can construct a dispersion estimate and associated $t-$statistic for the important spatial lag parameter ρ. An advantage of using the univariate simplex optimization algorithm in this application is that the value of ρ must lie between:

$$1/\lambda_{min} < \rho < 1/\lambda_{max}$$

where λ_{min} and λ_{max} are the minimum and maximum eigenvalues of the standardized spatial weight matrix W. We can impose this restriction using **fmin** as already demonstrated.

Given a maximum likelihood estimate for ρ, we can compute estimates for β, σ^2, and construct the theoretical information matrix which we use to produce a variance-covariance matrix for the estimates. The following code fragment from the function **sar** provides an example of this approach.

```
% construct information matrix, (page 80-81 Anselin, 1982)
B = eye(n) - p*W;
BI = inv(B); WB = W*BI;
pterm = trace(WB*WB + WB*WB');
xpx = zeros(nvar+2,nvar+2);               % bhat,bhat
xpx(1:nvar,1:nvar) = (1/sige)*(x'*x);     % bhat,rho
xpx(1:nvar,nvar+1) = (1/sige)*x'*W*BI*x*bhat;
xpx(nvar+1,1:nvar) = xpx(1:nvar,nvar+1)'; % rho,rho
xpx(nvar+1,nvar+1) = (1/sige)*bhat'*x'*BI'*W'*W*BI*x*bhat + pterm;
xpx(nvar+2,nvar+2) = n/(2*sige*sige);     %sige,sige
xpx(nvar+1,nvar+2) = (1/sige)*trace(WB);  % rho,sige
xpx(nvar+2,nvar+1) = xpx(nvar+1,nvar+2);
xpxi = inv(xpx);
tmp = diag(xpxi(1:nvar+1,1:nvar+1));
bvec = [results.beta
        results.rho];
results.tstat = bvec./(sqrt(tmp));
```

It should be noted that a similar approach could be taken with the Box-Cox model to produce an estimate for the dispersion of the transformation parameter λ. Fomby, Hill and Johnson (1984) provide the theoretical information matrix for the Box-Cox model.

Summarizing our discussion of the MATLAB function **fmin**, we should not have a great deal of difficulty since single parameter optimization problems should be easy to solve. The ability to impose restrictions on the domain of the parameter over which we are optimizing is often useful (or required) given theoretical knowledge of the likelihood function, or the nature of the problem being solved. Despite the seeming simplicity of a univariate likelihood function that depends on a single parameter, a number of problems in econometrics take this form once other parameters are concentrated out of the likelihood function. The greatest problem with this approach to optimization is that no measure of dispersion for the estimated parameter is available without knowledge of the theoretical information matrix.

10.1.2 Multivariate simplex optimization

As an example of multivariate simplex optimization, we illustrate the case of maximum likelihood estimation of the AR(1) autoregressive error model:

$$y_t = X\beta + u_t \tag{10.7}$$
$$u_t = \rho u_{t-1} + \varepsilon \tag{10.8}$$

We maximize the likelihood function over the parameters β, ρ in the model (with σ^2 concentrated out). Another example of using **fmins** is the function **sac** that estimates a two-parameter version of the spatial autoregressive model. We also implement a Box-Cox model with separate parameters for the $y-$ and $X-$transformations using this approach in a function **box_cox2**.

For the case of the autoregressive error model, we set up the likelihood function in a file **ar1_like** where we have concentrated out the parameter σ^2:

```
function llike = ar1_like(param,y,x)
% PURPOSE: log-likelihood for a regression model with AR(1) errors
%---------------------------------------------------
% USAGE:    like = ar1_like(b,y,x)
% where:     b = parameter vector (m x 1)
%            y = dependent variable vector (n x 1)
%            x = explanatory variables matrix (n x m)
%---------------------------------------------------
% NOTE: this function returns a scalar equal to -log(likelihood)
%       b(1,1) contains rho parameter
%       sige is concentrated out
%---------------------------------------------------
% REFERENCES: Green, 1997 page 600
%---------------------------------------------------
[n k] = size(x);
rho = param(1,1); beta = param(2:2+k-1,1);
ys = y - rho*lag(y); xs = x - rho*lag(x);
ys(1,1) = sqrt(1-rho*rho)*y(1,1);
xs(1,:) = sqrt(1-rho*rho)*x(1,:);
term1 = -(n/2)*log((ys - xs*beta)'*(ys - xs*beta));
term2 = 0.5*log(1-rho*rho);
like = term1+term2;
llike = -like;
```

A regression function **ols_ar1** performs error checking on the user input, establishes optimization options for the user, and supplies starting values based on the Cochrane-Orcutt estimation function **olsc**. The code to carry out Cochrane-Orcutt estimation, set up optimization options, and call **fmins** looks as follows:

```
% use cochrane-orcutt estimates as initial values
reso = olsc(y,x);
parm = zeros(nvar+2,1);
parm(1,1) = reso.rho;              % initial rho
parm(2:2+nvar-1,1) = reso.beta;    % initial bhat's
parm(2+nvar,1) = reso.sige;        % initial sigma
options(1,1) = 0;      % no print of intermediate results
options(2,1) = 0.0001; % simplex convergence criteria
options(3,1) = 0.0001; % convergence criteria for function value
options(4,1) = 1000;   % default number of function evaluations
[beta foptions] = fmins('ar1_like',parm,options,[],y,x);
niter = foptions(1,10);
llike = foptions(1,8);
rho = beta(1,1);
```

As in the example for the function **sar**, we obtain an estimate of the precision of the important parameter ρ using the theoretical information matrix (in code not shown here).

Some points to note about using the simplex optimization algorithm **fmins**. Good starting values are often necessary to produce a reasonable solution and the time necessary depends to a large extent on the quality of these initial values. The example provided is somewhat unrealistic in that the superior quality of Cochrane-Orcutt estimates as starting values creates a situation where the function **fmins** has to do a minimum of work.

In more realistic applications such as the Box-Cox model with one parameter,for the $y-$transformation and another for the $X-$transformation, the function **fmins** performs more slowly and less accurately. As an illustration of this, you can experiment with the function **box_cox2** that implements the two-parameter Box-Cox model using **fmins**.

10.2 EM algorithms for optimization

Another approach to solving maximum likelihood estimation problems is to rely on an EM, or expectation-maximization algorithm of Dempster et al. (1977). This algorithm proceeds by successively maximizing the current conditional expectation of the log likelihood function, where missing data (or parameters) are replaced by their expectation prior to maximization. Given estimates from the maximization (the M-step), a new set of expected values are computed and substituted in the likelihood function (the E-step). This process continues until convergence to a set of values that do not increase the likelihood. An advantage of the EM algorithm is that each step is guaranteed to increase the likelihood function value. A disadvantage is that the algorithm tends to move slowly during the final steps.

As an example of this type of problem, consider a simple switching regression model:

$$
\begin{aligned}
y_{1t} &= X_{1t}\beta_1 + \varepsilon_{1t} && (10.9)\\
y_{2t} &= X_{2t}\beta_2 + \varepsilon_{2t}\\
y_{3t} &= X_{3t}\beta_3 + \varepsilon_{3t}\\
y_t &= y_{1t} \ \text{ if } \ y_{3t} \le 0\\
y_t &= y_{2t} \ \text{ if } \ y_{3t} > 0\\
\varepsilon_{1t} &\sim N(0, \sigma_1^2)\\
\varepsilon_{2t} &\sim N(0, \sigma_2^2)\\
\varepsilon_{3t} &\sim N(0, 1) && (10.10)
\end{aligned}
$$

This model suggests that the observed value of y_t is generated by two different latent unobservable variables y_{1t}, y_{2t}. The variable y_{3t} is also a latent variable that serves to classify y_t into either regime.

An EM algorithm for this model involves iteratively estimating a system that relies on simple least-squares (OLS) and weighted least-squares (WLS) regressions. Given initial values for the parameters in the model: $\beta_1, \beta_2, \beta_3, \sigma_1, \sigma_2$, we can begin by calculating weights for the WLS regression using:

$$
\begin{aligned}
w_1(y_t) &= \lambda_t[f_1(y_t)/h(y_t)] && (10.11)\\
w_2(y_t) &= (1-\lambda_t)[f_2(y_t)/h(y_t)]
\end{aligned}
$$

Where w_1 denotes a scalar weight associated with observation t and $f_i, i = 1, 2$ is the normal probability density function for each regime evaluated at observation t:

$$
f_i(y_t) = (2\pi)^{-1/2}\sigma_i^{-1}\exp\{-(y_t - x_t'\beta_i)^2/2\sigma_i^2\} \qquad (10.12)
$$

λ_t represents the cumulative normal density function of ε_{3t}, which we evaluate for each observation t using: $\Phi(-x'_{3t}\beta_3)$ to determine the probability that $y_t = y_{1t}$. Finally, $h(y_t)$ represents the probability density function for the observed variable y_t calculated from:

$$h(y_t) = \lambda_t f_1(y_t) + (1 - \lambda_t) f_2(y_t) \tag{10.13}$$

Given the above procedure for calculating weights, the EM-algorithm proceeds as follows:

1. Using the initial values β_i, σ_i, determine the weights, w_{1i} as discussed above for regimes $i = 1, 2$ and form: $W_i = \text{diag}(w_{1i}, w_{2i}, \ldots, w_{ni})$.

2. Perform WLS regressions:

$$\beta_i = [X_i' W_i X_i]^{-1} [X_i' W_i y] \tag{10.14}$$

3. Compute a value for β_3 using and OLS regression:

$$\begin{aligned} \beta_3 &= (X_3' X_3)^{-1} X_3' \varepsilon_3 \qquad (10.15) \\ \varepsilon_{3t} &= x_{3t}' \beta_3 - w_1(y_t)(f_3(0)/\lambda_t) + w_2(y_t) f_3(0)/(1 - \lambda_t) \end{aligned}$$

4. Compute values for $\sigma_i^2, i = 1, 2$ using the WLS regressions:

$$\sigma_i^2 = [1/\sum w_i(y_t)](y - X_i\beta_i)' W_i (y - X_i\beta_i) \tag{10.16}$$

5. Go to step 1, and replace the initial values with the new values of β_i, σ_i.

The routine first generates a set of weights based on arbitrary starting values for all parameters in the model. Given these weights, we can determine the unobserved y_1, y_2, y_3 values and produce estimates $\beta_1, \beta_2, \beta_3, \sigma_1, \sigma_2$ that maximize the likelihood function.

Note that by replacing the unobservable y_{3t} with the conditional expectation based on y_t values (represented by ε_{3t}), the problem of classifying observed y_t values into the two regimes is simple. Further, the task of maximizing the likelihood function with respect to $\beta_i, i = 1, 2, 3, \sigma_i, i = 1, 2$ involves relatively simple OLS and WLS regressions. Given the estimated $\beta_i, i = 1, 2, 3, \sigma_i, i = 1, 2$, a new set of weights can be produced and the process repeated until convergence in the estimated parameters occurred.

This procedure is implemented in the MATLAB function **switch_em** which has the following documentation:

```
 PURPOSE: Switching Regime regression (EM-estimation)
          y1 = x1*b1 + e1
          y2 = x2*b2 + e2
          y3 = x3*b3 + e3; e3 =N(0,1)
          y = y1 if y3 <= 0, y = y2 if y3 > 0
---------------------------------------------------
 USAGE: results = switch_em(y,x1,x2,x3,crit)
 where: y = dependent variable vector (nobs x 1)
        x1 = independent variables matrix (nobs x k1)
        x2 = independent variables matrix (nobs x k2)
        x3 = independent variables matrix (nobs x k3)
        b1 = (optional) initial values for b1
        b2 = (optional) initial values for b2
        b3 = (optional) initial values for b3
      crit = (optional) convergence criteria (default = 0.001)
---------------------------------------------------
 RETURNS: a structure
        results.meth  = 'switch_em'
        results.beta1 = bhat1
        results.beta2 = bhat2
        results.beta3 = bhat3
        results.t1    = t-stats for bhat1
        results.t2    = t-stats for bhat2
        results.t3    = t-stats for bhat3
        results.yhat1 = predicted values regime 1
        results.yhat2 = predicted values regime 2
        results.r1    = residuals regime 1
        results.r2    = residuals regime 2
        results.sig1  = e1'*e1/(n1-k1)
        results.sig2  = e2'*e2/(n2-k2)
        results.rsqr1 = rsquared, regime 1
        results.rsqr2 = rsquared, regime 2
        results.nobs  = nobs
        results.k1    = k1, variables in x1
        results.k2    = k2, variables in x2
        results.k3    = k3, variables in x3
        results.nvar  = k1+k2+k3
        results.y     = y data vector
        results.prob1 = probability of regime 1
        results.prob2 = probability of regime 2
        results.iter  = # of iterations in EM-algorithm
        results.crit  = convergence criterion
        results.like  = likelihood function value
 --------------------------------------------------
```

Some points about the code are:

1. We need starting values for the initial parameters $\beta_1, \beta_2, \sigma_1, \sigma_2$. These are based on separate regressions involving the positive and negative y values on the associated X values. The initial values for β_3 are based on a regression of y on X_3. This approach works well when y has a mean close to zero, but another method based on Goldfeld and Quandt's (1973) method of moments estimation would be preferable.

2. During EM-iteration we need to check that the weights associated with one of the two regimes do not become zero for all observations, effectively producing a single regime result. Once the weights for one of the two regimes become zero at all observations, there is no possibility of returning to positive weight values for the zero-weight regime, so we terminate with an error message.

3. We set a default maximum # of iterations and convergence criterion, but the user can change these values as input arguments. In the event that the iterations exceed the maximum, we display a warning (using the MATLAB **warn** function) to the user and break out of the EM-loop. The use of a warning rather than an error (based on a call to the MATLAB **error** function) allows the user to examine a printout of the estimates, and perhaps set better starting values for another run.

4. We return a log likelihood function value by evaluating the likelihood at the converged EM-estimates.

The code for **switch_em** is:

```
% code for EM-estimation of the switching regime regression
if nargin == 4;     bflag = 0; crit = 0.001; maxit = 1000;
elseif nargin == 7; bflag = 1; crit = 0.001; maxit = 1000;
elseif nargin == 8; bflag = 1; maxit = 1000;
elseif nargin == 9; bflag = 1;
else, error('Wrong # of arguments to switch_em');
end;
[n1 k1] = size(x1); [n2 k2] = size(x2); [n3 k3] = size(x3);
if n1 ~= n2,     error('switch_em: x1, x2 have different nobs');
elseif n2 ~= n3, error('switch_em: x2, x3 have different nobs');
end;
nobs = n1; converge = 1.0; iter = 0;
if bflag == 0 % user supplied NO initial values
% get  starting values using positive and negative y's
ypos = find(y > 0); yneg = find(y <= 0);
res = ols(y(ypos,1),x1(ypos,:)); sig1 = res.sige; b1 = res.beta;
res = ols(y(yneg,1),x2(yneg,:)); sig2 = res.sige; b2 = res.beta;
% b3 starting values using ad-hockery
```

```
res = ols(y,x3); b3 = res.beta;
else % user supplied b1,b2,b3 we need to find sig1,sig2
    sig1 = (y - x1*b1)'*(y - x1*b1)/nobs;
    sig2 = (y - x2*b2)'*(y - x2*b2)/nobs;
end;
while converge > crit % start EM-loop
f1=norm_pdf(((y-x1*b1)/sig1)/sig1); f2=norm_pdf(((y-x2*b2)/sig2)/sig2);
lamda=norm_cdf(-x3*b3); h=(lamda.*f1)+((1-lamda).*f2);
w1=lamda.*f1./h; w2=ones(nobs,1)-w1;
ep=x3*b3+(f2-f1).*norm_pdf(-x3*b3./h); t1=sum(w1); t2=sum(w2);
if t1 <= 1; error('switch_em: regime 1 weights all near zero'); end;
if t2 <= 1; error('switch_em: regime 2 weights all near zero'); end;
w1 = sqrt(w1); w2 = sqrt(w2); xx1=matmul(x1,w1); xx2=matmul(x2,w2);
y1=y.*w1; y2=y.*w2; b01 = b1; b02 = b2; b03 = b3;
b1=inv(xx1'*xx1)*xx1'*y1; b2=inv(xx2'*xx2)*xx2'*y2; b3=inv(x3'*x3)*x3'*ep;
sig01 = sig1; sig02 = sig2; sig1=y1-xx1*b1; sig2=y2-xx2*b2;
sig1=sqrt(sig1'*sig1/t1); sig2=sqrt(sig2'*sig2/t2);
% check for convergence
c = max(abs(b1-b01)); c = max(c,max(abs(b2-b02)));
c = max(c,max(abs(b3-b03))); c = max(c,max(sig1-sig01));
converge = max(c,max(sig2-sig02));
iter = iter + 1;
 if iter > maxit; warn('switch_em: max # of iterations exceeded');
 break; end;
end; % end of EM-loop
% compute t-statistics
tmp = sig1*(inv(xx1'*xx1)); results.t1 = b1./sqrt(diag(tmp));
tmp = sig2*(inv(xx2'*xx2)); results.t2 = b2./sqrt(diag(tmp));
tmp = inv(x3'*x3);% sig3=1; results.t3 = b3./sqrt(diag(tmp));
results.meth  = 'switch_em'; % return results
results.iter  = iter; results.crit  = converge;
results.beta1 = b1; results.beta2 = b2; results.beta3 = b3;
results.nobs  = nobs; results.k1 = k1; results.k2 = k2; results.k3 = k3;
results.nvar  = k1+k2+k3;    results.y = y;
results.prob1 = w1.^2;       results.prob2 = w2.^2;
results.yhat1 = xx1*b1;      results.yhat2 = xx2*b2;
results.r1    = y1-xx1*b1;   results.r2 = y2-xx2*b2;
results.sig1  = sig1;        results.sig2 = sig2;
% compute R-squared based on two regimes
sigu1 = 0; sigu2 = 0; y1sum = 0; y2sum = 0; y1s = 0; y2s = 0;
prob1 = results.prob1; nobs1 = 0; nobs2 = 0;
for i=1:nobs
    if prob1(i,1) > 0.5, nobs1 = nobs1+1;
        sigu1 = sigu1 + results.r1(i,1)*results.r1(i,1);
        y1sum = y1sum + y(i,1); y1s = y1s + y1sum*y1sum;
    else, nobs2 = nobs2+1;
        sigu2 = sigu2 + results.r2(i,1)*results.r2(i,1);
        y2sum = y2sum + y(i,1); y2s = y2s + y2sum*y2sum;
    end;
```

```
end;
results.rsqr1 = 1 - sigu1/y1s; results.rsqr2 = 1 - sigu2/y2s;
like = log((1-lamda).*f1 + lamda.*f2); results.like = sum(like);
```

