

EViews 4 Command and Programming Reference



Quantitative Micro Software

EViews 4 Command and Programming Reference

Copyright © 1994–2002 Quantitative Micro Software, LLC

All Rights Reserved

Printed in the United States of America

ISBN 1-880411-29-6

Revised for EViews 4.1 - February 2002

This software product, including program code and manual, is copyrighted, and all rights are reserved by Quantitative Micro Software, LLC. The distribution and sale of this product are intended for the use of the original purchaser only. Except as permitted under the United States Copyright Act of 1976, no part of this product may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of Quantitative Micro Software.

Disclaimer

The authors and Quantitative Micro Software assume no responsibility for any errors that may appear in this manual or the EViews program. The user assumes all responsibility for the selection of the program to achieve intended results, and for the installation, use, and results obtained from the program.

Trademarks

Windows, Windows 95/98/2000/NT/Me, and Microsoft Excel are trademarks of Microsoft Corporation. PostScript is a trademark of Adobe Corporation. X11.2 and X12-ARIMA Version 0.2.7 are seasonal adjustment programs developed by the U. S. Census Bureau. Tramo/Seats is copyright by Agustin Maravall and Victor Gomez. All other product names mentioned in this manual may be trademarks or registered trademarks of their respective companies.

Quantitative Micro Software, LLC

4521 Campus Drive, #336, Irvine CA, 92612-2699

Telephone: (949) 856-3368

Fax: (949) 856-2044

e-mail: sales@eviews.com

web: www.eviews.com

March 11, 2002

Table of Contents

CHAPTER 1. INTRODUCTION	1
Using Commands	1
Batch Program Use	2
How to Use this Manual	2
CHAPTER 2. OBJECT AND COMMAND BASICS	5
Object Declaration	5
Object Commands	6
Object Assignment Statements	9
More on Declaring Objects	9
Auxiliary Commands	10
Managing Workfiles and Databases	11
Managing Objects	14
Basic Command Summary	17
CHAPTER 3. OBJECT, VIEW AND PROCEDURE REFERENCE	19
Coef	20
Equation	21
Graph	25
Group	26
Logl	29
Matrix	31
Model	32
Pool	34
Rowvector	36
Sample	37
Scalar	38
Series	39
Sspace	40
Sym	44
System	45
Table	48
Var	49
Vector	52

CHAPTER 4. MATRIX LANGUAGE	55
Declaring Matrices	55
Assigning Matrix Values	56
Copying Data Between Objects	59
Matrix Expressions	66
Matrix Commands and Functions	69
Matrix Views and Procs	71
Matrix Operations versus Loop Operations	73
Summary of Automatic Resizing of Matrix Objects	74
Matrix Function and Command Summary	76
CHAPTER 5. WORKING WITH TABLES	79
Declaring a Table	79
Controlling Appearance	79
Filling Cells	80
Table Example	82
Table Summary	84
CHAPTER 6. EVIEW'S PROGRAMMING	85
Program Basics	85
Simple Programs	88
Program Variables	89
Program Arguments	96
Control of Execution	97
Multiple Program Files	106
Subroutines	107
Programming Summary	114
CHAPTER 7. SAMPLE PROGRAMS	117
Descriptive Statistics by Year	117
Rolling ADF Test	119
Calculating Cumulative Sums	121
Time Series Operations on a Sample	122
Creating Dummy Variables with a Loop	123
Extracting Test Statistics in a Loop	124
Between Group Estimation for Pooled Data	126
Hausman Test for Fixed Versus Random Effects	128

Regression Output Table	130
CHAPTER 8. COMMAND REFERENCE	135
CHAPTER 9. MATRIX AND STRING REFERENCE	397
CHAPTER 10. PROGRAMMING LANGUAGE REFERENCE	421
APPENDIX A. OPERATOR AND FUNCTION REFERENCE	435
Operators	435
Date and Observation Functions	436
Basic Mathematical Functions	437
Time Series Functions	438
Descriptive Statistics	439
Additional and Special Functions	441
Trigonometric Functions	444
Statistical Distribution Functions	444
INDEX	449

Chapter 1. Introduction

EViews provides you with both a Windows and a command line interface for working with your data. Almost every operation that can be accomplished using menus may also be entered into the command window, or placed in programs for batch processing. You are free to choose the mixture of techniques which best fits your particular style of work.

The *Command and Programming Reference* (CPR) documents the use of commands and programs to perform various tasks in EViews—the companion *User's Guide* describes in greater detail the general features of EViews, with an emphasis on the interactive Windows interface.

In addition to providing a basic command reference, the *Command and Programming Reference* documents the use of EViews' powerful batch processing language and advanced programming features. With EViews, you can create and store commands in programs that automate repetitive tasks, or generate a record of your research project.

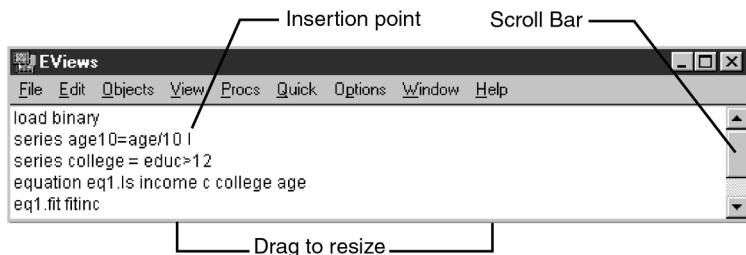
Using Commands

Commands may be used *interactively*, or executed in *batch* mode.

Interactive Use

To work interactively, you should type a command into the command window, then press ENTER to execute the command immediately. If you enter an incomplete command, EViews will open a dialog box prompting you for additional information.

The *command window* is located just below the main menu bar at the top of the EViews window. Unless you are editing an object or have a dialog box open, EViews will assume that anything you type on the keyboard belongs in the command window. The blinking vertical insertion bar at the left end of the command window indicates that EViews is expecting a command.



A command that you enter in the window will be executed as soon as you press ENTER. The insertion point need not be at the end of the command line when you press ENTER. EViews will execute the entire line containing the insertion point.

As you enter commands, EViews will create a list in the command window. You can scroll up to an earlier command, edit it, and hit ENTER. The modified command will be executed again. You may also use standard Windows copy-and-paste between the command window and any other window.

The contents of the command area may also be saved directly into a text file for later use: make certain that the command window is active by clicking anywhere in the window, and then select **File/Save As...** from the main menu.

You can resize the command window so that a larger number of previously executed commands are visible. Use the mouse to move the cursor to the bottom of the window, hold down the mouse button, and drag the bottom of the window downwards.

Batch Program Use

You can assemble a number of commands into a *program*, and then execute the commands in batch mode. Each command in the program will be executed in the order that it appears in the program. Using batch programs allows you to make use of advanced capabilities such as looping and condition branching, and subroutine and macro processing. Programs also are an excellent way to document a research project since you will have a record of each step of the project.

One way to create a program file in EViews is to select **File/New/Program**. EViews will open an untitled program window into which you may enter your commands. You can save the program by clicking on the **Save** or **SaveAs** button, navigating to the desired directory, and entering a file name. EViews will append the extension .PRG to the name you provide.

Alternatively, you can use your favorite text (ASCII) editor to create a program file containing your commands. The commands in this program may then be executed from within EViews.

How to Use this Manual

[Chapters 2](#) and [3](#) constitute the basic EViews command reference:

- [Chapter 2, “Object and Command Basics”, on page 5](#) explains the basics of using commands to work with EViews objects, and provides a cross-referenced listing of basic EViews commands associated with various tasks.

- [Chapter 3, “Object, View and Procedure Reference”, on page 19](#) provides a listing of commands, views, and procedures associated with each object.

[Chapter 3](#) will probably be your primary reference for working with EViews since it provides a convenient outline and summary of the built-in features associated with each EViews object.

The next sections provide documentation of more advanced EViews features:

- [Chapter 4, “Matrix Language”, on page 55](#) describes the EViews matrix language and provides a summary of the available matrix operators, functions, and commands.
- [Chapter 5, “Working with Tables”, on page 79](#) documents the table object and describes the basics of working with strings in EViews.
- [Chapter 6, “EViews Programming”, on page 85](#) describes the basics of using programs for batch processing and documents the programming language.
- [Chapter 7, “Sample Programs”, on page 117](#) contains annotated sample EViews programs to perform a variety of common tasks.

The remaining sections contain dictionary-style reference material for all of the EViews commands, functions, and operators:

- [Chapter 8, “Command Reference”, beginning on page 135](#) provides a full alphabetized listing of basic commands, views and procedures. This material contains the primary reference material for working with EViews.
- [Chapter 9, “Matrix and String Reference”, beginning on page 397](#) is an alphabetical listing of the commands and functions associated with the EViews matrix language.
- [Chapter 10, “Programming Language Reference”, on page 421](#) contains an alphabetical listing of the keywords and functions associated with the EViews programming language.

While this reference manual is not designed to be read from cover-to-cover, we recommend that before beginning extensive work using commands, you spend some time reading through [Chapter 2, “Object and Command Basics”](#), which describes the basics of using commands to work with objects. A solid understanding of this material is important for getting the most out of EViews.

If you wish to use programs in EViews, you should, at the very least, examine the first part of [Chapter 6, “EViews Programming”](#), which describes the basics of creating, loading, and running a batch program.

Chapter 2. Object and Command Basics

This chapter provides an overview of the command method of working with EViews and EViews objects. If you are new to EViews, you may find it useful to consult the *User's Guide* (especially [Chapters 1](#) and [3](#)) for a more detailed introduction to EViews and a discussion of objects, their views, and procedures.

The command line interface of EViews is comprised of a set of single line commands, each of which may be classified as one of the following:

- object declarations.
- object commands.
- object assignment statements.
- auxiliary commands.

An EViews program is composed of a sequence of these commands, and may also contain the following:

- control variable assignment statements.
- program control statements.

The use of control variables and program control statements is discussed in detail in the programming guide in [Chapter 6, “EViews Programming”, on page 85](#). The following sections provide an overview of the first four types of commands.

Object Declaration

The first step is to create or declare an object. A simple declaration has the form:

```
object_type object_name
```

where `object_name` is the name you would like to give to the new object and `object_type` is one of the following type identifiers:

coef	(coefficient vector)
equation	
graph	
group	(collection of series)
matrix	
model	

pool	(time series, cross-section)
rowvector	
sample	
scalar	
series	
sspace	(state space)
sym	(symmetric matrix)
system	
table	
text	
var	(vector autoregression, error correction)
vector	

For example, the declaration:

```
series lgdp
```

creates a new series called LGDP, while the command

```
equation eq1
```

creates a new equation object called EQ1.

Matrix objects are typically declared with their dimension in parentheses after the object type. For example

```
matrix(5,5) x
```

creates a 5×5 matrix named X, while

```
coef(10) results
```

creates a 10 element coefficient vector named RESULTS.

Note that in order to create an object you must have a workfile currently open in EViews. You can open or create a workfile interactively from the File Menu, or you can use the `load` or `workfile` commands to perform the same operations inside a program. See “[Workfile Basics](#)” on page 33 of the *User’s Guide* for details.

Object Commands

An *object command* is a command which accesses an object’s views and procedures (procs). Object commands have two main parts, a *display action* followed by a *view specification*. The view specification describes the view or procedure of the object to be acted

upon. The display action determines what is to be done with the output from the view or procedure.

The *complete* syntax for an object command has the form:

```
action(act_opt) object.view_proc(view_proc_opt) arg_list
```

where

actionis one of four verbs (do, freeze, print, show)

act_optan option that modifies the default behavior of the action

objectthe name of the object to be acted upon

view_procthe object view or procedure to be acted upon

view_proc_optan option that modifies the default behavior of the view or procedure

arg_lista list of view or procedure arguments, generally separated by spaces

The four possible actions are:

- `do` executes procedures without opening a window. If the object's window is not currently displayed, no output is generated. If the objects window is already open, `do` is equivalent to `show`.
- `freeze` creates a table or graph from the object view window.
- `print` prints the object view window.
- `show` displays the object view in a window.

In most cases, some of the components of the general object command are not necessary since some views and procs do not require an argument list or options.

Furthermore, you need not explicitly specify an action. If no action is provided, the `show` action is assumed for views and the `do` action is assumed for procedures. For example, when using the command to display the series view for a line graph:

```
gdp.line
```

EViews implicitly adds a `show` command.

```
show gdp.line
```

Alternatively, for the equation procedure `ls`:

```
eq1.ls cons c gdp
```

there is an implicit `do` action.

```
do eq1.ls cons c gdp
```

In some cases, you may wish to modify the default behavior by explicitly describing the action. For example,

```
print eq1.ls cons c gdp
```

both performs the implicit `do` action and then sends the output from the proc to the printer.

Examples:

```
show gdp.line  
print(1) group1.stats  
freeze(output1) eq1.ls cons c gdp  
do eq1.forecast eq1f
```

The first example opens a window displaying a line graph of the series GDP. The second example prints (in landscape mode) descriptive statistics for the series in GROUP1. The third example creates a table named OUTPUT1 from the estimation results of EQ1 for a least squares regression of CONS on GDP. The final example executes the forecast procedure of EQ1, putting the forecasted values into the series EQ1F and suppressing any procedure output.

Of these four examples, only the first opens a window and displays output on the screen.

Output Control

As discussed above, the display action determines the destination for view and procedure output. Here we note in passing a few extensions to these general rules.

You may specify that a view be simultaneously printed and displayed on your screen by the letter “p” as an option to the object command. For example, the expression

```
gdp.correl(24,p)
```

is equivalent to the two commands,

```
show gdp.correl(24)  
print gdp.correl(24)
```

since `correl` is a series view. The “p” option can be combined with other options, separated by commas. So as not to interfere with other option processing, we strongly recommend that the “p” option should *always be specified after any required options*.

Note that the `print` command accepts the “l” or “p” option to indicate landscape or portrait orientation. For example,

```
print(1) gdp.correl(24)
```

Printer output can be redirected to a text file or frozen output. See the `output` command in [Chapter 8](#), and the discussion in [Appendix A, “Global Options”, beginning on page 651](#) of the *User’s Guide*, for details.

The `freeze` command used without options creates an untitled graph or table from a view specification:

```
freeze gdp.line
```

You also may provide a name for the frozen object in parentheses after the word `freeze`. For example,

```
freeze(figure1) gdp.bar
```

names the frozen bar graph of GDP as “figure1”.

Object Assignment Statements

Object assignment statements are commands which assign new data to an EViews object. Object assignment statements have the syntax:

```
object_name = expression
```

where `object_name` identifies the object whose data is to be modified and `expression` is an expression which evaluates to an object of an appropriate type.

The nature of the assignment varies depending on what type of object is on the left hand side of the equal sign. To take a simple example, consider the assignment statement:

```
x = 5 * log(y) + z
```

where X, Y and Z are series. This assignment statement will take the log of each element of Y, multiply each value by 5, add the corresponding element of Z, and, finally, assign the result into the appropriate element of X.

More on Declaring Objects

Object declarations can often be combined with either object commands or object assignment statements to create and initialize an object in a single line. For example:

```
series lgdp = log(gdp)
```

creates a new series called LGDP and initializes its elements with the log of the series GDP. Similarly, the command

```
equation eq1.ls y c x1 x2
```

creates a new equation object called EQ1 and initializes it with the results from regressing the series Y against a constant term, the series X1 and the series X2.

Additional examples,

```
scalar elas = 2
series tr58 = @trend(1958)
group nipa gdp cons inv g x
equation cnsfnc2.ls log(cons)=c(1)+c(2)*yd
vector beta = @inverse(x*x)*(x*y)
```

An object can be declared multiple times so long as it is always declared to be of the same type. The first declaration will create the object, subsequent declarations will have no effect unless the subsequent declaration also specifies how the object is to be initialized.

For example,

```
smp1 @first 1979
series dummy = 1
smp1 1980 @last
series dummy=0
```

creates a series named DUMMY that has the value 1 prior to 1980 and the value 0 thereafter.

Redeclaration of an object to a different type is not allowed and will generate an error.

Auxiliary Commands

Auxiliary commands are commands which are either unrelated to a particular object (*i.e.*, not views or procs), or which act on an object in a way that is generally independent of the type or contents of the object (often acting symmetrically across objects of all types). Auxiliary commands typically follow the syntax:

```
command(option_list) argument_list
```

where `command` is the name of the view or procedure to be executed, `option_list` is a list of options separated by commas, and `argument_list` is a list of arguments generally separated by spaces.

An example of an auxiliary command might be:

```
store(d=c:\newdata\db1) gdp m x
```

which will store the three objects GDP, M and X in the database named DB1 in the directory C:\NEWDATA.

There is also a set of auxiliary commands which performs operations that create new untitled or unnamed objects. For example, the command:

```
ls y c x1 x2
```

will regress the series Y against a constant term, the series X1 and the series X2, and create a new unnamed equation object to hold the results.

Although this latter set of auxiliary commands can sometimes be useful for carrying out simple tasks, their overuse will tend to make it difficult to manage your work. Unnamed objects cannot be referenced by name from within a program, cannot be saved to disk, and cannot be deleted except through the graphical Windows interface. Wherever possible, you should favor using named rather than unnamed objects for your work. For example, we may replace the above auxiliary command with:

```
equation eq1.ls y c x1 x2
```

to create the named equation object EQ1.

Managing Workfiles and Databases

There are two types of object containers: *workfiles* and *databases*. All EViews objects must be held in an object container, so before you begin working with objects you must create a workfile or database. Workfiles and databases are described in depth in [Chapter 3, “EViews Basics”](#) and [Chapter 6, “EViews Databases”](#) of the *User’s Guide*.

Managing Workfiles

To declare and create a new workfile, follow the `workfile` command with a name for the workfile, an option for the frequency of the workfile, and the start and end dates. The workfile frequency type options are:

a	annual.
s	semi-annual.
q	quarterly.
m	monthly.
w	weekly.

d	daily (5 day week).
7	daily (7 day week).
u	undated.

For example,

```
workfile macro1 q 1965:1 1995:4
```

creates a new quarterly workfile named MACRO1 from the first quarter of 1965 to the fourth quarter of 1995.

```
workfile cps88 u 1 1000
```

creates a new undated workfile named CPS88 with 1000 observations.

Note that if you have multiple workfiles open, the workfile command may be used to change the active workfile.

To save your workfile, type “save” followed by a workfile name. If any part of the path or workfile name has spaces, you should enclose the entire expression in quotation marks. The active workfile will be saved in the default path under the given name. You may optionally provide a path to save the workfile in a different directory:

```
save a:\mywork
```

If necessary, enclose the path name in quotations. To close the workfile, type “close” followed by the workfile name. For example,

```
close mywork
```

closes the workfile window of MYWORK.

To open a previously saved workfile, follow the load keyword with the name of the workfile. You can optionally include a path designation to open workfiles that are not saved in the default path. For example,

```
load "c:\mywork\proj1"
```

You may also use the open command to open a previously saved workfile. To use open, you have to type the full workfile name including the extension .WK1:

```
open proj2.wk1
```

Managing Databases

To create a new database, follow the dbcreate keyword with a name for the new database. Alternatively, you could use the db keyword followed by a name for the new database. The two commands differ only when the named database already exists. If you use

`dbccreate` and the named database already exists on disk, EViews will error indicating that the database already exists. If you use `db` and the named database already exists on disk, EViews will simply open the existing database. Note that the newly opened database (either by `dbccreate` or `db`) will become the default database.

For example,

```
dbccreate mydata1
```

creates a new database named MYDATA1 in the default path, opens a new database window, and makes MYDATA1 the default database.

```
db c:\evdata\usdb
```

opens the USDB database in the specified directory if it already exists. If it does not, EViews creates a new database named USDB, opens its window, and makes it the default database.

You can also use `dbopen` to open an existing database and to make it the default database. For example,

```
dbopen findat
```

opens the database named FINDAT in the default directory. If the database does not exist, EViews will error indicating that the specified database cannot be found. You can also use `open` to open an existing database. To use `open`, you must provide the full name of the database, including the file extension .EDB:

```
open findat.edb
```

You may use `dbrename` to rename an existing database. Follow the `dbrename` keyword by the current (old) name and a new name.

```
dbrename templ newmacro
```

To delete an existing database, use the `dbdelete` command. Follow `dbdelete` by the name of the database to delete.

```
dbdelete c:\data\usmacro
```

`dbcopy` makes a copy of the existing database. Follow `dbcopy` by the name of the source file and the name of the destination file.

```
dbcopy c:\evdata\macro1 a:\macro1
```

`dbpack`, `dbrepair`, and `dbrebuild` are database maintenance commands. See also [Chapter 6, “EViews Databases”, beginning on page 107](#) of the *User’s Guide* for a detailed description.

Managing Objects

In the course of a program you will often need to manage the objects in a workfile by copying, renaming, deleting and storing them to disk. EViews provides a number of auxiliary commands which perform these operations. The following discussion introduces you to the use of these commands; a full description of each command is provided in [Chapter 8, “Command Reference”, beginning on page 135](#).

Copying Objects

You can create a duplicate copy of an object using the `copy` command. The `copy` command is an auxiliary command with the format:

```
copy source_name dest_name
```

where `source_name` is the name of the object you wish to duplicate, and `dest_name` is the name you want attached to the new copy of the object.

The `copy` command may also be used to copy objects in databases and to move objects between workfiles and databases.

Copy with Wildcard Characters

EViews supports the use of wildcard characters (“?” for a single character match and “*” for a pattern match) in destination specifications when using the `copy` and `rename` commands. Using this feature, you can copy or rename a set of objects whose names share a common pattern in a single operation. This can be useful for managing series produced by model simulations, series corresponding to pool cross-sections, and any other situation where you have a set of objects which share a common naming convention.

A destination wildcard pattern can be used only when a wildcard pattern has been provided for the source, and the destination pattern must always conform to the source pattern in that the number and order of wildcard characters must be exactly the same between the two. For example, the following patterns

Source Pattern	Destination Pattern
x*	y*
c	b
x*12?	yz*f?abc

conform to each other, while these patterns do not

Source Pattern	Destination Pattern
a*	b
*x	?y
x*y*	*x*y*

When using wildcards, the destination name is formed by replacing each wildcard in the destination pattern by the characters from the source name that matched the corresponding wildcard in the source pattern. Some examples should make this principle clear:

Source Pattern	Destination Pattern	Source Name	Destination Name
*_base	*_jan	x_base	x_jan
us_*	*	us_gdp	gdp
x?	x?f	x1	x1f
_	**f	us_gdp	usgdpf
??*f	??_*	usgdpf	us_gdp

Note, as shown in the second example, that a simple asterisk for the destination pattern does not mean to use the unaltered source name as the destination name. To copy objects between containers preserving the existing name, either repeat the source pattern as the destination pattern

```
copy x* db1::x*
```

or omit the destination pattern entirely

```
copy x* db1::
```

If you use wildcard characters in the source name and give a destination name without a wildcard character, EViews will keep overwriting all objects which match the source pattern to the name given as destination.

For additional discussion of wildcards, see [Appendix C, “Wildcards”, on page 657](#) of the *User’s Guide*.

Renaming Objects

You can give an object a different name using the `rename` command. The `rename` command has the format:

```
rename source_name dest_name
```

where `source_name` is the original name of the object and `dest_name` is the new name you would like to give to the object.

`rename` can also be used to rename objects in databases.

You may use wildcards when renaming series. The name substitution rules are identical to those described above for `copy`.

Deleting Objects

Objects may be removed from the workfile using the `delete` command. The `delete` command has the format:

```
delete name_pattern
```

where `name_pattern` can either be a simple name such as “XYZ”, or a pattern containing the wildcard characters “?” and “*”, where “?” means to match any one character, and “*” means to match zero or more characters. When a pattern is provided, all objects in the workfile with names matching the pattern will be deleted. [Appendix C, “Wildcards”, on page 657](#) of the *User’s Guide* describes further the use of wildcards.

`delete` can also be used to remove objects from databases.

Saving Objects

All named objects will be saved automatically in the workfile when the workfile is saved to disk. You can store and retrieve the current workfile to and from disk using the `save` and `load` commands. Unnamed objects will not be saved as part of the workfile.

You can also save objects for later use by storing them in a database. The `store` command has the format:

```
store(option_list) object1 object2 ...
```

where `object1`, `object2`, ..., are the names of the objects you would like to store in the database. If no options are provided, the series will be stored in the current default database (see [Chapter 6](#) of the *User’s Guide* for a discussion of the default database). You can store objects into a particular database by using the option “`d = db_name`” or by prepending the object name with a database name followed by a double colon “`::`”, such as:

```
store db1::x db2::x
```

Fetch Objects

You can retrieve objects from a database using the `fetch` command. The `fetch` command has the same format as the `store` command:

```
fetch(option_list) object1 object2 ...
```

To specify a particular database use the “`d =` ” option or the “`::`” extension as for `store`.

Basic Command Summary

The following list summarizes the EViews basic commands. The full descriptions of these commands are given in [Chapter 8, “Command Reference”, beginning on page 135](#).

A list of views and procedures available for each object may be found in the next chapter. Commands for working with matrix objects are listed in [Chapter 4, “Matrix Language”, on page 55](#), and EViews programming expressions are described in [Chapter 6, “EViews Programming”, beginning on page 85](#).

Object Declarations

You may define EViews objects using the following commands/declarations (see the corresponding entries in [Chapter 8, “Command Reference”, beginning on page 135](#) for syntax details and additional discussion): `coef`, `equation`, `graph`, `group`, `log1`, `matrix`, `model`, `pool`, `rowvector`, `sample`, `scalar`, `series`, `sspace`, `sym`, `system`, `table`, `text`, `var`, `vector`.

Command Actions

- `do` execute procedures ([p. 192](#)).
- `freeze` create view object ([p. 216](#)).
- `print` print view ([p. 286](#)).
- `show` display objects ([p. 328](#)).

Object Container, Data, and File Commands

- `ccopy` copy series from DRI database ([p. 156](#)).
- `cd`, `chdir` change default directory ([p. 156](#)).
- `cfetch` fetch series from DRI database ([p. 160](#)).
- `clabel` display DRI series description ([p. 162](#)).
- `close` close object, program, or workfile ([p. 163](#)).
- `create` create a new workfile ([p. 176](#)).
- `data` enter data from keyboard ([p. 178](#)).
- `db` open or create a database ([p. 180](#)).
- `dbcopy` make copy of a database ([p. 181](#)).
- `dbcreate` create a new database ([p. 181](#)).
- `dbdelete` delete a database ([p. 182](#)).
- `dbopen` open a database ([p. 183](#)).
- `dbpack` pack a database ([p. 184](#)).
- `dbrebuild` rebuild a database ([p. 184](#)).
- `dbrename` rename a database ([p. 185](#)).
- `dbrepair` repair a database ([p. 185](#)).

diconvert convert the entire DRI database to an EViews database ([p. 194](#)).
expand expand workfile range ([p. 204](#)).
fetch fetch objects from databases or databank files ([p. 205](#)).
hconvert convert an entire Haver Analytics database to an EViews database ([p. 226](#)).
load load a workfile ([p. 244](#)).
open open a file ([p. 275](#)).
range reset the workfile range ([p. 290](#)).
read read data from a foreign disk file ([p. 291](#)).
save save workfile to disk ([p. 308](#)).
sort sort the workfile ([p. 336](#)).
workfile create or change active workfile ([p. 381](#)).
write write series to a disk file ([p. 383](#)).

Object Utility Commands

close close window of an object, program, or workfile ([p. 163](#)).
copy copy objects ([p. 168](#)).
delete delete objects ([p. 188](#)).
output redirect printer output ([p. 279](#)).
rename rename object ([p. 293](#)).

Global Commands

exit exit the EViews program ([p. 203](#)).
param set parameter values ([p. 281](#)).
rndseed set the seed of the random number generator ([p. 303](#)).
smpl set current workfile sample ([p. 332](#)).
setconvert set default frequency conversion methods ([p. 321](#)).
tic reset the timer ([p. 363](#)).
toc display elapsed time (since timer reset) in seconds ([p. 364](#)).

Table Commands

setcell format and fill in a table cell ([p. 319](#)).
setcolwidth set width of a table column ([p. 321](#)).
setline place a horizontal line in table ([p. 326](#)).

Chapter 3. Object, View and Procedure Reference

The following is a reference guide to the views, procedures, and data members for each of the objects found in EViews:

 Coef (p. 20)	 Model (p. 32)	 Sspace (p. 40)
 Equation (p. 21)	 Pool (p. 34)	 Sym (p. 44)
 Graph (p. 25)	 Rowvector (p. 36)	 System (p. 45)
 Group (p. 26)	 Sample (p. 37)	 Table (p. 48)
 Logl (p. 29)	 Scalar (p. 38)	 Var (p. 49)
 Matrix (p. 31)	 Series (p. 39)	 Vector (p. 52)

To use these views, procedures, and data members you should list the name of the object followed by a period, and then enter the name of the view, procedure, or data member, along with any options or arguments:

```
object_name.view_name(options) arguments  
object_name.proc_name(options) arguments  
object_name.data_member
```

For example, to display the line graph view of the series object CONS, you can enter the command:

```
cons.line
```

To perform a dynamic forecast using the estimates in the equation object EQ1, you may enter:

```
eq1.forecast y_f
```

To save the coefficient covariance matrix from EQ1, you can enter:

```
sym cov1=eq1.@cofcov
```

Each of the views and procedures is documented more fully in the alphabetical listing found in [Chapter 8, “Command Reference”, on page 135](#).

Coef

Coefficient vector. Coefficients are used to represent the parameters of equations and systems.

There are two ways to create a coef. First, enter the `coef` keyword, followed by a name to be given to the coefficient vector. The dimension of the coef may be provided in parentheses after the keyword:

```
coef alpha  
coef(10) beta
```

If no dimension is provided, the resulting coef will contain a single element.

You may also combine a declaration with an assignment statement. If you do not provide an explicit assignment statement, new coefs are initialized to zero.

See also [param \(p. 281\)](#) for information on initializing coefficients, and the entries for each of the estimation objects ([equation](#), [pool](#), [sspace](#), [system](#), and [var](#)) for additional methods of accessing coefficients.

Coef Views

- [**bar**](#) bar graph of coefficient vector plotted against the coefficient index [\(p. 150\)](#).
- [**label**](#) label view [\(p. 238\)](#).
- [**line**](#) line graph of coefficient vector plotted against the coefficient index [\(p. 241\)](#).
- [**sheet**](#) spreadsheet [\(p. 327\)](#).
- [**spike**](#) spike graph [\(p. 338\)](#).
- [**stats**](#) descriptive statistics [\(p. 344\)](#).

Coef Procs

- [**displayname**](#) set display name [\(p. 192\)](#).
- [**read**](#) import data into coefficient vector [\(p. 291\)](#).
- [**write**](#) export data from coefficient vector [\(p. 383\)](#).

Coef Data Members

- (i) i -th element of the coefficient vector. Simply append “(i)” to the matrix name (without a “.”).

Coef Examples

The coefficient vector declaration

```
coef(10) coef1=3
```

creates a 10 element coefficient vector COEF1, and initializes all values to 3.

Suppose MAT1 is a 10×1 matrix, and VEC1 is a 20 element vector. Then

```
coef mycoef1=coef1
coef mycoef2=mat1
coef mycoef3=vec1
```

create, size, and initialize the coefficient vectors MYCOEF1, MYCOEF2 and MYCOEF3.

Coefficient elements may be referred to by an explicit index. For example:

```
vector(10) mm=beta(10)
scalar shape=beta(7)
```

fills the vector MM with the value of the tenth element of BETA, and assigns the seventh value of BETA to the scalar SHAPE.

Equation

Equation object. Equations are used for single equation estimation, testing, and forecasting.

To declare an equation object, enter the keyword **equation**, followed by a name:

```
equation eq01
```

and an optional specification:

```
equation r4cst.ls r c r(-1) div
equation wcd.ls q=c(1)*n^c(2)*k^c(3)
```

Equation Methods

arch autoregressive conditional heteroskedasticity (ARCH and GARCH) ([p. 145](#)).

binary binary dependent variable models (includes probit, logit, gompit) models ([p. 152](#)).

censored censored and truncated regression (includes tobit) models ([p. 158](#)).

count count data modeling (includes poisson, negative binomial and quasi-maximum likelihood count models) ([p. 173](#)).

gmm generalized method of moments ([p. 221](#)).

ls linear and nonlinear least squares regression (includes weighted least squares and ARMAX) models ([p. 245](#)).

- ordered** ordinal dependent variable models (includes ordered probit, ordered logit, and ordered extreme value models) ([p. 277](#)).
tsls linear and nonlinear two-stage least squares (TSLS) regression models (includes weighted TSLS, and TSLS with ARMA errors) ([p. 368](#)).

Equation Views

- archtest** LM test for the presence of ARCH in the residuals ([p. 147](#)).
auto Breusch-Godfrey serial correlation Lagrange Multiplier (LM) test ([p. 149](#)).
chow Chow breakpoint and forecast tests for structural change ([p. 161](#)).
coefcov coefficient covariance matrix ([p. 165](#)).
correl correlogram of the residuals ([p. 172](#)).
correlnsq correlogram of the squared residuals ([p. 173](#)).
derivs derivatives of the equation specification ([p. 189](#)).
garch conditional standard deviation graph (only for equations estimated using ARCH) ([p. 219](#)).
grads examine the gradients of the objective function ([p. 223](#)).
hist histogram and descriptive statistics of the residuals ([p. 229](#)).
label label information for the equation ([p. 238](#)).
means descriptive statistics by category of the dependent variable (only for binary, ordered, censored and count equations) ([p. 266](#)).
predict prediction (fit) evaluation table (only for binary and ordered equations) ([p. 285](#)).
representation text showing specification of the equation ([p. 294](#)).
reset Ramsey's RESET test for functional form ([p. 297](#)).
resids display, in tabular form, the actual and fitted values for the dependent variable, along with the residuals ([p. 299](#)).
results table of estimation results ([p. 300](#)).
rls recursive residuals least squares (only for equations estimated by ordinary least squares, without ARMA terms) ([p. 300](#)).
testadd likelihood ratio test for adding variables to equation ([p. 355](#)).
testdrop likelihood ratio test for dropping variables from equation ([p. 358](#)).
testfit performs Hosmer and Lemeshow and Andrews goodness-of-fit tests (only for equations estimated using binary) ([p. 360](#)).
wald Wald test for coefficient restrictions ([p. 378](#)).
white White test for heteroskedasticity ([p. 379](#)).

Equation Procs

displaynameset display name ([p. 192](#)).
fitstatic forecast ([p. 212](#)).
forecastdynamic forecast ([p. 214](#)).
makederivsmake group containing derivatives of the equation specification
([p. 251](#)).
makergarchcreate conditional variance series (only for ARCH equations)
([p. 252](#)).
makegradsmake group containing gradients of the objective function
([p. 253](#)).
makelimitscreate vector of estimated limit points (only for ordered models)
([p. 257](#)).
makemodelcreate model from estimated equation ([p. 257](#)).
makeregsmake group containing the regressors ([p. 258](#)).
makeresidsmake series containing residuals from equation ([p. 259](#)).
updatecoefsupdate coefficient vector(s) from equation ([p. 372](#)).

Equation Data Members

Scalar Values

@aicAkaike information criterion.
@cofcov(i,j)covariance of coefficient estimates i and j .
@coefs(i) i -th coefficient value.
@dwDurbin-Watson statistic.
@f F -statistic.
@hqHannan-Quinn information criterion.
@jstat J -statistic — value of the GMM objective function (for GMM).
@loglvalue of the log likelihood function.
@meandepmean of the dependent variable.
@ncoefnumber of estimated coefficients.
@r2R-squared statistic.
@rbar2adjusted R-squared statistic.
@regobsnumber of observations in regression.
@schwarzSchwarz information criterion.
@sddepstandard deviation of the dependent variable.
@sestandard error of the regression.
@ssrsum of squared residuals.
@stderrs(i)standard error for coefficient i .

@tstats(i) *t*-statistic value for coefficient *i*.
c(i) *i*-th element of default coefficient vector for equation (if applicable).

Vectors and Matrices

@coefcov covariance matrix for coefficient estimates.
@coefs coefficient vector.
@stderrs vector of standard errors for coefficients.
@tstats vector of *t*-statistic values for coefficients.

Equation Examples

To apply an estimation method (proc) to an existing equation object:

```
equation ifunc  
ifunc.ls r c r(-1) div
```

To declare and estimate an equation in one step, combine the two commands:

```
equation value.tsls log(p) c d(x) @ x(-1) x(-2)  
equation drive.logit ifdr c owncar dist income  
equation countmod.count patents c rdd
```

To estimate equations by list, using ordinary and two-stage least squares:

```
equation ordinary.ls log(p) c d(x)  
equation twostage.tsls log(p) c d(x) @ x(-1) x(-2)
```

You can create and use other coefficient vectors:

```
coef(10) a  
coef(10) b  
equation eq01.ls y=c(10)+b(5)*y(-1)+a(7)*inc
```

The fitted values from EQ01 may be saved using

```
series fit = eq01.@coefs(1) + eq01.@coefs(2)*y(-1) +  
eq01.@coefs(3)*inc
```

or by issuing the command

```
eq01.fit fitted_vals
```

To perform a Wald test:

```
eq01.wald a(7)=exp(b(5))
```

You can save the *t*-statistics and covariance matrix for your parameter estimates:

```
vector eqstats=eq01.@tstats
```

```
matrix eqcov=eq01.@coefcov
```

Graph

Graph view object.

Graphs may be created by declaring a graph using one of the graph methods described below, or by freezing the graphical view of an object. For example:

```
graph myline.line ser1  
graph myscat.scat ser1 ser2  
graph myxy.xyline grp1
```

declare and create the graph objects MYLINE, MYSCAT and MYXY. Alternatively, you can use the `freeze` command to create graph objects:

```
freeze(myline) ser1.line  
group grp2 ser1 ser2  
freeze(myscat) grp2.scat  
freeze(myxy) grp1.xyline
```

which are equivalent to the declarations above. For details, see [freeze \(p. 216\)](#).

Graph View

[label](#)label information for the graph (p. 238).

Graph Procs (to set graph type)

[bar](#)bar graph (p. 150).
[errbar](#)error bar graph (p. 201).
[graph](#)create a graph or merged graph (p. 224).
[hilo](#)high-low(-open-close) graph (p. 228).
[line](#)line-symbol graph (p. 241).
[pie](#)pie chart (p. 284).
[scat](#)scatter plot (p. 311).
[spike](#)spike graph (p. 338).
[xyline](#)XY line graph with one or more X series plotted against one or more Y (p. 394).

Graph Procs

[addtext](#)place arbitrary text on the graph (p. 140).
[align](#)align the placement of multiple graphs (p. 142).
[dates](#)controls labeling of the bottom date/time axis in time plots (p. 178).

displayname set display name ([p. 192](#)).
draw draw lines and shaded areas on the graph ([p. 193](#)).
label label information for the graph ([p. 238](#)).
legend control the appearance and placement of legends ([p. 240](#)).
metafile save graph to a Windows metafile ([p. 268](#)).
name change the series name for legends or axis labels ([p. 271](#)).
options change the option settings of the graph ([p. 275](#)).
scale manually scale the axis of the graph ([p. 309](#)).
setelem set individual line, bar and legend options for each series in the graph ([p. 323](#)).
template use template graph ([p. 355](#)).

Graph Examples

You can declare your graph

```
graph abc.xyline(m) unemp gnp inf  
graph bargraph.bar(d,l)unemp gnp
```

or freeze any graphical view:

```
freeze(mykernel) ser1.kdensity
```

You can change the graph type,

```
graph mygraph.line ser1  
mygraph.hist
```

or combine multiple graphs,

```
graph xyz.merge graph1 graph2
```

Group

Group of series. Groups are used for working with collections of series objects.

To declare a group, enter the keyword **group**, followed by a name, and optionally, a list of series or expressions:

```
group salesvrs  
group nipa cons(-1) log(inv) g x
```

Additionally, a number of object procedures will automatically create a group.

Group Views

bar single or multiple bar graph view of all series ([p. 150](#)).

cause pairwise Granger causality tests ([p. 154](#)).
cdfplot distribution (cumulative, survivor, quantile) graphs ([p. 157](#)).
coint Johansen cointegration test ([p. 166](#)).
cor correlation matrix between series ([p. 171](#)).
correl correlogram of the first series in the group ([p. 172](#)).
cov covariance matrix between series ([p. 175](#)).
cross cross correlogram of the first two series ([p. 177](#)).
drop drop one or more series from the group ([p. 195](#)).
dtable dated data table ([p. 196](#)).
errbar error bar graph view ([p. 201](#)).
freq frequency table n -way contingency table ([p. 217](#)).
hilo high-low(-open-close) chart ([p. 228](#)).
kerfit scatter of the first series against the second series with kernel fit ([p. 237](#)).
label label information for the group ([p. 238](#)).
line single or multiple line graph view of all series ([p. 241](#)).
linefit scatter of the first series against the second series with regression line ([p. 242](#)).
nnfit scatter of the first series against the second series with nearest neighbor fit ([p. 272](#)).
pie pie chart view ([p. 284](#)).
pcomp principal components analysis ([p. 281](#)).
qqplot quantile-quantile plots ([p. 288](#)).
scat single scatter diagram of the series in the group ([p. 311](#)).
scatmat matrix of all pairwise scatter plots ([p. 313](#)).
sheet spreadsheet view of the series in the group ([p. 327](#)).
spike spike graph ([p. 338](#)).
stats descriptive statistics ([p. 344](#)).
testbtw tests of equality for mean, median, or variance, between series in group ([p. 356](#)).
xyline XY line graph with one or more X series plotted against one or more Y ([p. 394](#)).

Group Procs

add add one or more series to the group ([p. 137](#)).
displayname set display name ([p. 192](#)).
resample resample from rows of group ([p. 295](#)).

Group Data Members

- (i) *i*-th series in the group. Simply append “(i)” to the group name (without a “.”).
- @comobs number of observations in the current sample for which each series in the group has a non-missing value (observations in the common sample).
- @count number of series in the group.
- @minobs number of non-missing observations in the current sample for the shortest series in the group.
- @maxobs number of non-missing observations in the current sample for the longest series in the group.
- @seriesname(i) string containing the name of the *i*-th series in the group.

Group Examples

To create a group G1, you can enter

```
group g1 gdp income
```

To change the contents of an existing group, you can repeat the declaration, or use the add and drop commands:

```
group g1 x y  
g1.add w z  
g1.drop y
```

The following commands produce a cross-tabulation of the series in the group, display the covariance matrix, and test for equality of variance:

```
g1.freq  
g1.cov  
g1.testbtw(var,c)
```

You can index selected series in the group:

```
show g1(2).line  
series sum=g1(1)+g1(2)
```

To create a scalar containing the number of series in the group, use

```
scalar nsers=g1.@count
```

Logl

Likelihood object. Used for performing maximum likelihood estimation of user-specified likelihood functions.

To declare a logl object, use the `logl` keyword, followed by a name to be given to the object.

Logl Method

`ml`maximum likelihood estimation ([p. 269](#)).

Logl Views

`append`add line to the specification ([p. 143](#)).

`checkderivs`compare user supplied and numeric derivatives ([p. 160](#)).

`coefcov`coefficient covariance matrix ([p. 165](#)).

`grads`examine the gradients of the log likelihood ([p. 223](#)).

`label`label view of likelihood object ([p. 238](#)).

`results`estimation results ([p. 300](#)).

`spec`likelihood specification ([p. 337](#)).

`wald`Wald coefficient restriction test ([p. 378](#)).

Logl Procs

`displayname`set display name ([p. 192](#)).

`makegrads`make group containing gradients of the log likelihood ([p. 253](#)).

`makemodel`make model ([p. 257](#)).