An example program that generates regression data based on two regimes and estimates the parameters using **switch_em** is:

```
% --- Example 10.2 EM estimation of switching regime model
% generate data from switching regression model
nobs = 100; n1 = 3; n2 = 3; n3 = 3; sig1 = 1; sig2 = 1;
b1 = ones(n1,1); b2 = ones(n2,1)*5; b3 = ones(n3,1);
x1 = randn(nobs,n1); x2 = randn(nobs,n2); x3 = randn(nobs,n3);
for i=1:nobs;
 if x3(i,:)*b3 <= 0, y(i,1) = x1(i,:)*b1 + randn(1,1)*2;
 else,               y(i,1) = x2(i,:)*b2 + randn(1,1)*2;
 end;
end;
result = switch_em(y,x1,x2,x3,b1,b2,b3);
vnames1 = strvcat('y1','x1_1','x1_2','x1_3');
vnames2 = strvcat('y2','x2_1','x2_2','x2_3');
vnames3 = strvcat('x3_1','x3_2','x3_3');
vnames = strvcat(vnames1,vnames2,vnames3);
prt(result,vnames);
plt(result,vnames);
```

Example 10.2 generates a set of $y-$values based on $X_3\beta_3 \leq 0$ for regime 1 and otherwise regime 2. The regime 1 relation contains true parameters (β_1) equal to unity, whereas regime 2 sets the parameters (β_2) equal to five. The values used for β_3 were unity, which should produce a roughly equal number of observations in the two regimes, given that the values of X_3 are normally distributed around a mean of zero.

The printout of estimation results from example 10.2 and the plot produced by the **plt** function are shown below.

```
EM Estimates - Switching Regression model
Regime 1 equation
Dependent Variable =            y1
R-squared      =    0.9925
sigma^2        =    4.2517
Nobs, Nvars    =    100,     3
***************************************
        Variable      Coefficient      t-statistic    t-probability
            x1_1       0.92050544       3.40618422       0.00095997
            x1_2       0.61875719       2.25506692       0.02637700
            x1_3       1.15667358       4.21939446       0.00005514
Regime 2 equation
Dependent Variable =            y2
```

```
R-squared      =    0.9934
sigma^2        =    2.4245
Nobs, Nvars    =    100,     3
***************************************
        Variable      Coefficient      t-statistic    t-probability
            x2_1       4.78698864      19.75899413       0.00000000
            x2_2       4.67524699      21.70327247       0.00000000
            x2_3       5.26189140      24.83553712       0.00000000
Switching equation
Conv criterion =   0.00097950
# iterations   =     53
# obs regime 1 =     51
# obs regime 2 =     49
log Likelihood =    -136.7960
Nobs, Nvars    =    100,     3
***************************************
        Variable      Coefficient      t-statistic    t-probability
            x3_1       1.04653177      10.97136877       0.00000000
            x3_2       0.96811598       8.97270638       0.00000000
            x3_3       1.04817888      10.45553196       0.00000000
```

The estimated parameters are close to the true values of one and five for the two regimes as are the parameters of unity used in the switching equation. Figure 10.1 shows the actual y-values versus predictions. The plot shows predictions classified into regimes 1 and 2 based on the probabilities for each regime > 0.5. The results structure returned by **switch_em** returns predicted values for all n observations, but the graph only shows predicted values based on classification by probabilities for the two regimes.

Summarizing the EM approach to optimization, we require a likelihood function that can be maximized once some missing data values or parameters are replaced by their expectations. We then loop through an expectation-maximization sequence where the expected values of the parameters or missing sample data are substituted to produce a full-data likelihood function. The remaining parameters or observations are computed based on maximizing the likelihood function and these estimates are used to produce new expectations for the missing sample data or parameters. This process continues until convergence in the parameter estimates. A number of estimation problems have been structured in a form amenable to EM optimization. For example, Shumway and Stoffer (1982) provide an EM algorithm for estimation time-series state space models, van Norden and Schaller (1993) provide an EM algorithm for estimation of the Markov transition regime-switching model of Hamilton (1989). McMillen (1992) sets forth an EM approach to estimating spatial autoregressive logit/probit models.

An interesting point is that for most cases where an EM approach to es-

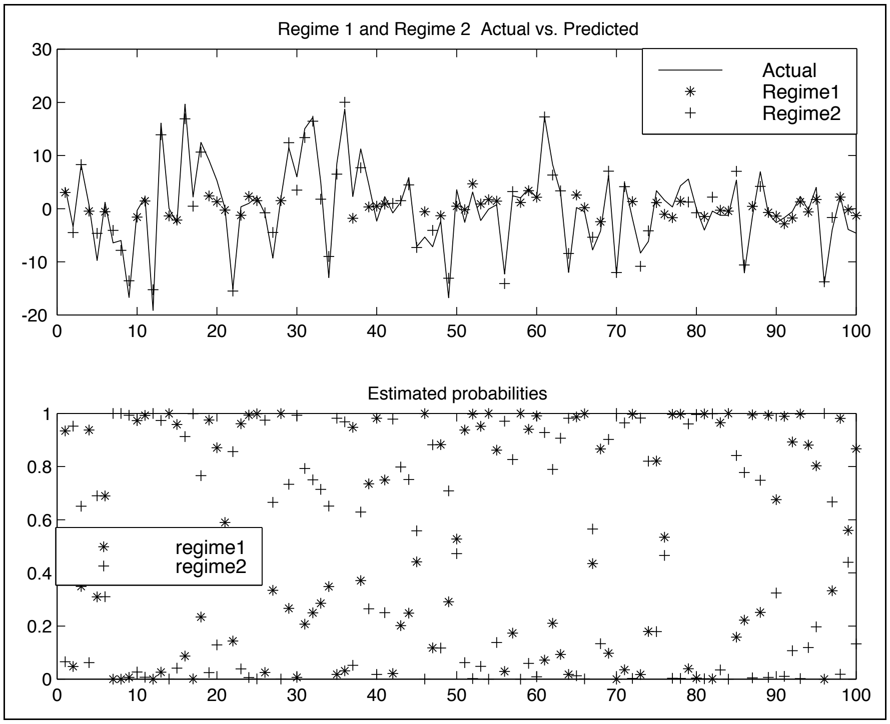

Figure 10.1: Plots from switching regime regression

timation exists, a corresponding Gibbs sampling estimation procedure can be devised. As an example, the function **sarp_g** implements a Gibbs sampling approach to estimating the spatial autoregressive logit/probit model that corresponds to an EM approach set forth in McMillen (1992).

For the switching regression example illustrated in this section, a Gibbs sampling approach would carry out multivariate normal draws for $\beta_1, \beta_2, \beta_3$ using means and variances based on the least-squares and weighted least-squares estimates presented. Chi-squared draws for σ_1, σ_2 would be made based on the sum of squared residuals from the weighted least-squares estimates and a new value of λ_t would be calculated based on these parameter values. This process of updating the estimates $\beta_1, \beta_2, \beta_3, \sigma_1, \sigma_2$ and the weights based on λ_t would be continued to produce a large sample of draws for the parameters in the model. Albert and Chib (1993) and Kim (1994) provide Gibbs sampling approaches to Hamilton's dynamic Markov switching model.

An EM approach to Hamilton's Markov switching model can be found in the functions **hmarkov_em** and **emhergott** contributed by Matt Hergott.

One criticism levelled against EM algorithms is that they are slow to converge. This seems less and less of a problem with the speed enhancements occurring in computing technology. Nonetheless, some problems can be more efficiently handled if the EM algorithm is used at the outset to steer the optimization problem towards high quality initial parameter values. After some initial EM loops, one can switch to a maximum likelihood approach based on the type of gradient routines discussed in the next section. This avoids the problem of slow convergence near the end of the EM algorithm, while taking advantage of a desirable aspect of the EM algorithm — it is guaranteed to produce a sequence of estimates that monotonically improve the value of the likelihood function.

10.3 Multivariate gradient optimization

By far the most difficult optimization problems are those that cannot be reduced to lower-dimensional optimization problems amenable to simplex algorithms, or handled by the EM algorithm, or Gibbs sampling. As we saw in Chapter 7 for the case of logit/probit models, this may not be a problem if analytical gradients and second derivatives are available.

For problems where analytical gradients and second derivatives are not available or convenient, there are functions **frpr_min, dfp_min, pow_min** that represent code from the *Numerical Recipes* book converted to MATLAB, a function **maxlik** that attempts to handle general maximum likelihood optimization problems, as well as the function **minz** by Mike Cliff and **optsolv**. The motivation for providing a host of different general purpose optimization functions is that in some problems one function works better than others.

Some differences between these functions are:

1. The **frpr_min, dfp_min, pow_min** functions begin by setting the hessian matrix to the identity and building up the hessian estimate with updating. This avoids inversion of the hessian matrix which can become non-positive definite in some problems. The **maxlik** function inverts the hessian matrix but relies on the function **invpd** to force non-positive definite matrices to be positive definite prior to inversion. (The small eigenvalues of the non-positive definite matrix are arbitrarily augmented to accomplish this.)

2. The **frpr_min, dfp_min, pow_min** functions use a function **linmin**

that can encounter problems, simply print an error message and quit.

3. The **optsolv** function is based on a public domain MATLAB function **solvopt** that implements a modification of Shor's r-algorithm. This function is used here to provide an example of how we use a 'wrapper function' **optsolv** to incorporate a public domain algorithm in the *Econometrics Toolbox* so it conforms in terms of calling arguments and documentation. The original **solvopt** algorithm was implemented by faculty at the University of Graz in Austria. Knowledge about the performance and quality of the algorithm in econometric maximum likelihood applications is sketchy.

4. The **minz** function incorporated in the *Econometrics Toolbox* differs slightly from that found on Mike Cliff's Internet site in order to make the calling arguments conformable with the other optimization algorithms in the toolbox.

As an illustration of using all of these optimization functions consider solving the tobit estimation problem. First, we create a MATLAB function to evaluate the log likelihood function. This function requires two optional input arguments 'y,x' that we use to pass the sample data to the likelihood function. Our optimization functions allow for an arbitrary number of arguments that can be passed, making their use perfectly general. The log-likelihood function for the tobit model is shown below:

```
function like =  to_liked(b,y,x);
% PURPOSE: evaluate tobit log-likelihood
%          to demonstrate optimization routines
%-----------------------------------------------------
% USAGE:    like = to_liked(b,y,x)
% where:     b = parameter vector (k x 1)
%            y = dependent variable vector (n x 1)
%            x = explanatory variables matrix (n x m)
%-----------------------------------------------------
% NOTE: this function returns a scalar equal to the negative
%       of the log-likelihood function
%       k ~= m because we may have additional parameters
%           in addition to the m bhat's (e.g. sigma)
%-----------------------------------------------------
% error check
if nargin ~= 3,error('wrong # of arguments to to_like'); end;
   h = .000001;             % avoid sigma = 0
   [m junk] = size(b);
   beta = b(1:m-1);         % pull out bhat
   sigma = max([b(m) h]);   % pull out sigma
```

```
xb = x*beta;
llf1 = -(y-xb).^2./(2*sigma) - .5*log(2*pi*sigma);
xbs = xb./sqrt(sigma); cdf = .5*(1+erf(xbs./sqrt(2)));
llf2 = log(h+(1-cdf));
llf = (y > 0).*llf1 + (y <= 0).*llf2;
like = -sum(llf);% scalar result
```

Now we can create a function to read the data and call these optimization functions to solve the problem. The documentation for all of the optimization functions take a similar form, so they can be used interchangeably when attempting to solve maximum likelihood problems. The documentation for **dfp_min** is shown below. There are four required input arguments, the function name given as a string, a vector of starting parameter values, a structure variable containing optimization options, and a variable argument list of optional arguments that will be passed to the function. In this case, the variable argument list contains the data vector for y and data matrix X. Other optimization options can be input as fields in the structure variable, and these differ depending on the optimization function being called. For example, the **dfp_min** function allows the user to provide optional arguments in the structure fields for the maximum number of iterations, a flag for printing intermediate results and a convergence tolerance.

```
% PURPOSE: DFP minimization routine to minimize func
%          (Converted from Numerical Recipes book dfpmin routine)
%---------------------------------------------------
% USAGE: [pout,fout, hessn, niter] = dfp_min(func,b,info,varargin)
% Where: func   = likelihood function to minimize
%        b      = parameter vector fed to func
%        info       = a structure with fields:
%        info.maxit = maximum # of iterations (default = 500)
%        info.tol   =  tolerance for convergence (default = 1e-7)
%        info.pflag = 1 for printing iterations, 0 for no printing
%        varargin   = list of arguments passed to function
%---------------------------------------------------
% RETURNS: pout = (kx1) minimizing vector
%          fout = value of func at solution values
%         hessn = hessian evaluated at the solution values
%         niter = # number of iterations
%---------------------------------------------------
% NOTE: func must take the form func(b,varargin)
%       where:     b = parameter vector (k x 1)
%           varargin = arguments passed to the function
```

Example 10.3 shows a program that calls five of the optimization functions to solve the tobit estimation problem. A comparison of the time taken by each optimization function, the estimates produced, and the log-likelihood function values and the hessian matrices evaluated at the solution are presented.

```
% --- Example 10.3 Maximum likelihood estimation of the Tobit model
n=200; k=5; randn('seed',20201); x = randn(n,k); beta = ones(k,1);
y = x*beta + randn(n,1); % generate uncensored data
% now censor the data
for i=1:n, if y(i,1) < 0, y(i,1) = 0.0; end; end;
% use ols for starting values
res = ols(y,x); b = res.beta; sige = res.sige;
 parm = [b
        sige];  % starting values
info.maxit = 100;
% solve using frpr_min routine
tic; [parm1,like1,hess1,niter1] = frpr_min('to_liked',parm,info,y,x);
disp('time taken by frpr routine'); toc;
% solve using dfp_min routine
tic; [parm2,like2,hess2,niter2] = dfp_min('to_liked',parm,info,y,x);
disp('time taken by dfp routine'); toc;
% solve using pow_min routine
tic; [parm3,like3,hess3,niter3] = pow_min('to_liked',parm,info,y,x);
disp('time taken by powell routine'); toc;
% solve using maxlik routine
tic; [parm4,like4,hess4,grad,niter4,fail] = maxlik('to_liked',parm,info,y,x);
disp('time taken by maxlik routine'); toc;
% solve using minz routine
infoz.call = 'other';
infoz.maxit = 500;
infoz.prt = 0;
tic;
[parm5,infoz,stat] = minz('to_liked',parm,infoz,y,x);
disp('time taken by minz routine'); toc;
niter5 = stat.iter;
like5 = stat.f;
hess5 = stat.Hi;

% formatting information for mprint routine
in.cnames = strvcat('fprf','dfp','powell','maxlik','minz');
in.fmt = '%8.4f';
fprintf(1,'comparison of bhat estimates \n');
mprint([parm1(1:k,1) parm2(1:k,1) parm3(1:k,1) parm4(1:k,1) parm5(1:k,1)],in);
fprintf(1,'comparison of sige estimates \n');
mprint([parm1(k+1,1) parm2(k+1,1) parm3(k+1,1) parm4(k+1,1) parm5(k+1,1)],in);
fprintf(1,'comparison of likelihood functions \n');
mprint([like1 like2 like3 like4 like5],in);
in.fmt = '%4d'; fprintf(1,'comparison of # of iterations \n');
mprint([niter1 niter2 niter3 niter4 niter5],in);
fprintf(1,'comparison of hessians \n'); in2.fmt = '%8.2f';
```

```
fprintf(1,'fprf hessian');   mprint(hess1,in2);
fprintf(1,'dfp hessian');    mprint(hess2,in2);
fprintf(1,'powell hessian'); mprint(hess3,in2);
fprintf(1,'maxlik hessian'); mprint(hess4,in2);
fprintf(1,'minz hessian');   mprint(hess5,in2);
```

The output from example 10.3 is shown below, where we see that all four functions found nearly identical results. A primary difference was in the time needed by the alternative optimization methods.

```
time taken by frpr routine   = 3.9557
time taken by dfp routine    = 3.6244
time taken by powell routine = 12.8823
time taken by maxlik routine = 0.9818
time taken by minz routine   = 1.3015

comparison of bhat estimates
    fprf      dfp   powell   maxlik     minz
  0.8822   0.8822   0.8820   0.8822   0.8828
  0.8509   0.8509   0.8508   0.8509   0.8515
  1.0226   1.0226   1.0224   1.0226   1.0247
  0.9057   0.9057   0.9055   0.9057   0.9067
  0.9500   0.9500   0.9497   0.9500   0.9517

comparison of sige estimates
    fprf      dfp   powell   maxlik     minz
  0.8808   0.8809   0.8769   0.8808   0.8844

comparison of likelihood functions
    fprf      dfp   powell   maxlik     minz
153.0692 153.0692 153.0697  153.0692 153.0700

comparison of # of iterations
  fprf    dfp powell maxlik   minz
     8      7      5     10     11

comparison of hessians
fprf hessian
  137.15   -6.66  -30.32   -2.11    2.97   -2.50
   -6.66  136.03  -17.15   -4.10   -5.99   -2.13
  -30.32  -17.15  126.68   -7.98  -10.56   -5.10
   -2.11   -4.10   -7.98  126.65   -4.78   -5.57
    2.97   -5.99  -10.56   -4.78  145.19   -8.04
   -2.50   -2.13   -5.10   -5.57   -8.04   71.06
dfp hessian
  137.12   -6.66  -30.31   -2.11    2.97   -2.52
   -6.66  136.00  -17.15   -4.10   -5.98   -2.11
  -30.31  -17.15  126.65   -7.97  -10.57   -5.14
```

```
   -2.11   -4.10   -7.97  126.63   -4.77   -5.54
    2.97   -5.98  -10.57   -4.77  145.16   -8.06
   -2.52   -2.11   -5.14   -5.54   -8.06   71.01
powell hessian
  137.69   -6.68  -30.43   -2.12    3.00   -2.38
   -6.68  136.57  -17.24   -4.12   -6.00   -2.10
  -30.43  -17.24  127.14   -8.00  -10.61   -5.27
   -2.12   -4.12   -8.00  127.14   -4.79   -5.58
    3.00   -6.00  -10.61   -4.79  145.77   -8.03
   -2.38   -2.10   -5.27   -5.58   -8.03   72.27
maxlik hessian
  137.14   -6.66  -30.31   -2.11    2.97   -2.50
   -6.66  136.02  -17.15   -4.10   -5.99   -2.12
  -30.31  -17.15  126.66   -7.98  -10.57   -5.12
   -2.11   -4.10   -7.98  126.64   -4.77   -5.56
    2.97   -5.99  -10.57   -4.77  145.18   -8.04
   -2.50   -2.12   -5.12   -5.56   -8.04   71.04
minz hessian
  136.60   -6.63  -30.19   -2.09    2.95   -2.52
   -6.63  135.49  -17.08   -4.08   -5.96   -2.14
  -30.19  -17.08  126.15   -7.94  -10.53   -5.31
   -2.09   -4.08   -7.94  126.16   -4.76   -5.63
    2.95   -5.96  -10.53   -4.76  144.61   -8.21
   -2.52   -2.14   -5.31   -5.63   -8.21   70.00
```

A point to note about the approach taken to designing the optimization functions is that we can easily add new algorithms from the Internet or other sources to the *optimization function library*. As an illustration of this, we demonstrate how to add a new MATLAB optimization function named "solvopt" which was placed in the public domain on the Internet by Alexei Kuntsevich and Franz Kappel.

We construct a 'wrapper function' that we name **optsolv** to call this function with the appropriate input arguments.

The first step is to change the function input arguments and documentation to conform to the others in our *optimization function library*. The function **solvopt** was written to handle constrained as well as unconstrained optimization problems, but we are only interested in using the function for unconstrained problems. The new format for the **optsolv** function is shown below and it takes the same form as our other functions with the exception of the optimization options. This function has the responsibility for interpreting the input arguments and passing them along to the function **solvopt** that does the real work of optimization.

```
function [x,f,hessn,gradn,niter]=optsolv(fun,x,info,varargin)
 PURPOSE: a modified version of Shor's r-algorithm to minimize func
```

```
---------------------------------------------------
 USAGE: [x,f,hess,gradn,niter] = optsolv(func,b,info)
 Where: func   = likelihood function to minimize (<=36 characters long)
        b      = parameter vector fed to func
        info       = a structure with fields:
        info.maxit = maximum # of iterations (default = 1000)
        info.btol  =  b tolerance for convergence (default = 1e-7)
        info.ftol  =  func tolerance for convergence (default = 1e-7)
        info.pflag = 1 for printing iterations, 0 for no printing
        varargain  = arguments passed to the function
---------------------------------------------------
 RETURNS: x    = (kx1) minimizing vector
          f    = value of func at solution values
         hessn = hessian evaluated at the solution values
         gradn = gradient evaluated at the solution
         niter = # number of iterations
---------------------------------------------------
 NOTE: - func must take the form func(b,P0,P1,...)
         where:   b = parameter vector (k x 1)
          P0,P1,... = arguments passed to the function
---------------------------------------------------
```

These modifications required that we parse our input option arguments into a vector variable named 'options' that the **solvopt** function uses. This was accomplished with the following code fragment:

```
  if ~isstruct(info)
    error('optsolv: options should be in a structure variable');
  end;
% default options
options=[-1,1.e-4,1.e-6,1000,0,1.e-8,2.5,1e-11];
app = 1;    % no user supplied gradients
constr = 0; % unconstrained problem
% parse options
fields = fieldnames(info);
nf = length(fields); xcheck = 0; ycheck = 0;
  for i=1:nf
    if strcmp(fields{i},'maxit'),     options(4) = info.maxit;
    elseif strcmp(fields{i},'btol'),  options(2) = info.btol;
    elseif strcmp(fields{i},'ftol'),  options(3) = info.ftol;
    elseif strcmp(fields{i},'pflag'), options(5) = info.pflag;
    end;
end;
funfcn = fcnchk(funfcn,length(varargin));
```

We simply feed the user-input options directly into the options vector used by the function.

A second step involves use of a MATLAB function **fcnchk** shown as the last line in the code fragment above. This function sets up a string argument for the function name so we can pass it to the gradient and hessian functions. This MATLAB function is used by **fmin** and **fmins** along with the approach we take to passing the function name on to other sub-functions used by the optimization algorithm. A good way to discover programming tricks is to examine the functions crafted by the MathWorks folks, which is what was done here.

Given the function name string, 'funfcn', we can use this in a function call to the function **apprgrdn** that **solvopt** uses to compute numerical gradients. We have to change this function to rely on a function name argument based on the string in 'funfcn' and our variable argument list that will be passed to this function. This function will need to call our likelihood function and properly pass the variable argument list. The original calls to **apprgrdn** were in the form:

```
g=apprgrdn(b,fp,fun,deltax,1);
```

which do not supply the y and X data arguments required by our function. The modified function **apprgrdn** was written to accept a call with a variable arguments list.

```
g=apprgrdn(parm,f,funfcn,deltaparm,varargin{:});
```

The third step involves changing likelihood function calls from inside the **apprgrdn** function The calls to the likelihood function in the original **apprgrdn** function were in the form:

```
fi=feval(fun,b);
```

These were modified with the code shown below. First we modify the function to rely on the MATLAB **fcnchk** function to provide a string argument for the function name. Next, we call the likelihood function using the string 'funfcn' containing its name, a vector of parameters 'x0', and the variable argument list 'varargin:'.

```
funfcn = fcnchk(funfcn,length(varargin));
fi=feval(funfcn,x0,varargin{:});
```

A fourth step relates to the fact that many optimization routines crafted for non-econometric purposes do not provide a hessian argument. This can be remedied with the function **hessian** from the *optimization function library*. It evaluates the hessian at a given parameter vector using a likelihood function argument. The documentation for **hessian** is:

```
 PURPOSE: computes hessian matrix of 2nd partial derivative
          for the likelihood function f evaluated at b
---------------------------------------------------
 USAGE: hessn = hessian(f,b,varargin)
 where:    f = a string containing the likelihood function
           b = a parameter vector (k x 1)
    varargin = arguments that will be passed to the function
---------------------------------------------------
 NOTE: f must take the form f(b,P0,P1,...)
       where: b = parameter vector (k x 1)
      P0,P1,... = optional variables passed to likelihood
---------------------------------------------------
 RETURNS: hess = hessian matrix (k x k)
---------------------------------------------------
```

Notice that it was designed to work with likelihood functions matching the format of those in the *optimization function library*. We can produce a hessian matrix evaluated at the solution vector with a call to **hessian**, for optimization algorithms like **solvopt** that don't provide this information.

An example of using this new optimization routine is shown below, where we compare the solution times and results with the **maxlik** function demonstrated previously.

```
% ----- Example 10.4 Using the optsolv() function
n=200; k=5; randn('seed',20211); x = randn(n,k); beta = ones(k,1);
y = x*beta + randn(n,1); % generate uncensored data
for i=1:n, if y(i,1) < 0,y(i,1) = 0.0; end; end;
res = ols(y,x); b = res.beta; sige = res.sige;
parm = [b
        sige];  % starting values
info.maxit = 100;
% solve using optsolv routine
tic;
[parm1,like1,hess1,grad1,niter1] = optsolv('to_liked',parm,info,y,x);
disp('time taken by optsolv routine'); toc;
% solve using maxlik routine
tic;
[parm2,like2,hess2,grad2,niter2,fail] = maxlik('to_liked',parm,info,y,x);
disp('time taken by maxlik routine'); toc;
in.fmt = '%9.3f'; in.cnames = strvcat('optsolv','maxlik');
fprintf(1,'comparison of estimates \n');
mprint([parm1 parm2],in);
fprintf(1,'comparison of # of iterations \n');
in.fmt = '%8d';
mprint([niter1 niter2],in);
fprintf(1,'comparison of hessians \n'); in2.fmt = '%8.3f';
mprint(hess1,in2); mprint(hess2,in2);
```

The results from example 10.4 are shown below, indicating that the **solvopt** function might be a valuable addition to the *optimization function library*.

```
time taken by optsolv routine = 7.4620
time taken by maxlik routine  = 1.1209

comparison of estimates
  optsolv    maxlik
    1.082     1.082
    1.145     1.145
    1.168     1.168
    1.021     1.021
    1.155     1.155
    0.867     0.867

comparison of # of iterations
 optsolv   maxlik
      23       10

comparison of hessians
optsolv
 145.137  -5.641  -6.786   7.562 -14.985  -7.246
  -5.641 125.331 -11.404  -4.000 -16.000  -1.483
  -6.786 -11.404 145.101  31.777 -10.786  -0.818
   7.562  -4.000  31.777 129.980 -28.289  -1.575
 -14.985 -16.000 -10.786 -28.289 152.888  -1.994
  -7.246  -1.483  -0.818  -1.575  -1.994  76.805
maxlik
 145.145  -5.641  -6.787   7.562 -14.985  -7.248
  -5.641 125.339 -11.405  -4.000 -16.001  -1.486
  -6.787 -11.405 145.110  31.779 -10.787  -0.815
   7.562  -4.000  31.779 129.989 -28.291  -1.575
 -14.985 -16.001 -10.787 -28.291 152.898  -1.985
  -7.248  -1.486  -0.815  -1.575  -1.985  76.824
```

10.4 Chapter summary

We provided some simple examples of maximum likelihood estimation for models involving univariate and multivariate parameter vectors. Simplex and gradient methods as well as the EM algorithm were demonstrated. Beyond providing examples, a design was set forth to allow new optimization algorithms to be incorporated into the *optimization function library* in a consistent, easy-to-use fashion. A demonstration of re-coding an existing optimization algorithm from the Internet to work with the function library was provided in the last section of the chapter.

Chapter 10 Appendix

Optimization functions discussed in this chapter discussed in this chapter are in the subdirectory **optimize**.

```
Optimization functions library

 --------------- optimization functions -----------------

 dfp_min    - Davidson-Fletcher-Powell
 frpr_min   - Fletcher-Reeves-Polak-Ribiere
 maxlik     - general all-purpose optimization routine
 minz       - general purpose optimziation routine
 pow_min    - Powell conjugate gradient
 optsolv    - yet another general purpose optimization routine

 --------------- demonstration programs -----------------

 optim1_d     - dfp, frpr, pow, maxlik demo
 optim2_d     - optsolv demo
 optim3_d     - fmins demo
 optim4_d     - minz demo

 --------------- support programs -----------------------

 apprgrdn      - computes gradient for optsolv
 box_like1     - used by optim3_d
 gradnt        - computes gradient
 gradt         - computes gradient
 hessian       - evaluates hessian
 hessz         - used by minz
 linmin        - line minimization routine (used by dfp, frpr, pow)
 numz          - used by minz
 stepz         - used by minz
 tol_liked     - used by optim1_d, optim2_d, optim4_d
 --------------------------------------------------------
```

Chapter 11

Handling sparse matrices

This chapter provides demonstrations of using the MATLAB sparse matrix algorithms to solve econometric problems. A sparse matrix is one that contains a large proportion of zeros. The illustrations used involve spatial econometric estimation problems where large but sparse matrices are often encountered.

Section 11.1 discusses the computational savings associated with intelligent handling of sparse matrices. In Section 11.2 we illustrate how the MATLAB sparse matrix algorithms can be used to solve a spatial econometric estimation problem involving a 3,107 by 3,107 matrix. The log-likelihood function for this model requires that we evaluate the determinant of this large but sparse matrix. Another illustration involving the use of sparse matrix algorithms in a Gibbs sampling situation is provided in Section 11.3.

11.1 Computational savings with sparse matrices

As an example of a sparse matrix, consider the first-order contiguity matrix for a sample of 3,107 U.S. counties used in Pace and Berry (1997). Recall, a first-order contiguity matrix has zeros on the main diagonal, rows that contain zeros in positions associated with non-contiguous observational units and ones in positions reflecting neighboring units that are (first-order) contiguous. An example is shown in (11.1) for a sample of five areas.

$$W = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{11.1}$$

Information regarding first-order contiguity is recorded for each observation as ones for areas that are neighbors (e.g., observations 2 and 3 are neighbors to 1) and zeros for those that are not (e.g., observations 4 and 5 are not neighbors to 1). By convention, zeros are placed on the main diagonal of the spatial weight matrix.

For the case of our 3,107 county sample, this matrix would be sparse since the largest number of neighbors to any county is 8 and the average number of neighbors is 4. A great many of the elements in the contiguity matrix W are zero, meeting the definition of a sparse matrix.

To understand how sparse matrix algorithms conserve on storage space and computer memory, consider that we need only record the non-zero elements of a sparse matrix for storage. Since these represent a small fraction of the total 3107x3107 = 9,653,449 elements in the weight matrix, we save a trememdous amount of computer memory. In fact for our example of the 3,107 U.S. counties, only 12,429 non-zero elements were found in the first-order spatial contiguity matrix, representing a very small fraction (about 0.4 percent) of the total elements.

MATLAB provides a function **sparse** that can be used to construct a large sparse matrix by simply indicating the row and column positions of non-zero elements and the value of the matrix element for these non-zero row and column elements. Continuing with our example, we can store the first-order contiguity matrix in a single data file containing 12,429 rows with 3 columns that take the form:

```
row column value
```

This represents a considerable savings in computational space when compared to storing a matrix containing 9,653,449 elements. A handy utility function in MATLAB is **spy** which allows one to produce a specially formatted graph showing the sparsity structure associated with sparse matrices. We demonstrate by executing **spy(W)** on our weight matrix W from the Pace and Berry data set, which produced the graph shown in Figure 11.1. As we can see from the figure, most of the non-zero elements reside near the diagonal.