`updatecoefs`update coefficient vector(s) from likelihood ([p. 372](#)).

Logl Statements

The following statements can be included in the specification of the likelihood object.

These statements are optional, except for “@logl” which is required. See [Chapter 18, “The Log Likelihood \(LogL\) Object”, on page 471](#) of the *User’s Guide* for further discussion.

`@byeqn`evaluate specification by equation.

`@byobs`evaluate specification by observation (default).

`@deriv`specify an analytic derivative series.

`@derivstep`set parameters to control step size.

`@logl`specify the likelihood contribution series.

`@param`set starting values.

`@temp`remove temporary working series.

Logl Data Members

Scalar Values (system data)

@aic Akaike information criterion.
@cofcov(i,j) covariance of coefficients i and j .
@coefs(i) coefficient i .
@hq Hannan-Quinn information criterion.
@logl value of the log likelihood function.
@ncoefs number of estimated coefficients.
@regobs number of observations used in estimation.
@sc Schwarz information criterion.
@stderrs(i) standard error for coefficient i .
@tstats(i) t -statistic value for coefficient i .
coef_name(i) i -th element of default coefficient vector for likelihood.

Vectors and Matrices

@cofcov covariance matrix of estimated parameters.
@coefs coefficient vector.
@stderrs vector of standard errors for coefficients.
@tstats vector of t -statistic values for coefficients.

Logl Examples

To declare a likelihood named LL1:

```
logl ll1
```

To define a likelihood function for OLS (not a recommended way to do OLS!):

```
ll1.append @logl logl1  
ll1.append res1 = y-c(1)-c(2)*x  
ll1.append logl1 = log(@dnorm(res1/@sqrt(c(3))))-log(c(3))/2
```

To estimate LL1 by maximum likelihood (the “showstart” option displays the starting values):

```
ll1.ml(showstart)
```

To save the estimated covariance matrix of the parameters from LL1 as a named matrix COV1:

```
matrix cov1=ll1.@coefcov
```

Matrix

Matrix (two-dimensional array).

There are several ways to create a matrix object. You can enter the `matrix` keyword (with an optional row and column dimension) followed by a name:

```
matrix scalarimat  
matrix(10,3) results
```

Alternatively, you can combine a declaration with an assignment statement, in which case the new matrix will be sized accordingly.

Lastly, a number of object procedures create matrices.

Matrix Views

`bar`single or multiple bar graph of each column against the row index ([p. 150](#)).
`cor`correlation matrix by columns ([p. 171](#)).
`cov`covariance matrix by columns ([p. 175](#)).
`errbar`error bar graph view ([p. 201](#)).
`hilo`high-low(-open-close) chart ([p. 228](#)).
`label`label information for the matrix ([p. 238](#)).
`line`single or multiple line graph of each column by the row index ([p. 241](#)).
`pie`pie chart view ([p. 284](#)).
`scat`scatter diagrams of the columns of the matrix ([p. 311](#)).
`sheet`spreadsheet view of the matrix ([p. 327](#)).
`spike`spike graph ([p. 338](#)).
`stats`descriptive statistics by column ([p. 344](#)).
`xyline`XY line graph with one or more X columns plotted against one or more Y ([p. 394](#)).

Matrix Procs

`displayname`set display name ([p. 192](#)).
`fill`fill the elements of the matrix ([p. 208](#)).
`read`import data from disk ([p. 291](#)).
`write`export data to disk ([p. 383](#)).

Matrix Data Members

(i,j) (i,j)-th element of the matrix. Simply append “(i, j)” to the matrix name (without a “.”).

Matrix Examples

The following assignment statements create and initialize matrix objects:

```
matrix copymat=results  
matrix covmat1=eq1.@coefcov  
matrix(5,2) count  
count.fill 1,2,3,4,5,6,7,8,9,10
```

as does the procedure

```
eq1.mak ecoefcov covmat2
```

You can declare and initialize a matrix in one command:

```
matrix(10,30) results=3  
matrix(5,5) other=results1
```

Graphs and covariances may be generated for the columns of the matrix:

```
copymat.line  
copymat.cov
```

and statistics computed for the rows of a matrix

```
matrix rowmat=@transpose(copymat)  
rowmat.stats
```

You can use explicit indices to refer to matrix elements:

```
scalar diagsum=cov1(1,1)+cov1(2,2)+cov(3,3)
```

Model

Set of simultaneous equations used for forecasting and simulation.

Declare an object by entering the keyword `model`, followed by a name:

```
model mymod
```

declares an empty model named MYMOD. To fill MYMOD, open the model and edit the specification view, or use the append view. Note that models are not used for estimation of unknown parameters.

See also the section on model keywords in “[Text View](#)” on page 624 of the *User’s Guide*.

Model Views

blockdisplay model block structure ([p. 154](#)).
eqsview of model organized by equation ([p. 201](#)).
labelview or set label information for the model ([p. 238](#)).
msgdisplay model solution messages ([p. 270](#)).
textshow text showing equations in the model ([p. 363](#)).
traceview of trace output from model solution ([p. 365](#)).
varsview of model organized by variable ([p. 377](#)).

Model Procs

addassignassign add factors to equations ([p. 138](#)).
addinitinitialize add factors ([p. 139](#))
appendappend a line of text to a model ([p. 143](#)).
controlsolve for values of control variable so that target matches trajectory ([p. 168](#)).
displaynameset display name ([p. 192](#)).
excludespecifies (or merges) excluded series to the active scenario ([p. 203](#)).
makegraphmake graph object showing model series ([p. 254](#)).
makegroupmake group out of model series and display dated data table ([p. 255](#)).
mergemerge other objects into the model ([p. 267](#)).
overridespecifies (or merges) override series to the active scenario ([p. 280](#)).
scenarioset the active, alternate, or comparison scenario ([p. 313](#)).
solvesolve the model ([p. 334](#)).
solveoptset solve options for model ([p. 335](#)).
specDisplays the text specification view ([p. 337](#)).

Model Examples

The commands

```
model mod1  
mod1.append y=324.35+x  
mod1.append x=-234+7.3*z  
mod1.solve(m=100,c=.008)
```

create, specify, and solve the model MOD1. The command

```
mod1(g).makegraph gr1 x y z
```

plots the endogenous series X, Y, and Z, in the active scenario for model MOD1.

Pool

Pooled time series, cross-section object. Used when working with data with both time series and cross-section structure.

To declare a pool object, use the `pool` keyword, followed by a pool name, and optionally, a list of pool members. Pool members are short text identifiers for the cross section units:

```
pool mypool  
pool g7 _can _fr _ger _ita _jpn _us _uk
```

Pool Methods

ls estimate linear regression models including cross-section weighted least squares, and fixed and random effects models ([p. 245](#)).

Pool Views

add add cross section members to pool ([p. 137](#)).

cofcov coefficient covariance matrix ([p. 165](#)).

define define cross section identifiers ([p. 188](#)).

describe calculate pool descriptive statistics ([p. 190](#)).

drop drop cross section members from pool ([p. 195](#)).

label label information for the pool object ([p. 238](#)).

representation text showing equations in the model ([p. 294](#)).

residcor residual correlation matrix ([p. 297](#)).

residcov residual covariance matrix ([p. 298](#)).

resids table or graph of residuals for each pool member ([p. 299](#)).

results table of estimation results ([p. 300](#)).

sheet spreadsheet view of series in pool ([p. 327](#)).

wald Wald coefficient restriction test ([p. 378](#)).

Pool Procs

delete delete pool series ([p. 188](#)).

displayname set display name ([p. 192](#)).

fetch fetch series into workfile using a pool ([p. 205](#)).

genr generate pool series using the "?" ([p. 220](#)).

makegroup create a group of series from a pool ([p. 255](#)).

makemodel creates a model object from the estimated pool ([p. 257](#)).

makeresidsmake series containing residuals from pool ([p. 259](#)).
makestatsmake descriptive statistic series ([p. 263](#)).
makesystemcreates a system object from the pool for other estimation methods
([p. 264](#)).
readimport pool data from disk ([p. 291](#)).
storestore pool series in database/bank files ([p. 347](#)).
storeupdate coefficient vector(s) from pool ([p. 347](#)).
writeexport pool data to disk ([p. 383](#)).

Pool Data Members

String Values

@idname(i)*i*-th cross-section identifier.

Scalar Values

@aicAkaike information criterion.
@cofcov(i,j)covariance of coefficients *i* and *j*.
@coefs(i)coefficient *i*.
@dwDurbin-Watson statistic.
@effects(i)estimated fixed or random effect for the *i*-th cross-section member
(only for fixed or random effects).
@f*F*-statistic.
@logllog likelihood.
@meandepmean of the dependent variable.
@ncoeftotal number of estimated coefficients.
@ncrosstotal number of cross sectional units.
@ncrossestnumber of cross sectional units in last estimated pool equation.
@r2R-squared statistic.
@rbar2adjusted R-squared statistic.
@regobstotal number of observations in regression.
@schwarzSchwarz information criterion.
@sddepstandard deviation of the dependent variable.
@sestandard error of the regression.
@ssrsum of squared residuals.
@stderrs(i)standard error for coefficient *i*.
@totalobs total number of observations in the pool. For a balanced sample
this is “@regobs*@ncrossest”.
@tstats(i)*t*-statistic value for coefficient *i*.

c(i) *i*-th element of default coefficient vector for the pool.

Vectors and Matrices

@coefcov covariance matrix for coefficients of equation.

@coefs coefficient vector.

@effects vector of estimated fixed or random effects (only for fixed or random effects estimation).

@stderrs vector of standard errors for coefficients.

@tstats vector of *t*-statistic values for coefficients.

Pool Examples

To read data using the pool object

```
mypool1.read(b2) data.xls x? y? z?
```

and to delete and store pool series you may enter

```
mypool1.delete x? y?  
mypool1.store z?
```

Descriptive statistics may be computed using the command

```
mypool1.describe(m) z?
```

To estimate a pool equation using least squares and to access the *t*-statistics, enter

```
mypool1.ls y? c z? @ w?  
vector mypool1.@tstats
```

Rowvector

Row vector. (One dimensional array of numbers).

There are several ways to create a rowvector object. First, you can enter the `rowvector` keyword (with an optional dimension) followed by a name:

```
rowvector scalarmat  
rowvector(10) results
```

The resulting rowvector will be initialized with zeros.

Alternatively, you may combine a declaration with an assignment statement. The new vector will be sized and initialized accordingly:

```
rowvector(10) y=3  
rowvector z=results
```

Rowvector Views

barbar graph of each column (element) of the data against the row index ([p. 150](#)).
fillfill elements of the vector ([p. 208](#)).
labellabel information for the rowvector ([p. 238](#)).
lineline graph of each column (element) of the data against the row index ([p. 241](#)).
sheetspreadsheet view of the vector ([p. 327](#)).
spikespike graph ([p. 338](#)).
statsdescriptive statistics ([p. 344](#)).

Rowvector Procs

displaynameset display name ([p. 192](#)).
readimport data from disk ([p. 291](#)).
writeexport data to disk ([p. 383](#)).

Rowvector Data Members

(i)*i*-th element of the vector. Simply append “(i)” to the matrix name (without a “.”).

Rowvector Examples

To declare a rowvector and to fill it with data read from an Excel file:

```
rowvector(10) mydata  
mydata.read(b2) thedata.xls
```

To access a single element of the vector using direct indexing:

```
scalar result1=mydata(2)
```

The rowvector may be used in standard matrix expressions:

```
vector transdata=@transpose(mydata)  
scalar inner=@transpose(mydata)*mydata  
scalar inner1=@inner(mydata)
```

Sample

Sample of observations. Description of a set of observations to be used in operations.

To declare a sample object, use the keyword `sample`, followed by a name and a sample string:

```
sample mysample 1960:1 1990:4
```

```
sample altsample 120 170 300 1000 if x>0
```

Sample Procs

`set` reset the sample range ([p. 319](#)).

Sample Example

To change the observations in a sample object, you can use the `set` proc:

```
mysample.set 1960:1 1980:4 if y>0  
sample thesamp 1 10 20 30 40 60 if x>0  
thesamp.set @all
```

To set the current sample to use a sample, enter a `smp1` statement, followed by the name of the sample object:

```
smp1 mysample  
equation eq1.ls y x c
```

Scalar

Scalar (single number). A scalar holds a single numeric value. Scalar values may be used in standard EViews expressions in place of numeric values.

To declare a scalar object, use the keyword `scalar`, followed by a name, an “=” sign and a scalar expression or value.

Scalar objects have no views or procedures, and do not open windows. The value of the scalar may be displayed in the status line at the bottom of the EViews window.

Scalar Examples

You can declare a scalar and examine its contents in the status line:

```
scalar pi=3.14159  
scalar shape=beta(7)  
show shape
```

or you can declare a scalar and use it in an expression:

```
scalar inner=@transpose(mydata)*mydata  
series x=1/@sqrt(inner)*y
```

Series

Series of observations. An EViews series contains a set of observations on a variable.

To declare a series, use the keyword `series`, followed by a name, and optionally, by an “`=`” sign and a valid series expression:

```
series y  
series x=3*z
```

If there is no assignment, the series will be initialized to contain NAs.

Series Views

- `bar` bar graph of the series ([p. 150](#)).
- `bdstest` BDS independence test ([p. 152](#)).
- `cdfplot` distribution (cumulative, survivor, quantile) functions ([p. 157](#)).
- `correl` correlogram, autocorrelation and partial autocorrelation functions ([p. 172](#)).
- `edftest` empirical distribution function tests ([p. 198](#)).
- `freq` one-way tabulation ([p. 217](#)).
- `hist` descriptive statistics and histogram ([p. 229](#)).
- `kdensity` kernel density estimate ([p. 236](#)).
- `label` label information for the series ([p. 238](#)).
- `line` line graph of the series ([p. 241](#)).
- `qqplot` quantile-quantile plot ([p. 288](#)).
- `seasplot` seasonal line graph ([p. 316](#)).
- `sheet` spreadsheet view of the series ([p. 327](#)).
- `spike` spike graph ([p. 338](#)).
- `statby` statistics by classification ([p. 340](#)).
- `stats` descriptive statistics and histogram ([p. 229](#)).
- `testby` equality test by classification ([p. 357](#)).
- `teststat` simple hypothesis tests ([p. 362](#)).
- `uroot` unit root test ([p. 373](#)).

Series Procs

- `displayname` set display name ([p. 192](#)).
- `hpfilter` Hodrick-Prescott filter ([p. 231](#)).
- `seas` seasonal adjustment only for quarterly and monthly time series ([p. 315](#)).
- `resample` resample from the observations in the series ([p. 295](#)).

- smooth** exponential smoothing ([p. 330](#)).
- tramoseats** seasonal adjustment using Tramo/Seats ([p. 365](#)).
- x11** seasonal adjustment by Census X11 method only for quarterly and monthly time series ([p. 387](#)).
- x12** seasonal adjustment by Census X12 method only for quarterly and monthly time series ([p. 388](#)).

Series Data Members

- (i) i -th element of the series from the beginning of the workfile (when used on the left-hand side of an assignment, or when the element appears in a matrix, vector, or scalar assignment).

Series Element Functions

- @elem(ser, j)** function to access the j -th observation of the series SER, where j identifies the date or observation.

Series Examples

You can declare a series in the usual fashion:

```
series b=income*@mean(z)  
series blag=b(1)
```

Note that the last example, above, involves a series expression so that B(1) is treated as a one-period lead of the entire series, not as an element operator. In contrast,

```
scalar blag1=b(1)
```

evaluates the first observation on B in the workfile.

Once a series is declared, views and procs are available:

```
a.qqplot  
a.statby(mean, var, std) b
```

To access individual values:

```
scalar quarterlyval = @elem(y, "1980:3")  
scalar undatedval = @elem(x, 323)
```

Sspace

State space object. Estimation and evaluation of state space models using the Kalman filter.

To declare a sspace object, use the `sspace` keyword, followed by a valid name.

Sspace Method

mlmaximum likelihood estimation or filter initialization ([p. 269](#)).

Sspace Views

appendadd line to the specification ([p. 143](#)).

coefcovcoefficient covariance matrix ([p. 165](#)).

endogtable or graph of actual signal variables ([p. 200](#)).

gradsexamine the gradients of the log likelihood ([p. 223](#)).

labellabel information for the state space object ([p. 238](#)).

residcorstandardized one-step ahead residual correlation matrix ([p. 297](#)).

residcovstandardized one-step ahead residual covariance matrix ([p. 298](#)).

residsone-step ahead actual, fitted, residual graph ([p. 299](#)).

resultstable of estimation and filter results ([p. 300](#)).

signalgraphsdisplay graphs of signal variables ([p. 329](#)).

spectext representation of state space specification ([p. 337](#)).

statefinaldisplay the final values of the states or state covariance ([p. 343](#)).

stategraphsdisplay graphs of state variables ([p. 342](#)).

stateinitdisplay the initial values of the states or state covariance ([p. 343](#)).

structureexamine coefficient or variance structure of the specification
([p. 349](#)).

waldWald coefficient restriction test ([p. 378](#)).

Sspace Procs

displaynameset display name ([p. 192](#)).

forecastperform state and signal forecasting ([p. 214](#)).

makeendogmake group containing actual values for signal variables ([p. 251](#)).

makefiltermake new Kalman Filter ([p. 252](#)).

makegradsmake group containing the gradients of the log likelihood ([p. 253](#)).

makemodelmake a model object containing equations in sspace ([p. 257](#)).

makesignalsmake group containing signal and residual series ([p. 260](#)).

makestatesmake group containing state series ([p. 262](#)).

sspacedeclare sspace object ([p. 339](#)).

updatecoefsupdate coefficient vector(s) from sspace ([p. 372](#)).

Sspace Data Members

Scalar Values

@coefcov(i,j)covariance of coefficients i and j .

- @coefs(i) coefficient i .
- @eqregobs(k) number of observations in signal equation k .
- @sddep(k) standard deviation of the signal variable in equation k .
- @ssr(k) sum-of-squared standardized one-step ahead residuals for equation k .
- @stderrs(i) standard error for coefficient i .
- @tstats(t) t -statistic value for coefficient i .

Scalar Values (system level data)

- @aic Akaike information criterion for the system.
- @hq Hannan-Quinn information criterion for the system.
- @logl value of the log likelihood function.
- @ncoefs total number of estimated coefficients in the system.
- @neqns number of equations for observable variables.
- @regobs number of observations in the system.
- @sc Schwarz information criterion for the system.
- @totalobs sum of “@eqregobs” from each equation.

Vectors and Matrices

- @coefcov covariance matrix for coefficients of equation.
- @coefs coefficient vector.
- @stderrs vector of standard errors for coefficients.
- @tstats vector of t -statistic values for coefficients.

State and Signal Results

The following functions allow you to extract the filter and smoother results for the estimation sample and place them in matrix objects. In some cases, the results overlap those available thorough the sspace procs, while in other cases, the matrix results are the only way to obtain the results.

Note also that since the computations are only for the estimation sample, the one-step-ahead predicted state and state standard error values *will not* match the final values displayed in the estimation output. The latter are the predicted values for the first out-of-estimation sample period.

- @pred_signal matrix or vector of one-step ahead predicted signals.
- @pred_signalcov .. matrix where every row is the @vech of the one-step ahead predicted signal covariance.
- @pred_sigalse.... matrix or vector of the standard errors of the one-step ahead predicted signals.
- @pred_err..... matrix or vector of one-step ahead prediction errors.

@pred_errcovmatrix where every row is the @vech of the one-step ahead prediction error covariance.

@pred_errcovinv ..matrix where every row is the @vech of the inverse of the one-step ahead prediction error covariance.

@pred_erresematrix or vector of the standard errors of the one-step ahead prediction errors.

@pred_errstdmatrix or vector of standardized one-step ahead prediction errors.

@pred_statematrix or vector of one-step ahead predicted states.

@pred_statecov.....matrix where each row is the @vech of the one-step ahead predicted state covariance.

@pred_statese.....matrix or vector of the standard errors of the one-step ahead predicted states.

@pred_stateerrmatrix or vector of one-step ahead predicted state errors.

@curr_err.....matrix or vector of filtered error estimates.

@curr_gain.....matrix or vector where each row is the @vec of the Kalman gain.

@curr_statematrix or vector of filtered states.

@curr_statecovmatrix where every row is the @vech of the filtered state covariance.

@curr_statesematrix or vector of the standard errors of the filtered state estimates.

@sm_signalmatrix or vector of smoothed signal estimates.

@sm_sigmalcov.....matrix where every row is the @vech of the smoothed signal covariance.

@sm_signalsematrix or vector of the standard errors of the smoothed signals.

@sm_signalerr.....matrix or vector of smoothed signal error estimates.

@sm_sigmalerrcov.matrix where every row is the @vech of the smoothed signal error covariance.

@sm_signalerrse...matrix or vector of the standard errors of the smoothed signal error.

@sm_signalerrstd .matrix or vector of the standardized smoothed signal errors.

@sm_statematrix or vector of smoothed states.

@sm_statecov.....matrix where each row is the @vech of the smoothed state covariances.

@sm_statesematrix or vector of the standard errors of the smoothed state.

@sm_stateerr.....matrix or vector of the smoothed state errors.

@sm_stateerrcov...matrix where each row is the @vech of the smoothed state error covariance.

`@sm_stateerrse`.... matrix or vector of the standard errors of the smoothed state errors.
`@sm_stateerrstd`... matrix or vector of the standardized smoothed state errors .
`@sm_crosserrcov` . matrix where each row is the `@vec` of the smoothed error cross-covariance.

Sspace Examples

The one-step-ahead state values and variances from SS01 may be saved using

```
vector ss_state=ss01.@pred_state  
matrix ss_statecov=ss01.@pred_statecov
```

Sym

Symmetric matrix (symmetric two-dimensional array).

Declare by providing a name after the `sym` keyword, with the optionally specified dimension in parentheses:

```
sym(10) symmatrix
```

You may optionally assign a scalar, a square matrix or another `sym` in the declaration. If the square matrix is not symmetric, the `sym` will contain the lower triangle. The `sym` will be sized and initialized accordingly.

Sym Views

`bar`..... single or multiple bar graph of each column against the row index ([p. 150](#)).
`cor`..... correlation matrix by columns ([p. 171](#)).
`cov` covariance matrix by columns ([p. 175](#)).
`errbar` error bar graph view ([p. 201](#)).
`hilo`..... high-low(-open-close) chart ([p. 228](#)).
`label` label information for the symmetric matrix ([p. 238](#)).
`line`..... single or multiple line graph of each column against the row index ([p. 241](#)).
`pie`..... pie chart view ([p. 284](#)).
`scat`..... scatter diagrams of the columns of the matrix ([p. 311](#)).
`sheet`..... spreadsheet view of the matrix ([p. 327](#)).
`spike`..... spike graph ([p. 338](#)).
`stats`..... descriptive statistics by column ([p. 344](#)).
`xyline`..... XY line graph with one or more X columns plotted against one or more Y ([p. 394](#)).

Sym Procs

displaynameset display name ([p. 192](#)).
fillfill the elements of the matrix ([p. 208](#)).
readimport data from disk ([p. 291](#)).
writeexport data to disk ([p. 383](#)).

Sym Data Members

(i,j)(i,j)-th element of the matrix. Simply append “(i,j)” to the matrix name (without a “.”).

Sym Examples

The declaration

```
sym results(10)  
results=3
```

creates the 10×10 matrix RESULTS and initializes each value to be 3. The following assignment statements also create and initialize sym objects:

```
sym copymat=results  
sym covmat1=eq1.@coefcov  
sym(3,3) count  
count.fill 1,2,3,4,5,6,7,8,9,10
```

Graphs, covariances, and statistics may be generated for the columns of the matrix:

```
copymat.line  
copymat.cov  
copymat.stats
```

You can use explicit indices to refer to matrix elements:

```
scalar diagsum=cov1(1,1)+cov1(2,2)+cov(3,3)
```

System

System of equations for estimation.

Declare a system object by entering the keyword **system**, followed by a name:

```
system mysys
```

To fill a system, open the system and edit the specification view, or use **append**. Note that systems are not used for simulation. See “[Model](#)” ([p. 32](#)).

System Methods

3sls three-stage least squares ([p. 136](#)).
fiml full information maximum likelihood ([p. 210](#)).
gmm generalized method of moments ([p. 221](#)).
ls ordinary least squares ([p. 245](#)).
sur seemingly unrelated regression ([p. 350](#)).
tsls two-stage least squares ([p. 368](#)).
wls weighted least squares ([p. 380](#)).
wtsls weighted two-stage least squares ([p. 385](#)).

System Views

coefcov coefficient covariance matrix ([p. 165](#)).
derivs derivatives of the system equations ([p. 189](#)).
endog table or graph of endogenous variables ([p. 200](#)).
label label information for the system object ([p. 238](#)).
residcor residual correlation matrix ([p. 297](#)).
residcov residual covariance matrix ([p. 298](#)).
resids residual graphs ([p. 299](#)).
results table of estimation results ([p. 300](#)).
spec text representation of system specification ([p. 337](#)).
wald Wald coefficient restriction test ([p. 378](#)).

System Procs

append add a line of text to the system specification ([p. 143](#)).
displayname set display name ([p. 192](#)).
makeendog make group of endogenous series ([p. 251](#)).
makemodel create a model from the estimated system ([p. 257](#)).
makeresids make series containing residuals from system ([p. 259](#)).
updatecoefs update coefficient vector(s) from system ([p. 372](#)).

System Data Members

Scalar Values (individual equation data)

@coefcov(i, j) covariance of coefficients i and j .
@coefs(i) coefficient i .
@dw(k) Durbin-Watson statistic for equation k .
@eqncoef(k) number of estimated coefficients in equation k .
@eqregobs(k) number of observations in equation k .

@meandep(k) mean of the dependent variable in equation k .
@ncoef(k) total number of estimated coefficients in equation k .
@r2(k) R-squared statistic for equation k .
@rbar2(k) adjusted R-squared statistic for equation k .
@sddep(k) standard deviation of dependent variable in equation k .
@se(k) standard error of the regression in equation k .
@ssr(k) sum of squared residuals in equation k .
@stderrs(i) standard error for coefficient i .
@tstats(i) t -statistic for coefficient i .
c(i) i -th element of default coefficient vector for system (if applicable).

Scalar Values (system level data)

@aic Akaike information criterion for the system (if applicable).
@detresid determinant of the residual covariance matrix.
@hq Hannan-Quinn information criterion for the system (if applicable).
@jstat J -statistic — value of the GMM objective function (for GMM estimation).
@logl value of the log likelihood function for the system (if applicable).
@ncoefs total number of estimated coefficients in system.
@neqn number of equations.
@regobs number of observations in the sample range used for estimation
(" @regobs " will differ from " @eqregobs " if the unbalanced sample is non-overlapping).
@schwarz Schwarz information criterion for the system (if applicable).
@totalobs sum of " @eqregobs " from each equation.

Vectors and Matrices

@cofcov covariance matrix for coefficients of equation.
@coefs coefficient vector.
@stderrs vector of standard errors for coefficients.
@tstats vector of t -statistic values for coefficients.

System Examples

To estimate a system using GMM and to create residual series for the estimated system:

```
sys1.gmm(i,m=7,c=.01,b=v)
sys1.makeresids consres incres saveres
```

To test coefficients using a Wald test:

```
sys1.wald c(1)=c(4)
```

To save the coefficient covariance matrix:

```
sym covs=sys1.@coefcov
```

Table

Table object. Formatted two-dimensional table for output display.

To declare a table object, use the keyword `table`, followed by an optional row and column dimension, and then the object name:

```
table onelement  
table(10,5) outtable
```

If no dimension is provided, the table will contain a single element.

Alternatively, you may declare a table using an assignment statement. The new table will be sized and initialized, accordingly:

```
table newtable=outtable
```

Lastly, you may use the `freeze` command to create tables from tabular views of other objects ([p. 216](#)).

Table Views

- `label` label information for the table object ([p. 238](#)).
- `sheet` view table ([p. 327](#)).

Table Procs

- `displayname` set display name ([p. 192](#)).

Table Data Members

- `(i,j)` the (i,j) -th element of the table, formatted as a string.

Table Commands

- `setcell` format and fill in a table cell ([p. 319](#)).
- `setcolwidth` set width of a table column ([p. 321](#)).
- `setline` place a horizontal line in table ([p. 326](#)).

Table Examples

```
table(5,5) mytable  
%strval = mytable(2,3)  
mytable(4,4) = "R2"
```

```
mytable(4,5) = @str(eq1.@r2)
```

Var

Vector autoregression and error correction object.

To declare a var use the keyword `var`, followed by a name and, optionally, by an estimation specification:

```
var finvar  
var empvar.ls 1 4 payroll hhold gdp  
var finec.ec(e,2) 1 6 cp div r
```

Var Methods

- `ec` estimate a vector error correction model ([p. 196](#)).
- `ls` estimate an unrestricted VAR ([p. 245](#)).

Var Views

- `arlm` serial correlation LM test ([p. 148](#)).
- `arroots` inverse roots of the AR polynomial ([p. 148](#)).
- `coint` Johansen cointegration test ([p. 166](#)).
- `correl` residual autocorrelations ([p. 172](#)).
- `decomp` variance decomposition ([p. 186](#)).
- `endog` table or graph of endogenous variables ([p. 200](#)).
- `impulse` impulse response functions ([p. 232](#)).
- `jbera` residual normality test ([p. 234](#)).
- `label` label information for the var object ([p. 238](#)).
- `laglen` lag order selection criteria ([p. 239](#)).
- `qstats` residual portmanteau tests ([p. 289](#)).
- `representation` text describing var specification ([p. 294](#)).
- `residcor` residual correlation matrix ([p. 297](#)).
- `residcov` residual covariance matrix ([p. 298](#)).
- `resids` residual graphs ([p. 299](#)).
- `results` table of estimation results ([p. 300](#)).
- `testexog` exogeneity (Granger causality) tests ([p. 359](#)).
- `testlags` lag exclusion tests ([p. 361](#)).
- `white` White heteroskedasticity test ([p. 379](#)).

Var Procs

- `append` append restriction text ([p. 143](#)).

cleartext clear restriction text ([p. 163](#)).
displayname set display name ([p. 192](#)).
makecoint make group of cointegrating relations ([p. 250](#)).
makeendog make group of endogenous series ([p. 251](#)).
makemodel make model from the estimated var ([p. 257](#)).
makeresids make residual series ([p. 259](#)).
makesystem make system from var ([p. 264](#)).
svar estimate structural factorization ([p. 351](#)).

Var Data Members

Scalar Values (individual level data)

@eqlogl(k) log likelihood for equation k .
@eqncoef(k) number of estimated coefficients in equation k .
@eqregobs(k) number of observations in equation k .
@meandep(k) mean of the dependent variable in equation k .
@r2(k) R-squared statistic for equation k .
@rbar2(k) adjusted R-squared statistic for equation k .
@sddep(k) std. dev. of dependent variable in equation k .
@se(k) standard error of the regression in equation k .
@ssr(k) sum of squared residuals in equation k .
a(i,j) adjustment coefficient for the j -th cointegrating equation in the i -th equation of the VEC (where applicable).
b(i,j) coefficient of the j -th variable in the i -th cointegrating equation (where applicable).
c(i,j) coefficient of the j -th regressor in the i -th equation of the var, or the coefficient of the j -th first-difference regressor in the i -th equation of the VEC.

Scalar Values (system level data)

@aic Akaike information criterion for the system.
@detresid determinant of the residual covariance matrix.
@hq Hannan-Quinn information criterion for the system.
@logl log likelihood for system.
@ncoefs total number of estimated coefficients in the var.
@neqn number of equations.
@regobs number of observations in the var.
@sc Schwarz information criterion for the system.

@svarcvgtypeReturns an integer indicating the convergence type of the structural decomposition estimation: 0 (convergence achieved), 2 (failure to improve), 3 (maximum iterations reached), 4 (no convergence—structural decomposition not estimated).
@svaroveridover-identification LR statistic from structural factorization.
@totalobssum of “@eqregobs” from each equation (“@regobs*@neqn”).

Vectors and Matrices

@coefmatcoefficient matrix (as displayed in output table).
@coefsematrix of coefficient standard errors (corresponding to the output table).
@cointsestandard errors of cointegrating vectors.
@cointveccointegrating vectors.
@impfactfactorization matrix used in last impulse response view.
@lrrspaccumulated long-run responses from last impulse response view.
@lrrspsestandard errors of accumulated long-run responses.
@residcov(sym) covariance matrix of the residuals.
@svaramatestimated A matrix for structural factorization.
@svarbmatestimated B matrix for structural factorization.
@svarcovabcovariance matrix of stacked A and B matrix for structural factorization.
@svarrcovrestricted residual covariance matrix from structural factorization.

Var Examples

To declare a var estimate a VEC specification and make a residual series:

```
var finec.ec(e,2) 1 6 cp div r  
finec.makeresids
```

To estimate an ordinary var, to create series containing residuals, and to form a model based upon the estimated var:

```
var empvar.ls 1 4 payroll hhold gdp  
empvar.makeresids payres hholdres gdpres  
empvar.makemodel(inmdl) cp fcp div fdiv r fr
```

To save coefficients in a scalar:

```
scalar coef1=empvar.b(1,2)
```

Vector

Vector. (One dimensional array of numbers).

There are several ways to create a vector object. Enter the `vector` keyword (with an optional dimension) followed by a name:

```
vector scalarmat  
vector(10) results
```

Alternatively, you may declare a vector using an assignment statement. The vector will be sized and initialized, accordingly:

```
vector(10) myvec=3.14159  
vector results=vec1
```

Vector Views

bar..... bar graph of data against the row index ([p. 150](#)).
label label information for the vector object ([p. 238](#)).
line line graph of the data against the row index ([p. 241](#)).
sheet spreadsheet view of the vector ([p. 327](#)).
spike spike graph ([p. 338](#)).
stats descriptive statistics ([p. 344](#)).

Vector Procs

displayname..... set display name ([p. 192](#)).
fill fill elements of the vector ([p. 208](#)).
read import data from disk ([p. 291](#)).
write export data to disk ([p. 383](#)).

Vector Data Members

(i) (i)-th element of the vector. Simply append “(i)” to the matrix name (without a “.”).

Vector Examples

To declare a vector and to fill it with data read in from an Excel file:

```
vector(10) mydata  
mydata.read(b2) thedata.xls
```

To access a single element of the vector using direct indexing:

```
scalar result1=mydata(2)
```

The vector may be used in standard matrix expressions:

```
rowvector transdata=@transpose(mydata)
scalar inner=@transpose(mydata)*mydata
```


Chapter 4. Matrix Language

EViews provides you with tools for working directly with data contained in matrices and vectors. You can use the EViews matrix language to perform calculations that are not available using the built-in views and procedures.

The following objects can be created and manipulated using the matrix command language:

- coef: column vector of coefficients to be used by equation, system, pool, logl, and sspace objects
- matrix: two-dimensional array
- rowvector: row vector
- scalar: scalar
- sym: symmetric matrix (stored in lower triangular form)
- vector: column vector

We term these objects *matrix objects* (despite the fact that some of these objects are not matrices).

Declaring Matrices

You must declare matrix objects prior to use. Detailed descriptions of declaration statements for the various matrix objects are provided in [Chapter 8, “Command Reference”, beginning on page 135](#).

Briefly, a declaration consists of the object *keyword*, followed either by size information in parentheses and the name to be given to the object, followed (optionally) by an assignment statement. If no assignment is provided, the object will be initialized to have all zero values.

The various matrix objects require different sizing information. A matrix requires the number of rows and the number of columns. A sym requires that you specify a single number representing both the number of rows and the number of columns. A vector, rowvector, or coef declaration can include information about the number of elements. A scalar requires no size information. If size information is not provided, EViews will assume that there is only one element in the object.

For example:

```
matrix(3,10) xdata
```

```
sym(9) moments
vector(11) betas
rowvector(5) xob
```

creates a 3×10 matrix XDATA, a symmetric 9×9 matrix MOMENTS, an 11×1 column vector BETAS, and a 1×5 rowvector XOB. All of these objects are initialized to zero.

To change the size of a matrix object, you can repeat the declaration statement. Furthermore, if you use an assignment statement with an existing matrix object, the target will be resized as necessary. For example:

```
sym(10) bigz
matrix zdata
matrix(10,2) zdata
zdata = bigz
```

will first declare ZDATA to be a matrix with a single element, and then redeclare ZDATA to be a 10×2 matrix. The assignment statement in the last line will resize ZDATA so that it contains the contents of the 10×10 symmetric matrix BIGZ.

Assigning Matrix Values

There are three ways to assign values to the elements of a matrix: you may assign values to specific matrix elements, you may fill the matrix using a list of values, or you may perform matrix assignment.

Element assignment

The most basic method of assigning matrix values is to assign a value for a specific row and column element of the matrix. Simply enter the matrix name, followed by the row and column indices, in parentheses, and then an assignment to a scalar value.

For example, suppose we declare the 2×2 matrix A:

```
matrix(2,2) a
```

The first command creates and initializes the 2×2 matrix A so that it contains all zeros. Then after entering the two commands:

```
a(1,1) = 1
a(2,1) = 4
```

we have

$$A = \begin{bmatrix} 1 & 0 \\ 4 & 0 \end{bmatrix}. \quad (4.1)$$

You can perform a large number of element assignments by placing them inside of programming loops:

```
vector(10) y
matrix (10,10) x
for !i = 1 to 10
    y(!i) = !i
    for !j = 1 to 10
        x(!i,!j) = !i + !j
    next
next
```

Note that the `fill` procedure provides an alternative to using loops for assignment.

Fill assignment

The second assignment method is to use the `fill` procedure to assign a list of numbers to each element of the matrix in the specified order. By default, the procedure fills the matrix column by column, but you may override this behavior.

You should enter the name of the matrix object, followed by a period, the `fill` keyword, and then a *comma delimited* list of values. For example, the commands:

```
vector(3) v
v1.fill 0.1, 0.2, 0.3
matrix(2,4) x
matrix.fill 1, 2, 3, 4, 5, 6, 7, 8
```

create the matrix objects

$$V = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \end{bmatrix}, \quad X = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \end{bmatrix} \quad (4.2)$$

If we replace the last line with

```
matrix.fill(b=r) 1,2,3,4,5,6,7,8
```

then X is given by

$$X = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}. \quad (4.3)$$

In some situations, you may wish to repeat the assignment over a list of values. You may use the “l” option to fill the matrix by repeatedly looping through the listed numbers until the matrix elements are exhausted. Thus,

```
matrix(3,3) y  
y.fill(l) 1, 0, -1
```

creates the matrix

$$Y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \quad (4.4)$$

See [fill \(p. 208\)](#) for a complete description of the `fill` procedure.

Matrix assignment

You can copy data from one matrix object into another using assignment statements. To perform an assignment, you should enter the name of the target matrix followed by the equal sign “=”, and then a matrix object expression. The expression on the right-hand side should either be a numerical constant, a matrix object, or an expression that returns a matrix object.

There are a variety of rules for how EViews performs the assignment that depend upon the types of objects involved in the assignment.

Scalar values on the right-hand side

If there is a scalar on the right-hand side of the assignment, every element of the matrix object is assigned the value of the scalar.

Examples:

```
matrix(5,8) first  
scalar second  
vec(10) third  
first = 5  
second = c(2)  
third = first(3,5)
```

Since declaration statements allow for initialization, you can combine the declaration and assignment statements. Examples:

```
matrix(5,8) first = 5  
scalar second = c(2)  
vec(10) third = first(3,5)
```

Same object type on right-hand side

If the *source* object on the right is a matrix or vector, and the *target* or *destination* object on the left is of the same type, the target will be resized to have the same dimension as the source, and every source element will be copied. For example:

```
matrix(10,2) zdata = 5
matrix ydata = zdata
matrix(10,10) xdata = ydata
```

declares that ZDATA is a 10×2 matrix filled with 5's. YDATA is automatically resized to be a 10×2 matrix and is filled with the contents of ZDATA.

Note that even though the declaration of XDATA calls for a 10×10 matrix, XDATA is a 10×2 matrix of 5's. This behavior occurs because the declaration statement above is equivalent to issuing the two commands,

```
matrix(10,10) xdata
xdata = ydata
```

which will first declare the 10×10 matrix XDATA, and then automatically resize it to 10×2 when you fill it with the values for YDATA.

The next section discusses assignment statements in the more general case, where you are converting between object types. In some cases, the conversion is automatic; in other cases, EViews provides you with additional tools to perform the conversion.

Copying Data Between Objects

In addition to the basic assignment statements described in the previous section, EViews provides you with a large set of tools for copying data to and from matrix objects.

At times, you may wish to move data between different types of matrix objects. For example, you may wish to take the data from a vector and put it in a matrix. EViews has a number of built-in rules which make these conversions automatically.

At other times, you may wish to move data between a matrix object and an EViews series or group object. There are a separate set of tools which allow you to convert data across a variety of object types.

Copying data from matrix objects

Data may be moved between different types of matrix objects using assignments. If possible, EViews will resize the target object so that it contains the same information as the object on the right side of the equation.

The basic rules governing expressions of the form “Y = X” may be summarized as follows:

- Object type of Y does not change.
- The target object Y will, if possible, be resized to match the object X; otherwise, EViews will issue an error. Thus, assigning a vector to a matrix will resize the matrix, but assigning a matrix to a vector will generate an error if the matrix has more than one column.
- The data in X will be copied to Y.

Specific exceptions to the rules given above are:

- If X is a scalar, Y will keep its original size and will be filled with the value of X.
- If X and Y are both vector or rowvector objects, Y will be changed to the same type as X.

[“Summary of Automatic Resizing of Matrix Objects” on page 74](#) contains a complete summary of the conversion rules for matrix objects.

Here are some simple examples illustrating the rules for matrix assignment:

```
vector(3) x
x(1) = 1
x(2) = 2
x(3) = 3
vector y = x
matrix z = x
```

Y is now a 3 element vector because it has the same dimension and values as X. EViews automatically resizes the Z Matrix to conform to the dimensions of X so that Z is now a 3×1 matrix containing the contents of X: Z(1,1) = 1, Z(2,1) = 2, Z(3,1) = 3.

Here are some further examples where automatic resizing is allowed:

```
vector(7) y = 2
scalar value = 4
matrix(10,10) w = value
w = y
matrix(2,3) x = 1
rowvector(10) t = 100
x = t
```

W is declared as a 10×10 matrix of 4's, but it is then reset to be a 7×1 matrix of 2's. X is a 1×10 matrix of 100's.

Lastly, consider the commands

```
vector(7) y = 2
rowvector(12) z = 3
coef(20) beta
Y = z
z = beta
```

Y will be a rowvector of length 3, containing the original contents of Z, and Z will be a column vector of length 20 containing the contents of BETA.

There are some cases where EViews will be unable to perform the specified assignment because the resize operation is not defined. For example, suppose that X is a 2×2 matrix. Then the assignment statement

```
vector(7) y = x
```

will result in an error. EViews cannot change Y from a vector to a matrix and there is no way to assign the 4 elements of the matrix X to the vector Y. Other examples of invalid assignment statements involve assigning matrix objects to scalars or syms to vector objects.

Copying data from parts of matrix objects

In addition to the standard rules for conversion of data between objects, EViews provides functions for extracting and assigning parts of matrix objects. Matrix functions are described in greater detail later in this chapter. For now, note that some functions take a matrix object and perhaps other parameters as arguments and return a matrix object.

A comprehensive list of the EViews commands and functions that may be used for matrix object conversion appears in “[Utility Functions and Commands](#)” on page 76. However, a few examples will provide you with a sense of the type of operations that may be performed.

Suppose first that you are interested in copying data from a matrix into a vector. The following commands will copy data from M1 and SYM1 into the vectors V1, V2, V3, and V4.

```
matrix(10, 10) m1
sym(10) sym1
vector v1 = @vec(m1)
vector v2 = @columnextract(m1, 3)
vector v3 = @rowextract(m1, 4)
vector v4 = @columnextract(sym1, 5)
```

The @vec function creates a 100 element vector, V1, from the columns of M1 stacked one on top of another. V2 will be a 10 element vector containing the contents of the third col-

umn of M1 while V3 will be a 10 element vector containing the fourth row of M1. The @vec, @rowextract, and @columnextract functions also work with sym objects. V4 is a 10 element vector containing the fifth column of SYM1.

You can also copy data from one matrix into a smaller matrix using @subextract. For example:

```
matrix(20,20) m1=1  
matrix m2 = @subextract(m1,5,5,10,7)  
matrix m3 = @subextract(m1,5,10)  
matrix m4 = m1
```

M2 is a 6×3 matrix containing a submatrix of M1 defined by taking the part of the matrix M1 beginning at row 5 and column 5 and ending at row 10 and column 7. M3 is the 16×11 matrix taken from M1 at row 5 and column 10 to the last element of the matrix (row 20 and column 20). In contrast, M4 is defined to be an exact copy of the full 20×20 matrix.

Data from a matrix may be copied into another matrix object using the commands colplace, rowplace, and matplace. Consider the commands:

```
matrix(100,5) m1 = 0  
matrix(100,2) m2 = 1  
vector(100) v1 = 3  
rowvector(100) v2 = 4  
matplace(m1,m2,1,3)  
colplace(m1,v1,3)  
rowplace(m1,v2,80)
```

The matplace command places M2 in M1 beginning at row 1 and column 3. V1 is placed in column 3 of M1, while V2 is placed in row 80 of M1.

Copying data between matrix objects and other objects

The previous sections described techniques for copying data between matrix objects such as vectors, matrices and scalars. In this section, we describe techniques for copying data between matrix objects and other EViews objects such as series and groups.

Keep in mind that there are two primary differences between the ordinary series or group objects and the matrix objects. First, operations involving series and groups use information about the current workfile sample, while matrix objects do not. Second, there are important differences in the handling of missing values (NAs) between the two types of objects.

Direct Assignment

The easiest method to copy data from series or group objects to a matrix object is to use direct assignment. Place the destination matrix object on the left side of an equal sign, and place the series or group to be converted on the right.

If you use a series object on the right, EViews will only include the observations from the current sample to make the vector. If you place a group object on the right, EViews will create a rectangular matrix, again only using observations from the current sample.

While very convenient, there are two principal limitations of this approach. First, EViews will only include observations in the current sample when copying the data. Second, observations containing missing data (NAs) for a series, or for any series in the group, are not placed in the matrix. Thus, if the current sample contains 20 observations but the series or group contains missing data, the dimension of the vector or matrix will be less than 20. Below, we provide you with methods which allow you to override the current sample and to retain missing values.

Examples:

```
smpl 1963:3 1993:6
fetch hsf gmpyq
group mygrp hsf gmpyq
vector xvec = gmpyq
matrix xmat = mygrp
```

These statements create the vector XVEC and the two column matrix XMAT containing the non-missing series and group data from 1963:3 to 1993:6. Note that if GMPYQ has a missing value in 1970:01, and HSF contains a missing value in 1980:01, both observations for both series will be excluded from XMAT.

When performing matrix assignment, you may refer to an element of a series, just as you would refer to an element of a vector, by placing an index value in parentheses after the name. An index value *i* refers to the *i*-th element of the series from the beginning of the workfile *range*. For example, if the range of the current annual workfile is 1961 to 1980, the expression GNP(6) refers to the 1966 value of GNP. These series element expressions may be used in assigning specific series values to matrix elements, or to assign matrix values to a specific series element. For example:

```
matrix(5,10) x
series yser = nrnd
x(1,1) = yser(4)
yser(5) = x(2,3)
yser(6) = 4000.2
```

assigns the fourth value of the series YSER to X(1,1), and assigns to the fifth and sixth values of YSER, the X(2,3) value and the scalar value “4000.2”, respectively.

While matrix assignments allow you to refer to elements of series as though they were elements of vectors, you cannot generally use series in place of vectors. Most vector and matrix operations will error if you use a series in place of a vector. For example, you cannot perform a `rowplace` command using a series name.

Furthermore, note that when you are not performing matrix assignment, a series name followed by a number in parentheses will indicate that the lag/lead operator be applied to the entire series. Thus, when used in generating series or in an equation, system, or model specification, GNP(6) refers to the sixth lead of the GNP series. To refer to specific elements of the GNP series in these settings, you should use the `@elem` function.

Copy using `@convert`

The `@convert` function takes a series or group object and, optionally, a sample object, and returns a vector or rectangular matrix. If no sample is provided, `@convert` will use the workfile sample. The sample determines which series elements are included in the matrix. Example:

```
smp1 61 90
group groupx inv gdp m1
vector v = @convert(gdp)
matrix x = @convert(groupx)
```

X is a 30×3 matrix with the first column containing data from INV, the second column from GDP, and the third column from M1.

As with direct assignment, the `@convert` function excludes observations for which the series or any of the series in the group contain missing data. If, in the example above, INV contains missing observations in 1970 and 1980, V would be a 29 element vector while X would be a 28×3 matrix. This will cause errors in subsequent operations that require V and X to have a common row dimension.

There are two primary advantages of using `@convert` over direct assignment. First, since `@convert` is a function, it may be used in the middle of a matrix expression. Second, an optional second argument allows you to specify a sample to be used in conversion. For example,

```
sample s1.set 1950 1990
matrix x = @convert(grp,s1)
sym y = @inverse(@inner(@convert(grp,s1)))
```

performs the conversion using the sample defined in S1.

Copy data between Series and Matrices

EViews also provides three useful commands that perform explicit conversions between series and matrices with control over both the sample, and the handling of NAs.

stom (Series TO Matrix) takes a series or group object and copies its data to a vector or matrix using either the current workfile sample, or the optionally specified sample. As with direct assignment, the **stom** command excludes observations for which the series or any of the series in the group contain missing data.

Example:

```
sample smpl_cnvrt.set 1950 1995
smpl 1961 1990
group group1 gnp gdp money
vector(46) vec1
matrix(3,30) mat1
stom(gdp,vec1,smpl_cnvrt)
stom(group1,mat1)
```

While the operation of **stom** is similar to **@convert**, **stom** is a command and cannot be included in a matrix expression. Furthermore, unlike **@convert**, the destination matrix or vector must already exist and have the proper dimension.

stomna (Series TO Matrix with NAs) works identically to **stom**, but does not exclude observations for which there are missing values. The elements of the series for the relevant sample will map directly into the target vector or matrix. Thus,

```
smpl 1951 2000
vector(50) gvector
stom(gdp,gvector)
```

will always create a 50 element vector GVECTOR that contains the values of GDP from 1951 to 2000, including observations with NAs.

mtos (Matrix TO Series) takes a matrix or vector and copies its data into an existing series or group, using the current workfile sample or a sample that you provide.

Examples:

```
mtos(mat1,group1)
mtos(vec1,resid)
mtos(mat2,group1,smpl1)
```

As with **stom** the destination series or group must already exist and the destination dimension given by the sample must match that of the source vector or matrix.

Matrix Expressions

A *matrix expression* is an expression which combines matrix objects using mathematical operators or relations, functions, and parentheses. While we discuss matrix functions in great detail below, some examples will demonstrate the relevant issues.

Examples:

```
@inner (@convert (grp,s1))  
mat1*vec1  
@inverse (mat1+mat2)*vec1  
mat1 > mat2
```

EViews uses the following rules to determine the order in which the expression will be evaluated:

- You may nest any number of pairs of parentheses to clarify the order of operations in a matrix expression.
- If you do not use parentheses, the operations are applied in the following order:
 1. Unary negation operator and functions
 2. Multiplication and division operators
 3. Addition and subtraction operators
 4. Comparison operators: “ $>=$ ”, “ $>$ ”, “ $<=$ ”, “ $<$ ”, “ $<>$ ”

Examples:

```
@inverse (mat1+mat2) +@inverse (mat3+mat4)  
vec1*@inverse (mat1+mat2)*@transpose (vec1)
```

In the first example, the matrices MAT1 and MAT2 will be added and then inverted. Similarly the matrices MAT3 and MAT4 are added and then inverted. Finally, the two inverses will be added together. In the second example, EViews first inverts MAT1 + MAT2 and uses the result to calculate a quadratic form with VEC1.

Matrix Operators

EViews provides standard mathematical operators for matrix objects.

Negation (-)

The unary minus changes the sign of every element of a matrix object, yielding a matrix or vector of the same dimension. Example:

```
matrix jneg = -jpos
```

Addition (+)

You can add two matrix objects of the same type and size. The result is a matrix object of the same type and size. Example:

```
matrix(3,4) a
matrix(3,4) b
matrix sum = a + b
```

You can add a square matrix and a sym of the same dimension. The upper triangle of the sym is taken to be equal to the lower triangle. Adding a scalar to a matrix object adds the scalar value to each element of the matrix or vector object.

Subtraction (-)

The rules for subtraction are the same as the rules for addition. Example:

```
matrix(3,4) a
matrix(3,4) b
matrix dif = a - b
```

Subtracting a scalar object from a matrix object subtracts the scalar value from every element of the matrix object.

Multiplication (*)

You can multiply two matrix objects if the number of columns of the first matrix is equal to the number of rows of the second matrix.

Example:

```
matrix(5,9) a
matrix(9,22) b
matrix prod = a * b
```

In this example, PROD will have 5 rows and 22 columns.

One or both of the matrix objects can be a sym. Note that the product of two sym objects is a matrix, not a sym. The @inner function will produce a sym by multiplying a matrix by its own transpose.

You can premultiply a matrix or a sym by a vector if the number of columns of the matrix is the same as the number of elements of the vector. The result is a vector whose dimension is equal to the number of rows of the matrix.

Example:

```
matrix(5,9) mat
vector(9) vec
vector res = mat * vec
```

In this example, RES will have 5 elements.

You can premultiply a rowvector by a matrix or a sym if the number of elements of the rowvector is the same as the number of rows of the matrix. The result is a rowvector whose dimension is equal to the number of columns of the matrix.

Example:

```
rowvector rres
matrix(5,9) mat
rowvector(5) row
rres = row * mat
```

In this example, RRES will have 9 elements.

You can multiply a matrix object by a scalar. Each element of the original matrix is multiplied by the scalar. The result is a matrix object of the same type and dimensions as the original matrix. The scalar can come before or after the matrix object. Examples:

```
matrix prod = 3.14159*orig
matrix xxx = d_mat*7
```

Division (/)

You can divide a matrix object by a scalar. Example:

```
matrix z = orig/3
```

Each element of the object ORIG will be divided by 3.

Comparison Operators (=, >, >=, <, <=, <>)

Two matrix objects of the same type and size may be compared using the comparison operators. The result is a scalar logical value. Every pair of corresponding elements is tested, and if any pair fails the test, the value zero or false is returned; otherwise, the value one or true is returned.

Example:

```
if result <> value then
    run crect
endif
```

It is possible for a vector to be not greater than, not less than, and not equal to a second vector. For example:

```
vector(2) v1
vector(2) v2
v1(1) = 1
v1(2) = 2
v2(1) = 2
v2(2) = 1
```

Since the first element of V1 is smaller than the first element of V2, V1 is not greater than V2. Since the second element of V1 is larger than the second element of V2, V1 is not less than V2. The two vectors are not equal.

Matrix Commands and Functions

EViews provides a number of commands and functions that allow you to work with the contents of your matrix objects. These commands and functions may be divided into roughly four distinct types:

1. Utility Commands and Functions
2. Matrix Algebra Functions
3. Descriptive Statistics Functions
4. Element Functions

The utility commands and functions provide support for creating, manipulating, and assigning values to your matrix objects. We have already seen the `@convert` function and the `stom` command, both of which convert data from series and groups into vectors and matrices.

The matrix algebra functions allow you to perform common matrix algebra manipulations and computations. Among other things, you can use these routines to compute eigenvalues, eigenvectors and determinants of matrices, to invert matrices, to solve linear systems of equations, and to perform singular value decompositions.

The descriptive statistics functions compute summary statistics for the data in the matrix object. You can compute statistics such as the mean, median, minimum, maximum, and variance, over all of the elements in your matrix.

The matrix element functions allow you create a new matrix containing the values of a function evaluated at each element of another matrix object. Most of the functions that are available in series expressions may be applied to matrix objects. You can compute the log-

arithm of every element of a matrix, or the cumulative normal distribution at every element of a vector.

A listing of the commands and functions is included in the matrix summary on [page 76](#). Functions for computing descriptive statistics for data in matrices are discussed in “[Descriptive Statistics](#)” on page 439. Additional details on matrix element computations are provided in “[Matrix Operators](#)” on page 66.

Functions versus Commands

A *function* generally takes arguments, and always returns a result. Functions are easily identified by the initial “@” character in the function name.

There are two basic ways that you can use a function. First, you may assign the result to an EViews object. This object may then be used in other EViews expressions, providing you access to the result in subsequent calculations. For example,

```
matrix y = @transpose(x)
```

stores the transpose of matrix X in the matrix Y. Since Y is a standard EViews matrix, it may be used in all of the usual expressions.

Second, you may use a function as part of a matrix expression. Since the function result is used *in-line*, it will not be assigned to a named object, and will not be available for further use. For example, the command

```
scalar z = vec1*@inverse(v1+v2)*@transpose(vec1)
```

uses the results of the @inverse and @transpose functions in forming the scalar expression assigned to Z. These function results will not be available for subsequent computations.

By contrast, a *command* takes object names and expressions as arguments, and operates on the named objects. Commands do not return a value.

Commands, which do not have a leading “@” character, must be issued alone on a line, rather than as part of a matrix expression. For example, to convert a series X to a vector V1, you would enter

```
stom(x,v1)
```

Because the command does not return any values, it may not be used in a matrix expression.

NA Handling

As noted above, most of the methods of moving data from series and groups into matrix objects will automatically drop observations containing missing values. It is still possible, however, to encounter matrices which contain missing values.

For example, the automatic NA removal may be overridden using the `stomna` command ([p. 346](#)). Additionally, some of the element operators may generate missing values as a result of standard matrix operations. For example, taking element-by-element logarithms of a matrix using `@log` will generate NAs for all cells containing nonpositive values.

EViews follows two simple rules for handling matrices that contain NAs. For all operators, commands, and functions, *except the descriptive statistics function*, EViews works with the full matrix object, processing NAs as required. For descriptive statistic functions, EViews automatically drops NAs when performing the calculation. These rules imply the following:

- Matrix operators will generate NAs where appropriate. Adding together two matrices that contain NAs will yield a matrix containing NAs in the corresponding cells. Multiplying two matrices will result in a matrix containing NAs in the appropriate rows and columns.
- All matrix algebra functions and commands will generate NAs, since these operations are undefined. For example, the Cholesky factorization of a matrix that contains NAs will contain NAs.
- All utility functions and commands will work as before, with NAs treated like any other value. Copying the contents of a vector into a matrix using `colplace` will place the contents, including NAs, into the target matrix.
- All of the matrix element functions will propagate NAs when appropriate. Taking the absolute value of a matrix will yield a matrix containing absolute values for non-missing cells and NAs for cells that contain NAs.
- The descriptive statistics functions are based upon the non-missing subset of the elements in the matrix. You can always find out how many values were used in the computations by using the `@OBS` function.

Matrix Views and Procs

The object listing in [Chapter 3, “Object, View and Procedure Reference”](#), on page 19 lists the various views and procs for all of the matrix objects.

Matrix Graph and Statistics Views

All of the matrix objects, with the exception of the scalar object, have windows and views. For example, you may display line and bar graphs for each column of the 10×5 matrix Z:

```
z.line  
z.bar(p)
```

each column will be plotted against the row number of the matrix.

Additionally, you can compute descriptive statistics for each column of a matrix, as well as the correlation and covariance matrix between the columns of the matrix:

```
z.stats  
z.cor  
z.cov
```

EViews performs listwise deletion by column, so that each group of column statistics is computed using the largest possible set of observations.

The full syntax for the commands to display and print these views is listed in the object reference.

Matrix input and output

EViews provides you with the ability to read and write files directly from matrix objects using the read and write procedures.

You must supply the name of the source file. If you do not include the optional path specification, EViews will look for the file in the default directory. The input specification follows the source file name. Path specifications may point to local or network drives. If the path specification contains a space, you must enclose the entire expression in double quotes “”.

In reading from a file, EViews first fills the matrix with NAs, places the first data element in the “(1,1)” element of the matrix, then continues to read the data by row or by column, depending upon the options set.

The following command reads data into MAT1 from an Excel file CPS88 in the network drive specified in the path directory. The data are read by column, and the upper left data cell is A2.

```
mat1.read(a2,s=sheet3) "\\\net1\dr_1\cps88.xls"
```

To read the same file by row, you should use the “t” option:

```
mat1.read(a2,t,s=sheet3) "\\\net1\dr 1\cps88.xls"
```

To write data from a matrix, use the `write` keyword, enter the desired options, then the name of the output file. For example:

```
mat1.write mydt.txt
```

writes the data in MAT1 into the ASCII file MYDT.TXT located in the default directory.

There are many more options for controlling reading and writing of data; [Chapter 4, “Basic Data Handling”, on page 55](#) of the *User’s Guide* provides extensive discussion. See also [read \(p. 291\)](#) and [write \(p. 383\)](#).

Matrix Operations versus Loop Operations

You can perform matrix operations using element operations and loops instead of the built-in functions and commands. For example, the inner product of two vectors may be computed by evaluating the vectors element-by-element:

```
scalar inprod1 = 0
for !i = 1 to @rows(vec1)
    inprod1 = inprod1 + vec1(!i)*vec2(!i)
next
```

however, this approach will generally be much slower than using the built-in function:

```
scalar inprod2 = @inner(vec1,vec2)
```

You should use the built-in matrix operators rather than loop operators whenever you can. The matrix operators are always much faster than the equivalent loop operations.

There will be cases when you cannot avoid using loop operations. For example, suppose you wish to subtract the column mean from each element of a matrix. Such a calculation might be useful in constructing a fixed effects regression estimator. First, consider a slow method involving only loops and element operations:

```
matrix(2000,10) x = @convert(mygrp1)
scalar xsum
for !i = 1 to @columns(x)
    xsum = 0
    for !j = 1 to @rows(x)
        xsum = xsum+x(!j,!i)
    next
    xsum = xsum/@rows(x)
    for !j = 1 to @rows(x)
```

```
x(!j,!i) = x(!j,!i)-xsum  
next  
next
```

The loops are used to compute a mean for each column of data in X, and then to subtract the value of the mean from each element of the column. A better and much faster method for subtracting column means uses the built-in operators:

```
matrix x = @convert(mygrp1)  
vector(@rows(x)) xmean  
for !i = 1 to @columns(x)  
    xmean = @mean(@columnextract(x,!i))  
    colplace(x,@columnextract(x,!i)-xmean,!i)  
next
```

This command extracts each column of X, computes the mean, and fills the vector XMEAN with the column mean. You then subtract the mean from the column and place the result back into the appropriate column of X. While you still need to loop over the control variable *i*, you avoid the need to loop over the elements of the columns.

Summary of Automatic Resizing of Matrix Objects

When you perform a matrix object assignment, EViews will resize, where possible, the destination object to accommodate the contents of the source matrix. This resizing will occur if the destination object type can be modified and sized appropriately and if the values of the destination may be assigned without ambiguity. You can, for example, assign a matrix to a vector and *vice versa*, you can assign a scalar to a matrix, but you cannot assign a matrix to a scalar since the EViews does not allow scalar resizing.

The following table summarizes the rules for resizing of matrix objects as a result of declarations of the form

```
object_type y = x
```

where *object_type* is an EViews object type, or as the result of an assignment statement for Y after an initial declaration, as in

```
object_type y  
y = x
```

Each row of the table corresponds to the specified type of the destination object, Y. Each column represents the type and size of the source object, X. Each cell of the table shows the type and size of object that results from the declaration or assignment.

Object type and size for source X		
Object type for Y	coef(p)	matrix(p, q)
coef(k)	coef(p)	<i>Invalid</i>
matrix(n, k)	matrix($p, 1$)	matrix(p, q)
rowvector(k)	rowvector(p)	<i>Invalid</i>
scalar	<i>Invalid</i>	<i>Invalid</i>
sym(k)	<i>Invalid</i>	sym(p) if $p = q$
vector(n)	vector(p)	<i>Invalid</i>

Object type and size for source X		
Object type for Y	rowvector(q)	scalar
coef(k)	coef(q)	coef(k)
matrix(n, k)	matrix($1, q$)	matrix(n, k)
rowvector(k)	rowvector(q)	rowvector(k)
scalar	<i>Invalid</i>	scalar
sym(k)	<i>Invalid</i>	<i>Invalid</i>
vector(n)	rowvector(q)	vector(n)

Object type and size for source X		
Object type for Y	sym(p)	vector(p)
coef(k)	<i>Invalid</i>	coef(p)
matrix(n, k)	matrix(p, p)	matrix($p, 1$)
rowvector(k)	<i>Invalid</i>	vector(p)
scalar	<i>Invalid</i>	<i>Invalid</i>
sym(k)	sym(p)	<i>Invalid</i>
vector(n)	<i>Invalid</i>	vector(p)

For example, consider the command

```
matrix(500, 4) y = x
```

where X is a coef of size 50. The object type is given by examining the table entry corresponding to row “matrix Y” ($n = 500, k = 4$), and column “coef X” ($p = 50$). The entry reads “matrix($p, 1$)”, so that the result Y is a 50×1 matrix.

Similarly, the command

```
vector(30) y = x
```

where X is a 10 element rowvector yields the 10 element rowvector Y. In essence, EViews first creates the 30 element rowvector Y, then resizes it to match the size of X, then finally assigns the values of X to the corresponding elements of Y.

Matrix Function and Command Summary

Utility Functions and Commands

- colplace** Places column vector into matrix ([p. 397](#)).
- @columnextract**... Extracts column from matrix ([p. 398](#)).
- @columns** Number of columns in matrix object ([p. 398](#)).
- @convert** Converts series or group to a vector or matrix after removing NAs ([p. 399](#)).
- @explode**..... Creates square matrix from a sym ([p. 402](#)).
- @filledmatrix**..... Creates matrix filled with scalar value ([p. 403](#)).
- @filledrowvector**.. Creates rowvector filled with scalar value ([p. 403](#)).
- @filledsym** Creates sym filled with scalar value ([p. 403](#)).
- @filledvector** Creates vector filled with scalar value ([p. 404](#)).
- @getmaindiagonal** Extracts main diagonal from matrix ([p. 404](#)).
- @identity** Creates identity matrix ([p. 404](#)).
- @implode** Creates sym from lower triangle of square matrix ([p. 405](#)).
- @makediagonal** ... Creates a square matrix with ones down a specified diagonal and zeros elsewhere ([p. 408](#)).
- matplace** Places matrix object in another matrix object ([p. 408](#)).
- mtos** Converts a matrix object to series or group ([p. 409](#)).
- @permute**..... Permutes the rows of the matrix ([p. 412](#)).
- @resample**..... Randomly draws from the rows of the matrix ([p. 412](#)).
- @rowextract**..... Extracts rowvector from matrix object ([p. 414](#)).
- rowplace** Places a rowvector in matrix object ([p. 414](#)).
- @rows** Returns the number of rows in matrix object ([p. 414](#)).
- stom** Converts series or group to vector or matrix after removing observations with NAs ([p. 415](#)).
- stomna**..... Converts series or group to vector or matrix without removing observations with NAs ([p. 416](#)).
- @subextract**..... Extracts submatrix from matrix object ([p. 417](#)).
- @transpose** Transposes matrix object ([p. 419](#)).

- @unitvector**Extracts column from an identity matrix ([p. 419](#)).
- @vec**Stacks columns of a matrix object ([p. 420](#)).
- @vech**Stacks the lower triangular portion of matrix by column ([p. 420](#)).

Matrix Algebra Functions

- @cholesky**Computes Cholesky factorization ([p. 397](#)).
- @cond**Calculates the condition number of a square matrix or sym ([p. 398](#)).
- @det**Calculate the determinant of a square matrix or sym ([p. 401](#)).
- @eigenvalues**Returns a vector containing the eigenvalues of a sym ([p. 402](#)).
- @eigenvectors**Returns a square matrix whose columns contain the eigenvectors of a matrix ([p. 402](#)).
- @inner**Computes the inner product of two vectors or series, or the inner product of a matrix object ([p. 405](#)).
- @inverse**Returns the inverse of a square matrix object or sym ([p. 406](#)).
- @issingular**Returns 1 if the square matrix or sym is singular, and 0 otherwise ([p. 406](#)).
- @kronecker**Computes the Kronecker product of two matrix objects ([p. 407](#)).
- @norm**Computes the norm of a matrix object or series ([p. 410](#)).
- @outer**Computes the outer product of two vectors or series, or the outer product of a matrix object ([p. 411](#)).
- @rank**Returns the rank of a matrix object ([p. 412](#)).
- @solvesystem**Solves system of linear equations, $Mx = v$, for x ([p. 415](#)).
- @svd**Performs singular value decomposition ([p. 418](#)).
- @trace**Computes the trace of a square matrix or sym ([p. 418](#)).

Matrix Descriptive Statistics Functions

- @cor**Computes correlation between two vectors, or between the columns of a matrix ([p. 400](#))
- @cov**Computes covariance between two vectors, or between the columns of a matrix ([p. 400](#)).

The remaining descriptive statistics functions that may be used with matrices and vectors are described in “[Descriptive Statistics](#)” on page 439.

Matrix Element Functions

EViews supports matrix element versions of the following categories of functions.

Category	Matrix Element Support
Basic Mathematical Functions (p. 437)	All, except for “@inv”
Additional and Special Functions (p. 441)	All
Trigonometric Functions (p. 444)	All
Statistical Distribution Functions (p. 444)	All

Chapter 5. Working with Tables

You can use EViews commands to generate custom tables of formatted output from your programs. A *table* is an object made up of rows and columns of cells, each of which can contain either a number or a string, as well as information used to control formatting for display or printing. The columns of a table can be set to different widths, and horizontal lines can be added to visually separate sections of the table.

After you have filled up all of the cells of a table, you can print the table object using the **print** command or by pressing the **Print** button on the table toolbar.

[Chapter 10, “Graphs, Tables, and Text Objects”, on page 243](#) of the *User’s Guide* describes table objects in detail.

Declaring a Table

To declare a table, indicate the number of rows and columns and provide a valid name. For example,

```
table(10,20) bestres
```

creates a table with 10 rows and 20 columns named BESTRES. You can change the size of a table by declaring it again. Redeclaring to a larger size does not destroy the contents of the table; any cells in the new table that existed in the original table will contain their previous values.

Tables are automatically resized when you attempt to fill a table cell outside the table’s current dimensions. This behavior is different from matrices which give an error when an out of range element is accessed.

Controlling Appearance

By default, each cell of a table will display approximately 10 characters. If a string is longer than the display width of the cell, part of the string may not be visible.

You can change the width of a column in a table using the **setcolwidth** command. Follow the **setcolwidth** keyword by the name of the table, the number of the column you wish to resize, and the approximate number of characters, all enclosed in parentheses. EViews measures units in terms of the width of a numeric character. Different characters have different widths, so the actual number of characters displayed may differ somewhat from the number which you specify. For example,

```
setcolwidth(bestres,2,12)
```

resizes the second column of table BESTRES to fit strings approximately 12 characters long.

You may also place horizontal separator (double) lines in the table using the `setline` command. Enter the `setline` keyword, followed by the name of the table, and the row number, all in parentheses. For example,

```
setline(bestres,8)
```

places a separator line in the eighth row of the table BESTRES.

Filling Cells

You can put a value into a cell of a table using an assignment statement. Each cell of the table can be assigned either a string or a numeric value.

Strings

To place a string value into a table cell, follow the table name by a row and column pair in parentheses, then an equal sign and a string expression:

```
bestres(1,6) = "convergence criterion"  
%strvar = "lm test"  
bestres(2,6) = %strvar  
bestres(2,6) = bestres(2,6) + " with 5 df"
```

Numbers

Numbers can be entered directly into cells, or they can be converted to strings before being placed in the table. If entered directly, the number will be displayed according to the numerical format set for that cell; if the format is changed, the number will be redisplayed according to the new format. If the number is first converted to a string, the number will be frozen in that form and cannot be reformatted.

By default, the number will be displayed with as many digits as will fit in the cell, with scientific notation used if necessary. If, instead, you use the `@str` function to first convert the number into a string, the string will contain the fewest decimal places required to represent the number. The following are some examples of cell assignments involving numbers:

```
tab1(3,4) = 15  
tab1(4,2) = "R-squared = " + @str(eq1.@r2)  
!ev = 10  
tab1(5,1) = "There are " + @str(!ev) + " events"
```

Cell Formatting

The `setcell` command is similar to cell assignment in that it allows you to set the contents of a cell. However, the `setcell` command also allows you to set formatting options for the cell. This is the only way to adjust formatting of a cell from the command line or within a program. The `setcell` command takes the following arguments:

- the name of the table,
- the row and the column of the cell,
- the number or string to be placed in the cell, and,
- optionally, a justification code or a numerical format code, or both.

The justification codes are:

- “c” for centered (default)
- “r” for right-justified
- “l” for left-justified

The numerical format code determines the format with which a number in a cell is displayed; cells containing strings will be unaffected. The format code can either be a positive integer, in which case it specifies the number of digits to be displayed after the decimal point, or a negative integer, in which case it specifies the total number of characters to be used to display the number. These two cases correspond to the **fixed decimal** and **fixed character** fields in the number format dialog.

Note that when using a negative format code, one character is always reserved at the start of a number to indicate its sign, and if the number contains a decimal point, that will also be counted as a character. The remaining characters will be used to display digits. If the number is too large or too small to display in the available space, EViews will attempt to use scientific notation. If there is insufficient space for scientific notation (six characters or less), the cell will contain asterisks to indicate an error.

Some examples of using `setcell`:

```
setcell(tabres,9,11,%label)
```

puts the contents of `%label` into row 9, column 11 of the table TABRES.

```
setcell(big_tabl,1,1,%info,"c")
```

inserts the contents of `%info` in BIG_TAB1(1,1), and displays the cell with centered justification.

```
setcell(tab1,5,5,!data)
```

puts the number `!data` into cell (5,5) of table TAB1, with default numerical formatting.

```
setcell(tab1,5,6,!data,4)
    puts the number !data into TAB1, with 4 digits to the right of the decimal
    point.

setcell(tab1,3,11,!data,"r",3)
    puts the number !data into TAB1, right-justified, with 3 digits to the right of
    the decimal point.

setcell(tab1,4,2,!data,-7)
    puts the number in !data into TAB1, with 7 characters used for display.
```

Table Example

Here we provide an extended example that shows how to construct a table using a program. The program constructs a table that displays the results of unit root tests applied to each series in a group.

We first write a subroutine that returns a vector of augmented Dickey-Fuller *t*-statistics for each series in the given group (subroutine construction is described in the next chapter).

```
subroutine local muroot(group g1, vector v1)
    ' get number of series in group
    !n = g1.@count
    ' declare vector to save results
    vector(!n) tstat
    ' do ADF test for each series in group
    for !i=1 to !n
        %str = g1.@seriesname(!i)
        series temp = {%str}
        equation eq_temp.ls d(temp) temp(-1) c
        tstat(!i) = eq_temp.@tstat(1)
    next
    ' copy results to global vector
    v1 = tstat
endsub
```

Note that we do not assign the results directly into the elements of the vector passed in as an argument. If we did this, we would need to know the number of series in the group before the function was called, so that we could correctly size the vector. Instead, we store the results in a temporary vector and copy this entire vector in a vector assignment at the end of the routine. This automatically resizes the vector to the required size.

The following program constructs a table to display the results of the unit root tests:

```
include c:\evdata\programs\muroot
load c:\evdata\macro
group grp1 ff tb3 tb10
vector tstat
call muroot(grp1,tstat)
' get number of series in group
scalar n = grp1.@count
' declare table and fill in headers
table(6,n+1) adf
setcell(adf,1,1,"ADF test (no lagged first differences with a
constant)")
setline(adf,2)
setcell(adf,3,1,"series")
setcell(adf,4,1,"t-stat")
setline(adf,5)
setcell(adf,6,1,"5% critical value is -2.86 (Davidson & MacKin-
non, Table 20.1)")
' fill in cells
for !i=1 to n
    setcell(adf,3,!i+1,grp1.@seriesname(!i))
    setcell(adf,4,!i+1,tstat(!i),3)
next
show adf
```

The first block (paragraph) of the program loads the workfile, creates a group of series, and carries out the unit root tests by calling the subroutine given above. The second block declares the table and fills in the cells that contain header information. The third block loops through the elements of the group to get the series name and the vector of *t*-statistics and places them in the appropriate cells of the table. Note that we display the *t*-statistics only up to the third decimal place.

	A	B	C	D	E
1	ADF test (no lagged first differences with a constant)				
2					
3	series	FF	TB3	TB10	
4	t-stat	-2.082	-2.109	-1.418	
5					
6	5% critical value is -2.86 (Davidson & MacKinnon, Table 20.1)				
7					
8					
9					

Table Summary

The following commands are documented in [Chapter 8, “Command Reference”](#).

setcell format and fill in a table cell ([p. 319](#)).

setcolwidth set width of a table column ([p. 321](#)).

setline place a horizontal line in table ([p. 326](#)).

Chapter 6. EViews Programming

EViews' programming features allow you to create and store commands in programs that automate repetitive tasks, or generate a record of your research project.

For example, you can write a program with commands that analyze the data from one industry, and then have the program perform the analysis for a number of other industries. You can also create a program containing the commands that take you from the creation of a workfile and reading of raw data, through the calculation of your final results, and construction of presentation graphs and tables.

If you have experience with computer programming, you will find most of the features of the EViews language to be quite familiar. The main novel feature of the EViews programming language is a macro substitution language which allows you to create object names by combining variables that contain portions of names.

Program Basics

Creating a Program

A program is not an EViews object within a workfile. It is simply a text file containing EViews commands. To create a new program, click **File/New/Program**. You will see a standard text editing window where you can type in the lines of the program. You may also open the program window by typing **program** in the command window, followed by an optional program name. For example

```
program firstprg
```

opens a program window named FIRSTPRG. Program names should follow standard EViews rules for file names.

A program consists of a one or more lines of text. Since each line of a program corresponds to a single EViews command, simply enter the text for each command and terminate the line by pressing the ENTER key.

If a program line is longer than the current program window, EViews will autowrap the text of the line. Autowrapping alters the appearance of the program line by displaying it on multiple lines, but does not change the contents of the line. While resizing the window will change the autowrap position, the text remains unchanged and is still contained in a single line.

If you wish to have greater control over the appearance of your lines, you can manually break long lines using the ENTER key, and then use the underscore continuation character

“_” as the last character on the line to join the multiple lines. For example, the three separate lines

```
equation eq1.ls _
y x c _
ar(1) ar(2)
```

are actually joined by the continuation character and are equivalent to the single line

```
equation eq1.ls y x c ar(1) ar(2)
```

Saving a Program

After you have created and edited your program, you will probably want to save it. Press the **Save** or **SaveAs** button on the program window toolbar. When saved, the program will have the extension .PRG.

Opening a Program

To load a program previously saved on disk, click on **File/Open/Program...**, navigate to the appropriate directory, change the **Files of Type** combo box to display Program .PRG files, and click on the desired name. Alternatively, from the command line, you can type `open` followed by the full program name, including the file extension .PRG. By default, EViews will look for the program in the default directory. If appropriate, include the full path to the file. The entire name should be enclosed in quotations if necessary. For example:

```
open mysp500.prg
open c:\mywork\evIEWS\myhouse.prg
```

opens MYSP500.PRG in the default directory, and MYHOUSE.PRG in the directory C:\MYWORK\EVIEWS.

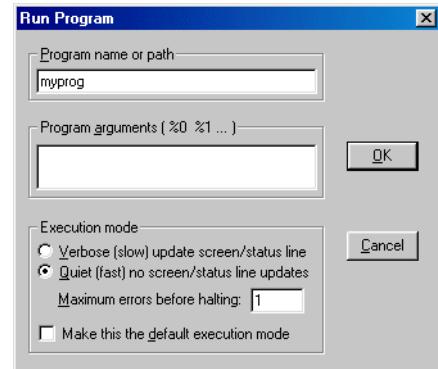
Executing a Program

When you enter a series of commands in the command window, line by line, we say that you are working in *interactive mode*. Alternatively, you can type all of the commands in a program and execute or run them collectively as a batch of commands. When you execute or run the commands from a program, we say that you are in *program (non-interactive) mode*.

There are several ways to execute a program. The easiest method is to execute your program by pushing the **Run** button on a program window. The Run dialog opens, where you can enter the program name and supply arguments. In addition, you can set the maximum number of errors before halting execution, and choose between quiet and verbose modes.

In verbose mode, EViews sends messages to the status line and continuously updates the work-file window as objects are created and deleted.

Quiet mode suppresses these updates, reducing the time spent writing to the screen.



By default, when EViews encounters an error, it will immediately terminate the program and display a message. If you enter a number into the **Maximum errors before halting** field, EViews will continue to execute the program until the maximum number of errors is reached. If there is a serious error so that it is impractical for EViews to continue, the program will halt even if the maximum number of errors is not reached.

You may also execute a program by entering the `run` command, followed by the name of the program file:

```
run mysp500
```

or

```
run c:\eviews\myprog
```

The use of the .PRG extension is not required since EViews will automatically append one. All of the run options described above may be set using command options. You may use the “v” option to run the program in verbose mode, and the “q” option to run the program in quiet mode. If you include a number as an option, EViews will use that number to indicate the maximum number of errors encountered before execution is halted. Any arguments to the program may be listed after the filename:

```
run(v, 500) mysp500
```

or

```
run(q) progarg arg1 arg2 arg3
```

See “[Handling Execution Errors](#)” on page 105 for discussion of execution errors and “[Multiple Program Files](#)” on page 106 for additional details on program execution.

You can also have EViews run a program automatically upon startup by choosing **File/Run** from the menu bar of the Windows Program Manager or **Start/Run** in Windows and then typing “eviews”, followed by the name of the program and the values of any arguments.

Stopping a Program

The F1 key halts execution of a program. It may take a few seconds for EViews to respond to the halt command.

Programs will also stop when they encounter a `stop` command, when they read the maximum number of errors, or when they finish processing a file that has been executed via a `run` statement.

If you include the `exit` keyword in your program, the EViews application will close.

Simple Programs

The simplest program is just a list of commands. Execution of the program is equivalent to typing the commands one by one into the command window.

While you could execute the commands by typing them in the command window, you could just as easily open a program window, type in the commands and click on the **Run** button. Entering commands in this way has the advantage that you can save the set of commands for later use, and execute the program repeatedly, making minor modifications each time.

Let us look at a simple example (the data series are provided in the database PROGDEMO in your EViews directory so that you can try out the program). Create a new program by typing

```
program myprog
```

in the command window. In the program window that opens for MYPROG, we are going to enter the commands to create a workfile, fetch a series from an EViews database named PROGDEMO, run a regression, compute residuals and a forecast, make a plot of the forecast, and save the results.

```
' housing analysis
workfile myhouse m 1968:3 1997:6
fetch progdemo::hsf
smpl 1968:5 1992:12
equation reg1.ls hsf c hsf(-1)
reg1.makeresid hsfres
smpl 1993:1 1997:6
reg1.forecast hsffit
```

```
freeze(hsfplot) hsffit.line  
save
```

The first line of the program is a comment, as denoted by the apostrophe “'”. In executing a program, EViews will ignore all text following the apostrophe until the end of the line.

HSF is total housing units started. We end up with a saved workfile named MYHOUSE containing the HSF series, an equation object REG1, a residual and forecast series, HSFRES and HSFFIT, and a graph HSFPLT of the forecasts.

You can run this program by clicking on **Run** and filling in the dialog box.

Now, suppose you wish to perform the same analysis, but for the S&P 500 stock price index (FSPCOM). Edit the program, changing MYHOUSE to MYSP500, and change all of the references of HSF to FSPCOM:

```
' s&p analysis  
workfile mysp500 m 1968:3 1997:6  
fetch progdemo::fspcom  
smpl 1968:5 1992:12  
equation reg1.ls fspcom c fspcom(-1)  
reg1.makeresid fspcomres  
smpl 1993:1 1997:6  
reg1.forecast fspcomfit  
freeze(fscomplot) fspcomfit.line  
save
```

Click on **Run** to execute the new analysis. Click on the **Save** button to save your program file as MYPROG.PRG in the EViews directory.

Since most of these two programs are identical, it seems like a lot of typing to make two separate programs. Below we will show you a way to handle these two forecasting problems with a single program.

Program Variables

While you can use programs just to edit, run, and re-run collections of EViews commands, the real power of the programming language comes from the use of *program variables* and *program control statements*.

Control Variables

Control variables are variables that you can use in place of numerical values in your EViews programs. Once a control variable is assigned a value, you can use it anywhere in a program that you would normally use a number.

The name of a control variable starts with an “!” mark. After the “!”, the name should be a legal EViews name of 15 characters or fewer. Examples of control variable names are:

```
!x  
!1  
!counter
```

You need not declare control variables before you refer to them, though you must assign them a value before use. Control variables are assigned in the usual way, with the control variable name on the left of an “=” sign and a numerical value or expression on the right. For example,

```
!x=7  
!12345=0  
!counter= 12  
!pi=3.14159
```

Once assigned a value, a control variable may appear in an expression. For example,

```
!counter = !counter + 1  
genr dnorm = 1/sqr(2*pi)*exp(-1/2*epsilon^2)  
scalar stdx = x/sqr(!varx)  
smpl 1950:1+!i 1960:4+!i
```

Control variables do not exist outside of your program and are automatically erased after a program finishes. As a result, control variables are not saved when you save the workfile. You can save the values of control variables by creating new EViews objects which contain the values of the control variable. For example,

```
scalar stdx = sqr(!varx)  
c(100) = !length  
sample years 1960+!z 1990
```

String Variables

A *string* is text enclosed in double quotes:

```
"gross domestic product"  
"3.14159"  
"ar(1) ar(2) ma(1) ma(2)"
```

A *string variable* is a variable whose value is a string of text. The name of a string variable starts with a “%” symbol. String variables are assigned by putting the string variable name on the left of an = sign and a string expression on the right. For example,

```
%value="value in millions of u.s. dollars"  
%armas="ar(1) ar(2) ma(1) ma(2)"  
%mysample=" 83:1 96:12"  
%dep=" hs"  
%pi=" 3.14159"
```

Once assigned a value, a string variable may appear in an expression. You can use strings to help you build up commands and variable names or as headings in tables. For example,

```
gnp.label %value  
smp1 %mysample  
equation eq1.ls %dep c %dep(-1)  
equation eq2.ls %dep c %dep(-1) %armas
```

EViews has a number of operators and functions for manipulating strings; a complete list is provided in [“Manipulating Strings” on page 92](#). Here is a quick example:

```
!repeat = 500  
%st1 = " draws from the normal"  
%st2 = "Cauchy "  
%st3 = @str(!repeat) +@left(%st1,16) + %st2 +"distribution"
```

In this example %ST3 is set to the value “500 draws from the Cauchy distribution”. Note the spaces before *draws* and after *Cauchy* in the double quotes.

String variables are like control variables in that they only exist during the time that your program is executing and are not stored in the workfile. You can save string data in a cell of a table, as described later in this chapter.