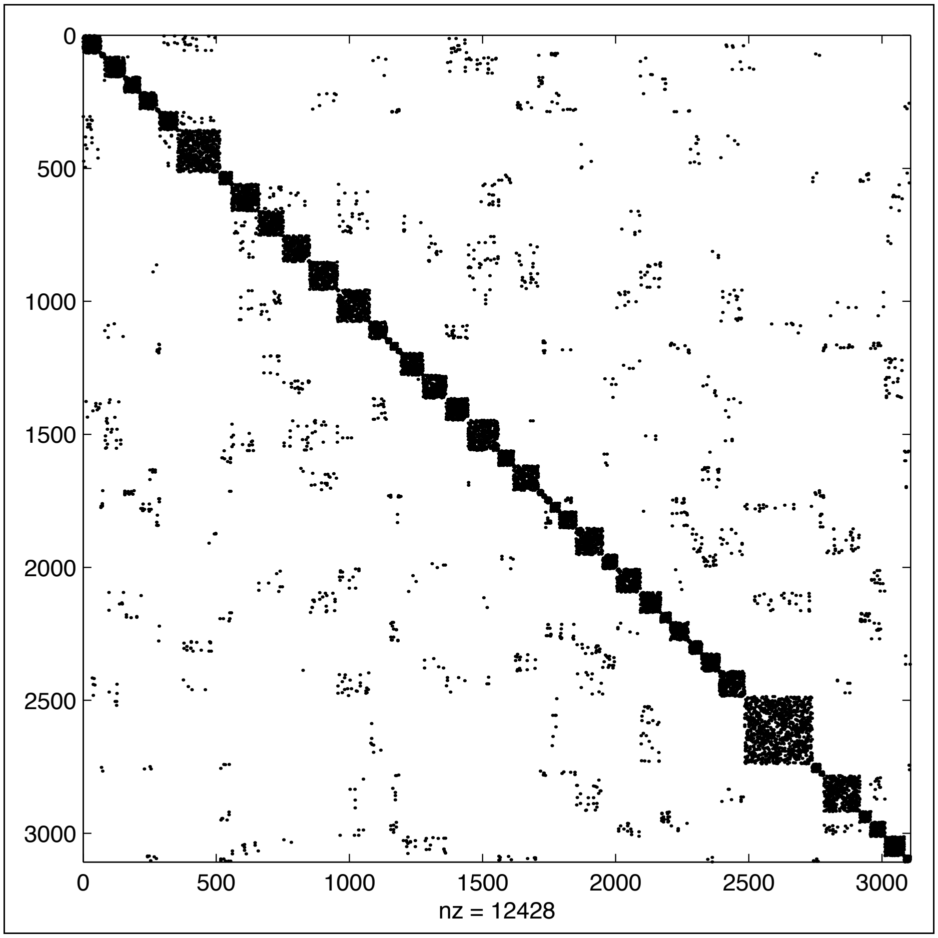

Figure 11.1: Sparsity structure of W from Pace and Berry

As an example of storing a sparse first-order contiguity matrix, consider example 11.1 below that reads data from the file 'ford.dat' in sparse format and uses the function **sparse** to construct a working spatial contiguity matrix W. The example also produces a graphical display of the sparsity structure using the MATLAB function **spy**.

```
% ----- Example 11.1 Using sparse matrix functions
load ford.dat; % 1st order contiguity matrix
               % stored in sparse matrix form
ii = ford(:,1);
jj = ford(:,2);
```

```
ss = ford(:,3);
clear ford;                   % clear out the matrix to save RAM memory
W = sparse(ii,jj,ss,3107,3107);
clear ii; clear jj; clear ss; % clear out these vectors to save memory
spy(W);
```

MATLAB does not automatically treat a matrix as sparse, users must provide a declaration using the **sparse** command. If we know a matrix is sparse and want to use this to our advantage when executing matrix operations, we convert the matrix to sparse form using: 'b = **sparse**(a)'. Subsequent MATLAB matrix operations such as multiplication, addition, subtraction, division and the MATLAB 'slash' operator (discussed below) involving 'b' and other sparse matrices will produce sparse matrix results. That is, operating on two sparse matrices will create a new sparse matrix as a result. Other operations that produce sparse matrix results when given a sparse matrix argument are the cholesky decomposition **chol**, the householder QR decomposition **qr** and the LU decomposition **lu**.

If one matrix in a binary operation is sparse and the other a full matrix, the result will be a full matrix. When you are uncertain about the nature of a matrix, the function **issparse** will provide an answer. This function returns a value of 1 if the matrix is sparse and 0 otherwise. Another useful function for dealing with sparse matrices is **nnz**, which returns the number of non-zero elements in a matrix.

To illustrate working with sparse matrices, example 11.2 generates a sparse matrix system which we solve using the MATLAB 'slash' operator that carries out Gaussian elimination to solve the problem. A full matrix solution is also provided, and we compare the time needed using the **tic** and **toc** commands.

```
% ----- Example 11.2 Solving a sparse matrix system
n = 500; density = 0.1; knum=0.5;
for i=1:5;
density = density*0.5;
X = sprandsym(n,density,knum);
y = randn(n,1);
tic; b = X\y; time1 = toc;
xf = full(X);
tic; b = xf\y; time2 = toc;
fprintf('# of non-zero elements in X = %6d \n',nnz(X));
fprintf('sparse matrix solution time = %8.3f \n',time1);
fprintf('full matrix solution   time = %8.3f \n',time2);
end;
```

The example relies on the sparse matrix function **sprandsym** to generate a symmetric matrix of dimension 'n=500' with random normal numbers distributed with a density provided by an input argument to the function. This 'density' argument represents the fraction of elements that will be non-zero. We produce a sequence of five matrix systems with increasing sparsity by varying the 'density' input argument each time through the 'for i=1:5' loop. A third argument allows us to specify the condition number for the generated matrix using a fraction 'knum' between zero and one such that the condition number will be 1/knum. We use this option to prevent ill-conditioning in our matrix system as the number of non-zero elements becomes smaller.

The results are shown below where we see that the full matrix operation takes roughly 3.3 seconds to solve all five systems. In contrast, the sparse matrix solution is around 10 times faster when we have a density of 0.05, taking only .356 seconds. As the sparsity increases, the time required to solve the problem decreases dramatically to 0.010 seconds, which is around 300 times faster. For the U.S counties the density is: 12,429/9,653,449 = 0.00128. By comparison, the smallest density in example 11.2 is 0.003125, so it should be clear that one could spend weeks of computer time solving large problems without sparse matrix algorithms or a few seconds with these algorithms.

```
# of non-zero elements in X =  11878
sparse matrix solution time =    0.356
full matrix solution   time =    3.260
# of non-zero elements in X =   5970
sparse matrix solution time =    0.080
full matrix solution   time =    3.370
# of non-zero elements in X =   2982
sparse matrix solution time =    0.020
full matrix solution   time =    3.284
# of non-zero elements in X =   1486
sparse matrix solution time =    0.013
full matrix solution   time =    3.310
# of non-zero elements in X =    744
sparse matrix solution time =    0.010
full matrix solution   time =    3.261
```

The subject of sparse matrix computational efficiency is beyond the scope of this text, but we can provide some insights that may be useful. Gaussian elimination includes a special preordering of the columns of the matrix labeled "minimum degree ordering", which reduces "fill-in" that occurs with matrix operations. The term "fill-in" refers to the introducion of non-zero elements as a result of matrix manipulation. As an illustration of fill-in, we carry out matrix multiplication of the sparse first-order contiguity matrix taken from Anselin's (1988) Columbus neighborhood crime data set. A comparison of the sparsity associated with the original first-order contiguity matrix W and the matrix product $W * W * W$ is shown in Figure 11.2.

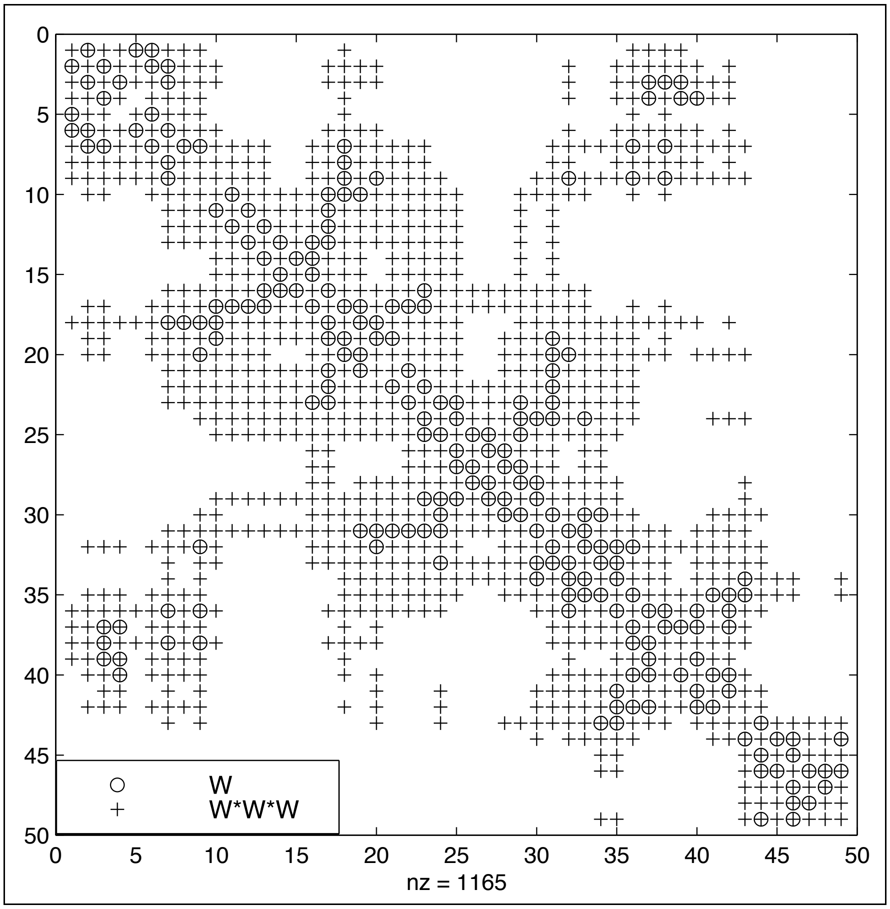

Figure 11.2: An illustration of fill-in from matrix multiplication

To illustrate the importance of preordering in matrix computations, we carry out matrix multiplication on a symmetric sparse random matrix generated using **sprandsym** from example 11.2. We produce a symmetric minimum degree ordering, using the MATLAB function **symmmd** and carry out matrix multiplication based on this ordering. We also rely on MATLAB to automatically do this for us and carry out matrix multiplication involving the sparse matrix. A comparison to the case of a full matrix multiplication is provided in terms of both floating point operations (flops) and the time required.

```
% ----- Example 11.3 Symmetric minium degree ordering operations
n = 500; density = 0.1; knum=0.5;
for i=1:5;
density = density*0.5;
X = sprandsym(n,density,knum);
% produce symmetric minimum degree ordering
mo = symmmd(X);
tic; flops(0); result = X(mo,mo)*X(mo,mo); time0 = toc; flops0 = flops;
tic; flops(0); result = X*X; time1 = toc; flops1 = flops;
xf = full(X);
tic; flops(0); result = xf*xf; time2 = toc; flops2 = flops;
fprintf('# of non-zero elements in X = %6d \n',nnz(X));
fprintf('ordered matrix solution time = %8.3f flops = %16d \n',time0,flops0);
fprintf('sparse matrix solution  time = %8.3f flops = %16d \n',time1,flops1);
fprintf('full matrix solution    time = %8.3f flops = %16d \n',time2,flops2);
end;
```

From the results presented below we see that the minimum degree ordering accounts for the greatly reduced number of floating point operations during sparse matrix multiplication. The time is slightly faster for the straightforward sparse matrix multiplication because the internal ordering is faster than our explicit ordering.

```
# of non-zero elements in X =  11922
ordered matrix solution time =    0.179 flops =         1228634
sparse matrix solution  time =    0.169 flops =         1228634
full matrix solution    time =    8.320 flops =       250000018
# of non-zero elements in X =   5956
ordered matrix solution time =    0.058 flops =          296022
sparse matrix solution  time =    0.047 flops =          296022
full matrix solution    time =    8.533 flops =       250000018
# of non-zero elements in X =   2976
ordered matrix solution time =    0.018 flops =           71846
sparse matrix solution  time =    0.010 flops =           71846
full matrix solution    time =    8.448 flops =       250000018
# of non-zero elements in X =   1502
ordered matrix solution time =    0.008 flops =           17238
sparse matrix solution  time =    0.006 flops =           17238
full matrix solution    time =    8.323 flops =       250000018
# of non-zero elements in X =    742
ordered matrix solution time =    0.004 flops =            2974
sparse matrix solution  time =    0.002 flops =            2974
full matrix solution    time =    8.470 flops =       250000018
```

As a graphical illustration of symmetric minimum degree ordering, Figure 11.3 shows the 3,107 county contiguity matrix sparsity structure in the original un-ordered form along with this structure for a symmetric minimum degree ordering of this matrix.

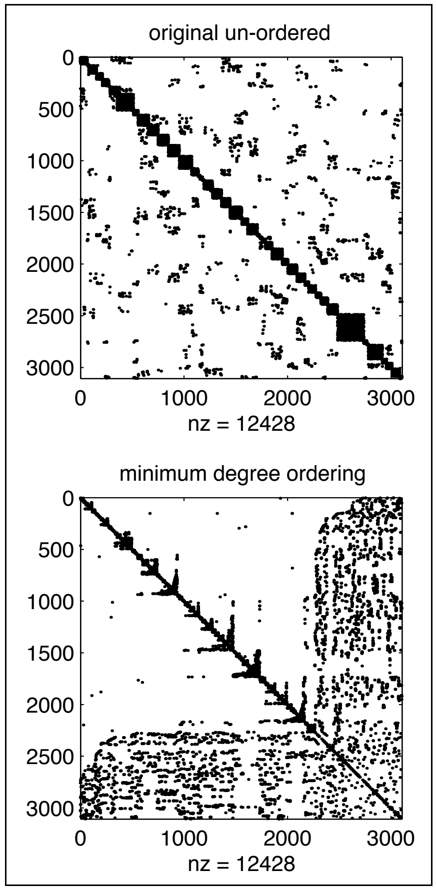

Figure 11.3: Minimum degree ordering versus unordered Pace and Barry matrix

MATLAB provides other orderings for non-symmetric matrices using the function **colmmd** and a Cuthill MacGee ordering can be generated using the function **symrcm**.

As a final example of sparse matrix functionality, we examine some code from a function **normw** in the spatial econometrics library that normalizes contiguity matrices to have row-sums of unity. One might approach this problem inefficiently using a loop over the rows as shown below.

```
rsum = sum(win');
wout = zeros(n1,n1);
for i=1:n1
wout(i,:) = win(i,:)/rsum(1,i);
end;
```

This code is inefficient because we need to allocate space for wout, which could be a very large but sparse matrix containing millions of elements, most of which are zero. Rather than allocate this memory, we can rely on the **find** command to extract only non-zero elements on which the standardization operates as shown below.

```
[n1 n2] = size(win);
if n1 ~= n2,
error('normw: W matrix must be square');
end;
d = zeros(n1,1);
    [junk,ind,val] = find(sum(win'));
    d(ind) = val.^(-1);
    D = diag(d);
    wout = D*win;
```

We find a vector of non-zero row-sums whose index is 'ind'. The index variable 'ind' is then used to standardize only the non-zero values by operating on the entire matrix with the diagonal matrix containing the inverse of non-zero row-sums.

11.2 Estimation using sparse matrix algorithms

MATLAB contains functions to carry out matrix operations on sparse matrices. To illustrate some of these, we construct a function **far** that provides maximum likelihood estimates for a first-order spatial autoregressive model shown in (11.2).

$$\begin{aligned} y &= \rho W_1 y + \varepsilon \\ \varepsilon &\sim N(0, \sigma^2 I_n) \end{aligned} \tag{11.2}$$

This model attempts to explain variation in y as a linear combination of contiguous or neighboring units with no other explanatory variables. The model is termed a first order spatial autoregression because its represents a spatial analogy to the first order autoregressive model from time series analysis, $y_t = \rho y_{t-1} + \varepsilon_t$, where total reliance is on the past period observations to explain variation in y_t.