You can convert a string variable containing a number into a number by using the @val function. For example,

```
%str = ".05"  
!level = @val(%str)
```

creates a control variable !LEVEL = 0.05. If the first character of the string is not a numeric character (and is not a plus or a minus sign), @val returns the value “NA”. Any characters to the right of the first non-digit character are ignored. For example,

```
%date = "04/23/97"  
scalar day=@val(@right(%date,5))
```

```
scalar month=@val(%date)  
creates scalar objects DAY = 23 and MONTH = 4.
```

Manipulating Strings

Strings and string variables may be concatenated using the “+” operator. For example,

```
%st1 = "The name "  
%st2 = "X"  
%st3 = %st1 + "is " + %st2
```

would leave the string “The name is X” in the string variable %ST3.

In addition, EViews provides a number of functions that operate on strings:

- @left: returns a string containing the specified number of characters at the left end of a string. If there are fewer characters than the number specified, @left will return the entire string. Put the string and the number of characters in parentheses.

```
@left("I did not do it",5)
```

returns the string “I did”.

- @mid: returns the specified number of characters starting from the specified location in the string and going to the right.

```
@mid("I doubt I did it",9,5)
```

returns the string “I did”.

If you omit the number of characters from @mid, or if there are fewer remaining characters than the number specified, the function will return all of the characters to the right of the specified location.

```
@mid("I doubt I did it",9)
```

returns the string “I did it”.

- @right: returns a string containing the specified number of characters at the right end of a string. If there are fewer characters than the number specified, @right will return the entire string. Put the string and the number of characters in parentheses.

```
@right("I doubt I did it",8)
```

returns the string “I did it”.

- @str: returns a string representing the given number.

```
!x=15
```

```
@str(!x)
```

returns the string “15”.

- `@val`: returns a number (scalar) when applied to a string representing a number. If the string has any non-digits, they are considered to terminate the number. If the first character is not a digit, the function returns zero.

```
%date = "02/13/95"  
!month = @val(%date)  
!year = @val(@right(%date, 2))  
!day = @val(@mid(%date, 4, 2))
```

return the numerical values !MONTH = 2, !YEAR = 97, and !DAY = 13.

- `@otod`: (*Obs TO Date*) returns a string representing the date associated with the observation at the given offset from the start of the workfile range. The first observation of the workfile has an offset of 1.

```
create q 50:1 90:4  
!x = 16  
%date = @otod(!x)
```

returns the string %DATE = “1953:4”.

- `@dtoo`: (*Date TO Obs*) returns the scalar offset from the beginning of the workfile associated with the observation given by the date string. The string must be a valid EViews date.

```
create d 2/1/90 12/31/95  
%date = "1/1/93"  
!t = @dtoo(%date)
```

returns the value !T = 762.

Combining the string functions listed above and the “+” operator allows you to build up strings with different meanings. To repeat the example used before,

```
!repeat = 500  
%st1 = " draws from the normal"  
%st2 = "Cauchy "  
%st3 = @str(!repeat) +@left(%st1, 16) + %st2 +"distribution"
```

In this example %ST3 is set to “500 draws from the Cauchy distribution”.

Replacement Variables

EViews allows you to construct command lines using the contents of string and control variables. For example if the string variable %X contains the string “GDP”:

```
%x = "gdp"
```

then the program line

```
ls %x c %x(-1)
```

would be interpreted by EViews as

```
ls gdp c gdp(-1)
```

Changing the contents of %X to “ML” changes the interpretation of the line to

```
ls m1 c m1(-1)
```

In this context we refer to the string variable %X as a *replacement variable* because it is replaced in the command line by its contents.

Replacement variables may be combined with letter and digits or with other replacement variables to form longer words. When you do this, you should use curly braces “{“ and “}” to delimit the replacement variables. For example,

```
%type = "Low"  
%vname = " Income"  
series inc1  
inc1.label { %type } { %vname }  
%type = "High"  
series inc2  
inc2.label { %type } { %vname }
```

will label the series INC1 as “Low Income” and series INC2 as “High Income”. In these cases, the term “{ %type } { %vname }” is a replacement variable since the command is constructed by replacing the string variables with their contents.

Control variables may also be used as replacement variables. For example,

```
!i = 1  
series y{ !x } = nrnd  
!j = 0  
series y{ !j }{ !i } = nrnd
```

is the same as the commands

```
series y1 = nrnd  
series y01 = nrnd
```

and will create two series Y1 and Y01 that contain a set of (pseudo-)random draws from a standard normal distribution.

An important use of replacement variables is in constructing object names. For example,

```
!a = 3
%b = "2"
%c = "temp"
series x{!a}
matrix(2,2) x{%b}
vector(3) x_{%c}1
```

declares a series named X3, a 2×2 matrix named X2, and a vector named X_TEMP1.

Replacement variables give you great flexibility in naming objects in your programs. We do suggest, however, that you avoid using the same base names to refer to different objects. Consider the following example which shows the potential for confusion:

```
' possibly confusing commands (avoid)
!a=1
series x{!a}
!a=2
matrix x{!a}
```

In this small code snippet it is easy to see that X1 is the series and X2 is the matrix. But in a more complicated program, where the assignment of !A may be separated from the series declaration by many program lines, it may be difficult to tell at a glance what $x\{!a\}$ represents. A better approach would be to use different names for different kinds of variables:

```
!a=1
series ser{!a}
!a=2
matrix mat{!a}
```

EViews functions perform type checking so that you need to use braces when you call a function with a replacement variable. Functions that take string arguments only allow strings to be passed in and functions that take series arguments only accept series. To pass a series into a series function via a replacement variable, you must enclose the replacement variable in braces so that EViews knows to use the object referenced by the string, rather than the string itself. For example, when trying to find the number of valid observations in a series named INCOME, you can use the @obs function

```
@obs(income)
```

If you wish to use the string variable %VAR to refer to the INCOME series, the proper syntax is

```
%var = "income"
```

```
@obs ({%var})
```

The function @obs (%var) will return an error since the @obs function requires a series, not a string as an argument.

When you pass series names to subroutines via replacement variables, you also need to use braces. See “[Subroutine with arguments](#)” on page 108.

Program Arguments

Program arguments are special string variables that are passed to your program when you run the program. Arguments allow you to change the value of string variables every time you run the program. You may use them in any context where a string variable is appropriate. Any number of arguments may be included in a program; they will be named “%0”, “%1”, “%2”, and so on.

When you run a program that takes arguments, you will also supply the values for the arguments. If you use the **Run** button or **File/Run**, you will see a dialog box where you can type in the values of the arguments. If you use the `run` command, you should list the arguments consecutively after the name of the program.

For example, suppose we have a program named REGPROG:

```
equation eq1  
smpl 1980:3 1994:1  
eq1.ls %0 c %1 %1(-1) time
```

To run REGPROG from the command line with `%0 = "lgdp"` and `%1 = "m1"`, we enter

```
run regprog lgdp m1
```

This program performs a regression of the variable LGDP, on C, M1, M1(-1), and TIME, by executing the command

```
eq1.ls lgdp c m1 m1(-1) time
```

Alternatively, you can run this program by clicking on the **Run** button on the program window, or selecting **File/Run....** In the Run Program dialog box that appears, type the name of the program in the Program name or path field and enter the values of the arguments in the Program arguments field. For this example, type “regprog” for the name of the program, and “lgdp” and “m1” for the arguments.

Any arguments in your program that are not initialized in the `run` command or Run Program dialog are treated as blanks. For example, suppose you have a one-line program named REGRESS:

```
ls y c time %0 %1 %2 %3 %4 %5 %6 %7 %8
```

The command

```
run regress x x(-1) x(-2)
```

executes

```
ls y c time x x(-1) x(-2)
```

while the command

```
run regress
```

executes

```
ls y c time
```

In both cases, EViews ignores arguments that are not included in your `run` command.

As a last example, we repeat our simple forecasting program from above, but use arguments to simplify our work. Suppose you have the program, MYPROG:

```
workfile %0 m 1968:3 1997:6
fetch progdemo::%1
smpl 1968:5 1992:12
equation reg1.ls %1 c %1(-1)
reg1.makeresid {%1}res
smpl 93:1 1997:6
reg1.forecast {%1}fit
freeze({%1}plot) {%1}fit.line
save
```

The results of running the two example programs at the start of this chapter can be duplicated by running MYPROG with arguments:

```
run myprog myhouse hsf
```

and

```
run myprog mysp500 fspcom
```

Control of Execution

EViews provides you with several ways to control the execution of commands in your programs. Controlling execution in your program means that you can execute commands selectively or repeat commands under changing conditions. The methods for controlling execution will be familiar from other computer languages.

IF Statements

There are many situations where you want to execute commands only if some condition is satisfied. EViews uses IF/ENDIF statements to indicate the condition to be met and the commands to be executed.

An IF statement starts with the `if` keyword, followed by an expression for the condition, and then the word `then`. You may use AND/OR statements in the condition, using parentheses to group parts of the statement as necessary.

If the expression is true, all of the commands until the matching `endif` are executed. If the expression is false, all of these commands are skipped. The expression may also take a numerical value. In this case, zero is equivalent to false and any non-zero value is considered to be true. For example:

```
if !stand=1 or (!rescale=1 and !redo=1) then
    series gnpstd = gnp/sqr(gvar)
    series constd = cons/sqr(cvar)
endif
if !a>5 and !a<10 then
    smpl 1950:1 1970:1+!a
endif
if !scale then
    series newage = age!/scale
endif
```

Note that all indentation is done for program clarity and has no effect on the execution of the program lines.

An IF statement can have an ELSE clause containing commands to be executed if the condition is false. If the condition is true, all of the commands up to the keyword `else` will be executed. If the condition is false, all of the commands between `else` and `endif` will be executed. For example,

```
if !scale>0 then
    series newage = age!/scale
else
    series newage = age
endif
```

IF statements can also be applied to string variables and can be nested:

```
if %0=="ca" or %0=="in" then
```

```
    series stateid = 1
else
    if %0="ma" then
        series stateid=2
    else
        if %0="id" then
            series stateid=3
        endif
    endif
endif
```

All string comparisons are case-insensitive. String comparisons are lexicographic and follow the ASCII ordering rules. Strings are considered equal if they are the same length and every character matches. One string is less than another if the corresponding characters come earlier in the alphabet. A string is greater than another if the corresponding characters come later in the alphabet, or if there is no corresponding character. For example, suppose we assign the string values

```
%1 = "a"
%2 = "b2"
```

then the following inequalities are true:

```
%1<"abc" and "abc"<%2 and %2<"d"
"259"<%1 and %1<"aa" and "aa"<%2 and %2<"ba"
" b"<%1 and %1<"a 1" and "a 1"<"b110" and "b110"<%2
```

The string “A” is less than “ABC” since the first characters of the strings match; the remaining characters “BC” make the string greater. “A” is greater than any digit since digits come before letters in the ASCII ordering.

In order to test whether a string contains any characters, test whether it is equal to an empty string. For example, the statement

```
if %str<>"" then
```

will execute the following lines if the string variable %STR is not empty.

To test whether a scalar contains a missing value, compare with NA. For example,

```
if !a <> na then
```

will execute the following lines if !A is not a missing value.

Note that inequality comparisons with NA are always evaluated as false in IF statements.

For example,

```
if 3 > na then  
if 3 <= na then
```

are both false.

You may have noticed that all of the above examples of IF statements involved scalar or string variable comparisons. You should take care when using the IF statement with series or matrices to note that the boolean is defined on the *entire* object and will evaluate to false unless every element of the element-wise comparison is true. Thus, if X and Y are series, the IF statement

```
if x<>y then  
[some program lines]  
endif
```

evaluates to false if any element of X is not equal to the corresponding value of Y in the default sample. If X and Y are identically sized vectors or matrices, the comparison is over all of the elements X and Y. The behavior is described in greater detail in “[Comparison Operators \(=, >, > =, <, < =, < >\)](#)” on page 68.

The FOR Loop

The FOR loop allows you to repeat a set of commands for different values of a control or string variable. The FOR loop begins with a `for` statement and ends with a `next` statement. Any number of commands may appear between these two statements.

The syntax of the FOR statement differs depending upon whether it uses control variables or string variables.

FOR Loops with Control Variables or Scalars

To repeat statements for different values of a control variable, the FOR statement involves setting a control variable equal to an initial value, followed by the word `to`, and then an ending value. After the ending value you may include the word `step` followed by a number indicating by how much to change the control variable each time the loop is executed. If you don’t include `step`, the step is assumed to be 1. For example,

```
for !j=1 to 10  
series decile{!j} = (income<level{!j})  
next
```

In this example, `STEP = 1` and the variable J is twice used as a replacement variable, first for the ten series declarations DECILE1 through DECILE10 and for the ten variables LEVEL1 through LEVEL10.

```

for !j=10 to 1 step -1
    series rescale{!j}=original/!j
next

```

In this example, the step is -1, and J is used as a replacement variable to name the ten constructed series RESCALE10 through RESCALE1 and as a scalar in dividing the series ORIGINAL.

The FOR loop is executed first for the initial value, unless that value is already beyond the terminal value. After it is executed for the initial value, the control variable is incremented by step and EViews compares the variable to the limit. If the limit is passed, execution stops.

One important use of FOR loops with control variables is to change the sample. If you add a control variable to a date in a smpl command, you will get a new date as many observations forward as the current value of the control variable. Here is a FOR loop that gradually increases the size of the sample and estimates a rolling regression:

```

for !horizon=10 to 72
    smpl 1970:1 1970:1+!horizon
    equation eq{!horizon}.ls sales c orders
next

```

One other important case where you will use loops with control variables is in accessing elements of a series or matrix objects. For example,

```

!rows = @rows(vec1)
vector cumsum1 = vec1
for !i=2 to !rows
    cumsum1(!i) = cumsum1(!i-1) + vec1(!i)
next

```

computes the cumulative sum of the elements in the vector VEC1 and saves it in the vector CUMSUM1.

To access an individual element of a series, you will need to use the @elem function and @otod to get the desired element

```

for !i=2 to !rows
    cumsum1(!i) = @elem(ser1, @otod(!i))
next

```

The @otod function returns the date associated with the observation index (counting from the beginning of the workfile), and the @elem function extract the series element associated with a given date.

You can nest FOR loops to contain loops within loops. The entire inner FOR loop is executed for each successive value of the outer FOR loop. For example,

```
matrix(25,10) xx
for !i=1 to 25
    for !j=1 to 10
        xx(!i,!j)=(!i-1)*10+!j
    next
next
```

You should avoid changing the control variable within a FOR loop. For example, consider

```
' potentially confusing loop (avoid doing this)
for !i=1 to 25
    vector a!i
    !i=!i+10
next
```

Here, both the FOR assignment and the assignment statement within the loop change the value of the control variable I. Loops of this type are difficult to follow and may produce unintended results. If you find a need to change a control variable inside the loop, consider using a WHILE loop as explained below.

You may execute FOR loops with scalars instead of control variables. However, you must first declare the scalar, and you may not use the scalar as a replacement variable. For example,

```
scalar i
scalar sum = 0
vector (10) x
for i=1 to 10
    x(i) = i
    sum = sum + i
next
```

In this example, the scalars I and SUM remain in the workfile after the program has finished running, unless they are explicitly deleted.

FOR Loops with String Variables

When you wish to repeat statements for different values of a string variable, you can use the FOR loop to let a string variable range over a list of string values. Give the name of the string variable followed by the list of values. For example,

```

for %y gdp gnp ndp nnp
    equation {%y}trend.ls %y c {%y}(-1) time
next

```

executes the commands

```

equation gdptrend.ls gdp c gdp(-1) time
equation gnptrend.ls gnp c gnp(-1) time
equation ndptrend.ls ndp c ndp(-1) time
equation nnptrend.ls nnp c nnp(-1) time

```

You can put multiple string variables in the same FOR statement—EViews will process the strings in sets. For example,

```

for %1 %2 %3 1955:1 1960:4 early 1970:2 1980:3 mid 1975:4
    1995:1 late
    smpl %1 %2
    equation {%3}eq.ls sales c orders
next

```

In this case, the elements of the list are taken in groups of three. The loop is executed three times for the different sample pairs and equation names:

```

smpl 1955:1 1960:4
equation earlyeq.ls sales c orders
smpl 1970:2 1980:3
equation mideq.ls sales c orders
smpl 1975:4 1995:1
equation lateeq.ls sales c orders

```

Note the difference between this construction and nested FOR loops. Here, all string variables are advanced at the same time, whereas with nested loops, the inner variable is advanced over all choices, while the outer variable is held constant. For example,

```

!eqno = 1
for %1 1955:1 1960:4
    for %2 1970:2 1980:3 1975:4
        smpl %1 %2
        'form equation name as eq1 through eq6
        equation eq{!eqno}.ls sales c orders
        !eqno=!eqno+1
next

```

```
next
```

Here, the equations are estimated over the samples 1955:1–1970:2 for EQ1, 1955:1–1980:3 for EQ2, 1955:1–1975:4 for EQ3, 1960:4–1970:2 for EQ4, 1960:4–1980:3 for EQ5, and 1960:4–1975:4 for EQ6.

The WHILE Loop

In some cases, we wish to repeat a series of commands several times, but only while one or more conditions are satisfied. Like the FOR loop, the WHILE loop allows you to repeat commands, but the WHILE loop provides greater flexibility in specifying the required conditions.

The WHILE loop begins with a `while` statement and ends with a `wend` statement. Any number of commands may appear between the two statements. WHILE loops can be nested.

The WHILE statement consists of the `while` keyword followed by an expression involving a control variable. The expression should have a logical (true/false) value or a numerical value. In the latter case, zero is considered false and any non-zero value is considered true.

If the expression is true, the subsequent statements, up to the matching `wend`, will be executed, and then the procedure is repeated. If the condition is false, EViews will skip the following commands and continue on with the rest of the program following the `wend` statement. For example,

```
!val = 1
!a = 1
while !val<10000 and !a<10
    smpl 1950:1 1970:1+!a
    series inc{!val} = income//!val
    !val = !val*10
    !a = !a+1
wend
```

There are four parts to this WHILE loop. The first part is the initialization of the control variables used in the test condition. The second part is the WHILE statement which includes the test. The third part is the statements updating the control variables. Finally the end of the loop is marked by the word `wend`.

Unlike a FOR statement, the WHILE statement does not update the control variable used in the test condition. You need to explicitly include a statement inside the loop that changes the control variable, or your loop will never terminate. Use the F1 key to break out of a program which is in an infinite loop.

In the example above of a FOR loop that changed the control variable, a WHILE loop provides a much clearer program:

```
!i = 1
while !i<=25
    vector a{!i}
    !i = !i + 11
wend
```

Handling Execution Errors

By default, EViews will stop executing after encountering any errors, but you can instruct the program to continue running even if errors are encountered (see “[Executing a Program](#)” on page 86). In the latter case, you may wish to perform different tasks when errors are encountered. For example, you may wish to skip a set of lines which accumulate estimation results when the estimation procedure generated errors.

To test for and handle execution errors, you should use the @errorcount function to return the number of errors encountered while executing your program:

```
!errs = @errorcount
```

The information about the number of errors may be used by standard program statements to control the behavior of the program.

For example, to test whether the estimation of a equation generated an error, you should compare the number of errors before and after the command:

```
!old_count = @errorcount
equation eq1.ls y x c
!new_count = @errorcount
if !new_count > !old_count then
    [various commands]
endif
```

Here, we perform a set of commands only if the estimation of equation EQ1 incremented the error count.

Other Tools

Occasionally, you will wish to stop a program or break out of a loop based on some conditions. To stop a program executing in EViews, use the stop command. For example, suppose you write a program that requires the series SER1 to have nonnegative values. The following commands check whether the series is nonnegative and halt the program if SER1 contains any negative value:

```
series test = (ser1<0)
if @sum(test) <> 0 then
    stop
endif
```

Note that if SER1 contains missing values, the corresponding value of TEST will also be missing. Since the @sum function ignores missing values, the program does not halt for SER1 that has missing values, as long as there is no negative value.

Sometimes, you do not wish to stop the entire program when a condition is satisfied; you just wish to exit the current loop. The `exitloop` command will exit the current `for` or `while` statement and continue running the program.

For example, suppose you computed a sequence of LR test statistics LR11, LR10, LR9, ..., LR1, say to test the lag length of a VAR. The following program sequentially carries out the LR test starting from LR11 and tells you the statistic that is first rejected at the 5% level:

```
!df = 9
for !lag = 11 to 1 step -1
    !pval = 1 - @cchisq(lr{!lag}, !df)
    if !pval<=.05 then
        exitloop
    endif
next
scalar lag=!lag
```

Note that the scalar LAG has the value 0 if none of the test statistics are rejected.

Multiple Program Files

When working with long programs, you may wish to organize your code using multiple files. For example, suppose you have a program file named `POWERS.PRG` which contains a set of program lines that you wish to use.

While you may be tempted to string files together using the `run` command, we caution you that EViews will stop after executing the commands in the referenced file. Thus, a program containing the lines

```
run powers.prg
series x = nrnd
```

will only execute the commands in the file `POWERS`, and will stop before generating the series X. This behavior is probably not what you intended.

You should instead use the `include` keyword to include the contents of a program file in another program file. For example, you can place the line

```
include powers
```

at the top of any other program that needs to use the commands in `POWERS`. `include` also accepts a full path to the program file and you may have more than one `include` statement in a program. For example the lines,

```
include c:\programs\powers.prg
include durbin_h
{more lines}
```

will first execute all of the commands in `C:\PROGRAMS\POWERS.PRG`, then will execute the commands in `DURBIN_H.PRG`, and then will execute the remaining lines in the program file.

Subroutines provide a more general, alternative method of reusing commands and using arguments.

Subroutines

A *subroutine* is a collection of commands that allows you to perform a given task repeatedly, with minor variations, without actually duplicating the commands. You can also use subroutines from one program to perform the same task in other programs.

Defining Subroutines

A subroutine starts with the keyword `subroutine` followed by the name of the routine and any arguments, and ends with the keyword `endsub`. Any number of commands can appear in between. The simplest type of subroutine has the following form:

```
subroutine z_square
series x = z^2
endsub
```

where the keyword `subroutine` is followed only by the name of the routine. This subroutine has no arguments so that it will behave identically every time it is used. It forms the square of the existing series Z and stores it in the new series X.

You can use the `return` command to force EViews to exit from the subroutine at any time. A common use of `return` is to exit from the subroutine if an unanticipated error is detected. The following program exits the subroutine if Durbin's *h* statistic for testing serial correlation with a lagged dependent variable cannot be computed (for details, see Greene, 1997, p.596, or Davidson and MacKinnon, 1993, p. 360):

```
subroutine durbin_h
```

```
equation eqn.ls cs c cs(-1) inc
scalar test=1-eqn.@regobs*eqn.@cov(2,2)
' an error is indicated by test being nonpositive
' exit on error
if test<=0 then
    return
endif
' compute h statistic if test positive
scalar h=(1-eqn.@dw/2)*sqr(eqn.@regobs/test)
endsub
```

Subroutine with arguments

The subroutines so far have been written to work with a specific set of variables. More generally, subroutines can take arguments. If you are familiar with another programming language, you probably already know how arguments allow you to change the behavior of the group of commands each time the subroutine is used. Even if you haven't encountered subroutines, you are probably familiar with similar concepts from mathematics. You can define a function, say

$$f(x) = x^2 \quad (6.1)$$

where f depends upon the argument x . The argument x is merely a place holder—it's there to define the function and it does not really stand for anything. Then if you want to evaluate the function at a particular numerical value, say 0.7839, you can write $f(0.7839)$. If you want to evaluate the function at a different value, say 0.50123, you merely write $f(0.50123)$. By defining the function, you save yourself from writing out the whole expression every time you wish to evaluate it for a different value.

To define a subroutine with arguments, you start with `subroutine`, followed by the subroutine name, a left parenthesis, the arguments separated by commas, and finally a right parenthesis. Each argument is specified by listing a type of EViews object, followed by the name of the argument. Control variables may be passed by the scalar type and string variables by the string type. For example,

```
subroutine power(series v, series y, scalar p)
    v = y^p
endsub
```

This subroutine generalizes the example subroutine Z_SQUARE. Calling POWER will fill the series given by the argument V with the power P of the series specified by the argument Y. So if you set V equal to X, Y equal to Z, and P equal to 2, you will get the equiva-

lent of the subroutine Z_SQUARE above. See the discussion below on how to call subroutines.

Subroutine Placement

Your subroutine definitions should be placed, in any order, at the beginning of your program. The subroutines will not be executed until they are executed by the program using a `call` statement. For example,

```
subroutine z_square
    series x=z^2
endsub
' start of program execution
load mywork
fetch z
call z_square
```

Execution of this program begins with the `load` statement; the subroutine definition is skipped and is executed only at the last line when it is “called.”

The subroutine definitions must not overlap—after the `subroutine` keyword, there should be an `endsub` before the next `subroutine` declaration. Subroutines may call each other, or even call themselves.

Alternatively, you may wish to place frequently used subroutines in a separate program file and use an `include` statement to insert them at the beginning of your program. If, for example, you put the subroutine lines in the file POWERS.PRG then you may put the line

```
include powers
```

at the top of any other program that needs to call Z_SQUARE or POWER. You can use the subroutines in these programs as though they were built-in parts of the EViews programming language.

Calling Subroutines

Once a subroutine is defined in your program, you may execute the commands in the subroutine by using the `call` keyword. `call` should be followed by the name of the subroutine, and a list of any argument values you wish to use, enclosed in parentheses and separated by commas. If the subroutine takes arguments, they must all be provided in the same order as in the declaration statement. Here is an example program file that calls subroutines:

```
include powers
load mywork
```

```
fetch z gdp
series x
series gdp2
series gdp3
call z_square
call power(gdp2,gdp,2)
call power(gdp3,gdp,3)
```

The first call fills the series X with the value of Z squared. The second call creates the series GDP2 which is GDP squared. The last call creates the series GDP3 as the cube of GDP.

When the subroutine argument is a scalar, the subroutine may be called with a scalar object, a control variable, a simple number (such as “10” or “15.3”), a matrix element (such as “mat1(1,2)”) or a scalar expression (such as “!y + 25”). Subroutines that take matrix and vector arguments can be called with a matrix name, and if not modified by the subroutine, may also take a matrix expression. All other arguments must be passed to the subroutine with a simple object (or string) name referring to a single object of the correct type.

Global and Local Variables

Subroutines work with variables and objects that are either *global* or *local*.

Global variables refer either to objects which exist in the workfile when the subroutine is called, or to the objects that are created in the workfile by a subroutine. Global variables remain in the workfile when the subroutine finishes.

A *local variable* is one that has meaning only within the subroutine. Local variables are deleted from the workfile once a subroutine finishes. The program that calls the subroutine will not know anything about a local variable since the local variable will disappear once the subroutine finishes and returns to the original program.

Global Subroutines

By default, subroutines in EViews are *global*. Any global subroutine may refer to any global object that exists in the workfile at the time the subroutine is called. Thus, if Z is a series in the workfile, the subroutine may refer to and, if desired, alter the series Z. Similarly, if Y is a global matrix that has been created by another subroutine, the current subroutine may use the matrix Y.

The rules for variables in global subroutines are:

- Newly created objects are global and will be included in the workfile when the subroutine finishes.

- Global objects may be used and updated directly from within the subroutine. If, however, a global object has the same name as an argument in a subroutine, the variable name will refer to the argument and not to the global variable.
- The global objects corresponding to arguments may be used and updated by referring to the arguments.

Here is a simple program that calls a global subroutine:

```
subroutine z_square
    series x = z^2
endsub
load mywork
fetch z
call z_square
```

Z_SQUARE is a global subroutine which has access to the global series Z. The new global series X contains the square of the series Z. Both X and Z remain in the workfile when Z_SQUARE is finished.

If one of the arguments of the subroutine has the same name as a global variable, the argument name takes precedence. Any reference to the name in the subroutine will refer to the argument, not to the global variable. For example,

```
subroutine sqseries(series z,string %name)
    series {%name} = z^2
endsub
load mywork
fetch z
fetch y
call sqseries(y,"y2")
```

In this example, there is a series Z in the original workfile and Z is also an argument of the subroutine. Calling SQSERIES with the argument set to Y tells EViews to use the argument rather than the global Z. Upon completion of the routine, a new series Y2 will contain the square of the series Y, not the square of the series Z.

Global subroutines may call global subroutines. You should make certain to pass along any required arguments when you call a subroutine from within a subroutine. For example,

```
subroutine wgtols(series y, series wt)
    equation eq1
    call ols(eq1, y)
    equation eq2
```

```
series temp = y/sqr(wt)
call ols(eq2,temp)
delete temp
endsub
subroutine ols(equation eq, series y)
eq.ls y c y(-1) y(-1)^2 y(-1)^3
endsub
```

can be run by the program:

```
load mywork
fetch cpi
fetch cs
call wgtols(cs,cpi)
```

In this example, the subroutine WGTOLS explicitly passes along the arguments for EQ and Y to the subroutine OLS; otherwise those arguments would not be recognized by OLS. If EQ and Y were not passed, OLS would try to find a global series named Y and a global equation named EQ, instead of using EQ1 and CS or EQ2 and TEMP.

You cannot use a subroutine to change the object type of a global variable. Suppose that we wish to declare new matrices X and Y by using a subroutine NEWXY. In this example, the declaration of the matrix Y works, but the declaration of matrix X generates an error because a series named X already exists:

```
subroutine newxy
matrix(2,2) x = 0
matrix(2,2) y = 0
endsub
load mywork
series x
call newxy
```

EViews will return an error indicating that the global series X already exists and is of a different type than a matrix.

Local Subroutines

All objects created by a global subroutine will be global and will remain in the workfile upon exit from the subroutine. If you include the word `local` in the definition of the subroutine, you create a local subroutine. All objects created by a local subroutine will be local and will be removed from the workfile upon exit from the subroutine. Local subroutines

are most useful when you wish to write a subroutine which creates many temporary objects that you do not want to keep.

The rules for variables in local subroutines are:

- You may not use or update global objects directly from within the subroutine.
- The global objects corresponding to arguments may be used and updated by referring to the arguments.
- All other objects in the subroutine are local and will vanish when the subroutine finishes.

If you want to save results from a local subroutine, you have to explicitly include them in the arguments. For example,

```
subroutine local ols(series y, series res, scalar ssr)
    equation temp_eq.ls y c y(-1) y(-1)^2 y(-1)^3
    temp_eq.makeresid res
    ssr = temp_eq.@ssr
endsub
```

This local subroutine takes the series Y as input and returns the series RES and scalar SSR as output. The equation object TEMP_EQ is local to the subroutine and will vanish when the subroutine finishes.

Here is an example program that calls this local subroutine:

```
load mywork
fetch hsf
equation eq1.ls hsf c hsf(-1)
eq1.makeresid rres
scalar rssr = eq1.@ssr
series ures
scalar ussr
call ols(hsf, ures, ussr)
```

Note how we first declare the series URES and scalar USSR before calling the local subroutine. These objects are global since they are declared outside the local subroutine. When we call the local subroutine by passing these global objects as arguments, the subroutine will update these global variables.

There is one exception to the general inaccessibility of global variables in local subroutines. When a global group is passed as an argument to a local subroutine, any series in the group is accessible to the local routine.

Local subroutines can call global subroutines and vice versa. The global subroutine will only have access to the global variables, and the local subroutine will only have access to the local variables, unless information is passed between the routines via arguments. For example, the subroutine

```
subroutine newols(series y, series res)
    include ols
    equation eq1.ls y c y(-1)
    eq1.makeresid res
    scalar rssr=eq1.@ssr
    series ures
    scalar ussr
    call ols(y, ures, ussr)
endsub
```

and the program

```
load mywork
fetch hsf
series rres
call newols(hsf, rres)
```

produce equivalent results. Note that the subroutine NEWOLS still does not have access to any of the temporary variables in the local routine OLS, even though OLS is called from within NEWOLS.

Programming Summary

Support Commands

- open** opens a program file from disk ([p. 275](#)).
- output** redirects print output to objects or files ([p. 279](#)).
- poff** turns off automatic printing in programs ([p. 427](#)).
- pon** turns on automatic printing in programs ([p. 427](#)).
- program** declares a program.
- run** runs a program ([p. 305](#)).
- statusline** sends message to the status line ([p. 428](#)).

Program Statements

- call** calls a subroutine within a program ([p. 421](#)).
- else** denotes start of alternative clause for IF ([p. 422](#)).
- endif** marks end of conditional commands ([p. 422](#)).

endsubmarks end of subroutine definition ([p. 422](#)).
exitloopexits from current loop ([p. 423](#)).
forstart of FOR execution loop ([p. 424](#)).
ifconditional execution statement ([p. 424](#)).
includeinclude subroutine in programs ([p. 425](#)).
nextend of FOR loop ([p. 425](#)).
returnexit subroutine ([p. 428](#)).
step(optional) step size of a FOR loop ([p. 429](#)).
stophalts execution of program ([p. 430](#)).
subroutinedeclares subroutine ([p. 430](#)).
thenpart of IF statement ([p. 431](#)).
toupper limit of FOR loop ([p. 432](#)).
wendend of WHILE loop ([p. 433](#)).
whilestart of WHILE loop ([p. 433](#)).

Support Functions

@datestring containing the current date ([p. 421](#)).
@errorcountnumber of errors encountered ([p. 423](#)).
@evpathstring containing the directory path for the EViews executable ([p. 423](#)).
@isobjectchecks for existence of object ([p. 425](#)).
@obsrangereturns number of observations in the current active workfile range (0 if no workfile in memory).
@obssmplreturns number of observations in the current active workfile sample (0 if no workfile in memory).
@tempopathstring containing the directory path for EViews temporary files ([p. 430](#)).
@timestring containing the current time ([p. 431](#)).
@toccalculates elapsed time (since timer reset) in seconds ([p. 432](#)).

String Functions

@dtooconverts date string to observation number ([p. 401](#)).
@leftextracts left portion of string ([p. 407](#)).
@midextracts middle portion of string ([p. 409](#)).
@todconverts observation number to date string ([p. 411](#)).
@rightextracts right portion of string ([p. 413](#)).
@strreturns a string representing a number ([p. 416](#)).
@strlenreturns the length of a string ([p. 417](#)).

@val..... returns the number represented by a string ([p. 419](#)).

Chapter 7. Sample Programs

In this chapter, we provide extended examples showing the use of the EViews programming language to perform the following tasks:

- Computing descriptive statistics by year.
- Rolling window unit-root (ADF) tests.
- Calculating cumulative sums.
- Performing time series operations on a sample of observations.
- Creating dummy variables with a loop.
- Extracting test statistics in a loop.
- Between group estimation for pooled data.
- Hausman test for fixed versus random effects.
- Reformatted regression output table.

In addition, there are a number of example files for creating and estimating user defined likelihood functions. These example files are as described in “[Additional Examples](#)” on [page 491](#).

We have included program and data files for each example in your EViews examples directory. You may open each file by clicking on **File/Open/Program...** and selecting the appropriate file. To run the program, either click on **Run** in the program window, or enter the command `run`, followed by the program name, in the command window.

Note that there may be additional programs in your EViews directory that were added after the manual went to press and therefore are not listed here. If you find that the examples here do not cover an issue of interest, you may wish to browse the contents of your EViews examples directory.

Descriptive Statistics by Year

(`descr1.prg`)

Suppose that you wish to compute descriptive statistics (mean, median, etc.) for each year of your monthly data in the series IP, URATE, M1, and TB10. While EViews does not provide a built-in procedure to perform this calculation, you can use the `statsby` view of a series to compute the relevant statistics in two steps: first create a year identifier series, and second compute the statistics for each value of the identifier.

```
' load the basics workfile
```

```
%evworkfile = @evpath + "\example files\basics"
load "{$evworkfile}"
smpl 1990:1 @last

' create a series containing the year identifier
series year = @year

' compute statistics for each year and
' freeze the output from each of the tables
for %var ip urate m1 tb10
    %name = "tab" + %var
    freeze(%name) { %var }.statby(min,max,mean,med) year
next
```

The results are saved in tables named TABIP, TABURATE, TABM1, and TABTB10. For example, TABIP contains:

Descriptive Statistics for IP
Categorized by values of YEAR
Date: 08/24/00 Time: 10:40
Sample: 1990:01 1995:04
Included observations: 64

YEAR	Mean	Median	Max	Min.	Std. Dev.	Obs.
1990	106.0696	106.3725	106.8150	104.5380	0.736456	12
1991	104.2815	104.5610	105.7480	102.0980	1.230377	12
1992	107.6821	107.7565	110.4470	104.8810	1.605744	12
1993	112.0888	111.7070	114.7110	110.6500	1.168270	12
1994	118.0779	118.1270	121.7000	114.7440	1.993301	12
1995	121.6750	121.8000	122.0000	121.1000	0.427201	4
All	110.3922	108.1350	122.0000	102.0980	5.805422	64

(desc2.prg)

The example above saves the values in a table. You may instead wish to place the annual values in a vector or matrix so that they can be used in other calculations. The following program creates a vector to hold the median values for each year, and loops through the calculation of the median for each year:

```
' load the basics workfile
%evworkfile = @evpath + "\example files\basics"
load "{$evworkfile}"
smpl 1990:1 @last

' create series with the year and find the maximum
series year = @year
```

```

!lastyear = @max(year)

' loop over the series names
for %var ip urate m1 tb10

    ' create matrix to hold the results
    !numrows = (!lastyear - 1990 + 1) + 1
    matrix(!numrows, 7) mat%var

    ' loop over each year and compute values
    for !i = 1990 to !lastyear
        !row = !i - 1990 + 1
        smpl if (year = !i)
        mat%var(!row,1) = !i
        mat%var(!row,2) = @mean({%var})
        mat%var(!row,3) = @med({%var})
        mat%var(!row,4) = @min({%var})
        mat%var(!row,5) = @max({%var})
        mat%var(!row,6) = @std({%var})
        mat%var(!row,7) = @obs({%var})
    next

    ' compute the total values
    mat%var(!numrows,1) = !i
    mat%var(!numrows,2) = @mean({%var})
    mat%var(!numrows,3) = @med({%var})
    mat%var(!numrows,4) = @min({%var})
    mat%var(!numrows,5) = @max({%var})
    mat%var(!numrows,6) = @std({%var})
    mat%var(!numrows,7) = @obs({%var})

next

```

Rolling ADF Test

(rollreg.prg)

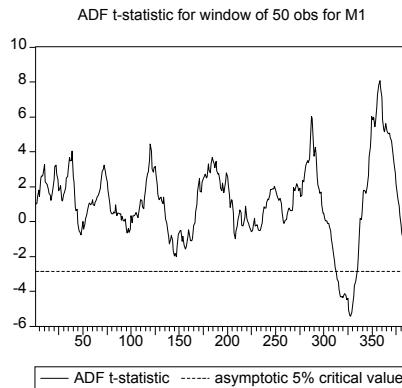
The following program runs a set of rolling ADF tests on the series M1 using a moving sample of fixed size. The basic techniques for working with rolling samples may be used in a variety of settings. The program stores and displays the *t*-statistics together with the asymptotic critical value (C2):

```
' load the data
%evworkfile = @evpath + "\example files\basics"
load "%evworkfile}"
smpl @all

' find size of workfile
series _temp = 1
!length = @obs(_temp)
delete _temp
' set fixed sample size
!ssize = 50

' initialize matrix to store results
matrix(!length-!ssize+1,2) adftstat
' run test regression for each subsample and store
' each ar(1) coefficient test includes a constant
' with no lagged first difference
for !i = 1 to !length-!ssize+1
    smpl @first+!i-1 @first+!i+!ssize-2
    equation temp.ls d(m1) c m1(-1)
    adftstat(!i,1) = temp.@tstat(2)
    ' 5% critical value from Davidson and MacKinnon,
    ' Table 20.1, page 708
    adftstat(!i,2) = -2.86
next

freeze(graph1) adftstat.line
graph1.name(1) ADF t-statistic
graph1.name(2) asymptotic 5% critical value
graph1.addtext(t) ADF t-statistic for window of 50 obs for M1
show graph1
```



Calculating Cumulative Sums

(cum_sum.prg)

EViews does not have a built-in function for calculating cumulative sums. You can easily calculate such sums with a few lines of commands.

Say we have a series X for which we would like to calculate the cumulative sum. If the series does not contain NAs, you can use:

```
smpl @first @first
series sum0 = x
smpl @first+1 @last
sum0 = sum0(-1) + x
```

If the series does contain NAs, you have two options, depending on how you would like the NAs to be treated. If you would like to continue accumulating the sum, ignoring the NA observations, you can use:

```
smpl @first @first
series sum1 = @nan(x, 0)
smpl @first+1 @last
sum1 = sum1(-1) + @nan(x, 0)
smpl @all if x = na
sum1 = na
```

If you would like to reset the sum to zero whenever an NA appears in the series you can use:

```
smpl @all
series sum2 = x
smpl if sum2(-1)<>na and x<>na
```

```
sum2 = sum2(-1) + x
```

The following sample output shows the differences between the three methods:

obs	X	SUM0	SUM1	SUM2
1	4.000000	4.000000	4.000000	4.000000
2	2.000000	6.000000	6.000000	6.000000
3	NA	NA	NA	NA
4	4.000000	NA	10.000000	4.000000
5	1.000000	NA	11.000000	5.000000
6	3.000000	NA	14.000000	8.000000
7	2.000000	NA	16.000000	10.000000
8	NA	NA	NA	NA
9	7.000000	NA	23.000000	7.000000
10	5.000000	NA	28.000000	12.000000

Time Series Operations on a Sample

(subset.prg)

Many operations in EViews can be performed on a subset of the workfile observations using the `smp1` command. To estimate an equation using only observations which have a positive value of X, you could simply use:

```
smp1 if x>=0  
eq1.ls y c x
```

Time series operations often rely on observations being adjacent, so that this method may not work as expected. In this example we demonstrate a general technique for working with a subset of observations from the workfile as though they were adjacent. There are three steps to using the method:

- create a “short version” of the data which contains only the subset of observations.
- use the “short version” of the data to get whatever results you need.
- create a “long version” of the results which match the observations in the original data.

Consider the simple case of creating a series which contains the differences between consecutive positive values of the series X, ignoring any intervening negative values.

The commands

```
smp1 if x>=0  
series dxp = d(x)
```

will not compute the desired values, since the `d` function computes differences between adjacent values, not between successive values in the sample.

The SUBSET.PRG program solves this problem using the general technique outlined above. The solution is stored in the series `DX`. The series `X_S` and `DX_S` are left in the workfile so you can follow the intermediate calculations.

```
' create sample
sample ss if x<>NA and x>=0

' create short x series
smpl @all
vector temp
stomna(x, temp, ss)
series x_s
mtos(temp, x_s)
!rows = @rows(temp)  'save number of elements
delete temp

' difference short x series
series dx_s = d(x_s)

' map back into long series dx
vector temp
stomna(dx_s, temp)
vector(!rows) temp  'trim to number of elements
series dx
mtos(temp, dx, ss)
delete temp

' display results
show x dx x_s dx_s
```

Creating Dummy Variables with a Loop

(`make_dum.prg`)

It is sometimes useful to create a set of dummy variables for a range of observations in a workfile. You can then include one or more of these dummies in an estimation simply by adding their names to the list of exogenous variables. It is easy to create these series by hand, though somewhat tedious.

The program MAKE_DUM.PRG automates this task. It should be run from the command line with arguments for the first and the last observation for which to create dummies. For example, to create dummies in a quarterly workfile from the first quarter of 1982 to the last quarter of 1983, the command would be:

```
run make_dum 1982:1 1983:4
```

As is, the program should work for annual, semiannual, quarterly, monthly and undated workfiles. It should be modified for weekly and daily workfiles.

```
'Set up start and end dates
%start = %0      'the first observation to dummy
%end = %1        'the last observation to dummy

'Generate dummy variables from 'start' to 'end'
for !i = @dtoo(%start) to @dtoo(%end)
    %obsstr = @otod(!i)
    if (@mid(%obsstr, 5, 1) = ":") then
        %name = "d_" + @left(%obsstr, 4) + "_" + @mid(%obsstr, 6)
    else
        %name = "d_" + %obsstr
    endif
    smpl @all
    series %name = 0
    smpl %obsstr %obsstr
    series %name = 1
next
```

Extracting Test Statistics in a Loop

(omitted.prg)

Most of the output from equation or system estimation can be accessed directly using EViews regression statistics functions (see [Chapter 3, “Object, View and Procedure Reference”, beginning on page 19](#)). This is not true of all of the output from EViews’ diagnostic tests.

If you would like to work with statistics from these views in a program, you should use the following general method that allows you to extract and store output from any object view. The basic steps in each step through the loop are:

- freeze the view into a temporary table.
- extract the cell containing the statistic and assign its contents to an element of a table or vector.

- delete the temporary table.

The following subroutine uses this approach to capture the F -statistic and probability value from an omitted variables test. The subroutine OMIT_TEST cycles through the series contained in a group, testing whether the first four lags of each series are jointly significant when added to the chosen equation. The output is stored in a table.

```
' test whether the first four lags of an omitted variable
' are jointly significant (eq1: the equation to test;
' g: a list of the series to include; results: output)
subroutine omit_test(equation eq1, group g, table results)
    !n = g.@count
    table(3, !n) results
    setcolwidth(results, 1, 12)
    setcell(results, 1, 1, "Four lag F-test for omitted
variables:", "l")
    results(2,2) = "F-stat"
    results(2,3) = "Probability"
    for !i = 1 to !n
        %name = g.@seriesname(!i)
        series temp_s = %name
        freeze(temp_t) eq1.testadd temp_s(-1) temp_s(-2) temp_s(-3)
temp_s(-4)
        results(!i+2, 1) = %name
        results(!i+2, 2) = temp_t(3, 2)
        results(!i+2, 3) = temp_t(3, 5)
        delete temp_t
        delete temp_s
    next
endsub
```

The following program uses this subroutine to carry out some tests on the data from the demo in [Chapter 2, “A Demonstration”, on page 15](#) of the *User’s Guide*. Both the subroutine and the program can be found in the file OMITTED.PRG.

```
open demo.wf1
smpl @all
equation eq1.ls dlog(m1) c dlog(m1(-1)) dlog(m1(-2)) dlog(m1(-3))
dlog(m1(-4))
group g1 dlog(gdp) dlog(rs) dlog(pr)
table tab1
call omit_test(eq1, g1, tab1)
show tab1
```

The program produces the following output:

```
Four lag F-test for omitted variables:  
          F-stat    Probability  
DLOG(GDP)  1.316803  0.265802  
DLOG(RS)   6.642363  0.000055  
DLOG(PR)   1.054600  0.380818
```

Between Group Estimation for Pooled Data

(between.prg)

Consider a pooled regression specification:

$$\log(y_{it}) = \alpha_i + \beta \log(x_{it}) + \epsilon_{it}, \quad (7.1)$$

where i is the cross-section index and t represents the time period. EViews provides built-in procedures to perform fixed effects (within-groups) estimation of β :

$$\log(\bar{y}_{it}) - \overline{\log(y_{it})} = \beta_W (\log(\bar{x}_{it}) - \overline{\log(x_{it})}) + (\bar{\epsilon}_{it} - \overline{\epsilon_{it}}), \quad (7.2)$$

where a bar over a variable name represents the average over time for the cross-section.

Alternatively, you may wish to estimate the between groups model:

$$\overline{\log(y_{it})} = \alpha_i + \beta_B \overline{\log(x_{it})} + \overline{\epsilon_{it}} \quad (7.3)$$

A quick way to obtain the between groups estimate is to use auto series in a pooled regression. If we run the pool regression

```
pool1.ls(c) @mean(log(ivm?)) @mean(log(mm?))
```

in the workfile POOL.WF1, we obtain:

```
Dependent Variable: @MEAN(LOG(IVM?))
Method: Pooled Least Squares
Date: 08/24/00 Time: 10:56
Sample: 1968:01 1995:12
Included observations: 336
Number of cross-sections used: 8
Total panel (balanced) observations: 2688
```

Variable	Coefficient	Std. Error	t-Statistic	Prob.
C	0.855345	0.092523	9.244701	0.0000
@MEAN(LOG(MM?))	0.959626	0.009793	97.99203	0.0000
R-squared	0.781421	Mean dependent var	9.890818	
Adjusted R-squared	0.781339	S.D. dependent var	0.847896	
S.E. of regression	0.396486	Sum squared resid	422.2431	
Log likelihood	-1326.401	F-statistic	9602.439	
Prob(F-statistic)	0.000000			

Although the point estimates are correct, the standard errors are not since EViews uses multiple observations for each cross-section unit.

We can, however, use a program to obtain the between estimator. The following program stores the cross-section specific means of each series in a matrix, creates a new workfile, converts the matrix to series, and runs the between groups regression:

```

' set number of cross-sections
%evworkfile = @evpath + "\example files\pool1"
load "%evworkfile"
smpl @all
!ncross = pool1.@ncross

' create group with variables
pool1.makegroup(tempgrp) log(ivm?) log(mm?)

' store means of two series in matrix
matrix(!ncross,2) means
series tempser
for !i = 1 to !ncross
    tempser = tempgrp(!i)
    means(!i,1) = @mean(tempser)
    tempser = tempgrp(!ncross+!i)
    means(!i,2) = @mean(tempser)
next
store(i) means

' create new workfile and fetch means matrix
workfile between u 1 !ncross
fetch(i) means

' convert matrix to series
series lc_mean
series ly_mean
group g1 lc_mean ly_mean
mtos(means,g1)

' run between groups regression and cleanup
equation eq_bet.ls lc_mean c ly_mean
delete tempgrp tempser
show eq_bet

```

The equation results for the between estimator are given by:

Dependent Variable: LC_MEAN

Method: Least Squares

Date: 08/24/00 Time: 10:55

Sample: 1 8

Included observations: 8

Variable	Coefficient	Std. Error	t-Statistic	Prob.
C	0.855345	1.957609	0.436934	0.6774
LY_MEAN	0.959626	0.207199	4.631412	0.0036
R-squared	0.781421	Mean dependent var	9.890818	
Adjusted R-squared	0.744991	S.D. dependent var	0.906271	
S.E. of regression	0.457653	Akaike info criterion	1.486906	
Sum squared resid	1.256676	Schwarz criterion	1.506766	
Log likelihood	-3.947623	F-statistic	21.44997	
Durbin-Watson stat	1.352621	Prob(F-statistic)	0.003572	

Note that the temporary group TEMPGRP and temporary series TEMPSER are deleted at the end of the program.

Hausman Test for Fixed Versus Random Effects

(hausman.prg)

The following program computes the Hausman test statistic for testing the null hypothesis of random effects against the alternative of fixed effects. The program estimates a fixed and random effects model with two slope regressors and stores the estimated coefficients and its covariance matrix. (The Grunfeld data used in the program is transcribed from Greene (1997), Table 15.1.).

```
' load the data
%evworkfile = @evpath + "\example files\grunfeld"
load "{$evworkfile}"
smpl @all

' estimate fixed effects and store results
pool1.ls(f) log(inv?) log(val?) log(cap?)
vector beta = pool1.@coefs
matrix covar = pool1.@cov

' keep only slope coefficients
vector b_fixed = @subextract(beta,1,1,2,1)
matrix cov_fixed = @subextract(covar,1,1,2,2)

' estimate random effects and store results
pool1.ls(r) log(inv?) log(val?) log(cap?)
beta = pool1.@coefs
```

```

covar = pool1.@cov

' keep only slope coefficients
vector b_gls = @subextract(beta,2,1,3,1)
matrix cov_gls = @subextract(covar,2,2,3,3)

' compute Hausman test stat
matrix b_diff = b_fixed - b_gls
matrix v_diff = cov_fixed - cov_gls
matrix qform = @transpose(b_diff) *@inverse(v_diff)*b_diff

if qform(1,1)>=0 then
    ' set table to store results
    table(4,2) result
    setcolwidth(result, 1, 17)
    setcell(result,1,1,"Hausman test","l")
    setcell(result,2,1,"(fixed versus random effects)","l")
    setline(result,3)

    !df = @rows(b_diff)
    setcell(result,4,1,"Chi-square (" +@str(!df)+" d.f.)","l")
    setcell(result,4,2,qform(1,1))
    setcell(result,5,1,"p-value","l")
    setcell(result,5,2,1-@cchisq(qform(1,1),!df))
    show result
else
    statusline "Quadratic form is negative"
endif

```

Note that the program stops if the computed test statistic is negative, which can happen in finite samples. Running this program yields the table

Hausman test (fixed versus random effects)	
Chi-square (2 d.f.)	8.2380468
p-value	0.0162604

Regression Output Table

(regrun.prg / regtab.prg)

EViews presents regression output in tabular form, with separate columns for the coefficient values and the standard errors. For example, the output view from the equation EQ1 in the workfile BASICS.WF1 is given by:

Dependent Variable: LOG(M1)				
Method: Least Squares				
Date: 08/18/97 Time: 14:02				
Sample: 1959:01 1989:12				
Included observations: 372				
Variable	Coefficient	Std. Error	t-Statistic	Prob.
C	-1.699912	0.164954	-10.30539	0.0000
LOG(IP)	1.765866	0.043546	40.55199	0.0000
TB3	-0.011895	0.004628	-2.570016	0.0106
R-squared	0.886416	Mean dependent var	5.663717	
Adjusted R-squared	0.885800	S.D. dependent var	0.553903	
S.E. of regression	0.187183	Akaike info criterion	-0.505429	
Sum squared resid	12.92882	Schwarz criterion	-0.473825	
Log likelihood	97.00980	F-statistic	1439.848	
Durbin-Watson stat	0.008687	Prob(F-statistic)	0.000000	

A commonly used alternative display format places the standard errors or *t*-statistics, enclosed in parentheses, below the coefficient estimates. The program file REGTAB.PRG contains a subroutine which takes an equation object and creates a table with this alternative output:

```
' subroutine to reformat regression output
subroutine regtab(equation eq1, table tab1, scalar format)

' assign useful values
!ncoef =eq1.@ncoef
!obs=eq1.@regobs

' create a temporary table with the results of the estimation
' and declare a new table
table(1,4) tab1
freeze(temp_table) eq1.results

setcolwidth(tab1, 1, 19)
setcolwidth(tab1, 2, format+4)
setcolwidth(tab1, 3, 19)
setcolwidth(tab1, 4, format+4)
```

```
' get the original header information
!line = 1
while temp_table(!line, 2) <> "Coefficient"
    setcell(tab1, !line, 1, temp_table(!line, 1), "l")
    !line = !line + 1
wend
setline(tab1, !line-1)
setcell(tab1, !line, 1, "Variable", "c")
setcell(tab1, !line, 2, "Estimate", "r")
setcell(tab1, !line, 3, "Estimate", "r")
setcell(tab1,!line+1, 2, "(s.e.)", "r")
setcell(tab1,!line+1, 3, "(t-stat)", "r")
!line = !line + 2
setline(tab1, !line)
!line = !line + 1
!vline = !line

' fill all of the coefficients and standard errors
' (or t-statistics)
for !i = 1 to !ncoef
    %variable = temp_table(!vline-1, 1)
    ' write coefficients
    setcell(tab1, !line, 1, %variable, "c")
    !vline = !vline + 1

    ' write coefficients
    !est=eq1.@coefs(!i)
    setcell(tab1, !line, 2, !est, "r", format)
    setcell(tab1, !line, 3, !est, "r", format)
    !line = !line + 1

    ' compute statistics
    !se = sqr(eq1.@covariance(!i, !i))
    !tstat = !est!/se
    !tprob = @tdist(!tstat, !obs-!ncoef)

    ' write standard error output
    setcell(tab1, !line, 2, !se, "r", format)
    %str_se = tab1(!line, 2)
    %str_se = "(" + %str_se + ")"
    tab1(!line, 2) = %str_se
```

```
' write t-statistic output
setcell(tab1, !line, 3, !tstat, "r", format)
%str_t = "(" + tab1(!line, 3) + ")"
if !tprob < .01 then
    %str_t = "***" + %str_t
else
    if !tprob < .05 then
        %str_t = "*" + %str_t
    endif
endif
tab1(!line, 3) = %str_t
!line = !line + 1
next

setline(tab1, !line)
!line = !line + 1

' original results at bottom of table
setcell(tab1, !line, 1, "R-squared", "l")
setcell(tab1, !line, 2, eq1.@r2, "r", format)
setcell(tab1, !line, 3, " Mean dependent var", "l")
setcell(tab1, !line, 4, eq1.@meandep, "r", format)
!line = !line + 1

setcell(tab1, !line, 1, "Adjusted R-squared", "l")
setcell(tab1, !line, 2, eq1.@rbar2, "r", format)
setcell(tab1, !line, 3, " S.D. dependent var", "l")
setcell(tab1, !line, 4, eq1.@sddep, "r", format)
!line = !line + 1

setcell(tab1, !line, 1, "S.E. of regression", "l")
setcell(tab1, !line, 2, eq1.@se, "r", format)
setcell(tab1, !line, 3, " Akaike info criterion", "l")
setcell(tab1, !line, 4, eq1.@aic, "r", format)
!line = !line + 1

setcell(tab1, !line, 1, "Sum squared resid", "l")
setcell(tab1, !line, 2, eq1.@ssr, "r", format)
setcell(tab1, !line, 3, " Schwarz criterion", "l")
setcell(tab1, !line, 4, eq1.@schwarz, "r", format)
```

```

!line = !line + 1

setcell(tab1, !line, 1, "Log likelihood", "l")
setcell(tab1, !line, 2, eq1.@logl, "r", format)
setcell(tab1, !line, 3, " F-statistic", "l")
setcell(tab1, !line, 4, eq1.@f, "r", format)
!line = !line + 1

setcell(tab1, !line, 1, "Durbin-Watson stat", "l")
setcell(tab1, !line, 2, eq1.@dw, "r", format)
setcell(tab1, !line, 3, " Prob(F-statistic)", "l")
setcell(tab1, !line, 4, @fdist(eq1.@f, (!ncoef-1), (!obs-!ncoef)),
       "r", format)
!line = !line + 1
setline(tab1, !line)

endsub

```

To call the subroutine, you can open the program REGRUN.PRG, which contains the following commands:

```

include regtab.prg
%evworkfile = @evpath + "\example files\basics"
load "{$evworkfile}"
table tab1
call regtab(eq1, tab1, 3)
show tab1

```

The first argument to the REGTAB subroutine is the equation object, the second is the table into which you wish to place the results, and the third argument is a numeric format code. Here we indicate that we wish to display three digits after the decimal. Table TAB1 will contain the reformatted output:

Dependent Variable: LOG(M1)

Method: Least Squares

Date: 08/18/97 Time: 14:02

Sample: 1959:01 1989:12

Included observations: 372

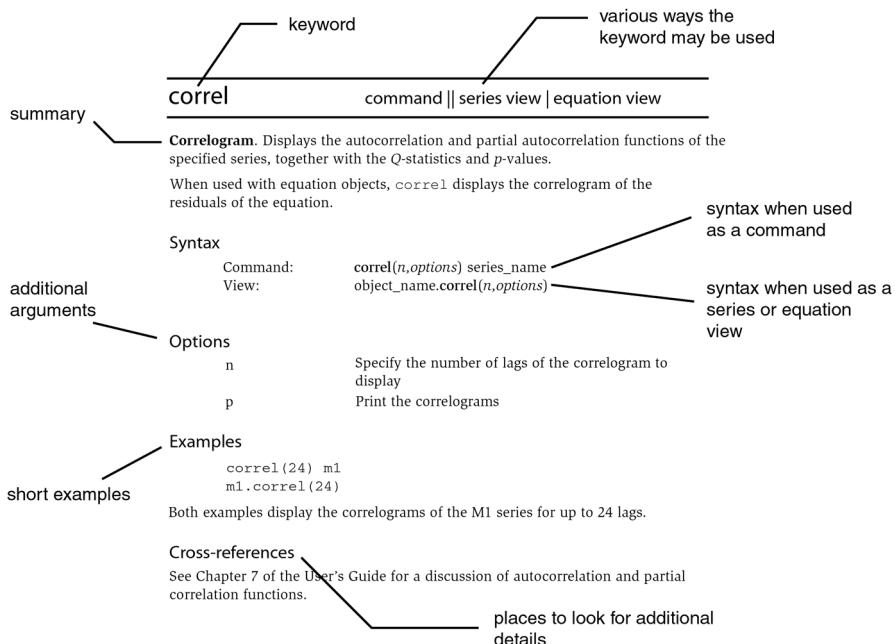
Variable	Estimate (s.e.)	Estimate (t-stat)
C	-1.700 (0.165)	-1.700 **(-10.305)
LOG(IP)	1.766 (0.044)	1.766 **(40.552)
TB3	-0.012 (0.005)	-0.012 *(-2.570)
R-squared	0.886	Mean dependent var 5.664
Adjusted R-squared	0.886	S.D. dependent var 0.554
S.E. of regression	0.187	Akaike info criterion -0.505
Sum squared resid	12.929	Schwarz criterion -0.474
Log likelihood	97.010	F-statistic 1439.848
Durbin-Watson stat	0.009	Prob(F-statistic) 0.000

Chapter 8. Command Reference

This chapter provides a complete alphabetical listing of the commands, views, and procedures in EViews.

Each entry is comprised of a keyword, followed by a listing of the ways in which the keyword may be used. The entry also provides a summary of behavior, as well as a description of the syntax. If appropriate, we also provide a listing of additional arguments, short examples, and cross-references.

For example, the listing for `correl` is given by:



Note that the keyword `correl` may be used in three distinct ways: as a command to compute the correlogram of a specified series, as a series view to display the correlogram for a series object, and as an equation view to display the correlogram of the residuals from the estimated equation. The syntax for each of these uses is documented separately.

In addition to the dictionary-style alphabetical listing provided here, there are several other places in the *Command and Programming Reference* that contain reference material:

- “[Basic Command Summary](#)” beginning on page 17, provides a listing of commands commonly used when managing objects in workfiles and databases.

- [Chapter 3, “Object, View and Procedure Reference”, beginning on page 19](#), lists all of the views and procedures classified by object, making it easy to see which views and procedures are available for a given object.
- In [“Matrix Function and Command Summary” on page 76](#), we list the basic functions and commands used when working with matrix objects.
- In [“Programming Summary” on page 114](#), we list the commands, functions and keywords used in the EViews programming and string processing.
- [Chapter 9, “Matrix and String Reference”, beginning on page 397](#) documents the matrix, vector, scalar, and string functions and commands.
- [Chapter 10, “Programming Language Reference”, beginning on page 421](#) contains the primary documentation for the commands and keywords used in the programming language. These entries may only be used in batch programs.
- [Appendix A, “Operator and Function Reference”, on page 435](#) contains a complete listing of the functions and mathematical operators used in forming series expressions, and in performing matrix element operations.

3sls	System Method
-------------	-------------------------------

Estimate a system of equations by three-stage least squares.

Syntax

System Method: `system_name.3sls(options)`

Options

i	Iterate simultaneously over the weighting matrix and coefficient vector.
s	Iterate sequentially over the weighting matrix and coefficient vector.
o (default)	Iterate the coefficient vector to convergence following one-iteration of the weighting matrix.
c	One step (iteration) of the coefficient vector following one-iteration of the weighting matrix.
<code>m = integer</code>	Maximum number of iterations.

<code>c = number</code>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
<code>l = number</code>	Set maximum number of iterations on the first-stage coefficient estimation to get the one-step weighting matrix.
<code>showopts / -showopts</code>	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
<code>deriv = keyword</code>	Set derivative methods. The argument <i>keyword</i> should be a one- or two-letter string. The first letter should either be “f” or “a” corresponding to fast or accurate numeric derivatives (if used). The second letter should be either “n” (always use numeric) or “a” (use analytic if possible). If omitted, EViews will use the global defaults.
<code>p</code>	Print estimation results.

Examples

```
sys1.3sls(i)
```

Estimates SYS1 by the 3SLS method, iterating simultaneously on the weighting matrix and the coefficient vector.

```
nlsys.3sls(showopts,m=500)
```

Estimates NLSYS by 3SLS with up to 500 iterations. The “showopts” option displays the starting values and other estimation options.

Cross-references

See [Chapter 19](#) of the *User’s Guide* for discussion of system estimation.

add	Group Proc Pool View
---------------------	--

Add series to a group or add cross section members to a pool.

Syntax

Group Proc:	<code>group_name.add ser1 [ser2 ser3 ...]</code>
Group Proc.	<code>group_name.add grp1 [grp2 ...]</code>
Pool Proc:	<code>pool_name.add id1 [id2 id3 ...]</code>

List the names of series or group of series to add to the group, or list the cross-section identifiers to add to the pool.

Examples

```
dummy.add d11 d12
```

Adds the two series D11 and D12 to the group DUMMY.

```
countries.add us gr
```

Adds US and GR as cross-section members of the pool object COUNTRIES.

Cross-references

See also [drop \(p. 195\)](#).

addassign	Model Proc
-----------	----------------------------

Assign add factors to equations.

Syntax

Model Proc: *model_name.addassign(options) equation_spec*

where *equation_spec* identifies the equations for which you wish to assign add factors. You may either provide a list of endogenous variables, or you can use one of the following shorthand keywords:

@all All equations.

@stochastic All stochastic equations (no identities).

@identity All identities.

The options identify the type of add factor to be used, and control the assignment behavior for equations where you have previously assigned add factors. *addassign* may be called multiple times to add different types of add factors to different equations. *addassign* may also be called to remove existing add factors.

Options

i Intercept shifts (default).

v Variable shift.

n None—remove add factors.

c Change existing add factors to the specified type—if the “c” option is not used, only newly assigned add factors will be given the specified type.

Examples

```
m1.addassign(v) @all
```

assigns a variable shift to all equations in the model while

```
m1.addassign(c, i) @stochastic
```

changes the stochastic equation add factors to intercept shifts.

```
m1.addassign(v) @stochastic
```

```
m1.addassign(v) y1 y2 y3
```

```
m1.addassign(i) @identity
```

assigns variable shifts to the stochastic equations and the equations for Y1, Y2, and Y3, and assigns intercept shifts to the identities.

Cross-references

See “[Using Add Factors](#)” on page 626 of the *User’s Guide*. See also [Chapter 23, “Models”, beginning on page 601](#) of the *User’s Guide* for a general discussion of models.

See [addinit \(p. 139\)](#).

addinit	Model Proc
----------------	----------------------------

Initialize add factors.

Syntax

Model Proc: `model_name.addinit(options) equation_spec`

where *equation_spec* identifies the equations for which you wish to initialize the add factors. You may either provide a list of endogenous variables, or you may use one of the following shorthand keywords:

@all	All equations
------	---------------

@stochastic	All stochastic equations (no identities)
-------------	--

@identity	All identities
-----------	----------------

The options control the type of initialization and the scenario for which you want to perform the initialization. `addinit` may be called multiple times to initialize various types of add factors in the different scenarios.

Options

v = z (default)	Set add factor values to zero
v = n	Set add factor values so that the equation has no residual when evaluated at actuals
v = b	Set add factors to the values of the baseline (override = actual)
s = a (default)	Set active scenario add factors
s = b	Set baseline scenario (actuals) add factors
s = o	Set active scenario override add factors

Examples

```
m1.addinit(v=b) @all
```

sets all of the add factors in the active scenario to the values of the baseline.

```
m1.addinit(v=z) @stochastic
```

```
m1.addinit(v=n) y1 y1 y2
```

first sets the active scenario stochastic equation add factors to zero, and then sets the Y1, Y2, and Y3 equation residuals to zero (evaluated at actuals).

```
m1.addinit(s=b, v=z) @stochastic
```

sets the baseline scenario add factors to zero.

Cross-references

See “[Using Add Factors](#)” on page 626 of the *User’s Guide*. See also [Chapter 23](#) of the *User’s Guide* for a general discussion of models.

See also [addassign \(p. 138\)](#).

addtext	Graph Proc
----------------	----------------------------

Place text in graphs.

Syntax

Graph Proc: graph_name.addtext(options) text

Follow the `addtext` keyword with the text to be placed in the graph.

Options

`font = n` Set the size of the font.

The following options control the position of the text.

`t` Top (above the graph and centered).

`l` Left rotated.

`r` Right rotated.

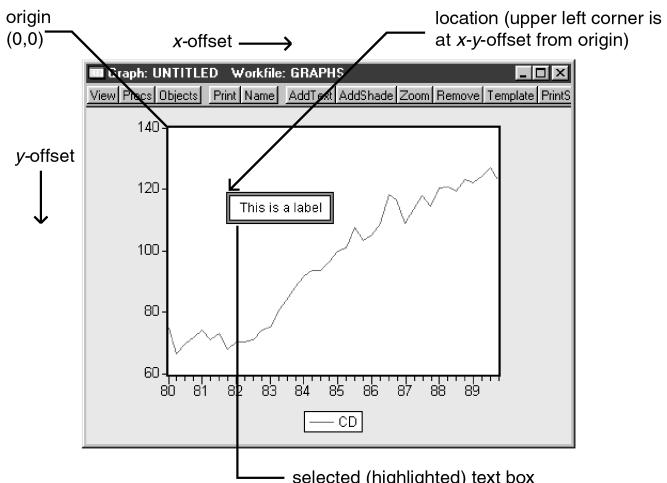
`b` Below and centered.

`x` Enclose text in box.

To place text within a graph, you can use explicit coordinates to specify the position of the upper left corner of the text.

Coordinates are set by a pair of numbers h, v in virtual inches. Individual graphs are always 4×3 virtual inches (scatter diagrams are 3×3 virtual inches) regardless of their current display size.

The origin of the coordinate is the upper left hand corner of the graph. The first number h specifies how many virtual inches to offset to the right from the origin. The second number v specifies how many virtual inches to offset below the origin. The upper left hand corner of the text will be placed at the specified coordinate.



Coordinates may be used with other options, but they must be in the first two positions of the options list. Coordinates are overridden by other options that specify location.

When `addtext` is used with a multiple graph, the text is applied to the whole graph, not to each individual graph.

Examples

```
freeze(g1) gdp.line
```

```
g1.addtext(t) Fig 1: Monthly GDP (78.1-95.12)
```

Places the text “Fig1: Monthly GDP (78.1-95.12)” centered above the graph G1.

```
g1.addtext(.2,.2,X) Seasonally Adjusted
```

Places the text “Seasonally Adjusted” in a box within the graph, slightly indented from the upper left corner.

Cross-references

See also [legend \(p. 240\)](#).

align	Graph Proc
--------------	----------------------------

Align placement of multiple graphs.

Syntax

Graph Proc: `graph_name.align(n,h,v)`

Options

You must specify three numbers (each separated by a comma) in parentheses in the following order: the first number n is the number of columns in which to place the graphs, the second number h is the horizontal space between graphs, and the third number v is the vertical space between graphs. Spacing is specified in virtual inches.

Examples

```
mygraph.align(3,1.5,1)
```

Aligns MYGRAPH with graphs placed in three columns, horizontal spacing 1.5 inches, and vertical spacing 1 inch.

```
var var1.ls 1 4 m1 gdp
freeze(impgra) var1.impulse(m,24) gdp @ gdp m1
impgra.align(2,1,1)
```

Estimates a VAR, freezes the impulse response functions as multiple graphs, and realigns the graphs. By default, the graphs are stacked in one column, and the realignment places the graphs in two columns.

Cross-references

For a detailed discussion of customizing graphs, see [Chapter 10, “Graphs, Tables, and Text Objects”, on page 243](#) of the *User’s Guide*.

See also [graph](#) (p. 224).

append

[Logl Proc](#) | [Model Proc](#) | [Sspace Proc](#) | [System Proc](#) | [Var Proc](#)

Append a specification line to a model, system, sspace, or var.

Syntax

Object Proc:	<code>object_name.append text</code>
Var Proc:	<code>var_name.append(options) text</code>

Type the text to be added after the `append` keyword. *For vars, you must specify the text type in the options argument.*

Options for Vars

One of the following options is required when using `append` as a var proc:

svar	Text for identifying restrictions for structural VAR.
coint	Text for restrictions on the cointegration relations and/or adjustment coefficients.

Examples

```
model macro2
macro2.merge eq_m1
macro2.merge eq_gdp
macro2.append assign @all f
macro1.append @trace gdp
macro2.solve
```

The first line declares a model object. The second and third lines merge existing equations into the model. The fourth and fifth line appends an `assign` statement and a trace of GDP to the model. The last line solves the model.

```
system macro1
macro1.append cons=c(1)+c(2)*gdp+c(3)*cons(-1)
macro1.append inv=c(4)+c(5)*tb3+c(6)*d(gdp)
macro1.append gdp=cons+inv+gov
macro1.append inst tb3 gov cons(-1) gdp(-1)
macro1.gmm
show macro1.results
```

The first line declares a system. The next three lines appends the specification of each endogenous variable in the system. The fifth line appends the list of instruments to be used in estimation. The last two lines estimate the model by GMM and display the estimation results.

```
vector(2) svec0=0  
sspace1.append @mprior svec0
```

This command appends a line in the state space object SSPACE1 to use the zero vector SVEC0 as initial values for the state vector.

Cross-references

See also [cleartext \(p. 163\)](#). See [Chapters 19](#) and [23](#) of the *User’s Guide* for a discussion of systems and models, respectively.

ar	Expression
----	------------

Autoregressive error specification.

The AR specification can appear in an ls or tsls command to indicate an autoregressive component. ar(1) indicates the first order component, ar(2) indicates the second order component, and so on.

Examples

The command

```
ls m1 c tb3 tb3(-1) ar(1) ar(4)
```

regresses M1 on a constant, TB3, and TB3 lagged once with a first order and fourth order autoregressive component. The command

```
tsls sale c adv ar(1) ar(2) ar(3) ar(4) @ c gdp
```

performs two-stage least squares of SALE on a constant and ADV with up to fourth order autoregressive components using a constant and GDP as instruments.

Cross-references

See [Chapter 13, “Time Series Regression”, on page 303](#) of the *User’s Guide* for details on ARMA and seasonal ARMA modeling.

See also [sar \(p. 307\)](#), [ma \(p. 249\)](#), and [sma \(p. 330\)](#).

arch	Command Equation Method
------	--

Estimate generalized autoregressive conditional heteroskedasticity (GARCH) models.

Syntax

Command: `arch(p,q,options) y x1 x2 x3 [@ p1 p2 @ t1 t2]`

Equation Method: `eq_name.arch(p,q,options) y x1 x2 x3 [@ p1 p2 @ t1 t2]`

ARCH estimates a GARCH(p, q) model with p ARCH terms and q GARCH terms. If you do not specify (p, q), then GARCH(1,1) is assumed by default. After the “arch” keyword, specify the dependent variable followed by a list of regressors in the mean equation.

By default, no exogenous variables (except for the intercept) are included in the conditional variance equation. If you want to include variance regressors in addition to the GARCH terms, list them after the mean equation regressors using an “@”-sign to separate the two lists.

If you choose the option for component ARCH models, you may specify exogenous variables for the permanent and transitory components separately. After the mean equation regressors, first list the regressors for the permanent component, followed by an “@”-sign, then the regressors for the transitory component. A constant term is always included in the permanent component.

Options

e, egarch	Exponential GARCH.
t, tarch	Threshold (asymmetric) ARCH.
c	Component (permanent and transitory) ARCH.
a	Asymmetric component (permanent and transitory) ARCH.
v	ARCH-M (ARCH in mean) with conditional variance in the mean equation.
m	ARCH-M (ARCH in mean) with conditional standard deviation in the mean equation.
h	Bollerslev-Wooldridge robust quasi-maximum likelihood (QML) covariance/standard errors.
z	Turn off backcasting for both initial MA innovations and initial variances.

b	Use Berndt-Hall-Hausman (BHHH) algorithm for maximization. The default is Marquardt.
m = <i>integer</i>	Set maximum number of iterations.
c = <i>scalar</i>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
s	Use the current coefficient values in C as starting values (see param).
s = <i>number</i>	Specify a number between zero and one to determine starting values as a fraction of preliminary LS estimates (out of range values are set to “s = 1”).
showopts / -showopts	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
deriv = <i>keyword</i>	Set derivative methods. The argument <i>keyword</i> should be a one-letter string (“f” or “a” corresponding to fast or accurate numeric derivatives).
p	Print estimation results.

Saved results

Most of the results saved for the `ls` command are also available after ARCH estimation; see [ls \(p. 245\)](#) for details.

Examples

The command

```
arch(4,0,m=1000,h) sp500 c
```

estimates an ARCH(4) model with a mean equation consisting of SP500 regressed on a constant. The procedure will perform up to 1000 iterations and, upon convergence, will report Bollerslev-Wooldridge robust QML standard errors.

The commands

```
c=.1  
equation arc1.arch(s,v) nys c @ r
```

estimate a GARCH(1,1)-in-mean specification, with the mean equation for NYS depending upon a constant and a conditional variance (GARCH) term. The conditional variance equation is the default GARCH(1,1) specification, with exogenous regressors R and a constant.

The first line sets the default coefficient vector to 0.1 and the “s” option uses these elements of the C coefficient vector as starting values.

Following estimation, we can save the estimated conditional variance as a series named GARCH1.

```
arc1.makegarch garch1
```

Cross-references

See [Chapter 16](#) of the *User’s Guide* for a discussion of ARCH models.

See also [garch \(p. 219\)](#) and [makegarch \(p. 252\)](#).

archtest	Command Equation View
-----------------	--

Test for autoregressive conditional heteroskedasticity (ARCH).

Carries out Lagrange Multiplier (LM) tests for ARCH in the residuals.

Syntax

Command:	<code>archtest(options)</code>
Equation View:	<code>eq_name.archtest(options)</code>

Options

You must specify the order of ARCH to test for. The number of lags to include should be provided in parentheses after the `arch` keyword.

Other Options:

<code>p</code>	Print output from the test.
----------------	-----------------------------

Examples

```
ls output c labor capital
archtest(4)
```

Regresses OUTPUT on a constant, LABOR, and CAPITAL and tests for ARCH up to order 4.

```
equation eq1.arch sp500 c
eq1.archtest(4)
```

Estimates a GARCH(1,1) model with mean equation of SP500 on a constant and tests for additional ARCH up to order 4. Note that performing an `archtest` after an `arch` estimation uses the standardized residuals (the residual of the mean equation divided by the estimated conditional standard deviation).

Cross-references

See “[ARCH LM Test](#)” on page 377 of the *User’s Guide* for further discussion of testing ARCH and [Chapter 16](#) of the *User’s Guide* for a general discussion of working with ARCH models in EViews.

arlm	Var View
------	--------------------------

Multivariate residual serial correlation LM test.

Syntax

Var View: var_name.arlm(*h, options*)

You must specify the highest order of lag *h* to test for serial correlation.

Options

name = *arg* Save LM statistics in named matrix object. The matrix has *h* rows and one column.

p Print test output.

Examples

```
var var1.ls 1 6 lgdp lm1 lcpi  
show var1.arlm(12, name=lmout)
```

The first line declares and estimates a VAR with 6 lags. The second line displays the serial correlation LM tests for lags up to 12 and stores the statistics in a matrix named LMOUT.

Cross-references

See “[Diagnostic Views](#)” on page 522 of the *User’s Guide* for other VAR diagnostics.

See also [qstats \(p. 289\)](#) for related multivariate residual autocorrelation portmanteau tests.

arroots	Var View
---------	--------------------------

Inverse roots of the characteristic AR polynomial.

Syntax

Var View: var_name.arroots(*options*)

Options

name = <i>arg</i>	Save roots in named matrix object. The matrix has two columns, where the first column is the real and the second column is the imaginary part of each root in the rows.
graph	Plots the roots together with a unit circle. The VAR is stable if all roots are inside the unit circle.
p	Print table of AR roots.

Examples

```
var var1.ls 1 6 lgdp lm1 lcpi
var1.arroots(graph)
```

The first line declares and estimates a VAR with 6 lags. The second line plots the AR roots of the estimated VAR.

```
var var1.ls 1 6 lgdp lm1 lcpi
'store roots
freeze(tab1) var1.arroots(name=roots)
```

The first line declares and estimates a VAR with 6 lags. The second line stores the roots in a matrix named ROOTS and the table view as a table named TAB1.

Cross-references

See “[Diagnostic Views](#)” on page 522 of the *User’s Guide* for other VAR diagnostics.

auto	Command Equation View
------	--

Serial correlation LM (Lagrange multiplier) test.

Carries out Breusch-Godfrey Lagrange Multiplier (LM) tests for serial correlation in the estimation residuals.

Syntax

Command: **auto**(*options*)

Equation View: eq_name.**auto**(*options*)

In command form, **auto** tests the residuals from the default equation.

Options

You must specify the order of serial correlation to test for. You should specify the number of lags in parentheses after the `auto` keyword, followed by any additional options.

Other Options:

p	Print output from the test.
---	-----------------------------

Examples

To regress OUTPUT on a constant, LABOR, and CAPITAL and test for serial correlation of up to order four:

```
ls output c labor capital  
auto(4)
```

The commands

```
output(t) c:\result\artest.txt  
equation eq1.ls cons c y y(-1)  
eq1.auto(12, p)
```

perform a regression of CONS on a constant, Y and lagged Y and test for serial correlation of up to order twelve. The first line redirects printed tables/text to the ARTEST.TXT file.

Cross-references

See “Serial Correlation LM Test” on page 305 of the *User’s Guide* for further discussion of the Breusch-Godfrey test.

bar	Command Coef View Graph Proc Group View Matrix View Series View Sym View Vector View
-----	--

Bar graph of series or each column of a vector/matrix.

Syntax

Command: `bar(options) ser1 ser2 ser3 ...`
Object View: `object_name.bar(options)`
Graph Proc: `graph_name.bar(options)`

Options

Template and printing options

<code>o = graph_name</code>	Use appearance options from the specified graph.
<code>t = graph_name</code>	Use appearance options and copy text and shading from the specified graph.
<code>p</code>	Print the bar graph.

Scale options

<code>a</code> (default)	Automatic scaling. The series are graphed in their original units and the range of the graph is chosen to accommodate the highest and lowest values of the series.
<code>d</code>	Dual scaling. The first series is scaled on the left and all other series are scaled on the right.
<code>s</code>	Stacked bar graph. Each bar represents the cumulative total of the series listed (may not be used with the “l” option).
<code>l</code>	Bar graph for the first series listed and a line graph for all subsequent series (may not be used with the “s” option).
<code>x</code>	Same as the “d” option (dual scaling).
<code>m</code>	Plot bars in multiple graphs.

Examples

Plot a bar graph of POP together with line graphs of GDP and CONS:

```
bar(x, o=mybar1) pop gdp cons
```

The bar graph is scaled on the left, while the line graphs are scaled on the right. The graph uses options from graph MYBAR1 as a template.

```
group mygrp oldsales newsales
mygrp.bar(s)
```

The first line defines a group of series and the second line graphs a stacked bar graph of the series in the group.

Cross-references

See “[Graph Templates](#)” on page 249 of the *User’s Guide* for a discussion of graph templates.

See also [freeze](#) (p. 216) and [graph](#) (p. 224).

bdstest	Series View
----------------	-----------------------------

Perform BDS test for independence.

The BDS test is a portmanteau test for time based dependence in a series. It can be used for testing against a variety of possible deviations from independence including linear dependence, non-linear dependence, or chaos.

Syntax

Series View: `series_name.bds(options)`

Options

<code>m = method</code>	Method for calculating ϵ .
<code>m = p</code>	Fraction of pairs.
<code>m = v</code>	Fixed value.
<code>m = s</code>	Standard deviations.
<code>m = r</code>	Fraction of range.
<code>e</code>	Value for calculating ϵ .
<code>d</code>	Maximum dimension.
<code>b</code>	Number of repetitions for bootstrap p -values. If option is omitted, no bootstrapping is performed.
<code>o = arg</code>	Name of output vector for final BDS z -statistics.
<code>p</code>	Print output.

Cross-references

See “[BDS Test](#)” on page 170 of the *User’s Guide* for additional discussion.

binary	Command Equation Method
---------------	--

Estimate binary dependent variable models.

Estimates models where the binary dependent variable Y is either zero or one (probit, logit, gompit).

Syntax

Command: **binary(*options*)** *y* *x1* *x2* *x3*

Equation Method: **eq_name.binary(*options*)** *y* *x1* *x2* *x3*

Options

d = n (default)	Maximize using normal likelihood function (probit).
d = l	Maximize using logistic likelihood function (logit).
d = x	Maximize using (Type I) extreme value likelihood function (Gompit).
q (default)	Use quadratic hill climbing for maximization algorithm.
r	Use Newton-Raphson for maximization algorithm.
b	Use Berndt-Hall-Hall-Hausman (BHHH) for maximization algorithm.
h	Quasi-maximum likelihood (QML) standard errors.
g	GLM standard errors.
m = integer	Set maximum number of iterations.
c = scalar	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
s	Use the current coefficient values in C as starting values.
s = number	Specify a number between zero and one to determine starting values as a fraction of EViews default values (out of range values are set to “s = 1”).
showopts / -showopts	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
p	Print results.

Examples

To estimate a logit model of Y on a constant, WAGE, EDU, and KIDS with QML standard errors:

```
binary(d=l,h) y c wage edu kids
```

This command uses the default quadratic hill climbing algorithm. The commands

```
param c(1) .1 c(2) .1 c(3) .1
```

```
equation probit1.binary(s) y c x2 x3
```

estimate a probit model of Y on a constant, X2, and X3, using the specified starting values.
The commands

```
coef beta_probit = probit1.@coefs  
matrix cov_probit = probit1.@coeffcov
```

store the estimated coefficients and coefficient covariances in the coefficient vector
BETA_PROBIT, and matrix COV_PROBIT.

Cross-references

See “[Binary Dependent Variable Models](#)” on page 421 of the *User’s Guide* for additional discussion.

block	Model View
-------	----------------------------

Display the model block structure view.

Show the block structure of the model identifying which blocks are recursive and which blocks are simultaneous.

Syntax

Model View: `model_name.block(options)`

Options

p Print the block structure view.

Cross-references

See “[Block Structure View](#)” on page 623 of the *User’s Guide* for details. See the remainder of [Chapter 23](#) of the *User’s Guide* for a general discussion of models.

See also [eqs \(p. 201\)](#), [text \(p. 363\)](#) and [vars \(p. 377\)](#) for alternative representations of the model.

cause	Command Group View
-------	---

Granger causality test.

Performs pairwise Granger causality tests between (all possible) pairs of the listed series or group of series.

Syntax

Command: `cause(n, options) ser1 ser2 ser3`

Group View: `group_name.cause(n, options)`

In command form, you should list the series or group of series in which to test Granger causality.

Options

You must specify the number of lags n to use for the test; specify an integer in parentheses after the `cause` keyword. Note that the regressors of the test equation are a constant and the specified lags of the pair of series under test.

Other options:

p	Print output of the test.
---	---------------------------

Examples

Compute Granger causality tests of whether GDP Granger causes M1 and whether M1 Granger causes GDP.

```
cause(4) gdp m1
```

The regressors of each test are a constant and four lags of GDP and M1. The commands

```
group macro m1 gdp r  
macro.cause(12,p)
```

print the result of six pairwise Granger causality tests for the three series in the MACRO group. The regressors of each test are a constant and twelve lags of the two series under test (and do not include lagged values of the third series in the group).

Cross-references

See “[Granger Causality](#)” on page 222 of the *User’s Guide* for a discussion of Granger’s approach to testing hypotheses about causality.