It is conventional to standardize the spatial weight matrix W so that the row sums are unity and to put the vector of sample observations y in deviations from the means form to eliminate the intercept term from the model. Anselin (1988) shows that ordinary least-squares estimation of this model will produce biased and inconsistent estimates. Beacuse of this, a maximum likelihood approach can be used to find an estimate of the parameter ρ using the likelihood function shown in (11.3).

$$L(y|\rho,\sigma^2) = \frac{1}{2\pi\sigma^{2(n/2)}}|I_n - \rho W| \exp\{-\frac{1}{2\sigma^2}(y-\rho Wy)'(y-\rho Wy)\} \quad (11.3)$$

In order to simplify the maximization problem, we obtain a concentrated log likelihood function based on eliminating the parameter σ^2 for the variance of the disturbances. This is accomplished by substituting $\hat{\sigma}^2 = (1/n)(y-\rho Wy)'(y-\rho Wy)$ in the likelihood (11.3) and taking logs which yields:

$$\mathrm{Ln}(L) \propto -\frac{n}{2}\ln(y-\rho Wy)'(y-\rho Wy) + \ln|I_n - \rho W| \quad (11.4)$$

This expression can be maximized with respect to ρ using a simplex univariate optimization routine. The estimate for the parameter σ^2 can be obtained using the value of ρ that maximizes the log-likelihood function (say, $\tilde{\rho}$) in: $\hat{\sigma}^2 = (1/n)(y-\tilde{\rho}Wy)'(y-\tilde{\rho}Wy)$.

Note that the log-likelihood function in (11.4) contains an expression for the determinant of $(I_n - \rho W)$, which for our example data sample is 3,107 by 3,107. We demonstrate a sparse matrix algorithm approach to evaluating this determinant that allows us to solve problems involving thousands of observations quickly with small amounts of computer memory.

Two implementation details arise with this approach to solving for maximum likelihood estimates. First, there is a constraint that we need to impose on the parameter ρ. This parameter can take on feasible values in the range (Anselin and Florax, 1994):

$$1/\lambda_{min} < \rho < 1/\lambda_{max}$$

where λ_{min} represents the minimum eigenvalue of the standardized spatial contiguity matrix W and λ_{max} denotes the largest eigenvalue of this matrix.

This suggests that we need to constrain our optimization procedure search over values of ρ within this range. Note that this requires we solve for the maximum and minimum eigenvalues of the matrix W which is 3,107 by 3,107, not a trivial problem.

The second implementation issue is that for problems involving a small number of observations, we used our knowledge of the theoretical information matrix to produce estimates of dispersion. This approach is computationally impossible when dealing with large scale problems involving thousands of observations. In these cases we can evaluate the numerical hessian matrix using the maximum likelihood estimates of ρ and σ^2 as well as a sparse matrix non-concentrated version of the likelihood function.

Our first task is to construct a function to evaluate the concentrated log likelihood based on the sparse matrix algorithms. This function named **f_far** is shown below.

```
function llike = f_far(rho,y,W)
% PURPOSE: evaluate the concentrated log-likelihood for the first-order
% spatial autoregressive model using sparse matrix algorithms
% ---------------------------------------------------
%  USAGE:llike = f_far(rho,y,W)
%  where: rho  = spatial autoregressive parameter
%         y    = dependent variable vector
%         W    = spatial weight matrix
% ---------------------------------------------------
%  RETURNS: a  scalar equal to minus the log-likelihood
%           function value at the parameter rho
%  --------------------------------------------------
%  SEE ALSO: far, f_sar, f_sac, f_sem
% ---------------------------------------------------
n = length(y); spparms('tight');
z = speye(n) - 0.1*sparse(W);
p = colmmd(z);
z = speye(n) - rho*sparse(W);
[l,u] = lu(z(:,p));
detval = sum(log(abs(diag(u))));
epe = y'*z'*z*y;
llike = (n/2)*log(pi) + (n/2)*log(epe) - detval;
```

The function solves the determinant of $(I - \rho W)$ using the LU decomposition implemented by the MATLAB **lu** command. This algorithm operates on sparse matrices in an intelligent way. The command **spparms** sets options for operation of the sparse matrix algorithms. The option 'tight' sets the minimum degree ordering parameters to particular settings, which lead to sparse matrix orderings with less fill-in, but make the ordering functions use more execution time. Some experimentation on my part with the various options that can be set has led me to believe this is an optimal setting for this type of model. The command **sparse** informs MATLAB that the matrix W is sparse and the command **speye** creates an identity matrix in sparse format. We set up an initial matrix based on $(I_n - 0.1W)$ from which we construct a column vector of minimum degree permutations for this sparse matrix. By executing the **lu** command with this vector, we manage to operate on a sparser set of LU factors than if we operated on the matrix $z = (I - \rho W)$.

Given this function to evaluate the log likelihood for very large spatial weight matrices W, we can now rely on the same **fmin** simplex optimization algorithm demonstrated in Chapter 10. Another place where we can rely on sparse matrix functions is in determining the minimum and maximum eigenvalues of the matrix W. We will use these values to set the feasible range for ρ in our call to the simplex search function **fmin**. The code for carrying this out is:

```
opt.tol = 1e-3; opt.disp = 0;
lambda = eigs(sparse(W),speye(n),2,'BE',opt);
lmin = 1/lambda(2);
lmax = 1/lambda(1);
```

The MATLAB function **eigs** works to compute selected eigenvalues for a sparse matrix, and we use the option 'BE' to compute only the maximum and minimum eigenvalues which we need to determine the feasible range for ρ. The options set as fields in the 'opt' structure variable indicate a tolerance to be used when solving the eigenvalue problem and to prohibit display of the iterative solution results. The default tolerance is 1e-10, but using the tolerance of 1e-3 speeds up the solution by a factor of four times. Note that we are not overly concerned about loosing a few decimal digits of accuracy, since it is unlikely that the maximum likelihood values of ρ are near these bounds on the feasible range. If we find a maximum likelihood estimate near these limits, we can always change the tolerance option argument to the **eigs** function.

Another point is that for the case of standardized weight matrices W, the maximum eigenvalue will always take on a value of unity, so we could save time by only computing one eigenvalue. However, not all researchers will use a row-standardized matrix W, so we make the function more generally useful by computing both eigenvalues.

The final issue we need to address is computing measures of dispersion for the estimates ρ and σ^2 in our estimation problem. As already noted, we cannot rely on the information matrix approach because this involves matrix operations on very large matrices. An approach that we take to produce measures of dispersion is to numerically evaluate the hessian matrix using the maximum likelihood estimates of ρ and σ^2. Our function **hessian** from Chapter 10 can be used to compute the hessian matrix given a non-concentrated log-likelihood function and maximum likelihood estimates for ρ and σ. This non-concentrated version of the log-likelihood function is shown below.

```
function llike = f2_far(parm,y,W)
% PURPOSE: evaluate the log-likelihood for ML rho,sigma values
% for the first-order spatial autoregressive model
% ---------------------------------------------------
%  USAGE: llike = f2_far(parm,y,W)
%  where: parm  = 2x1 vector with rho,sigma values
%          y    = dependent variable vector
%          W    = spatial weight matrix
% ---------------------------------------------------
%  RETURNS: a scalar equal to minus the log-likelihood
%           function value at the ML parameters rho,sigma
%  --------------------------------------------------
%  SEE ALSO: far, f2_sar, f2_sac, f2_sem
% ---------------------------------------------------
n = length(y); rho = parm(1,1); sige = parm(2,1);
spparms('tight'); z = speye(n) - 0.1*sparse(W);
p = colmmd(z);    z = speye(n) - rho*sparse(W);
[l,u] = lu(z(:,p));
detval = sum(log(abs(diag(u))));
epe = y'*z'*z*y;
llike = (n/2)*log(pi) + (n/2)*log(epe) +(n/2)*log(sige) - detval;
```

The function **far** that implements this approach to estimation of the first-order autoregressive model is shown below.

```
function results = far(y,W,convg,maxit)
%  PURPOSE: computes 1st-order spatial autoregressive estimates
%           model: y = p*W*y + e, using sparse matrix algorithms
% ---------------------------------------------------
%  USAGE: results = far(y,W,convg,maxit)
%  where:  y = dependent variable vector
%          W = standardized contiguity matrix
%      convg = (optional) convergence criterion (default = 1e-8)
%      maxit = (optional) maximum # of iterations (default = 500)
% ---------------------------------------------------
%  RETURNS: a structure
%         results.meth  = 'far'
%         results.rho   = rho
```

```
%         results.tstat = asymptotic t-stat
%         results.yhat  = yhat
%         results.resid = residuals
%         results.sige  = sige = (y-p*W*y)'*(y-p*W*y)/n
%         results.rsqr  = rsquared
%         results.lik   = -log likelihood
%         results.nobs  = nobs
%         results.nvar  = nvar = 1
%         results.y     = y data vector
%         results.iter   = # of iterations taken
%         results.romax  = 1/maximum eigenvalue of W
%         results.romin  = 1/minimum eigenvalue of W
%  --------------------------------------------------
%  SEE ALSO: prt(results), sar, sem, sac
% ---------------------------------------------------
options = zeros(1,18);
if nargin == 2 % set default optimization options
options(1,1) = 0; options(1,2) = 1.e-8; options(14) = 500;
elseif nargin == 3 % set user supplied convergence option
options(1,1) = 0; options(1,2) = convg; options(1,14) = 500;
elseif nargin == 4 % set user supplied convg and maxit options
options(1,1) = 0; options(1,2) = convg; options(1,14) = maxit;
else, error('Wrong # of arguments to far');
end;
[n junk] = size(y); results.y = y;      results.nobs = n;
results.nvar = 1;   results.meth = 'far';
opt.tol = 1e-3; opt.disp = 0;
lambda = eigs(sparse(W),speye(n),2,'BE',opt);
lmin = 1/lambda(2);        lmax = 1/lambda(1);
results.romax = lmax;      results.romin = lmin;
% step 1) maximize concentrated likelihood function;
[p options] = fmin('f_far',1/lmin,1/lmax,options,y,W);
if options(10) == options(14),
fprintf(1,'far: convergence not obtained in %4d iterations',options(14));
else, results.iter = options(10);
end;
Wy = sparse(W)*y; epe = (y - p*Wy)'*(y-p*Wy); sige = epe/n;
results.rho = p; results.yhat = p*Wy;
results.resid = y - results.yhat; results.sige = sige;
% asymptotic t-stats using numerical hessian
parm = [p
        sige];
hessn = hessian('f2_far',parm,y,W); xpxi = inv(hessn);
results.tstat = results.rho/sqrt(xpxi(1,1));
ym = y - mean(y); rsqr1 = results.resid'*results.resid;
rsqr2 = ym'*ym; results.rsqr = 1.0-rsqr1/rsqr2;   % r-squared
results.lik = -f2_far(p,y,W);                     % -log likelihood
```

As an example of using this function, we solve for maximum likelihood estimates using a large sample of 3,107 observations representing counties in the continental U.S. from Pace and Berry (1997). They examine presidential election results for this large sample of observations covering the U.S. presidential election of 1980 between Carter and Reagan. The variable we wish to explain using the first-order spatial autoregressive model is the proportion of total possible votes cast for both candidates. Only persons 18 years and older are eligible to vote, so the proportion is based on those voting for both candidates divided by the population over 18 years of age.

Example 11.4 shows a MATLAB program that reads in data from a file 'elect.dat' that contains 3,107 rows with the county-level sample data. Another file named 'ford.dat' holds the first-order contiguity matrix information in sparse matrix storage form. This allows us to store the information in three columns containing 12,429 rows with the non-zero values. We rely on the MATLAB **sparse** command to construct the first-order contiguity matrix from the compactly stored information.

```
% ----- Example 11.4 Using the far() function
%       with very large data set from Pace and Berry

load elect.dat;             % load data on votes
y = elect(:,7)./elect(:,8); % proportion of voters casting votes
ydev = y - mean(y);         % deviations from the means form
clear y;     % conserve on RAM memory
clear elect; % conserve on RAM memory
load ford.dat; % 1st order contiguity matrix stored in sparse matrix form
ii = ford(:,1); jj = ford(:,2); ss = ford(:,3);
n = 3107;
clear ford; % clear ford matrix to save RAM memory
W = sparse(ii,jj,ss,n,n);
clear ii; clear jj; clear ss; % conserve on RAM memory
tic; res = far(ydev,W); toc;
prt(res);
```

A breakdown of time needed to solve various aspects of the estimation problem is shown along with the output below. Our use of the simplex optimization algorithm required 13 function evaluations which took 10.6 seconds. The total time required to compute the estimates and measures of dispersion for ρ and σ, the $R-$squared statistics and log likelihood function value was around 100 seconds.

```
elapsed_time = 59.8226 % computing min,max eigenvalues
elapsed_time = 10.6622 % time required for simplex solution of rho
elapsed_time =  1.7681 % time required for hessian evaluation
elapsed_time =  1.7743 % time required for likelihood evaluation
```

```
First-order spatial autoregressive model Estimates
R-squared       =    0.5375
sigma^2         =    0.0054
Nobs, Nvars     =   3107,     1
log-likelihood  =        3506.3203
# of iterations =     13
min and max rho =   -1.0710,   1.0000
***************************************************************
Variable      Coefficient      t-statistic    t-probability
rho              0.721474        59.567710         0.000000
```

11.3 Gibbs sampling and sparse matrices

We have already used the first-order spatial autoregressive model to illustrate Metropolis-within-Gibbs sampling in Chapter 6. Here we extend the approach to include the use of sparse matrix algorithms that will allow us to solve very large models of the type illustrated in the previous section.

We focus attention on implementation details concerned with constructing a MATLAB function **far_g** that will produce estimates for a Bayesian variant of the first-order spatial autoregressive model. This function will rely on a sparse matrix algorithm approach to handle problems involving large data samples. It will also allow for diffuse or informative priors and handle the case of heterogeneity in the disturbance variance.

The first thing we need to consider is that in order to produce a large number of draws, say 1,000, we would need to evaluate the conditional distribution of ρ given σ 2,000 times since we need to call this function twice during metropolis sampling. Each evaluation would require that we compute the determinant of the matrix $(I_n - \rho W)$, which we have already seen is a non-trivial task for large data samples. To avoid this, we rely on an approach suggested by Pace and Berry (1997). They suggested evaluating this determinant over a grid of values in the feasible range of ρ once at the outset. Given that we have carried out this evaluation and stored the values for the determinant and associated values of ρ, we can simply "look-up" the appropriate determinant in our function that evaluates the conditional distribution. Each call to the conditional distribution function will provide a value of ρ for which we need to evaluate the conditional distribution. If we already know the determinant for a grid of all feasible ρ values, we can simply look up the determinant value closest to the ρ value and use it during evaluation of the conditional distribution. This saves us the time involved in computing the determinant twice for each draw of ρ.

The code that we execute at the outset in our function **far_g** to compute determinant values over a grid of ρ values is shown below. After finding the minimum and maximum eigenvalues using our approach from the previous section, we define a grid based on increments of 0.01 over these values and evaluate the determinant over this grid.

```
opt.tol = 1e-3; opt.disp = 0;
lambda = eigs(sparse(W),speye(n),2,'BE',opt);
lmin = 1/lambda(2);       lmax = 1/lambda(1);
results.romax = lmax; results.romin = lmin;
% compute a detval vector based on Pace and Berry's approach
rvec = lmin:.01:lmax;
spparms('tight');
z = speye(n) - 0.1*sparse(W);
p = colmmd(z);
niter = length(rvec);
detval = zeros(niter,2);
for i=1:niter;
  rho = rvec(i); z = speye(n) - rho*sparse(W);
  [l,u] = lu(z(:,p));
  detval(i,1) = sum(log(abs(diag(u)))); detval(i,2) = rho;
end;
```

Note that we save the values of the determinant alongside the associated values of ρ in a 2-column matrix named **detval**. We will simply pass this matrix to the conditional distribution function **c_far** which is shown below:

```
function cout = c_far(rho,y,W,detval,sige,c,T)
% PURPOSE: evaluate the conditional distribution of rho given sige
%  1st order spatial autoregressive model using sparse matrix algorithms
% ---------------------------------------------------
%  USAGE:cout = c_far(rho,y,W,detval,c,T)
%  where:  rho  = spatial autoregressive parameter
%          y    = dependent variable vector
%          W    = spatial weight matrix
%        detval = an (ngrid,2) matrix of values for det(I-rho*W)
%                 over a grid of rho values
%                 detval(:,1) = determinant values
%                 detval(:,2) = associated rho values
%          sige = sige value
%          c    = optional prior mean for rho
%          T    = optional prior variance for rho
% ---------------------------------------------------
%  RETURNS: a conditional used in Metropolis-Hastings sampling
%  NOTE: called only by far_g
%  --------------------------------------------------
%  SEE ALSO: far_g, c_sar, c_sac, c_sem
```

```
% ------------------------------------------------
i1 = find(detval(:,2) <= rho + 0.005);
i2 = find(detval(:,2) <= rho - 0.005);
i1 = max(i1); i2 = max(i2);
index = round((i1+i2)/2);
detm = detval(index,1); n = length(y);
z = speye(n) - rho*sparse(W);
if nargin == 5, % diffuse prior
epe = (n/2)*log(y'*z'*z*y);
elseif nargin == 7 % informative prior
epe = (n/2)*log(y'*z'*z*y + (rho-c)^2/T);
end;
cout = -epe -(n/2)*log(sige) + detm;
```

In the function **c_far**, we find the determinant value that is closest to the ρ value for which we are evaluating the conditional distribution. This is very fast in comparison to calculating the determinant. Since we need to carry out a large number of draws, this approach works better than computing determinants for every draw.