See also [var](#) (p. 376).

ccopy	Command
-------	-------------------------

Copy series from the DRI Basic Economics database to the data bank.

Copies one or more series from the DRI Basic Economics Database to the EViews data bank (.DB) files. *You must have the DRI database installed on your computer to use this feature.*

Syntax

Command: `ccopy series_name`

Type the name of the series you want to copy after the `ccopy` keyword. The data bank file will be stored in the default directory with the same name as the series name in the DRI database. You can supply path information to indicate the directory for the data bank file.

Examples

The command

`ccopy lhur`

copies the DRI series LHUR to LHUR.DB file in the default path directory, while

`ccopy b:gdp c:\nipadata\gnet`

copies the GDP series to GDP.DB file in the B drive and the GNET series to the GNET.DB file in C:\NIPADATA.

Cross-references

See also [cfetch \(p. 160\)](#), [clabel \(p. 162\)](#), [store \(p. 347\)](#), [fetch \(p. 205\)](#).

cd, chdir	Command
-----------	-------------------------

Change default directory.

The `cd` command changes the current default working directory. The current working directory is displayed as “Path = ...” at the bottom right of the EViews window.

Syntax

Command: `cd path_name`

Examples

To change the default directory to “SAMPLE DATA” in the A drive:

```
cd "a:\sample data"
```

Notice that the quotes surround the entire directory name. If your name does not contain spaces, you may omit the quotes:

```
cd a:\test
```

changes the default directory to A:TEST.

Everything you save will be saved to the default directory, unless you specify a different directory in the `save` command.

Cross-references

See [Chapter 3, “EViews Basics”, on page 33](#) of the *User’s Guide* for further discussion of basic operations in EViews.

cdfplot	Group View Series View
----------------	--

Empirical distribution functions.

Displays empirical cumulative distribution functions, survivor functions, and quantiles with standard errors.

Syntax

Object View: `series_name.cdfplot(options)`

Note: due to the potential for very long computation times, standard errors will only be computed if there are fewer than 2500 observations.

Options

c (default)	Plot the empirical CDF.
s	Plot the empirical survivor function.
q	Plot the empirical quantiles.
a	Plot all CDF, survivor, and quantiles.
n	Do not include standard errors.

<code>q = arg</code> (default = “r”)	Compute quantiles using the definition: “b” (Blom), “r” (Rankit-Cleveland), “o” (simple fraction), “t” (Tukey), “v” (van der Waerden).
<code>o = arg</code>	Output matrix to store results. Each column of the results matrix corresponds to one of the values used in plotting: (in order) evaluation points, the value, standard errors. If there are multiple graphs, all of the columns for the first graph are presented prior to the columns for the subsequent graph.
<code>p</code>	Print the distribution function(s).

Examples

To plot the empirical cumulative distribution function of the series LWAGE:

```
lwage.cdfplot
```

Cross References

See [Chapter 9](#) of the *User’s Guide* for a discussion of empirical distribution graphs.

See [qqplot](#) (p. 288).

censored	Command Equation Method
-----------------	--

Estimation of censored and truncated models.

Estimates models where the dependent variable is either censored or truncated. The allowable specifications include the standard Tobit model.

Syntax

Command: `censored(options) y x1 x2 x3`

Equation Method: `eq_name.censored(options) y x1 x2 x3`

Options

<i>l = number</i> (default = 0)	Set value for the left censoring limit.
<i>r = number</i> (default = none)	Set value for the right censoring limit.
<i>l = series_name, i</i>	Set series name of the indicator variable for the left censoring limit.
<i>r = series_name, i</i>	Set series name of the indicator variable for the right censoring limit.
<i>t</i>	Estimate truncated model.
<i>d = n</i> (default)	Maximize using normal likelihood function.
<i>d = l</i>	Maximize using logistic likelihood function.
<i>d = x</i>	Maximize using (Type I) extreme value likelihood function.
<i>q</i> (default)	Use quadratic hill climbing for maximization algorithm.
<i>r</i>	Use Newton-Raphson for maximization algorithm.
<i>b</i>	Use Berndt-Hall-Hausman for maximization algorithm.
<i>h</i>	Quasi-maximum likelihood (QML) standard errors.
<i>g</i>	GLM standard errors.
<i>m = integer</i>	Set maximum number of iterations.
<i>c = scalar</i>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
<i>s</i>	Use the current coefficient values in C as starting values.
<i>s = number</i>	Specify a number between zero and one to determine starting values as a fraction of EViews default values (out of range values are set to “ <i>s = 1</i> ”).
<i>showopts / -showopts</i>	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
<i>p</i>	Print results.

Examples

The command

```
censored(h) hours c wage edu kids
```

estimates a censored regression model of HOURS on a constant, WAGE, EDU, and KIDS with QML standard errors. This command uses the default normal likelihood, with left-censoring at HOURS = 0 and no right censoring, and the quadratic hill climbing algorithm.

Cross-references

See [Chapter 17](#) of the *User’s Guide* for discussion of censored and truncated regression models.

cfetch	Command
---------------	-------------------------

Fetch a series from the DRI Basic Economics database into a workfile.

`cfetch` reads one or more series from the DRI Basic Economics Database into the active workfile. *You must have the DRI database installed on your computer to use this feature.*

Syntax

Command: **cfetch** series_name

Examples

```
cfetch lhur gdp gnet
```

reads the DRI series LHUR, GDP, and GNET into the current active workfile, performing frequency conversions if necessary.

Cross-references

EViews’ automatic frequency conversion is described in “[Frequency Conversion](#)” beginning on page 72 of the *User’s Guide*.

See also [ccopy \(p. 156\)](#), [clabel \(p. 162\)](#), [store \(p. 347\)](#), [fetch \(p. 205\)](#).

checkderivs	Log View
--------------------	--------------------------

Check derivatives of likelihood object.

Displays a table containing information on the numeric derivatives and user-supplied analytic derivatives (if available).

Syntax

Logl View: `logl_name.checkderiv(options)`

Options

<code>p</code>	Print the table of results.
----------------	-----------------------------

Examples

```
ll1.ml
ll1.checkderiv
```

estimates a likelihood object named LL1 and displays a table that evaluates the numeric derivatives.

Cross-references

See [Chapter 18, “The Log Likelihood \(LogL\) Object”, on page 471](#) for a general discussion of the likelihood object and the “@deriv” statement.

See also [grads](#) (p. 223) and [makegrads](#) (p. 253).

chow	Command Equation View
-------------	--

Chow test for stability.

Carries out Chow breakpoint and Chow forecast tests for parameter constancy.

Syntax

Command: `chow(options) obs1 obs2 obs3`

Equation View: `eq_name.chow(options) obs1 obs2 obs3`

You must provide the breakpoint (dates or observation numbers) to be tested. To specify more than one breakpoint, separate the breakpoints by a space.

Options

<code>f</code>	Chow forecast test. For this option, you must specify a single breakpoint to test (default performs breakpoint test).
----------------	---

<code>p</code>	Print the result of test.
----------------	---------------------------

Examples

The commands

```
ls m1 c gdp cpi ar(1)  
chow 1970:1 1980:1
```

perform a regression of M1 on a constant, GDP, and CPI with first order autoregressive errors, and test whether the parameters before the 1970's, during the 1970's, and after the 1970's are “stable” using the Chow breakpoint test.

To regress the log of SPOT on a constant, log of P_US, and log of P_UK and carry out the Chow forecast test starting from 1973, enter the commands:

```
equation ppp.ls log(spot) c log(p_us) log(p_uk)  
ppp.chow(f) 1973
```

You may also perform Chow tests on cross-section data. Suppose GENDER is a zero-one dummy variable and, when sorted, observations up to 533 have GENDER = 0 and for observations from 534 on GENDER = 1. Then the chow command tests whether the LWAGE equation has the same coefficients for observations with GENDER = 0 and GENDER = 1.

```
sort gender  
ls lwage c edu edu^2 union  
chow 534
```

Cross-references

See “[Chow's Breakpoint Test](#)” on page 380 of the *User's Guide* for further discussion.

See also [rls \(p. 300\)](#).

clabel	Command
---------------	-------------------------

Display a DRI Basic Economics database series description.

clabel reads the description of a series from the DRI Basic Economics Database and displays it in the status line at the bottom of the EViews window.

Use this command to verify the contents of a given DRI database series name. *You must have the DRI database installed on your computer to use this feature.*

Syntax

Command: **clabel** series_name

Examples

```
clabel lhur
```

displays the description of the DRI series LHUR on the status line.

Cross-references

See also [ccopy \(p. 156\)](#), [cfetch \(p. 160\)](#), [read \(p. 291\)](#), [fetch \(p. 205\)](#).

cleartext	Var Proc
------------------	--------------------------

Clear restriction text of var object.

Syntax

Var Proc:	<code>var_name.cleartext(arg)</code>
-----------	--------------------------------------

You must specify the text type you wish to clear as an argument.

Options

One of the following arguments is required:

svar	Clear text for identifying restrictions for a structural VAR.
coint	Clear text for restrictions on the cointegration relations and/or adjustment coefficients.

Examples

```
var1.cleartext(svar)
var1.append(svar) @lr2(@u1)=0
```

The first line clears the structural VAR identifying restrictions in VAR1. The next line specifies a new long-run restriction for a structural factorization.

Cross-references

See also [append \(p. 143\)](#).

close	Command
--------------	-------------------------

Close object, program, or workfile.

Closing an object eliminates its window. If the object is named, it is still displayed in the workfile as an icon, otherwise it is deleted. Closing a program or workfile eliminates its

window and removes it from memory. If a workfile has changed since you activated it, you will see a dialog box asking if you want to save it to disk.

Syntax

Command: `close object_name`

Examples

```
close gdp graph1 table2
```

closes the three objects GDP, GRAPH1, and TABLE2.

```
lwage.hist  
close lwage
```

opens the LWAGE window and displays the histogram view of LWAGE, then closes the window.

Cross-references

See [Chapter 1](#) of the *User's Guide* for a discussion of basic control of EViews.

<code>coef</code>	Object Declaration
-------------------	------------------------------------

Declare a coefficient (column) vector.

Note: vector and coefficient objects are both column vectors. However, only coef elements are allowed in specifications of models or for sspace estimation.

Syntax

Command: `coef(n) coef_name`

Follow the `coef` keyword with the number of coefficients in parentheses and a name for the object. If you omit the number of coefficients, EViews will create a vector of length 1.

Examples

```
coef(2) slope  
ls lwage=c(1)+slope(1)*edu+slope(2)*edu^2
```

The first line declares a coef object of length 2 named SLOPE. The second line runs a least squares regression and stores the estimated slope coefficients in SLOPE.

```
arch(2,2) sp500 c  
coef beta=c  
coef(6) beta
```

The first line estimates a GARCH(2,2) model using the default coef vector C. (The “C” in an equation specification refers to the constant term, a series of ones.) The second line declares a coef object named BETA and copies the contents of C to BETA. (The “C” in the assignment statement refers to the default coef vector). The third line resizes BETA to “chop off” all elements except the first six, which are the estimated parameters from the GARCH(2,2) model. Note that since EViews stores coefficients with equations for later use, you will generally not need to perform this operation to save your coefficient vectors.

Cross-references

See “[Coef](#)” on page 20 for a full description of the coef object.

See also [vector \(p. 377\)](#).

coefcov	Equation View Logl View Pool View Sspace View System View
----------------	---

Coefficient covariance matrix.

Displays the covariances of the coefficient estimates for objects containing an estimated equation or equations.

Syntax

Object View: `object_name.coefcov(options)`

Options

p	Print the coefficient covariance matrix.
---	--

Examples

The set of commands

```
equation eq1.ls lwage c edu edu^2 union
eq1.coefcov
```

declares and estimates an equation and displays the coefficient covariance matrix in a window. To store the coefficient covariance matrix as a (symmetric) matrix object, use “@coefcov”:

```
sym eqcov = eq1.@coefcov
```

Cross-references

See also [coef \(p. 164\)](#) and [spec \(p. 337\)](#).

cointCommand || [Group View](#) | [Var View](#)

Johansen's cointegration test.

Syntax

Command:	<code>coint(test_option,n,option) y1 y2 y3</code>
Command:	<code>coint(test_option,n,option) y1 y2 y3 @ x1 x2 x3</code>
Group View:	<code>group_name.coint(test_option,n,option)</code>
Var View:	<code>var_name.coint(test_option,n,option)</code>

In command form, you should enter the `coint` keyword followed by a list of series or group names within which you wish to test for cointegration. Each name should be separated by a space. To use exogenous variables, such as seasonal dummy variables in the test, list the names after an “@”-sign.

Note: the reported critical values assume no exogenous variables other than an intercept and trend.

When used as a group or var view, `coint` tests for cointegration among the series in the group or var. If the var object contains exogenous variables, the cointegration test will use those exogenous variables. However, if you explicitly list the exogenous variables with an “@”-sign, then only those that are listed will be used in the test.

Options

You must specify the test option followed by the number of lags n in parentheses separated by a comma. You must choose one of the following six test options:

- a No deterministic trend in the data, and no intercept or trend in the cointegrating equation.
- b No deterministic trend in the data, and an intercept but no trend in the cointegrating equation.
- c Linear trend in the data, and an intercept but no trend in the cointegrating equation.
- d Linear trend in the data, and both an intercept and a trend in the cointegrating equation.
- e Quadratic trend in the data, and both an intercept and a trend in the cointegrating equation.
- s Summarize the results of all 5 options (a-e).

restrict	Impose restrictions as specified by the append (coint) proc.
m = <i>integer</i>	Maximum number of iterations for restricted estimation (only valid if you choose the restrict option).
c = <i>scalar</i>	Convergence criterion for restricted estimation. (only valid if you choose the restrict option).
save = <i>mat_name</i>	Stores test statistics as a named matrix object. The save= option stores a $(k + 1) \times 4$ matrix, where k is the number of endogenous variables in the VAR. The first column contains the eigenvalues, the second column contains the maximum eigenvalue statistics, the third column contains the trace statistics, and the fourth column contains the log likelihood values. The i -th row of columns 2 and 3 are the test statistics for rank $i - 1$. The last row is filled with NAs, except the last column which contains the log likelihood value of the unrestricted (full rank) model.

Other Options:

p	Print output of the test.
---	---------------------------

Examples

```
coint(s,4) gdp m1 tb3
```

summarizes the results of the Johansen cointegration test among the three series GDP, M1, and TB3 for all five specifications of trend. The test equation uses lags of up to order four.

```
var1.coint(c,12) @
```

carries out the Johansen test for the series in the var object named VAR1. The “@”-sign without a list of exogenous variables ensures that the test does not include any exogenous variables in VAR1.

Cross-references

See “[Cointegration Test](#)” on page 537 of the *User’s Guide* for details on the Johansen test.

control	Model Proc
---------	----------------------------

Solve for values of control variable so that target series matches trajectory.

Syntax

Model Proc: `model_name.control control_var target_var trajectory`

Specify the name of the control variable, followed by the target variable, and then the trajectory you wish to achieve for the target variable. EViews will solve for the values of the control so that the target equals the trajectory over the current workfile sample.

Examples

```
m1.control myvar targetvar trajvar
```

will put into MYVAR the values that lead the solution of the model for TARGETVAR to match TRAJVAR for the workfile sample.

Cross-references

See “[Solve Control for Target](#)” on page 640 of the *User’s Guide*. See [Chapter 23](#) of the *User’s Guide* for a general discussion of models.

copy	Command
------	-------------------------

Copy an object. Duplicate objects within and across workfiles and databases.

Syntax

Command: `copy(options) source_object_name destination_object_name`

To make copies of the object within the active workfile, list the name of the original object followed by the name for the copy.

To make copies of objects within a database, you can precede the name of the object with the database name and a double colon “::”.

You can also copy objects between a workfile and a database, or between two databases. Simply prefix the object name with the database name and double colon “::”.

You can copy several objects with one command by using the wild card characters “?” (to match any single character) and “*” (to match zero or more characters).

Options

When copying a group object from a workfile to a database

<code>g = s</code>	Copy group definition and series (as separate objects).
<code>g = t</code>	Copy group definition and series (as one object).
<code>g = d</code>	Copy only the series (as separate objects).
<code>g = l</code>	Copy only the group definition.

When copying a group object from a database to a workfile:

<code>g = b</code>	Copy both group definition and series.
<code>g = d</code>	Copy only the series.
<code>g = l</code>	Copy only the group definition.

The following options control the frequency conversion method when copying series and group objects to a workfile, converting from *low* to *high* frequency:

<code>c = r</code>	Conversion by constant match average.
<code>c = d</code>	Conversion by constant match sum.
<code>c = q</code>	Conversion by quadratic match average.
<code>c = t</code>	Conversion by quadratic match sum.
<code>c = i</code>	Conversion by linear match last.
<code>c = c</code>	Conversion by cubic match last.

The following options control the frequency conversion method when copying series and group objects to a workfile, converting from *high* to *low* frequency:

<code>c = a</code>	Conversion by taking the average of the nonmissing observations.
<code>c = s</code>	Conversion by taking the sum of the nonmissing observations.
<code>c = f</code>	Conversion by taking the first nonmissing observation.
<code>c = l</code>	Conversion by taking the last nonmissing observation.
<code>c = x</code>	Conversion by taking the maximum nonmissing observation.
<code>c = m</code>	Conversion by taking the minimum nonmissing observation.

c = an, c = na	Conversion by taking the average, propagating missing values.
c = sn, c = ns	Conversion by taking the sum, propagating missing values.
c = fn, c = nf	Conversion by taking the first observation, propagating missing values.
c = ln, c = nl	Conversion by taking the last observation, propagating missing values.
c = xn, c = nx	Conversion by taking the maximum observation, propagating missing values.
c = mn, c = nm	Conversion by taking the minimum observation, propagating missing values.

Note that if no method is specified, the global or series specific default conversion method will be employed.

Examples

```
copy good_equation best_equation
```

makes a duplicate copy of GOOD_EQUATION named BEST_EQUATION in the current workfile.

```
copy gdp usdat::gdp
```

copies GDP in the current workfile to the database named USDAT with the name GDP. To copy GDP in the default database to the database named MACRO1 with the new name GDP_US, use the command

```
copy ::gdp macro1::gdp_us
```

The command

```
copy ::gdp ::gdp2
```

makes a backup copy of GDP in the default database with a different name GDP2.

```
copy gd* findat::
```

makes a duplicate of all objects in the current workfile with name starting with GD to the database named FINDAT.

Cross-references

See “[Copying Objects](#)” on page 115 of the *User’s Guide* for a discussion of copying and moving objects.

See also [fetch](#) (p. 205), [setconvert](#) (p. 321), and [store](#) (p. 347).

cor	Command Group View Matrix View Sym View
------------	--

Correlation matrix.

Syntax

Command: `cor(options) ser1 ser2 ser3`

Group View: `group_name.cor(options)`

Matrix View: `matrix_name.cor(options)`

In command form, EViews will create an untitled group from the listed series, and then display the correlation matrix view for that group.

When used as a matrix view, `cor` displays the correlation among the columns of the matrix.

Options

- | | |
|---|--|
| i | Compute correlations using pairwise samples (default is to use the common sample). |
| p | Print the correlation matrix. |

Examples

```
cor height weight age
```

displays a 3 by 3 correlation matrix for the three series HEIGHT, WEIGHT, and AGE.

```
group mygroup height weight age
mygroup.cor
```

displays the equivalent view using the group MYGROUP.

Cross-references

See also [cov](#) (p. 175), [@cor](#) (p. 400), and [@cov](#) (p. 400).

correl[Equation View](#) | [Group View](#) | [Series View](#) | [Var View](#)

Correlogram.

Displays the autocorrelation and partial correlation functions of the specified series together with the Q -statistics and p -values associated with each lag.

When used with equation objects, `correl` displays the correlogram of the residuals of the equation.

Syntax

Object View: `object_name.correl(n, options)`

You must specify the largest lag n to compute the autocorrelations.

Options

`p` Print the correlograms.

Var View Options:

`graph` Display correlograms (graphs).

`byser` Display autocorrelations in tabular form by series.

`bylag` Display autocorrelations in tabular form by lag.

Examples

```
m1.correl(24)
```

Displays the correlograms of the M1 series for up to 24 lags.

Cross-references

See [Chapter 7](#) of the *User's Guide* for a discussion of autocorrelation ([p. 167](#)) and partial correlation ([p. 168](#)) functions.

See also [correlnsq](#) ([p. 173](#)).

correlsq	Equation View
-----------------	-------------------------------

Correlogram of squared residuals.

Displays the autocorrelation and partial correlation functions of the squared residuals from an estimated equation, together with the Q -statistics and p -values associated with each lag.

Syntax

View: `equation_name.correl(n, options)`

Options

n Specify the number of lags of the correlograms to display.

p Print the correlograms.

Examples

`eq1.correl(24)`

displays the correlograms of the squared residuals for up to 24 lags, for equation EQ1.

Cross-references

See [Chapter 7](#) of the *User's Guide* for a discussion of autocorrelation ([p. 167](#)) and partial correlation ([p. 168](#)) functions.

See also [correl \(p. 172\)](#).

count	Command Equation Method
--------------	--

Estimates models where the dependent variable is a nonnegative integer count.

Syntax

Command: `count(options) y x1 x2 x3`

Equation Method: `eq_name.count(options) y x1 x2 x3`

Follow the `count` keyword by the name of the dependent variable and a list of regressors, each separated by a space.

Options

d = p (default)	Maximize using Poisson likelihood function.
d = n	Maximize using normal quasi-likelihood function.
d = e	Maximize using exponential quasi-likelihood function.
d = b	Maximize using negative binomial likelihood or quasi-likelihood function.
v = <i>positive number</i> (default = 1)	Specify fixed QML parameter for normal and negative binomial distributions.
q (default)	Use quadratic hill-climbing as the maximization algorithm.
r	Use Newton-Raphson as the maximization algorithm.
b	Use Berndt-Hall-Hausman as the maximization algorithm.
h	Quasi-maximum likelihood (QML) standard errors.
g	GLM standard errors.
m = <i>integer</i>	Set maximum number of iterations.
c = <i>scalar</i>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
s	Use the current coefficient values in C as starting values.
s = <i>number</i>	Specify a number between zero and one to determine starting values as a fraction of the EViews default values (out of range values are set to “s = 1”).

Examples

The command

```
count(d=n,v=2,g) y c x1 x2
```

estimates a normal QML count model of Y on a constant, X1, and X2 with fixed variance parameter 2 and GLM standard errors.

```
equation eq1.count arrest c job police  
eq1.makeresid(g) res_g
```

estimates a Poisson count model of ARREST on a constant, JOB, and POLICE and stores the generalized residuals as RES_G.

```
equation eq1.count(d=p) y c x1
eq1.fit yhat
```

estimates a Poisson count model of Y on a constant and X1, and saves the fitted values (conditional mean) in the series YHAT.

```
equation eq1.count(d=p, h) y c x1
```

estimates the same model with QML standard errors and covariances.

Cross-references

See “[Count Models](#)” on page 458 of the *User’s Guide* for additional discussion.

COV	Command Group View Matrix View Sym View
-----	--

Covariance matrix.

Syntax

Command:	cov(<i>options</i>) ser1 ser2 ser3
Group View:	group_name.cov(<i>options</i>)
Matrix View:	matrix_name.cov(<i>options</i>)

In command form, EViews will create an untitled group from the listed series, and then display the covariance matrix view for that group.

When used as a matrix view, cov displays the covariance among the columns of the matrix.

Options

- | | |
|---|---|
| i | Compute covariances using pairwise samples (default is to use the common sample). |
| p | Print the correlation matrix. |

Examples

```
group grp1 height weight age
grp1.cov
```

displays a 3×3 covariance matrix for the three series in GRP1.

Cross-references

See also [cor \(p. 171\)](#), [@cor \(p. 400\)](#), and [@cov \(p. 400\)](#).

create	Command
--------	-------------------------

Create a new workfile.

Syntax

Command: `create optional_name frequency start end`

You may provide an optional name for your workfile. If you do not, EViews will create an untitled workfile.

You must specify the frequency, and the starting and ending dates of your data. For undated data, you should specify the starting and ending observation numbers.

Options

You must choose one of the following options to specify the frequency of your workfile:

a	Annual
s	Semi-annual
q	Quarterly
m	Monthly
w	Weekly
d	Daily (5 day week)
7	Daily (7 day week)
u	Undated or irregular

Examples

```
create a 1880 90
```

Creates an annual workfile from 1880 to 1990.

```
create m 1990:1 2010:12
```

Creates a monthly workfile from January 1990 to December 2010.

```
create w 2/10/1951 3/17/1994
```

Creates a weekly workfile from the week starting February 10, 1951 to the week starting March 17, 1994.

```
create u 1 5000
```

Creates an undated workfile with 5000 observations.

Cross-references

See “[Creating a Workfile](#)” on page 34 of the *User’s Guide* and [Appendix B](#) of the *User’s Guide* for a discussion of frequencies and rules for composing dates in EViews.

See [workfile \(p. 381\)](#) for a more general command which allows you to either create a workfile or change the active workfile.

cross	Command Group View
--------------	---------------------------------------

Cross correlations.

Displays cross correlations (correlograms) for a pair of series.

Syntax

Command: `cross(n,options) ser1 ser2`

Group View: `group_name.cross(n,options)`

You must specify, as an option, the number of lags *n* to compute the cross correlations as an option. In command form, EViews will create an untitled group and display the cross correlation view for the group. When used as a group view, cross correlations will be computed for the first two series in the group.

Options

The number of lags *n* for which to compute the cross correlations must be specified as the first option. The following options may be specified inside the parentheses after the number of lags:

p	Print the cross correlogram.
---	------------------------------

Examples

```
cross(36) log(m1) dlog(cpi)
```

displays the cross correlogram between the log of M1 and the first difference of the log of CPI up to 36 leads and lags.

```
equation eq1.arch sp500 c  
eq1.makeresid(s) res_std  
cross(24) res_std^2 res_std
```

The first line estimates a GARCH(1,1) model and the second line retrieves the standardized residuals. The third line plots the cross correlogram up to 24 leads and lags between the squared standardized residual and the standardized residual. This correlogram provides a rough check of asymmetry in ARCH effects.

Cross-references

See “[Cross Correlations and Correlograms](#)” on page 221 of the *User’s Guide* for discussion.

data	Command
------	---------

Enter data from keyboard.

Opens an unnamed group window to edit one or more series.

Syntax

Command: **data** ser1 ser2 group1 group2

Follow the **data** keyword by a list of series names and/or a list of group names. You can list existing names or new names. Unrecognized names will cause new series to be added to the workfile. These series will be initialized with the value “NA”.

Examples

```
data group1 newx newy
```

opens a group window containing the series in group GROUP1 and the series NEWX and NEWY.

Cross-references

See “[Entering Data](#)” on page 64 of the *User’s Guide* for a discussion of the process of entering data from the keyboard.

dates	Graph Proc
-------	------------

Control labeling of the bottom date/time axis in time plots.

dates sets options that are specific to appearance of time/date labeling. Many of the options that also affect the appearance of the date axis are set by the [scale \(p. 309\)](#) com-

mand with the “b” option. These options include most tick control, label and font options, and grid lines.

Syntax

Graph Proc: graph_name.dates option list

Options

format(dateformat [,delimiter])	dateformat = “auto” or dateformat = “string”, where the string argument is one of the supported formats: “yy”, “yyyy:mm”, “yyyy:q”, “yy:q”, “mm:dd:yyyy”, “mm:dd:yy”. Each letter represents a single display digit representing year (“y”), quarter (“q”), month (“m”), and day (“d”). The optional delimiter argument allows you to control the date separator.
interval(step size [,steps][,align date])	where step size takes one of the following values: “auto” (steps and align date are ignored), “ends” (only label endpoints; steps and align date are ignored), “all” (label every point; the <i>steps</i> and <i>align_date</i> options are ignored), “obs” (steps are one month), “year” (steps are one year), “m” (steps are one month), “q” (steps are one quarter). <i>steps</i> is a number (default = 1) indicating the number of steps between labels. <i>align_date</i> is a date specified to receive a label. Note, the <i>align date</i> should be in the units of the data being graphed, but may lie outside the current sample or workfile range.
minor/-minor	[Allow/Do not allow] minor tick marks.
span/-span	[Allow/Do not allow] date labels to span an interval. Consider the case of a yearly label with monthly ticks. If <i>span</i> is on, the label is centered on the 12 monthly ticks. If the <i>span</i> option is off, year labels are put on the first quarter or month of the year.
p	Print graph object.

Examples

```
graph1.dates format(yyyy:mm)
```

will display dates with four-digit years followed by the default delimiter “:” and a two-digit month (e.g. – “1974:04”).

```
graph1.dates format(yy:mm, q)
```

will display a two-digit year followed by a “q” separator and then a two-digit month (e.g. – “74:04”)

```
graph1.interval(y, 2, 1951)
```

specifies labels every two years on odd numbered years.

Cross-references

See [Chapter 10](#) of the *User’s Guide* for a discussion of graph options.

See also [scale \(p. 309\)](#).

db	Command
-----------	-------------------------

Open or create a database.

If the specified database does not exist, a new (empty) database will be created and opened. The opened database will become the default database.

Syntax

Command: **db db_name**

Follow the **db** command by the name of the database to be opened or to be created (if it does not already exist). You can include a path name to work with a database not in the default path.

Options

See [dbopen \(p. 183\)](#) for a list of available options for working with foreign format databases.

Examples

```
db findat
```

opens the database FINDAT in the default path and makes it the default database from which to store and fetch objects. If the database FINDAT does not already exist, an empty database named FINDAT will be created and opened.

Cross-references

See [Chapter 6](#) of the *User's Guide* for a discussion of EViews databases.

See also [dbcreate \(p. 181\)](#) and [dbopen \(p. 183\)](#).

dbcopy	Command
---------------	-------------------------

Make a copy of an existing database.

Syntax

Command: **dbcopy** source_name copy_name

Follow the **dbcopy** command by the name of the existing database and a name for the copy. You should include a path name to copy a database that is not in the default directory. All files associated with the database will be copied.

Examples

```
dbcopy usdat c:\backup\usdat
```

makes a copy of all files associated with the database USDAT in the default path and stores it in the C:\BACKUP directory under the name USDAT.

Cross-references

See [Chapter 6](#) of the *User's Guide* for a discussion of EViews databases.

See also [dbrename \(p. 185\)](#) and [dbdelete \(p. 182\)](#).

dbcreate	Command
-----------------	-------------------------

Create a new database.

Syntax

Command: **dbcreate** db_name

Follow the **dbcreate** keyword by a name for the new database. You can include a path name to create a database not in the default directory. The new database will become the default database.

Examples

```
dbccreate macrodat
```

creates a new database named MACRODAT in the default path and makes it the default database from which to store and fetch objects. This command will issue an error message if a database named MACRODAT already exists. To open an existing database, use “dbopen” or “db”.

Options

See [dbopen \(p. 183\)](#) for available options for working with foreign format databases.

Cross-references

See [Chapter 6](#) of the *User’s Guide* for a discussion of EViews databases.

See also [db \(p. 180\)](#) and [dbopen \(p. 183\)](#).

dbdelete	Command
-----------------	-------------------------

Delete an existing database.

dbdelete deletes all files associated with the specified database.

Syntax

Command: dbdelete db_name

Follow the dbdelete keyword by the name of the database to be deleted. You can include a path name to delete a database not in the default path.

Options

See [dbopen \(p. 183\)](#) for available options for working with foreign format databases.

Examples

```
dbdelete c:\temp\testdat
```

deletes all files associated with the TESTDAT database in the specified directory.

Cross-references

See [Chapter 6](#) of the *User’s Guide* for a discussion of EViews databases.

See also [dbcopy \(p. 181\)](#) and [dbdelete \(p. 182\)](#).

dbopen	Command
---------------	-------------------------

Open an existing database.

Syntax

Command: **dbopen db_name**

Follow the **dbopen** keyword with the name of a database. You should include a path name to open a database not in the default path. The opened database will become the default database.

Examples

```
dbopen c:\data\us1
```

opens a database named US1 in the C:\DATA directory. The command

```
dbopen us1
```

opens a database in the default path.

If the specified database does not exist, EViews will issue an error message. You should use **db** or **dbcreate** to create a new database.

Options

To read a foreign format database, you must specify one of the following database types:

t = database_type	Specify the database type: “a” (AREMOS-tsd), “b” (dRIBase), “e” (EViews), “f” (FAME), “g” (GiveWin/PcGive), “h” (Haver Analytics), “l” (RATS portable), “r” (RATS 4.x), “t” (TSP portable)
--------------------------	--

The following options may be required when connecting to a remote server:

s = server_id

u = username

p = password

Cross-references

See [Chapter 6](#) of the *User’s Guide* for a discussion of EViews databases.

See also [db](#) (p. 180) and [dbcreate](#) (p. 181)

dbpack[Command](#)

Pack an existing database.

Syntax

Command: **dbpack db_name**

Follow the **dbpack** keyword by a database name. You can include a path name to pack a database not in the default path.

Examples

```
dbpack findat
```

Packs the database named FINDAT in the default path.

Cross-references

See “[Packing the Database](#)” on page 134 of the *User’s Guide* for additional discussion.

See also [dbrebuild \(p. 184\)](#) and [dbrepair \(p. 185\)](#)

dbrebuild[Command](#)

Rebuild an existing database.

Rebuild a seriously damaged database into a new database file.

Syntax

Command: **dbrebuild source_name dest_name**

Follow the **dbrebuild** keyword by the name of the database to be rebuilt, and then a new database name.

Examples

If you issue the command

```
dbrebuild testdat fixed_testdat
```

EViews will attempt to rebuild the database TESTDAT into the database FIXED_TESTDAT in the default directory.

Cross-references

Note that dbrepair may be able to repair the existing database if the damage is not particularly serious. You should attempt to repair the database *before* rebuilding. See “[Maintaining the Database](#)” on page 133 of the *User’s Guide* for a discussion.

See also [dbpack](#) (p. 184) and [dbrepair](#) (p. 185).

dbrename	Command
-----------------	-------------------------

Rename an existing database.

dbrename renames all files associated with the specified database.

Syntax

Command: **dbrename old_name new_name**

Follow the dbrename keyword with the current name of an existing database and the new name for the database.

Options

See [dbopen](#) (p. 183) for available options for working with foreign format databases.

Examples

```
dbrename testdat mydat
```

Renames all files associated with the TESTDAT database in the specified directory to MYDAT in the default directory.

Cross-references

See [Chapter 6](#) of the *User’s Guide* for a discussion of EViews databases.

See also [db](#) (p. 180) and [dbccreate](#) (p. 181). See also [dbcopy](#) (p. 181) and [dbdelete](#) (p. 182).

dbrepair	Command
-----------------	-------------------------

Repair an existing database.

This command is used to repair a damaged database.

Syntax

Command: **dbrepair db_name**

Follow the **dbrepair** keyword by the name of the database to repair. You should include a path name to repair a database not in the default path.

Examples

```
dbrepair testdat
```

EViews will attempt to repair the database TESTDAT in the default directory.

Cross-references

If the database is severely damaged, you may have to rebuild it into a new database using the **dbrebuild** command. See “[Maintaining the Database](#)” on page 133 of the *User’s Guide* for a discussion of EViews database maintenance.

See also [dbpack](#) (p. 184) and [dbrebuild](#) (p. 184).

decomp	Var View
---------------	--------------------------

Variance decomposition in VARs.

Syntax

Var View: **var_name.decomp(*n*, *options*) series_list @ @ ordering**

List the series names in the VAR whose variance decomposition you would like to compute. You may optionally specify the ordering for the factorization after two “@”-signs.

You must specify the number of periods *n* over which to compute the variance decompositions.

Options

In addition, there are the following options:

g (default)	Display combined graphs, with the decompositions for each variable shown in a graph.
m	Display multiple graphs, with each response-shock pair shown in a separate graph.
t	Show numerical results in table.

imp = <i>arg</i> (default “imp = chol”)	Type of factorization for the decomposition: Cholesky with d.f. correction (“imp = chol”), Cholesky without d.f. correction (“imp = mlechol”), or structural (“imp = struct”). The structural factorization is based on the estimated structural VAR. To use this option, you must first estimate the structural decomposition; see svar (p. 351) . The option “imp = mlechol” is provided for backward compatibility with EViews 3.x and earlier.
se = mc	Monte Carlo standard errors. Must specify the number of replications with the “rep = ” option. Currently available only when you have specified the Cholesky factorization (using the “imp = chol” option).
rep = <i>integer</i>	Number of Monte Carlo replications to be used in computing the standard errors. Must be used with the “se = mc” option.
matbys = <i>name</i>	Save responses by shocks (impulses) in named matrix. The first column is the response of the first variable to the first shock, the second column is the response of the second variable to the first shock, and so on.
matbyr = <i>name</i>	Save responses by response series in named matrix. The first column is the response of the first variable to the first shock, the second column is the response of the first variable to the second shock, and so on.
p	Print results.

If you use the “matbys = ” or “matbyr = ” options to store the results in a matrix, two matrices will be returned. The matrix with the specified name contains the variance decompositions, while the matrix with “_FSE” appended to the name contains the forecast standard errors for each response variable. If you have requested Monte Carlo standard errors, there will be a third matrix with “_SE” appended to the name which contains the variance decomposition standard errors.

Examples

```
var var1.ls 1 4 m1 gdp cpi
var1.decomp(10,t) gdp
```

The first line declares and estimates a VAR with three variables and lags from 1 to 4. The second line tabulates the variance decompositions of GDP up to 10 periods using the ordering as specified in VAR1.

```
var1.decomp(10,t) gdp @@ cpi gdp m1
```

performs the same variance decomposition as above using a different ordering.

Cross-references

See “[Variance Decomposition](#)” on page 529 of the *User’s Guide* for additional details.

See also [impulse](#) (p. 232).

define	Pool View
---------------	---------------------------

Define cross section members (identifiers) in a pool.

Syntax

Pool View: `pool_name.define id1 id2 id3`

List the cross section identifiers after the `define` keyword.

Examples

```
pool spot uk japn ger can  
spot.def uk ger ita fra
```

The first line declares a pool object named SPOT with cross section identifiers UK, JAPN, GER, and CAN. The second line redefines the identifiers to be UK, GER, ITA, and FRA.

Cross-references

See [Chapter 21](#) of the *User’s Guide* for a discussion of cross-section identifiers.

See also [add](#) (p. 137), [drop](#) (p. 195) and [pool](#) (p. 285).

delete	Command Pool Proc
---------------	--

Deletes objects from a workfile or a database, or removes identifiers from a pool.

Syntax

Command: `delete name1 name2`

Pool Proc: `pool_name.delete ser1? ser2?`

Follow the `delete` keyword by a list of the names of any objects you wish to remove from the current workfile. `delete` does *not* remove objects that have been stored on disk in EViews database files.

You can delete an object from a database by prefixing the name with the database name and a double colon. You can use a pattern to delete all objects from a workfile or database with names that match the pattern. Use the “?” to match any one character and the “*” to match zero or more characters.

When used as a pool procedure, `delete` allows you to delete series from the workfile using the pool operator “?”.

If you use `delete` in a program file, EViews will delete the listed objects without prompting you to confirm each deletion.

Examples

To delete all objects in the workfile with names beginning with “temp”:

```
delete temp*
```

To delete the objects CONS and INVEST from the database MACRO1:

```
delete macro1::cons macro1::invest
```

To delete all series in the workfile with names beginning with CPI that are followed by identifiers in the pool object MYPOOL.

```
mypool.delete cpi?
```

Cross-references

See “[Object Basics](#)” on page 41 of the *User’s Guide* for a discussion of working with objects, and [Chapter 6](#) of the *User’s Guide* for a discussion of EViews databases.

derivs	Equation View System View
--------	---

Examine derivatives of the equation specification.

Display information about the derivatives of the equation specification in tabular, graphical or summary form.

The (default) summary form shows information about how the derivative of the equation specification was computed, and will display the analytic expression for the derivative, or a note indicating that the derivative was computed numerically. The tabular form shows a spreadsheet view of the derivatives of the regression specification with respect to each

coefficient for each observation. The graphical form shows this information in a multiple line graph.

Syntax

Equation View: `equation_name.derivs(options)`

Options

<code>g</code>	Display multiple graph showing the derivatives of the equation specification with respect to the coefficients, evaluated at each observation.
<code>t</code>	Display spreadsheet view of the values of the derivatives with respect to the coefficients evaluated at each observation.
<code>p</code>	Print results.

Note that the “g” and “t” options may not be used at the same time.

Examples

To show a table view of the derivatives:

```
eq1.derivs(t)
```

To display and print the summary view

```
eq1.derivs(p)
```

Cross-references

See “[Derivative Computation Options](#)” on page 670 of the *User’s Guide* for details on the computation of derivatives.

See also [makederivs \(p. 251\)](#) for additional routines for examining derivatives, and [grads \(p. 223\)](#), and [makegrads \(p. 253\)](#) for corresponding routines for gradients.

describe	Pool View
-----------------	---------------------------

Computes and displays descriptive statistics for the pooled data.

Syntax

Pool View: `pool_name.describe(options) ser1? ser2`

List the name of pool series for which to compute the descriptive statistics. You may use the cross-section identifier “?” in the series names.

By default, statistics are computed for each pool series, on the stacked data, using only observations where *all* of the listed series have nonmissing data. A missing observation in any one series causes that observation to be dropped for all series. You may change this default treatment of NAs using the “i” and “b” options.

EViews also allows you to compute statistics with the cross-section means removed, statistics for each cross-sectional series in a pool series, and statistics for each period, taken across all cross-section units.

Options

- | | |
|---|---|
| m | Stack data and subtract cross-section specific means from each variable—this option provides the within estimators. |
| c | Do not stack data—compute statistics individually for each cross-sectional unit. |
| t | Time period specific—compute statistics for each period, taken over all cross-section identifiers. |
| i | Individual sample—includes every valid observation for the series even if data are missing from other series in the list. |
| b | Balanced sample—constraints each cross-section to have the <i>same observations</i> . If an observation is missing for any series, in any cross-section, it will be dropped for all cross-sections. |
| p | Print the descriptive statistics. |

Examples

```
pool1.describe(m) gdp? inv? cpi?
```

displays the “within” descriptive statistics of the three series GDP, INV, CPI for the POOL1 cross-section members.

```
pool1.describe(t) gdp?
```

computes the statistics for GDP for each period, taken across each of the cross-section identifiers.

Cross-references

See [Chapter 21](#) of the *User’s Guide* for a discussion of the computation of these statistics, and a description of individual and balanced samples.

displayname	Object Proc
-------------	-------------

Display names for objects.

Attaches a display name to an object which may be used in tables and graphs in place of the standard object name.

Syntax

Object Proc: `object_name.displayname display_name`

Display names are case-sensitive, and may contain a variety of characters, such as spaces, that are not allowed in object names.

Examples

```
hrs.displayname Hours Worked  
hrs.label
```

The first line attaches a display name “Hours Worked” to the object HRS and the second line displays the label view of HRS, including its display name.

```
gdp.displayname US Gross Domestic Product  
plot gdp
```

The first line attaches a display name “US Gross Domestic Product” to the series GDP. The line graph view of GDP from the second line will use the display name as the legend.

Cross-references

See [“Labeling Objects” on page 50](#) of the *User’s Guide* for a discussion of labels and display names.

See also [label \(p. 238\)](#) and [legend \(p. 240\)](#)

do	Command
----	---------

Execute without opening window.

Syntax

Command: `do procedure`

`do` is most useful in EViews programs where you wish to run a series of commands without opening windows in the workfile area.

Examples

```
output(t) c:\result\junk1
do gdp.adf(c,4,p)
```

The first line redirects table output to a file on disk. The second line carries out a unit root test of GDP and prints the results to the disk file.

Cross-references

See also [show \(p. 328\)](#).

draw	Graph Proc
-------------	----------------------------

Place horizontal or vertical lines and shaded areas on the graph.

Syntax

Graph Proc:	<code>graph_name.draw(object_type, axis_id [,options]) position1 [position2]</code>
-------------	---

where `object_type` may be one of the following:

line, l	A solid line
dashline, d	A dashed line
shade	A shaded area

and where `axis_id` may take the values:

left, l	Draw a horizontal line or shade using the left axis to define the drawing position
right, r	Draw a horizontal line or shade using the right axis to define the drawing position
bottom, b	Draw a vertical line or shade using the bottom axis to define the drawing position

If drawing a line, the drawing position is taken from `position1`. If drawing a shaded area, you must provide a `position1` and `position2` to define the vertical or horizontal boundaries of the shaded region.

Options

rgb(<i>n1,n2,n3</i>)	where <i>n1</i> , <i>n2</i> , and <i>n3</i> , are integers from 0 to 255 representing the RGB values of the line or shade. The default is black for lines and gray for shades. RGB values may be examined by calling up the color palette in the Graph Options dialog.
width(<i>n1</i>)	where <i>n1</i> is the line width in points (used only if object_type is “line” or “dashline”). The default is 0.5 points
p	Print the graph object

Examples

The command

```
graph1.draw(line, left, rgb(0,0,127)) 5.25
```

draws a horizontal blue line at the value “5.25” as measured on the left axis while

```
graph1.draw(shade, right) 7.1 9.7
```

draws a shaded horizontal region bounded by the right axis values “7.1” and “9.7”. You can also draw vertical regions by using the “bottom” axis_id:

```
graph1.draw(shade, bottom) 1980:1 1990:2
```

draws a shaded vertical region bounded by the dates “1980:1” and “1990:2”.

Cross-references

See [Chapter 10](#) of the *User’s Guide* for a discussion of graph options.

See also “[Graph](#)” (p. 25) for a summary of the graph object command language.

driconvert	Command
------------	-------------------------

Convert the entire DRI Basic Economics database into an existing EViews database.

You must create an EViews database to store the converted DRI data *before* you use this command. This command may be very time-consuming.

Syntax

Command: **driconvert db_name**

Follow the command by listing the name of an existing EViews database into which you would like to copy the DRI data. You can include a path name to specify a database not in the default path.

Examples

```
dbcreate dribasic
driconvert dribasic
driconvert c:\mydata\dribase
```

The first line creates a new (empty) database named DRIBASIC in the default directory. The second line reads all the data in the DRI Basic Economics database and copies them into in the DRIBASIC database. The last example copies the DRI data into the database DRIDBASIC that is located in the C:\MYDATA directory.

Cross-references

See [Chapter 6](#) of the *User's Guide* for a discussion of EViews databases.

See also [dbcreate \(p. 181\)](#) and [db \(p. 180\)](#).

drop	Group Proc Pool Proc
-------------	--

Drops series from a group or drop cross-section members from a pool.

Syntax

Group Proc:	group_name.drop ser1 ser2 ser3
Pool Proc:	pool_name.drop id1 id2 id3

List the series or cross-section members to be dropped from the group or pool.

Examples

```
group gdplags gdp(-1 to -4)
gdplags.drop gdp(-4) gdp(-3)
```

drops the two series GDP(-4) and GDP(-3) from the group GDPLAGS.

To drop the cross-section members JPN, KOR, and HK from the pool CROSSSC

```
crosssc.drop jpn kor hk
```

Cross-references

See also [add \(p. 137\)](#).

dtable**Group View**

Dated data report table.

This group view is designed to make tables for reporting and presenting data, forecasts, and simulation results. You can display various transformations and various frequencies of the data in the same table.

The **dtable** view is currently available only for annual, semi-annual, quarterly, or monthly workfiles.

Syntax

Group View: `group_name.dtable(options)`

Options

p Print the report table.

Examples

```
freeze(report) group1.table
```

freezes the default table view of GROUP1 and saves it as a table object named REPORT.

Cross-references

See “[Creating and Specifying a Dated Data Table](#)” on page 201 of the *User’s Guide* for a description of dated data tables and formatting options. Note that most of the options for formatting the table are only available interactively from the window.

ec**Var Method**

Estimate a vector error correction model (VEC).

Syntax

Var Method: `var_name.ec(trend, n) lag_pairs y1 y2`
`var_name.ec(trend, n) lag_pairs y1 y2 @ x1 x2`
`var_name.ec(trend, n) lag_pairs y1 y2`

Specify the order of the VEC by entering one or more pairs of lag intervals, then list the endogenous variables. *Note that the lag orders are those of the first differences, not the levels.*

els. If you are comparing results to another software program, you should be certain that the specifications for the lag orders are comparable.

You may include exogenous variables, such as seasonal dummies, in the VEC by including an “@”-sign followed by the list of series. *Do not include an intercept or trend* in the VEC specification, these terms should be specified using options, as described below.

You must specify the trend option and the number of cointegrating equations n to use (default is $n = 1$) in parentheses, separated by a comma. You must choose the trend from the following five alternatives:

a	No deterministic trend in the data, and no intercept or trend in the cointegrating equation.
b	No deterministic trend in the data, and an intercept but no trend in the cointegrating equation.
c (default)	Linear trend in the data, and an intercept but no trend in the cointegrating equation.
d	Linear trend in the data, and both an intercept and a trend in the cointegrating equation.
e	Quadratic trend in the data, and both an intercept and a trend in the cointegrating equation.
restrict	Impose restrictions as specified by the <code>append (coint) proc</code>
<i>m = integer</i>	Maximum number of iterations for restricted estimation (only valid if you choose the restrict option).
<i>c = scalar</i>	Convergence criterion for restricted estimation. (only valid if you choose the restrict option).

Options

p	Print the results view.
---	-------------------------

Examples

```
var macro1.ec 1 4 m1 gdp tb3
```

declares a var object MACRO1 and estimates a VEC with four lagged first differences, three endogenous variables and one cointegrating equation using the default trend option “c”.

```
var term.ec(b,2) 1 2 4 4 tb1 tb3 tb6 @ d2 d3 d4
```

declares a var object TERM and estimates a VEC with lagged first differences of order 1, 2, 4, three endogenous variables, three exogenous variables, and two cointegrating equations using trend option “b”.

Cross-references

See “[Vector Error Correction \(VEC\) Models](#)” on page 547 of the *User’s Guide* for a discussion of VECs.

See also [var \(p. 376\)](#) and [coint \(p. 166\)](#).

edftest	Series View
----------------	-----------------------------

Computes goodness-of-fit tests based on the empirical distribution function.

Syntax

Series View: series_name.edftest(*options*)

Options

General Options

type = *arg* Normal distribution (“type = normal”, default)

Chi-square distribution (“type = chisq”)

Exponential (“type = exp”)

Extreme Value – Type I maximum (“type = xmax”)

Extreme Value – Type I minimum (“type = xmin”)

Gamma (“type = gamma”)

Logistic (“type = logit”)

Pareto (“type = pareto”)

Uniform (“type = uniform”)

p1 = *number* Specify the value of the first parameter (as it appears in the dialog). If this option is not specified, the first parameter will be estimated

<i>p2 = number</i>	Specify the value of the second parameter (as it appears in the dialog). If this option is not specified, the first parameter will be estimated
<i>p3 = number</i>	Specify the value of the third parameter (as it appears in the dialog). If this option is not specified, the first parameter will be estimated
<i>p</i>	Print test results.

Estimation Options

The following options apply if iterative estimation of parameters is required:

<i>b</i>	Berndt-Hall-Hausman (BHHH) algorithm. Default is Marquardt.
<i>m = integer</i>	Maximum number of iterations.
<i>c = number</i>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
<i>showopts / -showopts</i>	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
<i>s</i>	Take starting values from the C coefficient vector. By default, EViews uses distribution specific starting values that typically are based on the method of the moments.

Examples

```
x.edftest
```

uses the default settings to test whether the series X comes from a normal distribution. Both the location and scale parameters are estimated from the data in X.

```
freeze(tab1) x.edftest(type=chisq, p1=5)
```

tests whether the series x comes from a χ^2 distribution with 5 degrees of freedom. The output is stored as a table object TAB1.

Cross-references

See “[Empirical Distribution Tests](#)” on page 164 of the *User’s Guide* for a description of the goodness-of-fit tests.

See also [qqplot \(p. 288\)](#).

endog[Sspace View](#) | [System View](#) | [Var View](#)

Displays a spreadsheet or graph view of the endogenous variables.

Syntax

Object View: object_name.endog(*options*)

Note that in EViews 4, `endog` and `makeendog` are no longer supported for model objects. See instead, [makegroup \(p. 255\)](#).

Options

g Multiple line graphs of the solved endogenous series.

p Print the table of solved endogenous series.

Examples

```
sys1.endog(g,p)
```

prints the graphs of the solved endogenous series.

Cross-references

See also [makeendog \(p. 251\)](#), [system \(p. 354\)](#), [sspace \(p. 339\)](#) and [var \(p. 376\)](#).

equation[Object Declaration](#)

Declare an equation object.

Syntax

Command: **equation** eq_name

Command: **equation** eq_name.method(*options*) *specification*

Follow the `equation` keyword with a name and an optional specification. If you wish to enter the specification, you should follow the new equation name with a period, an estimation method, and then the equation specification. Valid estimation methods are `arch`, `binary`, `censored`, `count`, `gmm`, `ls`, `ordered`, and `tsls`.

Examples

```
equation cobdoug.ls log(y) c log(k) log(l)
```

declares and estimates an equation object named COBDOUG.

The command

```
equation ces.ls log(y)=c(1)*log(k^c(2)+l^c(3))
```

declares an equation object named CES containing a nonlinear least squares specification.

```
equation demand.tsls q c p x @ x p(-1) gov
```

creates an equation object named DEMAND and estimates DEMAND using two-stage least squares with instruments X, lagged P, and GOV.

Cross-references

See the object reference entry for “[Equation](#)” (p. 21) for a summary of the equation object. [Chapter 11](#) of the *User’s Guide* provides basic information on estimation and equation objects. Additional estimation methods are described in the *User’s Guide*.

eqs	Model View
---------------------	----------------------------

View of model organized by equation.

Lists the equations in the model allowing you to access the equation specifications. This view also allows you to identify which equations are entered by text, or by link.

Syntax

Model View: `model_name.eq`

Cross-references

See “[Equation View](#)” on page 620 of the *User’s Guide* for details. See [Chapter 23](#) of the *User’s Guide* for a general discussion of models.

See also [block](#) (p. 154), [text](#) (p. 363), and [vars](#) (p. 377) for alternative representations of the model.

errbar	Coef View Graph Proc Group View Matrix View Sym View
------------------------	--

Display error bar graph.

Sets the graph type to error bar or displays an error bar view of the group. If there are two series in the graph or group, the error bar will show the high and low values in the bar. The optional third series will be plotted as a symbol. When used as a matrix view, the columns of the matrix replace the series.

Syntax

Graph Proc: `graph_name.errbar(options)`
Object View: `object_name.errbar(options)`

Options

Template and printing options

<code>o = graph_name</code>	Use appearance options from the specified graph.
<code>t = graph_name</code>	Use appearance options and copy text and shading from the specified graph.
<code>p</code>	Print the error bar graph.

Examples

The following set of commands

```
group g1 x y  
g1.errbar
```

displays the error bar view of G1 using the X series as the high value of the bar and the Y series as the low value.

```
group g2 plus2se minus2se estimate  
g2.errbar
```

displays the error bar view of G2 with the PLUS2SE series as the high value of the bar, the MINUS2SE series as the low value, and ESTIMATE as a symbol.

```
group g1 x y  
freeze(graph1) g1.line  
graph1.errbar
```

first creates a graph object GRAPH1 containing a line graph of the series in G1, then changes the graph type to an error bar.

Cross-references

See “[High-Low \(Open-Close\)](#)” on page 210 of the *User’s Guide*. See [Chapter 10](#) for details on graph objects and types.

See also [graph](#) (p. 224) for additional graph types.

exit	Command
-------------	-------------------------

Exit from EViews. Closes the EViews application.

You will be prompted to save objects and workfiles which have changed since the last time they were saved to disk. Be sure to save your workfile, if desired, since all changes that you do not save to a disk file will be lost.

Syntax

Command: **exit**

Cross-references

See also [close \(p. 163\)](#) and [save \(p. 308\)](#).

exclude	Model Proc
----------------	----------------------------

Specifies (or merges) excluded endogenous variables in the active scenario.

Syntax

Model Proc: `model_name.exclude(options) ser1(smpl) ser2(smpl) ...`

Follow the `exclude` keyword with the argument list containing the endogenous variables you wish to exclude from the solution along with an optional sample for exclusion. If a sample is not provided, the variable will be excluded for the entire solution sample.

Options

<code>m</code>	Merge into instead of replace the existing exclude list.
----------------	--

Examples

```
mod1.exclude fedfunds govexp("1990:01 1995:02")
```

will create an exclude list containing the variables FEDFUNDS and GOVEXP. FEDFUNDS will be excluded for the entire solution sample, while GOVEXP will only be excluded for the specified sample.

If you then issue the command

```
mod1.exclude govexp
```

EViews will replace the original exclude list with one containing only GOVEXP. To add excludes to an existing list, use the “m” option:

```
mod1.exclude(m) fedfunds
```

The excluded list now contains both GOVEXP and FEDFUNDs.

Cross-references

See the discussion in “[Specifying Scenarios](#)” on page 625 of the *User’s Guide*.

See also [model](#) (p. 269), [override](#) (p. 280) and [solveopt](#) (p. 335).

expand	Command
--------	-------------------------

Expands the current workfile (range) by setting new start and end dates or number of observations.

Syntax

Command: **expand** start end

Follow the **expand** keyword with the new starting and ending dates or observations. The new starting point cannot be later than the current starting point and the new ending point cannot be earlier than the current ending point.

The [range](#) (p. 290) command may be used for more general workfile resizing which also allows for contracting.

Examples

```
workfile mywork m 1957:1 1995:12
expand 1945:1 2020:12
```

The first line creates a monthly workfile from January 1957 to December 1995. The second line expands the workfile to start from January 1945 and to end at December 2020.

```
workfile cps88 u 1 5000
expand 1 50000
```

Creates an undated workfile with 5000 observations and then expands it to 50000 observations.

```
smp1 1945 1995
equation eq1.ls y c y(-1) ma(1)
expand 1945 2010
smp1 1996 2010
```

```
eq1.forecast yhat
```

The first two lines estimate a model of Y over the period 1945 to 1995. The third line expands the workfile to 2010. We then generate dynamic forecasts of Y for the period 1996 to 2010, using the estimated model.

Cross-references

See “[Workfile Basics](#)” on page 33 of the *User’s Guide* for a discussion of workfiles.

See also [range \(p. 290\)](#), [workfile \(p. 381\)](#), [smp1 \(p. 332\)](#).

fetch	Command Pool Proc
--------------	--

Fetch objects from databases or databank files into the workfile.

`fetch` reads one or more objects from EViews databases or databank files into the active workfile. The objects are loaded into the workfile under the same name as in the database or with the databank file name.

When used as a pool proc, EViews will first expand the list of series using the pool operator, and then perform the fetch.

If you fetch a series into a workfile with a different frequency, EViews will automatically apply the frequency conversion method attached to the series by `setconvert`. If the series does not have a conversion method set by `setconvert`, EViews will use the method set by **Options/Date-Frequency** in the main menu. You can override the conversion method by explicitly specifying a conversion method option in the `fetch` command.

Syntax

Command: `fetch(options) object_name_list`

Pool Proc: `pool_name.fetch(options) ser1? ser2?`

The `fetch` command keyword is followed by a list of object names separated by spaces. The default behavior is to fetch the objects from the default database. (*This is a change from versions of EViews prior to EViews 3.x where the default was to fetch from individual databank files*).

You can precede the object name with a database name and the double colon “::” to indicate a specific database source. If you specify the database name as an option in parentheses (see below), all objects without an explicit database prefix will be fetched from the specified database.

You may use wild card characters, “?” (to match a single character) or “*” (to match zero or more characters), in the object name list. All objects with names matching the pattern will be fetched.

You can optionally fetch from individual databank files or search among registered databases. To fetch from individual databank files that are not in the default path, you should include an explicit path. If you have more than one object with the same file name (for example, an equation and a series named CONS), then you should supply the full object file name.

Options

<code>d = db_name</code>	Fetch from specified database.
<code>d</code>	Fetch all registered databases in registry order.
<code>i</code>	Fetch from individual databank files.

The following options are available for fetch of group objects:

<code>g = b</code>	Fetch both group definition and series.
<code>g = d</code>	Fetch only the series in the group.
<code>g = l</code>	Fetch only the group definition.

The database specified by the double colon “::” takes precedence over the database specified by the “d = ” option.

In addition, there are a number of additional options for controlling automatic frequency conversion when performing a fetch. The following options control the frequency conversion method when copying series and group objects to a workfile, converting from *low* to *high* frequency:

<code>c = r</code>	Conversion by constant match average.
<code>c = d</code>	Conversion by constant match sum.
<code>c = q</code>	Conversion by quadratic match average.
<code>c = t</code>	Conversion by quadratic match sum.
<code>c = i</code>	Conversion by linear match last.
<code>c = c</code>	Conversion by cubic match last.

The following options control the frequency conversion method when copying series and group objects to a workfile, converting from *high* to *low* frequency:

c = a	Conversion by taking the average of the nonmissing observations.
c = s	Conversion by taking the sum of the nonmissing observations.
c = f	Conversion by taking the first nonmissing observation.
c = l	Conversion by taking the last nonmissing observation.
c = x	Conversion by taking the maximum nonmissing observation.
c = m	Conversion by taking the minimum nonmissing observation.
c = an, c = na	Conversion by taking the average, propagating missing values.
c = sn, c = ns	Conversion by taking the sum, propagating missing values.
c = fn, c = nf	Conversion by taking the first observation, propagating missing values.
c = ln, c = nl	Conversion by taking the last observation, propagating missing values.
c = xn, c = nx	Conversion by taking the maximum observation, propagating missing values.
c = mn, c = nm	Conversion by taking the minimum observation, propagating missing values.

Note that if no method is specified, the global or series specific default conversion method will be employed.

Examples

To fetch M1, GDP, UNEMP from the default database, use:

```
fetch m1 gdp unemp
```

To fetch M1 and GDP from the US1 database and UNEMP from the MACRO database, use the command:

```
fetch(d=us1) m1 gdp macro::unemp
```

You can fetch all objects with names starting with SP by searching all registered databases in the search order. The “c = f” option uses the first (nonmissing) observation to convert the frequency of any matching series with a higher frequency than the workfile frequency:

```
fetch(d,c=f) sp*
```

You can fetch M1 and UNEMP from individual databank files using:

```
fetch(i) m1 c:\data\unemp
```

To fetch all objects with names starting with CONS from the two databases USDAT and UKDAT:

```
fetch usdat::cons* ukdat::cons*
```

The command

```
fetch a?income
```

will fetch all series beginning with the letter “a”, followed by any single character, and ending with the string “income”.

Cross-references

See [Chapter 6](#) of the *User’s Guide* for a discussion of databases, databank files, and frequency conversion. [Chapter 4](#) of the *User’s Guide* discusses importing data. [Appendix C](#) of the *User’s Guide* describes the use of wildcard characters.

See also [setconvert \(p. 321\)](#), [store \(p. 347\)](#), and [copy \(p. 168\)](#).

fill	Coef Proc Matrix Proc Series Proc Sym Proc Vector Proc
------	--

Fill object with specified values.

`fill` can be used to set values of series, vectors and matrices in programs.

Syntax

Series Proc: `series_name.fill(options) n1, n2, n3 ...`

Coef, Vector Proc: `vector_name.fill(options) n1, n2, n3 ...`

Matrix, Sym Proc: `matrix_name.fill(options) n1, n2, n3 ...`

Follow the `fill` keyword with a list of values to place in the specified object. *Each value should be separated by a comma*. By default, series `fill` ignores the current sample and fills the series from the beginning of the workfile range. You may specify sample information using options.

Running out of values before the object is completely filled is not an error; the remaining cells or observations will be unaffected. If, however, you list more values than the object can hold, EViews will not modify any of the observations and will return an error message.

Options

Options for series fill

1	Loop repeatedly over the list of values as many times as it takes to fill the series.
<i>o = date, obs</i>	Set starting date or observation from which to start filling the series. Default is the beginning of the workfile range.
s	Fill the series only for the current workfile sample. The “s” option overrides the “o” option.
<i>s = sample_name</i>	fill the series only for the specified subsample. The “s” option overrides the “o” option.

Options for coef/vector/matrix fill

1	Loop repeatedly over the list of values as many times as it takes to fill the object.
<i>o = integer</i> (default = 1)	Fill the object from the specified element. Default is the first element.
b = c (default)	Fill the matrix by column.
b = r	Fill the matrix by row.

Examples

To generate a series D70 that takes the value 1, 2, and 3 for all observations from 1970:1:

```
series d70=0
d70.fill(o=1970:1,1) 1,2,3
```

Note that the last argument in the fill command above is the *letter “l”*. The next three lines generate a dummy series D70S that takes the value one and two for observations from 1970:1 to 1979:4:

```
series d70s=0
smp1 1970:1 1979:4
d70s.fill(s,1) 1,2
smp1 @all
```

Assuming a quarterly workfile, the following generates a dummy variable for observations in either the third and fourth quarter:

```
series d34
```

```
d34.fill(1) 0, 0, 1, 1
```

Note that this series could more easily be generated using `@seas`.

The following example declares a four element coefficient vector MC, initially filled with zeros. The second line fills MC with the specified values and the third line replaces from row 3 to the last row with -1.

```
coef(4) mc
mc.fill 0.1, 0.2, 0.5, 0.5
mc.fill(o=3,1) -1
```

The commands

```
matrix(2,2) m1
matrix(2,2) m2
m1.fill 1, 0, 1, 2
m2.fill(b=r) 1, 0, 1, 2
```

create the matrices

$$m1 = \begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}, \quad m2 = \begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix} \quad (8.1)$$

Cross-references

See [Chapter 4, “Matrix Language”, on page 55](#) of the *Command and Programming Reference* for a detailed discussion of vector and matrix manipulation in EViews.

fiml	System Method
------	-------------------------------

Estimation by full information maximum likelihood.

`fiml` estimates a system of equations by full information maximum likelihood (assuming a multivariate normal distribution).

Syntax

System Method: `system_name.fiml(options)`

Options

i	Iterate simultaneously over the covariance matrix and coefficient vector.
s (default)	Iterate sequentially over the covariance matrix and coefficient vector.
m = <i>integer</i>	Maximum number of iterations.
c = <i>number</i>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
b	Berndt-Hall-Hausman (BHHH) algorithm. Default method is Marquardt.
showopts / -showopts	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
deriv = <i>keyword</i>	Set derivative methods. The argument <i>keyword</i> should be a one- or two-letter string. The first letter should either be “f” or “a” corresponding to fast or accurate numeric derivatives (if used). The second letter should be either “n” (always use numeric) or “a” (use analytic if possible). If omitted, EViews will use the global defaults.
p	Print estimation results.

Examples

```
sys1.fiml
```

estimates SYS1 by FIML using the default settings. The command

```
sys1.fiml(d, s)
```

sequentially iterates over the coefficients and the covariance matrix.

Cross-references

See [Chapter 19](#) of the *User’s Guide* for a discussion of systems in EViews.

fitCommand || [Equation Proc](#)

Static forecasts. Computes fitted values from an estimated equation.

When the regressor contains lagged dependent values or ARMA terms, **fit** uses the actual values of the dependent variable instead of the lagged fitted values. You may instruct **fit** to compare the forecasted data to actual data, and to compute forecast summary statistics.

Not available for equations estimated using ordered methods—use **model** instead.

Syntax

- Command: **fit**(*options*) *yhat* *y_se*
Equation Proc: *eq_name*.**fit**(*options*) *yhat* *y_se*
ARCH Proc: *eq_name*.**fit**(*options*) *yhat* *y_se* *y_var*

Following the **fit** keyword you should type a name for the forecast series and, optionally, a name for the series containing the standard errors and, for ARCH specifications, a name for the conditional variance series.

Forecast standard errors are currently not available for binary, censored, and count models.

Options

g	Graph the fitted values together with the ± 2 standard error bands.
e	Produce the forecast evaluation table.
s	Ignore ARMA terms and use only the structural part of the equation to compute the fitted values.
i	Compute the fitted values of the index. Only for binary, censored and count models.
d	Forecast the dependent variable without normalizing the formula.
f = actual (default)	Fill observations outside the fit sample with actual values for the fitted variable.
f = na	Fill observations outside the fit sample with missing values (NAs)

Examples

```
equation eq1.ls cons c cons(-1) inc inc(-1)
```

```

eq1.fit c_hat c_se
genr c_up=c_hat+2*c_se
genr c_low=c_hat-2*c_se
line cons c_up c_low

```

The first line estimates a linear regression of CONS on a constant, CONS lagged once, INC, and INC lagged once. The second line stores the static forecasts and their standard errors as C_HAT and C_SE. The third and fourth lines compute the upper and lower bounds of the 95% forecast interval (assuming a normal distribution). The fifth line plots the actual series together with the 95% forecast interval.

```

equation eq2.binary(d=1) y c wage edu
eq2.fit yf
eq2.fit(I) xbeta
genr yhat=1-@clogit(-xbeta)

```

The first line estimates a logit of Y on a constant, WAGE, and EDU. The second line computes the fitted probabilities, and the third line computes the fitted values of the index. The fourth line computes the probabilities from the fitted index using the cumulative distribution function of the logistic distribution. Note that YF and YHAT should be identical.

Note that you cannot fit values from an ordered model. You must instead solve the values from a model. The following lines generate fitted probabilities from an ordered model:

```

equation eq3.ordered y c x z
eq3.model(oprob1)
solve oprob1

```

The first line estimates an ordered probit of Y on a constant, X, and Z. The second line makes a model from the estimated equation with a name OPROB1. The third line solves the model and computes the fitted probabilities that each observation falls in each category.

Cross-references

See [Chapter 14](#) of the *User's Guide* for a discussion of forecasting in EViews. See [Chapter 17](#) of the *User's Guide* for forecasting from binary, censored, truncated, and count models. See ["Forecasting" on page 580](#) of the *User's Guide* for a discussion of forecasting from sspace models.

To perform dynamic forecasting, use [forecast \(p. 214\)](#). See [model \(p. 269\)](#) and [solve \(p. 334\)](#) for forecasting from systems of equations or ordered equations.

forecastCommand || [Equation Proc](#) | [Sspace Proc](#)**Dynamic forecast.**

Computes (n -period ahead) dynamic forecasts of an estimated equation or forecasts of the signals and states for an estimated state space.

`forecast` computes the forecast for all observations in a specified sample. In some settings, you may instruct `forecast` to compare the forecasted data to actual data, and to compute summary statistics.

Syntax

Command:	<code>forecast(options) yhat y_se</code>
Equation Proc:	<code>eq_name.forecast(options) yhat y_se</code>
ARCH Proc:	<code>eq_name.forecast(options) yhat y_se y_var</code>
Sspace Proc:	<code>ss_name.forecast(options) keyword1 names1 [keyword2 names2] [keyword3 names3] ...</code>

When used with an equation, you should type a name for the forecast series and, optionally, a name for the series containing the standard errors and, for ARCH specifications, a name for the conditional variance series. Forecast standard errors are currently not available for binary or censored models. `forecast` is not available for models estimated using ordered methods.

When used with a sspace, you should enter a type-keyword followed by a list of names for the target series or a wildcard expression, and if desired, additional type-keyword/target pairs. The following are valid keywords: “@state”, “@statese”, “@signal”, “@signalse”. The first two instruct EViews to forecast the state series and the values of the state standard error series. The latter two instruct EViews to forecast the signal series and the values of the signal standard error series.

If a list is used to identify the targets in sspace forecasting, the number of target series must match the number of names implied by the keyword. Note that wildcard expressions may not be used for forecasting signal variables that contain expressions. The “*” wildcard expression may not be used for forecasting signal variables since this would overwrite the original data.

Options

Options for Equation forecasting

d	In models with implicit dependent variables, forecast the entire expression rather than the normalized variable.
e	Produce the forecast evaluation table.
g	Graph the forecasts together with the ± 2 standard error bands.
i	Compute the forecasts of the index. Only for binary, censored and count models.
s	Ignore ARMA terms and use only the structural part of the equation to compute the forecasts.
f = actual (default)	Fill observations outside the forecast sample with actual values for the fitted variable.
f = na	Fill observations outside the forecast sample with missing values (NAs)

Options for Sspace forecasting

i = arg (default = "o")	State initialization options: "o" (one-step), "e" (diffuse), "u" (user-specified), "s" (smoothed).
m = arg (default = "d")	Basic forecasting method: "n" (n-step ahead forecasting), "s" (smoothed forecasting), "d" (dynamic forecasting).
mprior = vector_name	Name of state initialization (used if option "i = u").
n = arg (default = 1)	Number of n-step forecast periods (only relevant if n-step forecasting is specified using the <i>method</i> option).
vprior = sym_name	Name of state covariance initialization (used if option "i = u").

Examples

The following lines

```
smp1 1970:1 1990:4
equation eq1.ls con c con(-1) inc
smp1 1991:1 1995:4
```

```
eq1.fit con_s  
eq1.forecast con_d  
plot con_s con_d
```

estimate a linear regression over the period 1970:1–1990:4, compute static and dynamic forecasts for the period 1991:1–1995:4, and plots the two forecasts in the same graph

```
equation eq1.ls m1 gdp ar(1) ma(1)  
eq1.forecast m1_bj bj_se  
eq1.forecast(s) m1_s s_se  
plot bj_se s_se
```

Estimates an ARMA(1,1) model, computes the forecasts and standard errors with and without the ARMA terms, and plots the two forecast standard errors.

The following command performs n-step forecasting of the signals and states from the sspace object, SS1:

```
ss1.forecast(method=n, n=4) @state * @signal y1f y2f
```

Here we save the state forecasts in the names specified in the sspace object, and we save the two signal forecasts in the series Y1F and Y2F.

Cross-references

For more information on equation forecasting in EViews, see [Chapter 14](#) of the *User’s Guide*. Details on forecasting for specialized techniques are provided in the corresponding chapters of the *User’s Guide*. For additional discussion of wildcards, see [Appendix C, “Wildcards”, on page 657](#) of the *User’s Guide*.

To perform static forecasting with equation objects see [fit \(p. 212\)](#). For multiple equation forecasting, see [model \(p. 269\)](#), and [solve \(p. 334\)](#).

freeze	Command
--------	-------------------------

Creates graph, table, or text objects from a view.

Syntax

Command: `freeze(name) object.view`

If you follow the keyword `freeze` with an object name but no view of the object, `freeze` will use the default view for the object. You can provide a name for the object containing the frozen view in parentheses.

Examples

The command

```
freeze gdp.uroot(4,t)
```

creates an untitled table that contains the results of the unit root test of the series GDP.

```
group rates tb1 tb3 tb6
freeze(gral) rates.line(m)
show gral.align(2,1,1)
```

freezes a graph named GRA1 that contains multiple line graphs of the series in the group RATES, realigns the individual graphs, and displays the resulting graph.

```
freeze(mygra) gral gra3 gra4
show mygra.align(2,1,1)
```

creates a graph object named MYGRA that combines three graph objects GRA1, GRA2, and GRA3, and displays MYGRA in two columns.

Cross-references

See “[Object Commands](#)” on page 6, and “[Object Basics](#)” on page 41 of the *User’s Guide* for further discussion of objects and views of objects. Freezing views is also described in [Chapter 10](#) of the *User’s Guide*.

freq	Group View Series View
------	--

Compute frequency tables.

When used as a series view (or for a group containing a single series), `freq` performs a one-way frequency tabulation. The options allow you to control binning (grouping) of observations.

When used with a group containing multiple series, `freq` produces an N -way frequency tabulation for all of the series in the group.

Syntax

Object View: `object_name.freq(options)`

Options

Options common to both one-way and N-way frequency tables

dropna (default) / keepna	[Drop/Keep] NA as a category.
v = <i>integer</i> (default = 100)	Make bins if the number of distinct values or categories exceeds the specified number.
nov	Do not make bins on the basis of number of distinct values; ignored if you set “v = <i>integer</i> .”
a = <i>number</i> (default = 2)	Make bins if average count per distinct value is less than the specified number.
noa	Do not make bins on the basis of average count; ignored if you set “a = <i>number</i> .”
b = <i>integer</i> (default = 5)	Maximum number of categories to bin into.
n, obs, count (default)	Display frequency counts.
nocount	Do not display frequency counts.
p	Print the table.

Options for one-way tables

total (default) / nototal	(Display/Do not display) totals.
pct (default) / nopct	(Display/Do not display) percent frequencies.
cum (default) / nocum	(Display/Do not) display cumulative frequency counts/percentages.

Options for N-way tables

table (default)	Display in table mode.
list	Display in list mode.
rowm (default) / norowm	(Display/Do not display) row marginals.
colm (default) / nocolm	(Display/Do not display) column marginals.
tabm (default) / notabm	(Display/Do not display) table marginals—only for more than two series.
subm (default) / nosubm	(Display/Do not display) sub marginals—only for “l” option with more than two series.
full (default) / sparse	(Full/Sparse) tabulation in list display.

totpct / nototpct (default)	(Display/Do not display) percentages of total observations.
tabpct / notabpct (default)	(Display/Do not display) percentages of table observations—only for more than two series.
rowpct / norowpct (default)	(Display/Do not display) percentages of row total.
colpct / nocolpct (default)	(Display/Do not display) percentages of column total.
exp / noexp (default)	(Display/Do not display) expected counts under full independence.
tabexp / notabexp (default)	(Display/Do not display) expected counts under table independence—only for more than two series.
test (default) / notest	(Display/Do not display) tests of independence.

Examples

```
hrs.freq(nov,noa)
```

tabulates each value (no binning) of HRS in ascending order with counts, percentages, and cumulatives.

```
inc.freq(v=20,b=10,noa)
```

tabulates INC excluding NAs. The observations will be binned if INC has more than 20 distinct values; EViews will create at most 10 equal width bins. The number of bins may be smaller than specified.

```
group labor lwage gender race
labor.freq(v=10,norowm,nocolm)
```

displays tables of LWAGE against GENDER for each bin/value of RACE.

Cross-references

See [Chapters 7](#) and [8](#) of the *User's Guide* for a discussion of frequency tables.

garch	Equation View
-------	-------------------------------

Conditional standard deviation graph.

Displays the conditional standard deviation graph of an equation estimated by ARCH.

Syntax

Equation View: `eq_name.garch(options)`

Options

p	Print the graph
---	-----------------

Examples

```
equation eq1.arch sp500 c  
eq1.garch
```

estimates a GARCH(1,1) model and displays the estimated conditional standard deviation graph.

Cross-references

ARCH estimation is described in [Chapter 16](#) of the *User’s Guide*.

See also [arch \(p. 145\)](#), [makegarch \(p. 252\)](#).

genr	Pool Proc
------	---------------------------

Generate series using pool objects.

This procedure allows you to generate multiple series using the cross-section identifiers in a pool. To generate values for a single series, see [series \(p. 317\)](#).

Syntax

Pool Proc: `pool_name.genr ser_name = expression`

You may use the cross section identifier “?” in the series name and/or in the expression on the right-hand side.

Examples

The commands

```
pool pool1  
pool1.add 1 2 3  
pool1.genr y? = x? - @mean(x?)
```

are equivalent to generating separate series for each cross-section:

```
series y1 = x1 - @mean(x1)  
series y2 = x2 - @mean(x2)
```

```
series y3 = x3 - @mean(x3)
```

Similarly,

```
pool pool2
pool2.add us uk can
pool2.genr y_? = log(x_?) - log(x_us)
```

generates three series Y_US, Y_UK, Y_CAN that are the log differences from X_US. Note that Y_US = 0.

The pool genr command simply loops across the cross-section identifiers, performing the appropriate substitution. Thus, the command

```
pool2.genr z=y_?
```

is equivalent to entering

```
series z=y_us
series z=y_uk
series z=y_can
```

so that the ordinary series Z will contain Y_CAN, the last series associated with the “Y_?”.

Cross-references

See [Chapter 21](#) of the *User’s Guide* for a discussion of the computation of pools, and a description of individual and balanced samples.

See [series \(p. 317\)](#) for a discussion of the expressions allowed in genr.

gmm	Command Equation Method System Method
------------	--

The equation or system object must be specified with a list of instruments.

Syntax

Command: **gmm**(*options*) *y x1 x2 @ z1 z2 z3*
 gmm(*options*) *formula @ z1 z2 z3*

Equation Method: *eq_name.gmm*(*options*) *y x1 x2 @ z1 z2 z3*
 eq_name.gmm(*options*) *formula @ z1 z2 z3*

System Method: *system_name.gmm*(*options*)

To use `gmm` as a command or equation method, follow the name of the dependent variable by a list of regressors, an “@-sign”, and a list of instrumental variables which are orthogonal to the residuals. Alternatively, you can specify a (nonlinear) formula using coefficients, an “@-sign”, and a list of instrumental variables which are orthogonal to the formula. There must be at least as many instrumental variables as there are coefficients to estimate.

Options

w	Use diagonal White’s weighting matrix for cross section data.
b = <i>arg</i> (default = “nw”)	Specify the bandwidth: “nw” (Newey-West fixed bandwidth based on the number of observations), “number” (user specified bandwidth), “v” (Newey-West automatic variable bandwidth selection), “a” (Andrews automatic selection).
q	Use the quadratic kernel. Default is to use the Bartlett kernel.
n	Prewhiten by a first order VAR before estimation.
i	Iterate simultaneously over the weighting matrix and the coefficient vector.
s	Iterate sequentially over the weighting matrix and coefficient vector.
o (default)	Iterate only on the coefficient vector with one step of the weighting matrix.
c	One step (iteration) of the coefficient vector following one step of the weighting matrix.
e	TSLS estimates with GMM standard errors.
m = <i>integer</i>	Maximum number of iterations.
c = <i>number</i>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
l = <i>number</i>	Set maximum number of iterations on the first-stage iteration to get the one-step weighting matrix.

`showopts / -showopts` [Do / do not] display the starting coefficient values and estimation options in the estimation output.

`deriv = keyword` Set derivative methods. The argument *keyword* should be a one- or two-letter string. The first letter should either be “f” or “a” corresponding to fast or accurate numeric derivatives (if used). The second letter should be either “n” (always use numeric) or “a” (use analytic if possible). If omitted, EViews will use the global defaults.

`p` Print results.

Examples

```
sys1.gmm(b=a,q,i)
```

estimates SYS1 by GMM with a quadratic kernel, Andrews automatic bandwidth selection, and iterates until convergence.

Cross-references

See [Chapters 12](#) and [19](#) of the *User’s Guide* for discussion of GMM estimation.

grads	Equation View Logl View Sspace View System View
-------	---

Gradients of the objective function.

Displays the gradients of the objective function (where available) for objects containing an estimated equation or equations.

The (default) summary form shows the value of the gradient vector at the estimated parameter values (if valid estimates exist) or at the current coefficient values. This latter usage makes grads useful in examining the behavior of the objective function at starting values. The tabular form shows a spreadsheet view of the gradients for each observation. The graphical form shows this information in a multiple line graph.

Syntax

Object View: `object_name.grads(options)`

Options

g	Display multiple graph showing the gradients of the objective function with respect to the coefficients evaluated at each observation (not available for systems).
t (default)	Display spreadsheet view of the values of the gradients of the objective function with respect to the coefficients evaluated at each observation (not available for systems).
p	Print results.

Examples

To show a summary view of the gradients:

```
eq1.grads
```

To display and print the table view

```
eq1.grads(t, p)
```

Cross-references

See also [derivs \(p. 189\)](#), [makederivs \(p. 251\)](#), and [makegrads \(p. 253\)](#).

graph

[Object Declaration](#) || [Graph Proc](#)

Creates named graph objects of the named series or group of series.

May also be used to merge multiple graphs into a single graph object using the `merge` proc.

Syntax

Command: `graph name.type(options) ser1 ser2 ser3`

Command: `graph name.merge graph1 graph2 graph3`

Follow the `graph` keyword with a name for the graph, a period, a type of the graph, and a list of series or group names to graph.

The second form of the `graph` command combines two or more existing graph objects into a named multiple graph object. Follow the `graph` keyword with a name for the graph, a period, the `merge` keyword, and a list of named graph objects to be merged.

Options

The valid types of a graph object are:

bar	Bar graph.
errbar	Error bar graph.
hilo	High-low(-open-close) graph.
line	Line graph.
pie	Pie graph.
scat	Scatterplot (same as XY, but lines are initially turned off, symbols turned on, and a 3x3 frame is used).
spike	Spike graph.
xy	XY line-symbol graph with one X plotted against one or more Y's using existing line-symbol settings.
xylene	Same as XY but sets settings to display lines.
xypair	XY line-symbol graph with pairs of X and Y plotted against each other.

Additional options will depend on the type of graph chosen. See the entry for each graph type for a list of the available options (e.g. see [bar \(p. 150\)](#) for details on bar graphs).

Additional Options

p	Print the graph.
---	------------------

Examples

```
graph gra1.line(m) gdp m1 inf
```

creates a multiple line graph object named GRA1.

```
graph mygra.combine gr_line gr_scat gr_pie
```

creates a multiple graph object named MYGRA that combines three graph objects named GR_LINE, GR_SCAT, and GR_PIE.

Cross-references

The graph object is described in greater detail in “[Graph](#)” (p. 25). See [Chapter 10](#) of the *User’s Guide* for a general discussion of graphs.

See also [freeze \(p. 216\)](#), [merge \(p. 267\)](#).

group	Object Declaration
-------	--------------------

Declare a group object containing a group of series.

Syntax

Command: **group** group_name ser1 ser2 ser3

Follow the group name with a list of series to be included in the group.

Examples

```
group g1 gdp cpi inv
group g1 tb3 m1 gov
g1.add gdp cpi
```

The first line creates a group named G1 that contains three series GDP, CPI, and INV. The second line redeclares group G1 to contain the three series TB3, M1, and GOV. The third line adds two series GDP and CPI to group G1 to make a total of five series. See [add \(p. 137\)](#).

```
group rhs d1 d2 d3 d4 gdp(0 to -4)
ls cons rhs
ls cons c rhs(6)
```

The first line creates a group named RHS that contains nine series. The second line runs a linear regression of CONS on the nine series in RHS. The third line runs a linear regression of CONS on C and only the sixth series GDP(-1) of RHS.

Cross-references

See “[Group](#)” on page 26 for a complete description of the group object. See also [Chapter 8](#) of the *User’s Guide* for additional discussion.

hconvert	Command
----------	---------

Convert an entire Haver Analytics Database into an EViews database.

Syntax

Command: **hconvert** path_haver db_name

You must have a Haver Analytics database installed on your computer to use this feature. You must also create an EViews database to store the converted Haver data *before* you use this command.

Be aware that this command may be very time-consuming.

Follow the command by a *full path name* to the Haver database and the name of an existing EViews database to store the Haver database. You can include a path name to the EViews database not in the default path.

Examples

```
dbcreate hdata  
hconvert d:\haver\haver hdata
```

The first line creates a new (empty) database named HDATA in the default directory. The second line converts all the data in the Haver database and stores it in the HDATA database.

Cross-references

See [Chapter 6, “EViews Databases”, on page 107](#) of the *User’s Guide* for a discussion of EViews and Haver databases.