Another point is that we allow for the case of a normally distributed informative prior on the parameter ρ in the model, which changes the conditional distribution slightly. The documentation for our function **far_g** is shown below.

```
 PURPOSE: Gibbs sampling estimates of the 1st-order Spatial
          model: y = rho*W*y + e,    e = N(0,sige*V),
          V = diag(v1,v2,...vn), r/vi = ID chi(r)/r, r = Gamma(m,k)
          rho = N(c,T),  sige = gamma(nu,d0)
----------------------------------------------------------------
 USAGE: result =  far_g(y,W,ndraw,nomit,prior,start)
 where: y = nobs x 1 independent variable vector
        W = nobs x nobs 1st-order contiguity matrix (standardized)
       prior = a structure variable for prior information input
               prior.rho,  prior mean for rho,   c above, default = diffuse
               priov.rcov, prior rho  variance , T above, default = diffuse
               prior.nu,   informative Gamma(nu,d0) prior on sige
               prior.d0    informative Gamma(nu,d0) prior on sige
                           default for above: nu=0,d0=0 (diffuse prior)
               prior.rval, r prior hyperparameter, default=4
               prior.m,    informative Gamma(m,k) prior on r
               prior.k,    informative Gamma(m,k) prior on r
               prior.rmin, (optional) min value of rho to use in sampling
               prior.rmax, (optional) max value of rho to use in sampling
       ndraw = # of draws
       nomit = # of initial draws omitted for burn-in
       start = (optional) (2x1) vector of rho, sige starting values
```

```
             (defaults, rho = 0.5, sige = 1.0)
---------------------------------------------------------------
 RETURNS: a structure:
          results.meth   = 'far_g'
          results.bdraw  = rho draws (ndraw-nomit x 1)
          results.sdraw  = sige draws (ndraw-nomit x 1)
          results.vmean  = mean of vi draws (nobs x 1)
          results.rdraw  = r-value draws (ndraw-nomit x 1)
          results.pmean  = rho prior mean    (if prior input)
          results.pstd   = rho prior std dev (if prior input)
          results.nu     = prior nu-value for sige (if prior input)
          results.d0     = prior d0-value for sige (if prior input)
          results.r      = value of hyperparameter r (if input)
          results.m      = m prior parameter (if input)
          results.k      = k prior parameter (if input)
          results.nobs   = # of observations
          results.ndraw  = # of draws
          results.nomit  = # of initial draws omitted
          results.y      = actual observations
          results.yhat   = mean of posterior for y-predicted (nobs x 1)
          results.time   = time taken for sampling
          results.accept = acceptance rate
          results.pflag  = 1 for prior, 0 for no prior
          results.rmax  = 1/max eigenvalue of W (or rmax if input)
          results.rmin  = 1/min eigenvalue of W (or rmin if input)
----------------------------------------------------------------
 NOTE: use either improper prior.rval
       or informative Gamma prior.m, prior.k, not both of them
----------------------------------------------------------------
```

As the documentation makes clear, there are a number of user options to facilitate different models. We can use the function for homoscedastic as well as heteroscedastic data samples as well as those containing outliers. This was accomplished using the approach of Geweke (1993) illustrated in Chapter 6. In addition, an informative prior can be used for the parameter σ.

Example 11.5 illustrates using the function with various input options on the large Pace and Berry data set. We set an r value of 5 which will capture heterogeneity if it exists.

```
% ----- Example 11.5 Gibbs sampling with sparse matrices
%       using a very large data set from Pace and Berry
load elect.dat;                   % load data on votes in 3,107 counties
y =  (elect(:,7)./elect(:,8)); % convert to per capita variables
ydev = y - mean(y);
clear elect;                      % conserve on RAM memory
load ford.dat; % 1st order contiguity matrix stored in sparse matrix form
```

```
ii = ford(:,1); jj = ford(:,2); ss = ford(:,3);
n = 3107;
clear ford; % clear ford matrix to save RAM memory
W = sparse(ii,jj,ss,n,n);
clear ii; clear jj; clear ss;  % conserve on RAM memory
prior.rval = 5; ndraw = 1100; nomit = 100;
res = far_g(ydev,W,ndraw,nomit,prior);
prt(res);
```

We also present the maximum likelihood results for comparison with the Gibbs sampling results. If there is no substantial heterogeneity in the disturbance, the two sets of estimates should be similar, as we saw in Chapter 6. From the results, we see that the estimates are similar, suggesting a lack of heterogeneity that would lead to different estimated values for ρ and σ.

```
% Maximum likelihood results
First-order spatial autoregressive model Estimates
R-squared       =    0.5375
sigma^2         =    0.0054
Nobs, Nvars     =   3107,     1
log-likelihood  =        3506.3203
# of iterations =     13
min and max rho =   -1.0710,   1.0000
***************************************************************
Variable      Coefficient      t-statistic    t-probability
rho              0.721474        59.567710         0.000000
% Gibbs sampling estimates
Gibbs sampling First-order spatial autoregressive model
R-squared       =    0.5711
sigma^2         =    0.0050
r-value         =      5
Nobs, Nvars     =   3107,     1
ndraws,nomit    =   1100,   100
acceptance rate =    0.8773
time in secs    =  378.3616
min and max rho =   -1.0710,   1.0000
***************************************************************
Variable      Coefficient      t-statistic    t-probability
rho              0.718541        42.026284         0.000000
```

Note that the time needed to produce 1100 draws was around 378 seconds, making this estimation method competitive with the maximum likelihood approach which took around 100 seconds.

11.4 Chapter summary

We illustrated the use of sparse matrix functions that represent an important feature of MATLAB. These functions allow the solution of problems involving large but sparse matrices with a minimum of both time and computer memory.

Our illustrations demonstrated the use of these functions using a spatial econometrics problem, but other examples where these functions would be of use surely exists in econometrics. Cases where block diagonal matrices are used would be one example.

Extensive use of the approach presented in this chapter has been made to implement a host of maximum likelihood and Gibbs sampling implementations of spatial autoregressive estimation functions contained in a *spatial econometrics library*.

Chapter 11 Appendix

The *spatial econometrics library* is in a subdirectory **spatial**.

```
spatial econometrics library
------- spatial econometrics functions -----------
casetti     - Casetti's spatial expansion model
darp        - Casetti's darp model
far         - 1st order spatial AR model    - y = pWy + e
far_g       - Gibbs sampling Bayesian far model
gwr         - geographically weighted regression
gwr_logit   - logit version of gwr
gwr_probit  - probit version of gwr
bgwr        - Bayesian geographically weighted regression
bgwrv       - robust geographically weighted regression
lmerror     - LM error statistic for regression model
lmsar       - LM error statistic for sar model
lratios     - Likelihood ratio statistic for regression models
moran       - Moran's I-statistic
normxy      - isotropic normalization of x-y coordinates
normw       - normalizes a spatial weight matrix
sac         - spatial model  - y = p*W1*y + X*b + u, u = c*W2*u + e
sac_g       - Gibbs sampling Bayesian sac model
sacp_g      - Gibbs sampling Bayesian sac probit model
sact_g      - Gibbs sampling Bayesian sac tobit model
sar         - spatial autoregressive model  - y = p*W*y + X*b + e
sar_g       - Gibbs sampling Bayesian sar model
sarp_g      - Gibbs sampling Bayesian sar probit model
sart_g      - Gibbs sampling Bayesian sar tobit model
sem         - spatial error model  - y = X*b +u, u=c*W + e
sem_g       - Gibbs sampling Bayesian spatial error model
semp_g      - Gibbs sampling Bayesian spatial error probit model
semt_g      - Gibbs sampling Bayesian spatial error tobit model
semo        - spatial error model (optimization solution)
sdm         - spatial Durbin model  y = a + X*b1 + W*X*b2 + e
sdm_g       - Gibbs sampling Bayesian sdm model
sdmp_g      - Gibbs sampling Bayesian sdm probit model
sdmt_g      - Gibbs sampling Bayesian sdm tobit model
slag        - creates spatial lags
```

```
walds        - Wald test for regression models
xy2cont      - constructs a contiguity matrix from x-y coordinates

------- demonstration programs  -----------
casetti_d   - Casetti model  demo
darp_d      - Casetti darp demo
darp_d2     - darp for all data observations
far_d       - demonstrates far using a small data set
far_d2      - demonstrates far using a large data set
far_gd      - far Gibbs sampling with small data set
far_gd2     - far Gibbs sampling with large data set
gwr_d       - geographically weighted regression demo
gwr_d2      - GWR demo with Harrison-Rubinfeld Boston data
bgwr_d      - demo of Bayesian GWR
bgwr_d2     - BGWR demo with Harrison-Rubinfeld Boston data
lmerror_d   - lmerror demonstration
lmsar_d     - lmsar demonstration
lratios_d   - likelihood ratio demonstration
moran_d     - moran demonstration
sac_d       - sac model demo
sac_d2      - sac model demonstration large data set
sac_gd      - sac Gibbs sampling demo
sac_gd2     - sac Gibbs demo with large data set
sacp_gd     - sac Gibbs probit demo
sact_gd     - sac Gibbs tobit demo
sact_gd2    - sac tobit right-censoring demo
sar_d       - sar model demonstration
sar_d2      - sar model demonstration large data set
sar_gd      - sar Gibbs sampling demo
sar_gd2     - sar Gibbs demo with large data set
sarp_gd     - sar probit Gibbs sampling demo
sart_gd     - sar tobit model Gibbs sampling demo
sart_gd2    - sar tobit right-censoring demo
sdm_d       - sdm model demonstration
sdm_d2      - sdm model demonstration large data set
sdm_gd      - sdm Gibbs sampling demo
sdm_gd2     - sdm Gibbs demo with large data set
sdmp_g      - sdm Gibbs probit demo
sdmt_g      - sdm Gibbs tobit demo
sem_d       - sem model demonstration
sem_d2      - sem model demonstration large data set
sem_gd      - sem Gibbs sampling demo
sem_gd2     - sem Gibbs demo with large data set
semo_d      - semo function demonstration
semo_d2     - semo demo with large data set
semp_gd     - sem Gibbs probit demo
semt_gd     - sem Gibbs tobit demo
semt_gd2    - sem tobit right-censoring demo
slag_d      - demo of slag function
```

```
walds_d      - Wald test demonstration
xy2cont_d    - xy2cont demo

------- support functions  -----------
anselin.dat - Anselin (1988) Columbus crime data
boston.dat  - Harrison-Rubinfeld Boston data set
latit.dat   - latittude for HR data
longi.dat   - longitude for HR data
c_far       - used by far_g
c_sem       - used by sem_g
c_sar       - used by sar_g
c_sdm       - used by sdm_g
c_sac       - used by sac_g
darp_lik1   - used by darp
darp_lik2   - used by darp
elect.dat   - Pace and Barry 3,107 obs data set
ford.dat    - Pace and Barry 1st order contiguity matrix
f_far       - far model likelihood (concentrated)
f_sac       - sac model likelihood (concentrated)
f_sar       - sar model likelihood (concentrated)
f_sem       - sem model likelihood (concentrated)
f_sdm       - sdm model likelihood (concentrated)
f2_far      - far model likelihood
f2_sac      - sac model likelihood
f2_sar      - sar model likelihood
f2_sem      - sem model likelihood
f3_sem      - semo model likelihood
f2_sdm      - sdm model likelihood
prt_gwr     - prints gwr_reg results structure
prt_spat    - prints results from spatial models
scoref      - used by gwr
wmat.dat    - Anselin (1988) 1st order contiguity matrix
```

References

Albert, J. and S. Chib. 1993. "Bayes inference via Gibbs sampling of autoregressive time series subject to Markov mean and variance shifts," *Journal of Business and Economic Statistics*, Vol. 11, pp. 1-15.

Albert, James A. and Siddartha Chib. 1993. "Bayesian Analysis of Binary and Polytomous Response Data". *Journal of the American Statistical Association*, Vol. 88, pp. 669-679.

Anselin, Luc. 1988. *Spatial Econometrics: Methods and Models*, (Dordrecht: Kluwer Academic Publishers).

Belsley, D.A., R.E. Welsch and E. Kuh, 1980. *Regression Diagnostics*, John Wiley & Sons, Inc. (New York: New York).

Best, N.G., M.K. Cowles, and S.K. Vines. 1995. *CODA: Manual version 0.30.* Biostatistics Unit, Cambridge U.K. http://www.mrc-bsu.cam.ac.uk

Box, George E.P., and George C. Tiao. 1992. *Bayesian Inference in Statistical Analysis*, (John Wiley and Sons: New York).

Casella, G. and E.I. George. 1992. "Explaining the Gibbs Sampler", *American Statistician*, Vol. 46, pp. 167-174.

Chib, Siddhartha. 1993. "Bayes regression with autoregressive error: A Gibbs sampling approach," *Journal of Econometrics*, Vol. 58, pp. 275-294.

Chib, Siddhartha. 1992. "Bayes inference in the Tobit censored regression model", *Journal of Econometrics.* Vol. 51, pp. 79-100.

Cook, R.D. and S. Weisberg, 1982. *Residuals and Influence in Regression*, Chapman and Hall, (New York, London).

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, pp. 1-38.

Dickey, David A., Dennis W. Jansen and Daniel L. Thornton. 1991. "A primer on cointegration with an application to money and income," *Federal Reserve Bulletin, Federal Reserve Bank of St. Louis*, March/April, pp. 58-78.

Doan, Thomas, Robert. B. Litterman, and Christopher A. Sims. 1984. "Forecasting and conditional projections using realistic prior distributions," *Econometric Reviews*, Vol. 3, pp. 1-100.

Engle, Robert F. and Clive W.J. Granger. 1987. "Co-integration and error Correction: Representation, Estimation and Testing," *Econoemtrica*, Vol. 55, pp. 251-76.

Estrella, Artuto. 1998. "A new measure of fit for equations with dichotmous dependent variable", *Journal of Business & Economic Statistics*, Vol. 16, no. 2, pp. 198-205.

Fomby, Thomas B., R. Carter Hill and Stanley R. Johnson. 1984. *Advanced Econometric Methods*, (Springer Verlag: New York).

Geman, S., and D. Geman. 1984. "Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images," IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 6, pp. 721-741.

Gelfand, Alan 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.

Gelfand, Alan E., Susan E. Hills, Amy Racine-Poon and Adrian F.M. Smith. 1990. "Illustration of Bayesian Inference in Normal Data Models Using Gibbs Sampling", *Journal of the American Statistical Association*, Vol. 85, pp. 972-985.

Gelman, Andrew, John B. Carlin, Hal S. Stern, and Donald B. Rubin. 1995. *Bayesian Data Analysis*, (London: Chapman & Hall).

Geweke, John. 1991. "Efficient Simulation from the Multivariate normal and Student-t Distributions Subject to Linear Constraints," in Computing Science and Statistics: Proceedings of the Twenty-Third Symposium on the Interface.

Geweke, John. 1992. "Evaluating the Accuracy of Sampling-Based Approaches to the Calculation of Posterior Moments", *Bayesian Statistics, 4*, J.M Bernardo, J.0. Berger, A.P. Dawid, and A.F.M Smith, (eds.), pp. 641-649. (Oxford: Clarendon Press).

Geweke, John. 1993. "Bayesian Treatment of the Independent Student t Linear Model", *Journal of Applied Econometrics*, Vol. 8, s19-s40.

Geweke, John and Michael Keane. 1997. "Mixture of Normals Probit Models", *Research Department Staff Report 237*, Federal Reserve Bank of Minneapolis.

Gilks, W.R., S. Richardson and D.J. Spiegelhalter. 1996. *Markov Chain Monte Carlo in Practice*, (London: Chapman & Hall).

Goldfeld S.M. and R. E. Quandt 1973. "A Markov model for switching regressions", *Journal of Econometrics*, Vol. 1, pp. 3-16.

Green, William H., 1997. *Econometric Analysis, Third Edition*, (Prentice Hall: New Jersey).

Hamilton, James D. 1989. "A new approach to economic analysis of nonstationary time series and the business cycle", *Econometrica,* Vol. 57, number 2, pp. 357-384.

Hanselmann, D. adn B. Littlefield. 1997. *The Student Edition of MATLAB, Version 5 User's Guide*. (New Jersey: Prentice Hall).

Hastings, W. K. 1970. "Monte Carlo sampling methods using Markov chains and their applications," *Biometrika*, Vol. 57, pp. 97-109.

Hoerl A.E., and R.W. Kennard, 1970. "Ridge Regression: Biased Estimation and Applications for Nonorthogonal Problems," *Technometrics*, Vol. 12, pp. 55-82.

Hoerl A.E., R.W. Kennard and K.F. Baldwin, 1975. "Ridge Regression: Some Simulations," *Communications in Statistics, A*, Vol. 4, pp. 105-23.

Johansen, Soren. 1988. "Statistical Analysis of Co-integration vectors," *Journal of Economic Dynamics and Control*, Vol. 12, pp. 231-254.

Johansen, Soren. 1995. *Likelihood-based Inference in Cointegrated Vector autoregressive Models*, Oxford: Oxford University Press.

Johansen, Soren and Katarina Juselius. 1990. 'Maximum likelihood estimation and inference on cointegration - with applications to the demand for money", *Oxford Bulletin of Economics and Statistics*, Vol. 52, pp. 169-210.

Judge G. C., R.C. Hill, W.E. Griffiths, H. Lutkepohl, T. Lee, 1988. *Introduction to the Theory and Practice of Econometrics,* John Wiley & Sons, Inc. (New York: New York).

Kim, Chaing-Jin. 1994. "Dynamic linear models with Markov switching", *Journal of Econometrics*, Vol. 60, pp. 1-22.

Lange, K.L., R.J.A. Little, and J.M.G. Taylor. 1989. "Robust Statistical Modeling Using the t Distribution," *Journal of the American Statistical Association*, Vol. 84, pp. 881-896.

Leamer, Edward E. 1983. "Model Choice and Specification Analysis", in *Handbook of Econometrics, Volume 1*, Zvi Griliches and Michael D. Intriligator, eds. (North-Holland: Amsterdam).

LeSage, James P. 1990. "A Comparison of the Forecasting Ability of ECM and VAR Models," *Review of Economics and Statistics*, Vol. 72, pp. 664-671.

LeSage, James P. 1997. "Bayesian Estimation of Spatial Autoregressive Models", *International Regional Science Review*, 1997 Vol. 20, number 1&2, pp. 113-129. Also available at www.econ.utoledo.edu.

LeSage, James P. 1998. "Bayesian Estimation of Spatial Probit/Tobit Models", Available at www.econ.utoledo.edu.

LeSage, James P. and Anna Krivelyova. 1999. "A Spatial Prior for Bayesian Vector Autoregressive Models," *Journal of Regional Science*, Vol. 39, number 2, pp. 297-317.

LeSage, James P. and Anna Krivelova. 1998. "A Random Walk Averaging Prior for Bayesian Vector Autoregressive Models," available at: www.econ.utoledo.edu.

LeSage, James P. and Michael Magura. 1991. "Using interindustry input-output relations as a Bayesian prior in employment forecasting models", *International Journal of Forecasting*, Vol. 7, pp. 231-238.

LeSage, James P. and Zheng Pan. 1995. 'Using Spatial Contiguity as Bayesian Prior Information in Regional Forecasting Models", *International Regional Science Review*, Vol. 18, no. 1, pp. 33-53.

Lindley, David V. 1971. "The estimation of many parameters," in *Foundations of Statistical Science*, V.P. Godambe and D.A. Sprout (eds.) (Toronto: Holt, Rinehart, and Winston).

Litterman, Robert B. 1986. "Forecasting with Bayesian Vector Autoregressions — Five Years of Experience," *Journal of Business & Economic Statistics*, Vol. 4, pp. 25-38.

MacKinnon, J.G. 1994 "Approximate Asymptotic Distribution Functions for unit-root and cointegration tests," *Journal of Business & Economic Statistics*, Vol. 12, pp. 167-176.

MacKinnon, J.G. 1996 "Numerical distribution functions for unit-root and cointegration tests," *Journal of Applied Econometrics*, Vol. 11, pp. 601-618.

McFadden, Daniel. 1984. "Econometric Analysis of Qualitative Response Models", in Zvi Griliches and Michael D. Intriligator, eds. *Handbook of Econometrics, Volume 2*, (North-Holland: Amsterdam).

McMillen, Daniel P. 1992. "Probit with Spatial Autocorrelation", *Journal of Regional Science*, Vol. 32, no. 3, pp. 335-348.

Metroplis, N., A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller and E. Teller. 1953. "Equation of state calculations by fast computing machines," *Journal of Chemical Physics*, Vol. 21, pp. 1087-1092.

Pace, R. Kelley and Ronald Barry. 1997. "Quick Computation of Spatial Autoregressive Estimators", forthcoming in *Geographical Analysis*.

Raftery, Adrian E., and Steven M. Lewis. 1992a. "How many iterations in the Gibbs sampler?", in *Bayesian Statistics*, Vol. 4, J.M. Bernardo, J.O. Berger, A.P. Dawid and A.F.M Smith, eds. (Oxford University Press: Oxford, pp. 763-773.)

Raftery, Adrian E., and Steven M. Lewis. 1992b. "One long run with diagnostics: Implementation strategies for Markov chain Monte Carlo", *Statistical Science*, Vol. 7, pp. 493-497.

Raftery, Adrian E., and Steven M. Lewis. 1995. "The number of iterations, convergence diagnostics and generic Metropolis algorithms", forthcoming in *Practical Markov Chain Monte Carlo*, W.R. Gilks, D.J. Spiegelhalter and S. Richardson, eds. (Chapman and Hall: London).

Raftery, Adrian E., David Madigan and Jennifer A. Hoeting. 1997. "Bayesian model averaging for linear regression models,", *Journal of the American Statistical Association*, Vol. 92, pp. 179-191.

Shoesmith, Gary L. 1992. "Cointegration, error Correction and Improved Regional VAR Forecasting," *Journal of Forecasting*, Vol. 11, pp. 91-109.

Shoesmith, Gary L. 1995. "Multiple Cointegrating Vectors, error Correction, and Litterman's Model" *International Journal of Forecasting*, Vol. 11, pp. 557-567.

Simon, S.D., and J.P. LeSage, 1988a. "The Impact of Collinearity Involving the Intercept Term on the Numerical Accuracy of Regression," *Computational Statistics in Economics and Management Science*, Vol. 1 no. 2, pp. 137-152.

Simon, S.D., and J.P. LeSage, 1988b. "Benchmarking Numerical Accuracy of Statistical Algorithms," with Stephen D. Simon, *Computational Statistics and Data Analysis*, Vol. 7, pp. 197-209.

Sims, Christopher A. 1980. "Macroeconomics and Reality," *Econometrica* Vol. 48, pp. 1-48.

Smith, A.F.M and G.O. Roberts (1992). "Bayesian Statistics without Tears: A Sampling-Resampling Perspective", *The American Statistician*, Vol. 46, pp. 84-88.

Spector, L., and M. Mazzeo. 1980. "Probit Analysis and Economic Education." *Journal of Economic Education*, Vol. 11, pp. 37-44.

Tanner, Martin A. 1991. *Tools for Statistical Inference*, (Springer-Verlag: New York).

Theil, Henri., 1971. *Principles of Econometrics*, (John Wiley & Sons: New York).

Theil, Henri and Arthur S. Goldberger. 1961. "On Pure and Mixed Statistical Estimation in Economics," *International Economic Review*, Vol. 2, 65-78.

van Norden, Simon and Huntley Schaller. 1993. "The predictability of stock market regime: evidence from the Toronto Stock Exchange," *Review of Economics and Statistics*.

Wampler, R.H., 1980. "Test Procedures and Test Problems for Least-Squares Algorithms," *Journal of Econometrics*, Vol. 12, pp. 3-22.

Wecker, William E. 1979. "Predicting the Turning Points of a Time Series," *Journal of Business,* Vol. 55, pp. 57-85.

Zellner, Arnold and Chanisk Hong. 1988. "Bayesian Methods for Forecasting Turning Points in Economic Time Series: Sensitivity of Forecasts to Asymmetricy of Loss Structures," in *Leading Economic Indicators: New Approaches and Forecasting Records*, (K. Lahiri and G. Moore, eds.), pp. 129-140. (England: Cambridge Univ. Press).

Zellner, Arnold, Chanisk Hong, and G. M. Gulati. 1990. "Turning Points in Economic Time Series, Loss Structures and Bayesian Forecasting", in *Bayesian Likelihood Methods in Statistics and Econometrics: Essays in Honor of George A. Barnard*, (S. Geisser, J. Hodges, S.J. Press, and A. Zellner, eds.), pp. 371-393. (Amsterdam: North-Holland Publ.).

Appendix: Toolbox functions

The *Econometric Toolbox* is organized in a set of directories, each containing a different library of functions. When your Internet browser unpacks the compressed file containing the *Econometric Toolbox* the files will be placed in the appropriate directories.

To install the toolbox:

1. create a single subdirectory in the MATLAB toolbox directory:

   ```
   C:\matlab\toolbox\econ
   ```

   Where we have used the name **econ** for the directory.

2. Copy the system of directories to this subdirectory.

3. Use the graphical path tool in MATLAB to add these directories to your path. (On a unix or linux system, you may need to edit your environment variables that set the MATLAB path.) the graphical path tool in MATLAB to add these directories to your path. (On a unix or linux system, you may need to edit your environment variables that set the MATLAB path.)

A listing of the contents file from each subdirectory is presented on the following pages.

The *regression function library* is in a subdirectory **regress**.

```
regression function library

------- regression program functions -----------

ar_g       - Gibbs sampling Bayesian autoregressive model
bma_g      - Gibbs sampling Bayesian model averaging
boxcox     - Box-Cox regression with 1 parameter
boxcox2    - Box-Cox regression with 2 parameters
egarchm    - EGARCH(p,q)-in-Mean regression model
emhergott  - EM estimates of Hamilton's markov switching model
garchs     - garchs(1,1,1) model using non-central t-distribution
hmarkov_em - Hamilton's markov switching model
hwhite     - Halbert White's heteroscedastic consistent estimates
lad        - least-absolute deviations regression
lm_test    - LM-test for two regression models
logit      - logit regression
mlogit     - multinomial logit regression
nwest      - Newey-West hetero/serial consistent estimates
ols        - ordinary least-squares
ols_g      - Gibbs sampling Bayesian linear model
olsar1     - Maximum Likelihood for AR(1) errors ols model
olsc       - Cochrane-Orcutt AR(1) errors ols model
olst       - regression with t-distributed errors
probit     - probit regression
probit_g   - Gibbs sampling Bayesian probit model
ridge      - ridge regression
robust     - iteratively reweighted least-squares
rtrace     - ridge estimates vs parameters (plot)
sur        - seemingly unrelated regressions
switch_em  - switching regime regression using EM-algorithm
theil      - Theil-Goldberger mixed estimation
thsls      - three-stage least-squares
tobit      - tobit regression
tobit_g    - Gibbs sampling Bayesian tobit model
tsls       - two-stage least-squares
waldf      - Wald F-test

-------- Demonstration programs ------------------

ar_gd        - demonstration of Gibbs sampling ar_g
bma_gd       - demonstrates Bayesian model averaging
boxcox_d     - demonstrates Box-Cox 1-parameter model
boxcox2_d    - demonstrates Box-Cox 2-parmaeter model
demo_all     - demos most regression functions
egarchm_d    - demos egarchm function
garchs_d     - demos garchs function
hmarkov_emd  - demos Hamilton's model
```

```
hmarkov_emd2 - another demo of Hamilton's model
hwhite_d     - H. White's hetero consistent estimates demo
lad_d        - demos lad regression
lm_test_d    - demos lm_test
logit_d      - demonstrates logit regression
mlogit_d     - demonstrates multinomial logit
nwest_d      - demonstrates Newey-West estimates
ols_d        - demonstrates ols regression
ols_d2       - Monte Carlo demo using ols regression
ols_gd       - demo of Gibbs sampling ols_g
olsar1_d     - Max Like AR(1) errors model demo
olsc_d       - Cochrane-Orcutt demo
olst_d       - olst demo
probit_d     - probit regression demo
probit_gd    - demo of Gibbs sampling Bayesian probit model
ridge_d      - ridge regression demo
robust_d     - demonstrates robust regression
sur_d        - demonstrates sur using Grunfeld's data
switch_emd   - demonstrates switching regression
theil_d      - demonstrates theil-goldberger estimation
thsls_d      - three-stage least-squares demo
tobit_d      - tobit regression demo
tobit_d2     - tobit right-censoring demo
tobit_gd     - demo of Gibbs sampling Bayesian tobit model
tobit_gd2    - Bayesian tobit right-censoring demo
tsls_d       - two-stage least-squares demo
waldf_d      - demo of using wald F-test function

-------- Support functions ------------------------

ar1_like     - used by olsar1   (likelihood)
bmapost      - used by bma_g
box_lik      - used by box_cox  (likelihood)
box_lik2     - used by box_cox2 (likelihood)
chis_prb     - computes chi-squared probabilities
dmult        - used by mlogit
egarchm_lik  - likelihood used by egarchm
garchs_llf   - likelihood used by garchs
herg_llf     - likelihood used by emhergott
herg_llf2    - likelihood used by emhergott
hmarkov_llf  - likelihood used by hmarkov_em
hmarkov_llf2 - likelihood used by hmarkov_em
fdis_prb     - computes F-statistic probabilities
find_new     - used by bma_g
grun.dat     - Grunfeld's data used by sur_d
grun.doc     - documents Grunfeld's data set
hessian      - used by tobit to determine numerical hessian
lo_like      - used by logit    (likelihood)
mcov         - used by hwhite
```

```
mderivs      - used by mlogit
mlogit_lik   - used by mlogit
nmlt_rnd     - used by probit_g
nmrt_rnd     - used by probit_g, tobit_g
norm_cdf     - used by probit, pr_like
norm_pdf     - used by prt_reg, probit
olse         - ols returning only residuals (used by sur)
plt_eqs      - plots equation systems
plt_reg      - plots regressions
pr_like      - used by probit   (likelihood)
prt_eqs      - prints equation systems
prt_gibbs    - prints Gibbs sampling models
prt_reg      - prints regressions
prt_swm      - prints switching regression results
sample       - used by bma_g
stdn_cdf     - used by norm_cdf
stdn_pdf     - used by norm_pdf
tdis_prb     - computes t-statistic probabilities
to_like      - used by tobit    (likelihood)
```

The utility functions are in a subdirectory **util**.

```
utility function library

accumulate  - accumulates column elements of a matrix
blockdiag   - creates a block diagonal matrix
cal         - associates obs # with time-series calendar
ccorr1      - correlation scaling to normal column length
ccorr2      - correlation scaling to unit column length
cols        - returns the # of columns in a matrix or vector
crlag       - circular lag function
cumprodc    - returns cumulative product of each column of a matrix
cumsumc     - returns cumulative sum of each column of a matrix
delif       - select matrix values for which a condition is false
diagrv      - replaces main diagonal of square matrix with vector
findnear    - finds matrix element nearest a scalar value
fturns      - finds turning-points in a time-series
growthr     - converts time-series matrix to growth rates
ical        - associates time-series dates with obs #
indexcat    - extract indices equal to a scalar or an interval
indicator   - converts a matrix to indicator variables
invccorr    - inverse for ccorr1, ccorr2
invpd       - makes a matrix positive-definite, then inverts
kernel_n    - normal kernal density estimates
lag         - generates a lagged variable vector or matrix
levels      - generates factor levels variable
lprint      - prints a matrix in LaTeX table-formatted form
lprintf     - enhanced lprint function
matadd      - adds non-conforming matrices, row or col compatible.
```

```
matdiv      - divides non-conforming matrices, row or col compatible.
matmul      - multiplies non-conforming matrices, row or col compatible.
matsub      - divides non-conforming matrices, row or col compatible.
mlag        - generates a var-type matrix of lags
mode        - calculates the mode of a distribution
mprint      - prints a matrix
mprint3     - prints coefficient, t-statistics matrices
mth2qtr     - converts monthly to quarterly data
nclag       - generates a matrix of non-contiguous lags
plt         - wrapper function, plots all result structures
prodc       - returns product of each column of a matrix
prt         - wrapper function, prints all result strucutres
recserar    - recursive AR series (like Gauss)
recsercp    - recursive series product (like Gauss)
roundoff    - rounds matrix to fixed number of decimal digits
rows        - returns the # of rows in a matrix or vector
sacf        - sample autocorrelation function estimates
sdiff       - seasonal differencing
sdummy      - generates seasonal dummy variables
selif       - select matrix values for which a condition is true
seqa        - a sequence of numbers with a beginning and increment
shist       - plots spline smoothed histogram
spacf       - sample partial autocorrelation estimates
stdc        - std deviations of columns  returned as a column vector
sumc        - returns sum of each column
tally       - computes frequencies of distinct levels
tdiff       - time-series differencing
trimc       - trims columns of a matrix (or vector) like Gauss
trimr       - trims rows of a matrix (or vector) like Gauss
tsdates     - time-series dates function
tsprint     - print time-series matrix
unsort      - unsorts a sorted vector or matrix
vec         - turns a matrix into a stacked vector
vech        - matrix from lower triangular columns of a matrix
xdiagonal   - spreads x(nxk) out to X(n*n x n*k) diagonal matrix
yvector     - repeats y(nx1) to form Y(n*n x 1)

-------- demonstration programs -------------

cal_d       - demonstrates cal function
fturns_d    - demonstrates fturns and plt
ical_d      - demonstrates ical function
lprint_d    - demonstrates lprint function
lprintf_d   - demonstrates lprintf function
mprint_d    - demonstrates mprint function
mprint3_d   - demonstrates mprint3 function
sacf_d      - demonstrates sacf
spacf_d     - demonstrates spacf
tsdate_d    - demonstrates tsdate function
```

```
tsprint_d   - demonstrates tsprint function
util_d      - demonstrated some of the utility functions
```

A set of graphing functions are in a subdirectory **graphs**.

```
graphing function library

-------- graphing programs --------------------------

tsplot      - time-series graphs
pltdens     - density plots
pairs       - scatter plot (uses histo)
plt         - plots results from all functions

-------- Demonstration programs ---------------------

tsplot_d   - demonstrates tsplot
pltdens_d  - demonstrates pltdens
plt_turnsd - demonstrates plt_turns
pairs_d    - demonstrates pairwise scatter
plt_d      - demonstrates plt on results structures

------- support functions ---------------------------

histo      - used by pairs
plt_turns  - used by plt to plot turning points
```

A library of routines in the subdirectory **diagn** contain the regression diagnostics functions.

```
regression diagnostics library

------- regression diagnostic programs -----------

arch             - ARCH(p) test
bkw              - BKW collinearity diagnostics
bpagan           - Breusch-Pagan heteroscedasticity test
cusums           - Brown,Durbin,Evans cusum squares test
dfbeta           - BKW influential observation diagnostics
diagnose         - compute diagnostic statistics
plt_dfb          - plots dfbetas
plt_dff          - plots dffits
plt_cus          - plots cusums
recresid         - compute recursive residuals
rdiag            - graphical residuals diagnostics
studentize       - standarization transformation
unstudentize     - reverses studentize transformation
qstat2           - Box-Ljung Q-statistic
```

```
------- demonstration programs --------------------

arch_d           - demonstrates arch
bkw_d            - demonstrates bkw
bpagan_d         - demonstrates bpagan
cusums_d         - demonstrates cusums
dfbeta_d         - demonstrates dfbeta, plt_dfb, plt_dff
diagnose_d       - demonstrates diagnose
recresid_d       - demonstrates recresid
rdiag_d          - demonstrates rdiag
unstudentize_d   - demonstrates studentize, unstudentize
qstat2_d         - demonstrates qstat2

------- support functions -------------------------

plt_cus          - plots cusums test results
plt_dfb          - plots dfbetas
plt_dff          - plots dffits
../util/plt      - plots everything
../regress/ols.m - least-squares regression
```

The vector autoregressive library is in a subdirectory **var_bvar**.

```
vector autoregressive function library

------- VAR/BVAR functions -------------------

becm_g   - Gibbs sampling BECM estimates
becmf    - Bayesian ECM model forecasts
becmf_g  - Gibbs sampling BECM forecasts
bvar     - BVAR model
bvar_g   - Gibbs sampling BVAR estimates
bvarf    - BVAR model forecasts
bvarf_g  - Gibbs sampling BVAR forecasts
ecm      - ECM (error correction) model estimates
ecmf     - ECM model forecasts
irf      - impulse response functions
lrratio  - likelihood ratio tests for lag length
recm     - ecm version of rvar
recm_g   - Gibbs sampling random-walk averaging estimates
recmf    - random-walk averaging ECM forecasts
recmf_g  - Gibbs sampling random-walk averaging forecasts
rvar     - Bayesian random-walk averaging prior model
rvar_g   - Gibbs sampling RVAR estimates
rvarf    - Bayesian RVAR model forecasts
rvarf_g  - Gibbs sampling RVAR forecasts
var      - VAR model
varf     - VAR model forecasts
```

```
------- demonstration programs  -----------

becm_d    - BECM model demonstration
becm_g    - Gibbs sampling BECM estimates demo
becmf_d   - becmf demonstration
becmf_gd  - Gibbs sampling BECM forecast demo
bvar_d    - BVAR model demonstration
bvar_gd   - Gibbs sampling BVAR demonstration
bvarf_d   - bvarf demonstration
bvarf_gd  - Gibbs sampling BVAR forecasts demo
ecm_d     - ECM model demonstration
ecmf_d    - ecmf demonstration
irf_d     - impulse response function demo
irf_d2    - another irf demo
lrratio_d - demonstrates lrratio
pftest_d  - demo of pftest function
recm_d    - RECM model demonstration
recm_gd   - Gibbs sampling RECM model demo
recmf_d   - recmf demonstration
recmf_gd  - Gibbs sampling RECM forecast demo
rvar_d    - RVAR model demonstration
rvar_gd   - Gibbs sampling rvar model demo
rvarf_d   - rvarf demonstration
rvarf_gd  - Gibbs sampling rvar forecast demo
var_d     - VAR model demonstration
varf_d    - varf demonstration

------- support functions  ------------------

johansen  - used by ecm,ecmf,becm,becmf,recm,recmf
lag       - does ordinary lags
mlag      - does var-type lags
nclag     - does contiguous lags (used by rvar,rvarf,recm,recmf)
ols       - used for VAR estimation
pftest    - prints Granger F-tests
pgranger  - prints Granger causality probabilities
prt       - prints results from all functions
prt_coint - used by prt_var for ecm,becm,recm
prt_var   - prints results of all var/bvar models
prt_varg  - prints results of all Gibbs var/bvar models
rvarb     - used for RVARF forecasts
scstd     - does univariate AR for BVAR
theil_g   - used for Gibbs sampling estimates and forecasts
theilbf   - used for BVAR forecasts
theilbv   - used for BVAR estimation
trimr     - used by VARF,BVARF, johansen (in /util/trimr.m)
vare      - used by lrratio (vare uses /regress/olse.m)
```

The co-integration library functions are in a subdirectory **coint**.

```
co-integration library

------ Co-integration testing routines --------

johansen    - carries out Johansen's co-integration tests
adf         - carries out Augmented Dickey-Fuller unit root tests
cadf        - carries out ADF tests for co-integration
phillips    - carries out Phillips-Peron co-integration tests
prt_coint   - prints results from adf,cadf,johansen

------ Demonstration programs -----------------

johansen_d  - demonstrates johansen
adf_d       - demonstrates adf
cadf_d      - demonstrates cadf
phillips_d  - demonstrates phillips

------ Support functions -----------------------

c_sja       - returns critical values for SJ maximal eigenvalue test
c_sjt       - returns critical values for SJ trace test
ztcrit      - returns critical values for adf test
rztcrit     - returns critical values for cadf test
detrend     - used by johansen to detrend data series
ptrend      - used by adf to create time polynomials
trimr       - /util/trimr.m  (like Gauss trimr)
cols        - /util/cols.m   (like Gauss cols)
rows        - /util/rows.m   (like Gauss rows)
tdiff       - /util/tdiff.m  differences
```

The Gibbs convergence diagnostic functions are in a subdirectory **gibbs**.

```
Gibbs sampling convergence diagnostics functions

--------- convergence testing functions ---------

apm       - Geweke's chi-squared test
coda      - convergence diagnostics
momentg   - Geweke's NSE, RNE
raftery   - Raftery and Lewis program Gibbsit for convergence

--------- demonstration programs ----------------

apm_d     - demonstrates apm
coda_d    - demonstrates coda
momentg_d - demonstrates momentg
raftery_d - demonstrates raftery
```

```
--------- support functions ---------------------

prt_coda  - prints coda, raftery, momentg, apm output  (use prt)
empquant  - These were converted from:
indtest   - Rafferty and Lewis FORTRAN program.
mcest     - These function names follow the FORTRAN subroutines
mctest    -
ppnd      -
thin      -
```

Distribution functions are in the subdirectory **distrib**.

```
Distribution functions library

------- pdf, cdf, inverse functions -----------

beta_cdf  - beta(a,b) cdf
beta_inv  - beta inverse (quantile)
beta_pdf  - beta(a,b) pdf
bino_cdf  - binomial(n,p) cdf
bino_inv  - binomial inverse (quantile)
bino_pdf  - binomial pdf
chis_cdf  - chisquared(a,b) cdf
chis_inv  - chi-inverse (quantile)
chis_pdf  - chisquared(a,b) pdf
chis_prb  - probability for chi-squared statistics
fdis_cdf  - F(a,b) cdf
fdis_inv  - F inverse (quantile)
fdis_pdf  - F(a,b) pdf
fdis_prb  - probabililty for F-statistics
gamm_cdf  - gamma(a,b) cdf
gamm_inv  - gamma inverse (quantile)
gamm_pdf  - gamma(a,b) pdf
hypg_cdf  - hypergeometric cdf
hypg_inv  - hypergeometric inverse
hypg_pdf  - hypergeometric pdf
logn_cdf  - lognormal(m,v) cdf
logn_inv  - lognormal inverse (quantile)
logn_pdf  - lognormal(m,v) pdf
logt_cdf  - logistic cdf
logt_inv  - logistic inverse (quantile)
logt_pdf  - logistic pdf
norm_cdf  - normal(mean,var) cdf
norm_inv  - normal inverse (quantile)
norm_pdf  - normal(mean,var) pdf
pois_cdf  - poisson cdf
pois_inv  - poisson inverse
pois_pdf  - poisson pdf
```

```
 stdn_cdf  - std normal cdf
 stdn_inv  - std normal inverse
 stdn_pdf  - std normal pdf
 tdis_cdf  - student t-distribution cdf
 tdis_inv  - student t inverse (quantile)
 tdis_pdf  - student t-distribution pdf
 tdis_prb  - probabililty for t-statistics

 ------- random samples -----------------------

 beta_rnd  - random beta(a,b) draws
 bino_rnd  - random binomial draws
 chis_rnd  - random chi-squared(n) draws
 fdis_rnd  - random F(a,b) draws
 gamm_rnd  - random gamma(a,b) draws
 hypg_rnd  - random hypergeometric draws
 logn_rnd  - random log-normal draws
 logt_rnd  - random logistic draws
 nmlt_rnd  - left-truncated normal draw
 nmrt_rnd  - right-truncated normal draw
 norm_crnd - contaminated normal random draws
 norm_rnd  - multivariate normal draws
 pois_rnd  - poisson random draws
 tdis_rnd  - random student t-distribution draws
 unif_rnd  - random uniform draws (lr,rt) interval
 wish_rnd  - random Wishart draws

 -------- demonstration and test programs ------

 beta_d    - demo of beta distribution functions
 bino_d    - demo of binomial distribution functions
 chis_d    - demo of chi-squared distribution functions
 fdis_d    - demo of F-distribution functions
 gamm_d    - demo of gamma distribution functions
 hypg_d    - demo of hypergeometric distribution functions
 logn_d    - demo of lognormal distribution functions
 logt_d    - demo of logistic distribution functions
 pois_d    - demo of poisson distribution functions
 stdn_d    - demo of std normal distribution functions
 tdis_d    - demo of student-t distribution functions
 trunc_d   - demo of truncated normal distribution function
 unif_d    - demo of uniform random distribution function

-------- support functions ---------------------

 betacfj   - used by fdis_prb
 betai     - used by fdis_prb
 bincoef   - binomial coefficients
 com_size  - test and converts to common size
```

```
gammalnj  - used by fdis_prb
is_scalar - test for scalar argument
```

Optimization functions are in the subdirectory **optimize**.

```
Optimization functions library

 --------------- optimization functions -----------------

 dfp_min    - Davidson-Fletcher-Powell
 frpr_min   - Fletcher-Reeves-Polak-Ribiere
 maxlik     - general all-purpose optimization routine
 pow_min    - Powell conjugate gradient
 solvopt    - yet another general purpose optimization routine

 --------------- demonstration programs -----------------

 optim1_d     - dfp, frpr, pow, maxlik demo
 optim2_d     - solvopt demo
 optim3_d     - fmins demo

 --------------- support functions -----------------------

 apprgrdn      - computes gradient for solvopt
 box_like1     - used by optim3_d
 gradt         - computes gradient
 hessian       - evaluates hessian
 linmin        - line minimization routine (used by dfp, frpr, pow)
 stepsize      - stepsize determination
 tol_like1     - used by optim1_d, optim2_d
 updateh       - updates hessian
```

A library of spatial econometrics functions are in the subdirectory **spatial**.

```
 ------- spatial econometrics functions -----------

 casetti    - Casetti's spatial expansion model
 darp       - Casetti's darp model
 far        - 1st order spatial AR model    - y = pWy + e
 far_g      - Gibbs sampling Bayesian far model
 gwr        - geographically weighted regression
 bgwr       - Bayesian geographically weighted regression
 lmerror    - LM error statistic for regression model
 lmsar      - LM error statistic for sar model
 lratios    - Likelihood ratio statistic for regression models
 moran      - Moran's I-statistic
```

```
sac        - spatial model  - y = p*W1*y + X*b + u, u = c*W2*u + e
sac_g      - Gibbs sampling Bayesian sac model
sar        - spatial autoregressive model  - y = p*W*y + X*b + e
sar_g      - Gibbs sampling Bayesian sar model
sarp_g     - Gibbs sampling Bayesian sar Probit model
sart_g     - Gibbs sampling Bayesian sar Tobit model
sem        - spatial error model  - y = X*b +u, u=c*W + e
sem_g      - Gibbs sampling Bayesian spatial error model
semo       - spatial error model (optimization solution)
sdm        - spatial Durbin model  y = a + X*b1 + W*X*b2 + e
sdm_g      - Gibbs sampling Bayesian spatial Durbin model
walds      - Wald test for regression models
xy2cont    - constructs a contiguity matrix from x-y coordinates

------- demonstration programs  -----------

casetti_d  - Casetti model  demo
darp_d     - Casetti darp demo
darp_d2    - darp for all data observations
far_d      - demonstrates far using a small data set
far_d2     - demonstrates far using a large data set
far_gd     - far Gibbs sampling with small data set
far_gd2    - far Gibbs sampling with large data set
gwr_d      - geographically weighted regression demo
gwr_d2     - GWR demo with Harrison-Rubinfeld Boston data
bgwr_d     - demo of Bayesian GWR
bgwr_d2    - BGWR demo with Harrison-Rubinfeld Boston data
lmerror_d  - lmerror demonstration
lmsar_d    - lmsar demonstration
lratios_d  - likelihood ratio demonstration
moran_d    - moran demonstration
sac_d      - sac model demo
sac_d2     - sac model demonstration large data set
sac_gd     - sac Gibbs sampling demo
sac_gd2    - sac Gibbs demo with large data set
sar_d      - sar model demonstration
sar_d2     - sar model demonstration large data set
sar_gd     - sar Gibbs sampling demo
sar_gd2    - sar Gibbs demo with large data set
sarp_gd    - sar Probit Gibbs sampling demo
sart_gd    - sar Tobit model Gibbs sampling demo
sdm_d      - sdm model demonstration
sdm_d2     - sdm model demonstration large data set
sdm_gd     - sdm Gibbs sampling demo
sdm_gd2    - sdm Gibbs demo with large data set
sem_d      - sem model demonstration
sem_d2     - sem model demonstration large data set
sem_gd     - sem Gibbs sampling demo
sem_gd2    - sem Gibbs demo with large data set
```

```
semo_d     - semo function demonstration
semo_d2    - semo demo with large data set
walds_d    - Wald test demonstration
xy2cont_d  - xy2cont demo

------- support functions  -----------

anselin.dat- Anselin (1988) Columbus crime data
boston.dat - Harrison-Rubinfeld Boston data set
latit.dat  - latittude for HR data
longi.dat  - longitude for HR data
c_far      - used by far_g
c_sem      - used by sem_g
c_sar      - used by sar_g
c_sdm      - used by sdm_g
c_sac      - used by sac_g
darp_lik1  - used by darp
darp_lik2  - used by darp
elect.dat  - Pace and Barry 3,107 obs data set
ford.dat   - Pace and Barry 1st order contiguity matrix
f_far      - far model likelihood (concentrated)
f_sac      - sac model likelihood (concentrated)
f_sar      - sar model likelihood (concentrated)
f_sem      - sem model likelihood (concentrated)
f_sdm      - sdm model likelihood (concentrated)
f2_far     - far model likelihood
f2_sac     - sac model likelihood
f2_sar     - sar model likelihood
f2_sem     - sem model likelihood
f3_sem     - semo model likelihood
f2_sdm     - sdm model likelihood
normxy     - isotropic normalization of x-y coordinates
prt_gwr    - prints gwr_reg results structure
prt_spat   - prints results from spatial models
scoref     - used by gwr
wmat.dat   - Anselin (1988) 1st order contiguity matrix
```