See also [dbcreate \(p. 181\)](#), [db \(p. 180\)](#), [hfetch \(p. 227\)](#) and [hlabel \(p. 230\)](#).

hfetch	Command
---------------	-------------------------

Fetch a series from a Haver Analytics database into a workfile.

hfetch reads one or more series from a Haver Analytics Database into the active workfile. *You must have a Haver Analytics database installed on your computer to use this feature.*

Syntax

Command: **hfetch(*database_name*) *series_name***

hfetch, if issued alone on a command line, will bring up a Haver dialog box which has fields for entering both the database name and the series names to be fetched. If you provide the database name (including the full path) in parentheses after the hfetch command, EViews will read the database and copy the series requested into the current workfile. It will also display information about the series on the statusline. The database name is optional if a default database has been specified.

hfetch can read multiple series with a single command. List the series names, each separated by a space.

Examples

```
hfetch(c:\data\us1) pop gdp xyz
```

reads the series POP, GDP, and XYZ from the US1 database into the current active workfile, performing frequency conversions if necessary.

Cross-references

Additional information on EViews frequency conversion is provided in “[Frequency Conversion](#)” on page 72 of the *User’s Guide*. See also [Chapter 6](#) of the *User’s Guide* for a discussion of EViews and Haver databases.

See also [dbccreate](#) (p. 181), [db](#) (p. 180), [hconvert](#) (p. 226) and [hlabel](#) (p. 230).

hilo	Command Graph Proc Group View Matrix View Sym View
-------------	---

Hi-low(-open-close) graph. Plot high-low, high-low-close, or high-low-open-close graph.

Syntax

- | | |
|--------------|--|
| Command: | <code>hilo(options) high_ser low_ser [close_ser]</code> |
| | <code>hilo(options) high_ser low_ser open_ser close_ser</code> |
| Graph Proc: | <code>graph_name.hilo(options)</code> |
| Object View: | <code>object_name.hilo(options)</code> |

For a high-low, or a high-low-close graph, follow the command name with the name of the high series, followed by the low series, and an optional close series. If four series names are provided, EViews will use them in the following order: high-low-open-close.

When used with a matrix or group or an existing graph, EViews will use the first series as the high series, the second series as the low series, and if present, the third series as the close. If there are four or more series, EViews will use them in the following order: high-low-open-close. When used as a matrix view, the columns of the matrix are used in place of the series.

Note that if you wish to display a high-low-open graph, you should use an “NA”-series for the close values.

Options

Template and printing options

<code>o = graph_name</code>	Use appearance options from the specified graph.
<code>t = graph_name</code>	Use appearance options and copy text and shading from the specified graph.
<code>p</code>	Print the bar graph.

Examples

```
hilo mshigh mslow
```

displays a high-low graph using the series MSHIGH and MSLOW.

The command

```
hilo(p) mshigh mslow msopen msclose
```

plots and prints the high-low-open-close graph of the four series.

```
group stockprice mshigh mslow msclose
stockprice.hilo
```

displays the high-low-close view of the group STOCKPRICE.

The command

```
hilo NA NA msopen msclose
```

displays an open-close graph.

Cross-references

See [Chapter 10](#) of the *User's Guide* for additional details on using graphs in EViews.

See also [graph \(p. 224\)](#).

hist	Command Equation View Series View
-------------	--

Histogram and descriptive statistics of a series.

When used as a command or a series view, `hist` computes descriptive statistics and displays a histogram for the series. When used as an equation view, `hist` displays a histogram and descriptive statistics of the residual series.

Syntax

Command: `hist(options) series_name`
Object View: `object_name.hist(options)`

Options

<code>p</code>	Print the histogram.
----------------	----------------------

Examples

`lwage.hist`

Displays the histogram and descriptive statistics of LWAGE.

Cross-references

See “[Histogram and Stats](#)” on page 152 of the *User’s Guide* for a discussion of the descriptive statistics reported in the histogram view.

<code>hlabel</code>	Command
---------------------	---------

Display a Haver Analytics database series description.

`hlabel` reads the description of a series from a Haver Analytics Database and displays it on the status line at the bottom of the EViews window. Use this command to verify the contents of a Haver database series name.

You must have a Haver Analytics database installed on your computer to use this feature.

Syntax

Command: `hlabel(database_name) series_name`

`hlabel`, if issued alone on a command line, will bring up a Haver dialog box which has fields for entering both the database name and the series names to be examined. If you provide the database name in parentheses after the `hlabel` command, EViews will read the database and find the key information about the series in question, such as the start date, end date, frequency, and description. This information is displayed in the status line at the bottom of the EViews window. Note that the *database_name* should refer to the full path to the Haver database but need not be specified if a default database has been specified in HAVERDIR.INI.

If several series names are placed on the line, `hlabel` will gather the information about each of them, but the information display may scroll by so fast that only the last will be visible.

Examples

```
hlabel(c:\data\us1) pop
```

displays the description of the series POP in the US1 database.

Cross-references

See [Chapter 6](#) of the *User's Guide* for a discussion of EViews and Haver databases.

See also [hfetch \(p. 227\)](#) and [hconvert \(p. 226\)](#).

hpf	Command Series Proc
------------	--

Hodrick-Prescott filter. **hpf** smooths a series using the Hodrick-Prescott filter.

Syntax

Command:	<code>hpf(<i>n, options</i>)</code> <i>series_name</i> <i>filtered_name</i>
Series Proc:	<code><i>series_name.hpf(<i>n, options</i>)</i></code> <i>filtered_name</i>

Follow the **hpf** keyword with the name of the series you want to filter and a name for the filtered series.

When used as a series procedure, **hpf** smooths the series without displaying the graph of the smoothed series.

Options

- | | |
|----------|--|
| <i>n</i> | You may specify the smoothing parameter; a larger number results in more smoothing. The default is “100” for annual, “1600” for quarterly, and “14400” for monthly data. |
| <i>p</i> | Print the graph of the smoothed series and the original series. |

Examples

The command

```
hpf(1000) gdp gdp_hp
```

smooths the GDP series with a smoothing parameter “1000” and saves the smoothed series as GDP_HP.

Cross-references

See “[Hodrick-Prescott Filter](#)” on page 195 of the *User’s Guide* for details.

impulse	Var View
----------------	--------------------------

Display impulse response functions of a VAR.

Syntax

Var View: `var_name.impulse(n, options) responses @ shocks @ ordering`

List the series names in the var whose responses you would like to compute. You may optionally specify the sources of shocks. To specify the shocks, list the series after an “@”. By default, EViews computes the responses to all possible sources of shocks using the ordering in the Var.

If you are using impulses from the Cholesky factor, you may change the Cholesky ordering by listing the order of the series after a second “@”.

You must specify the number of periods *n* for which you wish to compute the impulse responses.

Options

<code>g</code> (default)	Display combined graphs, with impulse responses of one variable to all shocks shown in one graph. If you choose this option, standard error bands will not be displayed.
<code>m</code>	Display multiple graphs, with impulse response to each shock shown in separate graphs.
<code>t</code>	Tabulate the impulse responses.
<code>a</code>	Accumulate the impulse responses.

<code>imp = arg</code> (default “ <code>imp = chol</code> ”)	Type of factorization for the decomposition: unit impulses, ignoring correlations among the residuals (“ <code>imp = unit</code> ”), non-orthogonal, ignoring correlations among the residuals (“ <code>imp = nonort</code> ”), Cholesky with d.f. correction (“ <code>imp = chol</code> ”), Cholesky without d.f. correction (“ <code>imp = mlechol</code> ”), Generalized (“ <code>imp = gen</code> ”), structural (“ <code>imp = struct</code> ”), or user specified (“ <code>imp = user</code> ”). The structural factorization is based on the estimated structural VAR. To use this option, you must first estimate the structural decomposition; see svar (p. 351) . For user specified impulses, you must specify the name of the vector/matrix containing the impulses using the “ <code>fname = </code> ” option. The option “ <code>imp = mlechol</code> ” is provided for backward compatibility with EViews 3.x and earlier.
<code>fname = name</code>	Specify name of vector/matrix containing the impulses. The vector/matrix must have k rows and 1 or k columns, where k is the number of endogenous variables.
<code>se = arg</code>	Standard error calculations: “ <code>se = a</code> ” (analytic), “ <code>se = mc</code> ” (Monte Carlo). If selecting Monte Carlo, you must specify the number of replications with the “ <code>rep = </code> ” option. Note the following: (1) Analytic standard errors are currently not available for (a) VECs and (b) structural decompositions identified by long-run restrictions. The “ <code>se = a</code> ” option will be ignored for these cases. (2) Monte Carlo standard errors are currently not available for (a) VECs and (b) structural decompositions. The “ <code>se = mc</code> ” option will be ignored for these cases.
<code>rep = integer</code>	Number of Monte Carlo replications to be used in computing the standard errors. Must be used with the “ <code>se = mc</code> ” option.

matbys = <i>name</i>	Save responses by shocks (impulses) in named matrix. The first column is the response of the first variable to the first shock, the second column is the response of the second variable to the first shock, and so on.
matbyr = <i>name</i>	Save responses by response series in named matrix. The first column is the response of the first variable to the first shock, the second column is the response of the first variable to the second shock, and so on.
p	Print the results.

Examples

```
var var1.ls 1 4 m1 gdp cpi  
var1.impulse(10,m) gdp @ m1 gdp cpi
```

The first line declares and estimates a VAR with three variables. The second line displays multiple graphs of the impulse responses of GDP to shocks to the three series in the VAR using the ordering as specified in VAR1.

```
var1.impulse(10,m) gdp @ m1 @ cpi gdp m1
```

displays the impulse response of GDP to a one standard deviation shock in M1 using a different ordering.

Cross-references

See [Chapter 20](#) of the *User’s Guide* for a discussion of variance decompositions in VARs.

See also [decomp](#) (p. 186).

jbera	Var View
-------	--------------------------

Multivariate residual normality test.

Syntax

Var View: `var_name.jbera(options)`

You must specify a factorization method by the “factor =” option.

Options

factor = chol	Factorization by the inverse of the Cholesky factor of the residual covariance matrix.
factor = cor	Factorization by the inverse square root of the residual correlation matrix (Doornik and Hansen 1994).
factor = cov	Factorization by the inverse square root of the residual covariance matrix (Urzua 1997).
factor = svar	Factorization matrix from structural VAR. You must first estimate the structural factorization to use this option; see svar (p. 351) .
name =	Save the test statistics in a named matrix object. See below for a description of the statistics contained in the stored matrix.
p	Print the test results.

The “name =” option stores the following matrix. Let the VAR have k endogenous variables. Then the stored matrix will have dimension $(k + 1) \times 4$. The first k rows contain statistics for each orthogonal component, where the 1st column is the third moments, the second column is the one-degree of freedom chi-square statistics for the third moments, the third column is the fourth moments, and the fourth column is the one-degree of freedom χ^2 statistics for the fourth moments. The sum of the second and fourth columns are the Jarque-Bera statistics reported in the last output table.

The $(k + 1)$ row contains statistics for the joint test. The second and fourth column of the $(k + 1)$ row is simply the sum of all the rows above in the corresponding column and are the $\chi^2(k)$ statistics for the joint skewness and kurtosis tests, respectively. These joint skewness and kurtosis statistics add up to the joint Jarque-Bera statistic reported in the output table, except for the “factor = cov” option. When this option is set, the joint Jarque-Bera statistic includes all cross moments (in addition to the pure third and fourth moments). The overall Jarque-Bera statistic for this option is stored in the first column of the $(k + 1)$ row (which will be a missing value for all other options).

Examples

```
var var1.ls 1 6 lgdp lm1 lcpi
show var1.jbera(factor=cor, name=jb)
```

The first line declares and estimates a VAR. The second line carries out the residual multivariate normality test using the inverse square root of the residual correlation matrix as the factorization matrix and stores the results in a matrix named JB.

Cross-references

See [Chapter 20](#) of the *User’s Guide* for a discussion of the test and other VAR diagnostics.

kdensity	Series View
-----------------	-----------------------------

Kernel density plots. Displays nonparametric kernel density estimates of the specified series.

Syntax

Series View: *series_name.kdensity(options)*

Options

<i>k = arg</i> (default “k = e”)	Kernel type: “e” (Epanechnikov), “r” (Triangular), “u” (Uniform), “n” (Normal–Gaussian), “b” (Biweight–Quartic), “t” (Triweight), “c” (Cosinus).
<i>s</i> (default)	Automatic bandwidth (Silverman).
<i>b = number</i>	Specify a number for the bandwidth.
<i>b</i>	Bracket bandwidth.
<i>n</i> (default = 100)	Number of points to evaluate.
<i>x</i>	Exact evaluation of kernel density.
<i>o = arg</i>	Name of matrix to hold results of kernel density computation. The first column of the matrix contains the evaluation points and the remaining columns contain the kernel estimates.
<i>p</i>	Print the kernel density plot.

Examples

```
lwage.kdensity(k=n)
```

plots the kernel density estimate of LWAGE using a normal kernel and the automatic bandwidth selection.

Cross-references

See [“Kernel Density” on page 229](#) of the *User’s Guide* for a discussion of kernel density estimation.

kerfit	Group View
--------	------------

Fits a kernel regression of the second series in the group (vertical axis) against the first series in the group (horizontal axis).

Syntax

Group View: `group_name.kerfit(options)`

Options

<code>k = arg</code> (default “k = e”)	Kernel type: “e” (Epanechnikov), “r” (Triangular), “u” (Uniform), “n” (Normal–Gaussian), “b” (Biweight–Quartic), “t” (Triweight), “c” (Cosinus).
<code>b = number</code>	Specify a number for the bandwidth.
<code>b</code>	Bracket bandwidth.
<code>integer</code> (default = 100)	Number of grid points to evaluate.
<code>x</code>	Exact evaluation of the polynomial fit.
<code>d = integer</code> (default = 1)	Degree of polynomial to fit. Set “d = 0” for Nadaraya–Watson regression.
<code>s = name</code>	Save fitted series.
<code>p</code>	Print the kernel fit plot.

Examples

```
group gg weight height
gg.kerfit(s=w_fit, 200)
```

Fits a kernel regression of HEIGHT on WEIGHT using 200 points and saves the fitted series as W_FIT.

Cross-references

See “[Scatter with Kernel Fit](#)” on page 238 of the *User’s Guide* for a discussion of kernel regression.

See also [linefit](#) (p. 242), [nnfit](#) (p. 272).

label

Object View | Object Proc

Display or change the label view of the object, including the last modified date and display name (if any).

As a procedure, `label` changes the fields in the object label.

SyntaxObject View: `object_name.label`Object Proc: `object_name.label(options) text`**Options**

To modify the label, you should specify one of the following options along with optional text. If there is no text provided, the specified field will be cleared:

c	Clears all text fields in the label.
d	Sets the description field to <i>text</i> .
s	Sets the source field to <i>text</i> .
u	Sets the units field to <i>text</i> .
r	Appends <i>text</i> to the remarks field as an additional line.
p	Print the label view.

Examples

The following lines replace the remarks field of LWAGE with “Data from CPS 1988 March File”:

```
lwage.label(r)  
lwage.label(r) Data from CPS 1988 March File
```

To append additional remarks to LWAGE, and then to print the label view:

```
lwage.label(r) Log of hourly wage  
lwage.label(p)
```

To clear and then set the units field, use:

```
lwage.label(u) Millions of bushels
```

Cross-references

See “[Labeling Objects](#)” on page 50 of the *User’s Guide* for a discussion of labels.

See also [displayname](#) (p. 192).

laglen	Var View
---------------	--------------------------

VAR lag order selection criteria.

Syntax

Var View: `var_name.laglen(m, options)`

You must specify the maximum lag order *m* to test for.

Options

`vname = arg` Save selected lag orders in named vector. See below for a description of the stored vector.

`mname = arg` Save lag order criteria in named matrix. See below for a description of the stored matrix.

`p` Print table of lag order criteria.

The “vname =” option stores a vector with 5 rows containing the selected lags from the following criteria: sequential modified LR test (row 1), final prediction error (row 2), Akaike information criterion (row 3), Schwarz information criterion (row 4), Hannan-Quinn information criterion (row 5).

The “mname =” option stores a $q \times 6$ matrix, where $q = m + 1$ if there are no exogenous variables in the VAR, otherwise $q = m + 2$. The first $(q - 1)$ rows contain the information displayed in the table view following the same order. In addition, the saved matrix has an additional row which contains the lag order selected from each column criterion. The first column (corresponding to the log likelihood values) of the last row is always an NA.

Examples

```
var var1.ls 1 6 lgdp lm1 lcpi
show var1.laglen(12,vname=v1)
```

The first line declares and estimates a VAR. The second line computes the lag length criteria up to a maximum of 12 lags and stores the selected lag orders in a vector named V1.

Cross-references

See [Chapter 20](#) of the *User’s Guide* for a discussion of the various criteria and other VAR diagnostics.

See also [testlags](#) (p. 361).

legend	Graph Proc
---------------	----------------------------

Set legend appearance and placement in graphs.

When `legend` is used with a multiple graph, the legend settings apply to all graphs. See [setelem](#) (p. 323) for setting legends for individual graphs in a multiple graph.

Syntax

Graph Proc: `graph_name.legend option_list`

Note: the syntax of the `legend` proc has changed considerably from version 3.1 of EViews. While not documented here, the EViews 3 options are still (for the most part) supported. However, we do not recommend using the old options as future support is not guaranteed.

Options

`columns(arg)` *arg* = “auto” — automatically choose number of columns for legend (*default*).

arg = “*n1*”— put legend in *n1* columns.

`display/-display` Display/do not display the legend.

`font(n1)` Size of font for legend.

`inbox/-inbox` Put legend in box/remove box around legend.

`position(arg)` *arg* = “left”, “l” — place legend on left side of graph.

arg = “right”, “r” — place legend on right side of graph.

arg = “botleft”, “bl” — place left-justified legend below graph.

arg = “botcenter”, “bc” — place centered legend below graph.

arg = “botright”, “br”— place right-justified legend below graph.

arg = “(*h, v*)” — The first number *h* specifies how many virtual inches to offset to the right from the origin. The second number *v* specifies how many virtual inches to offset below the origin. The origin is the upper left hand corner of the graph.

`p` Print the graph.

Examples

```
mygra1.legend display position(1) inbox
```

places the legend of MYGRA1 in a box to the left of the graph.

```
mygra1.legend position(.2,.2) -inbox
```

places the legend of MYGRA1 within the graph, indented slightly from the upper left corner with no box surrounding the legend text.

Cross-references

See [Chapter 10](#) of the *User's Guide* for a discussion of graph objects in EViews.

See also [setelem \(p. 323\)](#).

line	Command Coef View Graph Proc Group View Matrix View Series View Sym View Vector View
-------------	--

Line graph. Displays a line graph of the specified object.

The line graph view of a group plots all series in the group in a single graph. The line graph view of a matrix plots each column in the matrix in a single graph.

Syntax

Command: `line(options) object_name`

Object View: `object_name.line(options)`

Graph Proc: `graph_name.line(options)`

Options

Template and printing options

<code>o = graph_name</code>	Use appearance options from the specified graph.
-----------------------------	--

<code>t = graph_name</code>	Use appearance options and copy text and shading from the specified graph.
-----------------------------	--

<code>p</code>	Print the line graph.
----------------	-----------------------

Scale options (for multiple line views of group and matrix objects)

a (default)	Automatic single scale.
n	Normalized scale (zero mean and unit standard deviation).
d	Dual scaling with no crossing.
x	Dual scaling with possible crossing.
s	Stack so that the difference between lines corresponds to the value of a series.
m	Display multiple graphs.

Examples

```
group g1 gdp cons m1  
g1.line(d)
```

plots line graphs of the three series in group G1 with dual scaling (no crossing). The latter two series will share the same scale.

```
matrix1.line(t=mygra)
```

displays line graphs of the columns of MATRIX1 using the graph object MYGRA as a template.

```
line(m) gdp cons m1
```

creates an untitled graph object that contains three line graphs of GDP, CONS, and M1, with each plotted separately.

Cross-references

See [Chapter 10](#) of the *User's Guide* for a detailed discussion of graphs in EViews.

See also [graph \(p. 224\)](#) for additional graph types.

linefit	Group View
----------------	----------------------------

Scatter plot with bivariate fit.

Displays the scatter plot of the second series (horizontal axis) and the first series (vertical axis) with a regression fit line. You can specify various transformation methods and weighting for the bivariate fit.

Syntax

Group View: `group_name.linefit(options)`

Options

<code>yl</code>	Take the natural log of first series, y .
<code>yi</code>	Take the inverse of y .
<code>yp = number</code>	Take y to the power of the specified number.
<code>yb = number</code>	Take the Box-Cox transformation of y with the specified parameter.
<code>xl</code>	Take the natural log of x .
<code>xi</code>	Take the inverse of x .
<code>xp = number</code>	Take x to the power of the specified number.
<code>xb = number</code>	Take the Box-Cox transformation of x with the specified parameter.
<code>xd = integer</code>	Fit a polynomial of x up to the specified power.
<code>m = integer</code>	Set number of robustness iterations.
<code>s = name</code>	Save the fitted y series.
<code>p</code>	Print the scatter plot.

If the polynomial degree of x leads to singularities in the regression, EViews will automatically drop the high order terms to avoid collinearity.

Examples

```
group g1 inf unemp
g1.linefit(yl,xl,s=yfit)
```

displays a scatter plot of log UNEMP against log INF together with the fitted values from a regression of log UNEMP on the log INF. The fitted values are saved in a series named YFIT. Note that the saved fitted values are for the original UNEMP, not the log transform.

```
g1.linefit(yb=0.5,m=10)
```

The Box-Cox transformation of UNEMP with parameter 0.5 is regressed on INF with 10 iterations of bisquare weights.

Cross-references

See “[Scatter with Regression](#)” on page 234 of the *User’s Guide* for a discussion of scatter plot with regression fit.

See also [nnfit](#) (p. 272) and [kerfit](#) (p. 237).

load	Command
-------------	---------

Load a workfile. `load` reads in a previously saved workfile.

The workfile becomes the active workfile; existing workfiles in memory remain on the desktop but become inactive.

Syntax

Command: `load(options) workfile_name`

Options

`d` Load a DOS MicroTSP workfile.

`m` Load a Macintosh MicroTSP workfile.

Examples

```
load c:\data\marco
```

loads a previously saved EViews workfile MACRO.WF1 from the DATA directory in the C drive.

```
load c:\tsp\nipa.wf
```

loads a MicroTSP workfile NIPA.WF. If you do not use the workfile type option, you should add the extension .WF to the workfile name when loading a DOS MicroTSP workfile. An alternative method uses the type “d” option:

```
load(d) nipa
```

Cross-references

See “[Workfile Basics](#)” on page 33 of the *User’s Guide* for a discussion of workfile operations. [Chapter 4](#) of the *User’s Guide* describes various methods of moving data into EViews.

See also [open](#) (p. 275), [save](#) (p. 308), [read](#) (p. 291) and [fetch](#) (p. 205)

logit	Command
--------------	---------

Estimate binary models with logistic errors.

Equivalent to issuing the command, `binary` with the option “(d=l)”.

See [binary](#) (p. 152).

logl	Object Declaration
-------------	------------------------------------

Declare likelihood object.

Syntax

Command: **logl logl_name**

Options

p	Print the table of results.
----------	-----------------------------

Examples

```
logl ll1
```

declares a likelihood object named LL1.

```
ll1.append @logl logl1
ll1.append res1 = y-c(1)-c(2)*x
ll1.append logl1 = log(@dnorm(res1/@sqrt(c(3))))-log(c(3))/2
```

specifies the likelihood function for LL1 and estimates the parameters by maximum likelihood.

Cross-references

See [Chapter 18](#) of the *User's Guide* for further examples of the use of the likelihood object.

See also [ml](#) (p. 269).

ls	Command Equation Method Pool Method System Method Var Method
-----------	---

Estimation by linear or nonlinear least squares regression.

When used as a pool proc, ls estimates linear cross-section weighted least squares, and fixed and random effects models.

Syntax

Command: $\text{ls}(options) y x1 x2 x3$
 $\text{ls}(options) y = c(1)*x1 + c(2)*x2 + c(3)*x3$

Equation Method: $\text{eq_name.ls}(options) [specification]$

System Method: $\text{system_name.ls}(options)$

VAR Method: $\text{var_name.ls}(options)$

Pool Method: $\text{pool_name.ls}(options) y x1 x2 @ z1 z2$

For linear specifications, list the dependent variable first, followed by a list of the independent variables. Use a “C” if you wish to include a constant or intercept term; unlike some other programs, EViews does not automatically include a constant in the regression. You may add AR, MA, SAR and SMA error specifications and PDL specifications for polynomial distributed lags. If you include lagged variables, EViews will adjust the sample automatically, if necessary.

Both dependent and independent variables may be created from existing series using standard EViews functions and transformations. EViews treats the equation as linear in each of the variables and assigns coefficients C(1), C(2), and so forth to each variable in the list.

Linear or nonlinear single equations can also be specified by explicit equation. You should specify the equation as a formula. The parameters to be estimated should be called “C(1)”, “C(2)”, and so forth (assuming that you wish to use the default coefficient vector “C”). You may also declare an alternative coefficient vector using `coef` and use these coefficients in your expressions.

For equation and var objects, you can declare and estimate the object in a single command. Follow the declaration of the object type with the name of the new object, then the `ls` keyword, separated by a period.

When used as a pool method, `ls` carries out panel data estimation. Type the name of the dependent variable followed by one or two lists of regressors. The first list should contain ordinary and pool series that are restricted to have the same coefficient across all members of the pool. The second list, if provided, should contain variables that have different coefficients for each member of the pool. If there is a second list, the two lists must be separated by an “@”-sign.

For pool estimation, you may include AR terms as regressors (except when estimating random effects models using the “r” option) but not MA terms.

Options

Options for Equation, System, and Var estimation

w = <i>series_name</i>	Weighted Least Squares. Each observation will be weighted by multiplying by the specified series.
h	White's heteroskedasticity consistent standard errors.
n	Newey-West heteroskedasticity and autocorrelation consistent (HAC) standard errors.
m = <i>integer</i>	Set maximum number of iterations.
c = <i>scalar</i>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
s	Use the current coefficient values in C as starting values for equations with AR or MA terms (see PARAM).
s = <i>number</i>	Specify a number between zero and one to determine starting values for equations with AR or MA terms as a fraction of the preliminary LS or TSLS estimates made without AR or MA terms (out of range values are set to "s = 1").
z	Turn off backcasting in ARMA models.
showopts / -showopts	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
deriv = <i>keyword</i>	Set derivative methods. The argument <i>keyword</i> should be a one or two letter string. The first letter should either be "f" or "a" corresponding to fast or accurate numeric derivatives (if used). The second letter should be either "n" (always use numeric) or "a" (use analytic if possible). If omitted, EViews will use the global defaults.
p	Print basic estimation results.

Note: not all options are available for all methods. See the *User's Guide* for details on each estimation method.

Options for pool estimation

n	No intercept.
f	Fixed effect estimation (separate intercept for each cross section member).
r	Random effects estimation.
w	Weighted least squares, with weights estimated in preliminary regression using equal weights. Not available with the “r” option.
s	SUR (seemingly unrelated regression), with covariance matrix of pool members estimated in preliminary regression and then applied in generalized least squares in a second round. Not available with the “r” option.
i	Iterate to convergence (default if AR terms are included).
h	White heteroskedasticity consistent standard errors.
b	Estimate using a balanced sample.
p	Print estimation results.

Examples

```
equation eq1.ls m1 c uemp inf(0 to -4) @trend(1960:1)
```

estimates a linear regression of M1 on a constant, UEMP, INF (from current up to four lags), and a linear trend.

```
equation eq2.ls(z) d(tbill) c inf @seas(1) @seas(1)*inf ma(2)
```

regresses the first difference of TBILL on a constant, INF, a seasonal dummy, and an interaction of the dummy and INF with an MA(2) error term. The “z” option turns off backcasting.

```
coef(2) beta
param beta(1) .2 beta(2) .5 c(1) 0.1
equation eq3.ls(h) q = beta(1)+beta(2)*(1^c(1) + k^(1-c(1)))
```

estimates the nonlinear regression starting from the specified initial values. The “h” option reports heteroskedasticity consistent standard errors.

```
equation eq4.ls r=c(1)+c(2)*r(-1)+div(-1)^c(3)
sym betacov=eq4.@cov
```

declares and estimates a nonlinear equation and stores the coefficient covariance matrix in a symmetric matrix called BETACOV.

```
pool1.ls dy? inv? edu? year
```

estimates pooled OLS of DY on a constant, INV, EDU, and YEAR.

```
pool1.ls(f) dy? @ inv? edu? year ar(1)
```

estimates a fixed effects model without restricting any of the coefficients to be the same across pool members.

```
group rhs c dum1 dum2 dum3 dum4
ls cons rhs ar(1)
```

uses the group definition for RHS to regress CONS on C, DUM1, DUM2, DUM3, and DUM4 with an AR(1) correction.

Cross-references

[Chapters 11 and 12](#) of the *User’s Guide* discuss the various regression methods in greater depth. See [Chapter 21](#) of the *User’s Guide* for a discussion of pool estimation.

See “[Equation](#)” (p. 21), “[Pool](#)” (p. 34), “[System](#)” (p. 45), and “[Var](#)” (p. 49) for a complete description of these objects, including a list of saved results.

See [ar](#) (p. 144), [ma](#) (p. 249), [sar](#) (p. 307), [sma](#) (p. 330), and [pd1](#) (p. 282) for terms that may be used in `ls` specifications.

ma	Expression
----	------------

Moving average error specification.

The `ma` specification may be added in an “`ls`” or “`tsls`” specification to indicate a moving average error component. `ma(1)` indicates the first order component, `ma(2)` indicates the second order component, and so on.

Examples

```
ls(z) m1 c tb3 tb3(-1) ma(1) ma(2)
```

regresses M1 on a constant, TB3, and TB3 lagged once with first order and second order moving average error components. The “`z`” option turns off backcasting in estimation.

Cross-references

See [Appendix , “”, on page 303](#) of the *User’s Guide* for details on ARMA and seasonal ARMA modeling.

See also [sma](#) (p. 330), [ar](#) (p. 144), and [sar](#) (p. 307).

makecoint	Var Proc
------------------	--------------------------

Create group containing the estimated cointegrating relations from a VEC.

Syntax

Var Proc: var_name.makecoint
Var Proc: var_name.makecoint *group_name*

From the command window, the first form creates an untitled group object containing the estimated cointegrating relations in the VEC. The second form allows you to name the group object by following the `makecoint` proc with a name to be given to the group. The series contained in the group are named COINTEQ01, COINTEQ02, and so on. In batch mode (program files), the first form simply creates the series COINTEQ01, COINTEQ02, and so on.

This proc will return an error message unless you have estimated an error correction model with the `var` object.

Examples

```
var vec1.ec(b,2) 1 4 y1 y2 y3
vec1.makecoint gcoint
```

The first line estimates a VEC with 2 cointegrating relations. The second line creates a group named GCOINT which contains the two estimated cointegrating relations. The two cointegrating relations will be stored as series named COINTEQ01 and COINTEQ02 if these names have not yet been used in the workfile. If these names are already used, the next unused number will be appended to the prefix “COINTEQ” and will be used to name the series.

Cross-references

See [Chapter 20](#) of the *User’s Guide* for details.

See also [coint](#) (p. 166).

makederivs[Equation Proc](#)

Make a group containing individual series which hold the derivatives of the equation specification.

Syntax

Equation Proc: `equation_name.makederivs(options) [names]`

If desired, enclose the name of the group object to contain the series in parentheses following the command name.

The argument specifying the names of the series is also optional. If not provided, EViews will name the series “DERIV##” where ## is a number such that “DERIV##” is the next available unused name. If the names are provided, the number of names must match the number of target series.

Cross-references

See [Chapter 22](#) of the *User’s Guide* for details on state space estimation.

See also [derivs](#) (p. 189), [grads](#) (p. 223), [makegrads](#) (p. 253).

makeendog[Sspace Proc](#) | [System Proc](#) | [Var Proc](#)

Make a group out of the endogenous series.

Syntax

Object Proc: `object_name.makeendog name`

Following the keyword `makeendog`, you should provide a name for the group to hold the endogenous series. If you do not provide a name, EViews will create an untitled group.

Note that in EViews 4, `endog` and `makeendog` are no longer supported for model objects. See instead, [makegroup](#) (p. 255).

Examples

```
var1.makeendog grp_v1
```

creates a group named GRP_V1 that contains the endogenous series in VAR1.

Cross-references

See also [endog](#) (p. 200) and [makegroup](#) (p. 255).

makefilter[Sspace Proc](#)

Create a “Kalman filter” sspace object.

Creates a new sspace object with all estimated parameter values substituted out of the specification. This procedure allows you to use the structure of the sspace without reference to estimated coefficients or the estimation sample.

Syntax

Sspace Proc: `sspace_name.makefilter [filter_name]`

If you provide a name for the model in parentheses after the keyword, EViews will quietly create the named model in the workfile. If you do not provide a name, EViews will open an untitled model window if the command is executed from the command line.

Examples

```
ss1.makefilter(kfilter)
```

creates a new sspace object named KFILTER, containing the specification in SS1 with estimated parameter values substituted for coef elements.

Cross-references

See [Chapter 22](#) of the *User’s Guide* for details on state space models.

See also [makesignals](#) (p. 260) and [makestates](#) (p. 262).

makegarch[Equation Proc](#)

Generate conditional variance series.

`makegarch` saves the estimated conditional variance (from an equation estimated using ARCH) as a named series.

Syntax

Equation Proc: `eq_name.makegarch name`

Following the `makegarch` keyword you should provide a name for the saved series. If you do not provide a name, EViews will name the series GARCH01 (if GARCH01 already exists, it will be named GARCH02, and so on).

Examples

```
equation eq1.arch sp c
eq1.makegarch cvar
plot cvar^.5
```

estimates a GARCH(1,1) model, saves the conditional variance as a series named CVAR, and plots the conditional standard deviation. If you only want to view a plot of the conditional standard deviation use the garch view.

Cross-references

See [Chapter 16](#) of the *User's Guide* for a discussion of GARCH models.

See also [arch](#) (p. 145) [archtest](#) (p. 147) and [garch](#) (p. 219).

makegrads

[Equation Proc](#) | [Logl Proc](#) | [Sspace Proc](#) | [System Proc](#)

Make a group containing individual series which hold the gradients of the objective function.

Syntax

Object Proc: object_name.**makegrads**(*options*) [*names*]

The argument specifying the names of the series is also optional. If the argument is not provided, EViews will name the series “GRAD##” where ## is a number such that “GRAD##” is the next available unused name. If the names are provided, the number of names must match the number of target series.

Options

<i>n = arg</i>	Name of group to object to contain the series.
----------------	--

Examples

```
eq1.grads (n=out)
```

creates a group named OUT containing series named GRAD1, GRAD2, and GRAD3.

```
eq1.grads (n=out) g1 g2 g3
```

creates the same group, but names the series G1, G2 and G3.

Cross-references

See also [derivs](#) (p. 189), [makederivs](#) (p. 251), [grads](#) (p. 223).

makegraph**Model Proc**

Make graph object showing model series.

Syntax

Model Proc: `model_name.makegraph(options) graph_name model_vars`

where `graph_name` is the name of the resulting graph object, and `models_vars` are the names of the series. The list of `model_vars` may include the following special keywords:

<code>@all</code>	All model variables.
<code>@endog</code>	All endogenous model variables.
<code>@exog</code>	All exogenous model variables
<code>@addfactor</code>	All add factor variables in the model

Options

<code>a</code>	Include actuals.
<code>c</code>	Include comparison scenarios.
<code>d</code>	Include deviations.
<code>n</code>	Do not include active scenario (by default the active scenario is included).
<code>t = transform_type</code>	<ul style="list-style-type: none"> “<code>t = level</code>”—display levels in graph (default). “<code>t = pch</code>”—display percent change in graph. “<code>t = pcha</code>”—display percent change (annual rates) in graph. “<code>t = pchy</code>”—display 1-year percent change in graph. “<code>t = dif</code>”—display 1-period differences in graph. “<code>t = dify</code>”—display 1-year differences in graph.
<code>s = solution_type</code>	<ul style="list-style-type: none"> “<code>s = d</code>”—deterministic. “<code>s = m</code>”—mean of stochastic. “<code>s = s</code>”—mean and ± 2 std. dev. of stochastic. “<code>s = b</code>”—mean and confidence bounds of stochastic.

g = grouping

“g = v”—group series in graph by model variable
(default).

“g = s”—group series in graph by scenario.

“g = u”—ungrouped (each series in its own graph).

Examples

```
mod1.makegraph(a) gr1 y1 y2 y3
```

creates a graph containing the model series Y1, Y2, and Y3 in the active scenario and the actual Y1, Y2, and Y3.

```
mod1.makegraph(a,t=pchy) gr1 y1 y2 y3
```

plots the same graph, but with data displayed as 1-year percent changes.

Cross-references

See “[Displaying Data](#)” on page 642 of the *User’s Guide* for details. See [Chapter 23](#) of the *User’s Guide* for a general discussion of models.

See [makegroup \(p. 255\)](#).

makegroup	Model Proc Pool Proc
------------------	--

Make group out of pool and ordinary series using a pool object or make group out of model series and display dated data table.

Syntax

Pool Proc: `pool_name.makegroup(group_name) name1 name2 ...`

Model Proc: `model_name.makegroup(options) group_name model_vars`

When used as a pool proc, following the `makegroup` keyword, you should provide a name for the new group, and then list the ordinary and pool series to be placed in the group.

When used as a model proc, the `makegroup` keyword should be followed by any options, the name of the destination group, and the list of model variables to be created. The options control the choice of model series, and transformation and grouping features of the resulting dated data table view. The list of `model_vars` may include the following special keywords:

@all	All model variables.
@endog	All endogenous model variables.
@exog	All exogenous model variables
@addfactor	All add factor variables in the model

Options

For Model Proc

a	Include actuals
c	Include comparison scenarios
d	Include deviations
n	Do not include active scenario (by default the active scenario is included)
t = <i>transform_type</i>	 “t = level”—display levels (default) “t = pch”—display percent change “t = pcha”—display percent change (annual rates) “t = pchy”—display 1-year percent change “t = dif”—display 1-period differences “t = dify”—display 1-year differences
s = <i>solution_type</i>	 “s = d”—deterministic “s = m”—mean of stochastic “s = s”—mean and std. dev. of stochastic “s = b”—mean and confidence bounds of stochastic
g = <i>grouping</i>	 “g = v”—group series in table by model variable (default) “g = s”—group series in table by scenario

Examples

```
pool1.makegroup(g1) x? z y?
```

places the ordinary series Z, and all of the series represented by the pool series X? and Y?, in the group G1.

`model1.makegroup(a,n) group1 @endog`
places all of the actual endogenous series in the group GROUP1.

Cross-references

See “[Displaying Data](#)” on page 642 of the *User’s Guide* for details. See [Chapter 23](#) of the *User’s Guide* for a general discussion of models.

See [makegraph \(p. 254\)](#).

makelimits	Equation Proc
-------------------	-------------------------------

Create vector of limit points from ordered models.

`makelimits` creates a vector of the estimated limit points from equations estimated by `ordered`.

Syntax

Equation Proc: `eq_name.makelimits name`

Provide a name for the vector after the `makelimits` keyword. If you do not provide a name, EViews will name the vector LIMITS1 (if LIMITS1 already exists, it will be named LIMITS2, and so on).

Examples

```
equation eq1.ordered edu c age race gender
eq1.makelimit cutoff
```

Estimates an ordered probit and saves the estimated limit points in a vector named CUT-OFF.

Cross-references

See “[Ordered Dependent Variable Models](#)” on page 438 of the *User’s Guide* for a discussion of ordered models.

makemodel	Equation Proc Logl Proc Pool Proc Sspace Proc System Proc Var Proc
------------------	--

Make a model from an estimation object.

Syntax

Object Proc: `object_name.makemodel(name) assign_statement`

You should enter the name of the object, followed by a dot and the keyword `makemode1`. If you provide a name for the model in parentheses after the keyword, EViews will quietly create the named model in the workfile. If you do not provide a name, EViews will open an untitled model window if the command is executed from the command line.

Examples

```
var var3.ls 1 4 m1 gdp tb3  
var3.makemode1(varmod) @prefix s_
```

estimates a VAR and makes a model named VARMOD from the estimated var object. VARMOD includes an assignment statement “`assign @prefix s_`”. Use the command “`show varmod`” or “`varmod.spec`” to open the VARMOD window.

Cross-references

See [Chapter 23](#) of the *User’s Guide* for a discussion of specifying and solving models in EViews.

See also [append \(p. 143\)](#), [merge \(p. 267\)](#) and [solve \(p. 334\)](#). For sspace objects, see [makefilter \(p. 252\)](#).

makeregs	Equation Proc
-----------------	-------------------------------

Make regressor group.

Creates a group containing the dependent and independent variables from an equation specification.

Syntax

Equation Proc: `equation_name.makeregs name`

Follow the keyword `makeregs` with the name of the group.

Examples

```
equation eq1.ls y c x1 x2 x3 z  
eq1.makeregs reggroup
```

creates a group REGGROUP containing the series Y X1 X2 X3 and Z.

Cross-references

See also [group \(p. 226\)](#) and [equation \(p. 200\)](#).

makeresids[Equation Proc](#) | [Pool Proc](#) | [System Proc](#) | [Var Proc](#)

Create residual series.

Creates and saves residuals in the workfile from an object with estimated equation or equations.

Syntax

Equation Proc:	<code>equation_name.makeresids(<i>options</i>) res1</code>
Object Proc:	<code>object_name.makeresids res1 res2 res3</code>
Pool Proc:	<code>pool_name.makeresids <i>poolser</i></code>

Follow the object name with a period and the `makeresid` keyword and a list of names to be given to the stored residuals. For pool residuals, you may use a cross section identifier “?”.

For multiple equation objects, make sure to provide as many names as there are equations. If there are fewer names than equations, EViews creates the extra residual series with names RESID01, RESID02, and so on. If you do not provide any names, EViews will also name the residuals RESID01, RESID02, and so on.

[NOTE: `makeresids` is no longer supported for the `sspace` object— see [makesignals \(p. 260\)](#) for relevant replacement routines]

Options

The following options are available only for the residuals from single equation objects:

<code>o</code> (default)	Ordinary residuals.
<code>s</code>	Standardized residuals (available only after weighted estimation and GARCH, binary, ordered, censored, and count models).
<code>g</code> (default for ordered models)	Generalized residuals (available only for binary, ordered, censored, and count models).
<code>n = arg</code>	Create group object to hold the residual series.

Examples

```
equation eq1.ls y c m1 inf unemp
eq1.makeresids res_eq1
```

estimates a linear regression of Y on a constant, M1, INF, UNEMP and saves the residuals as a series named RES_EQ1.

```
var macro_var.ls 1 4 y m1 r  
macro_var.makeresids resay res_m1 riser
```

estimates an unrestricted VAR with four lags and endogenous variables Y, M1, R and stores the residuals as RES_Y, RES_M1, RES_R.

```
equation probit1.binary y c x z  
probit1.makeresids(g) res_g  
series score1=res_g  
series score2=res_g*x  
series score3=res_g*z
```

estimates a probit model of Y on a constant, X, Z and stores the generalized residuals as RES_G. Then the vector of scores are computed as SCORE1, SCORE2, SCORE3.

```
pool1.makeresids res1_?
```

The residuals of each pool member will have a name starting with “RES1_” and the cross-section identifier substituted for the “?”.

Cross-references

See “[Weighted Least Squares](#)” on page 279 of the *User’s Guide* for a discussion of standardized residuals after weighted least squares and [Chapter 17](#) of the *User’s Guide* for a discussion of standardized and generalized residuals in binary, ordered, censored, and count models.

For state space models, see [makesignals \(p. 260\)](#).

makesignals	Sspace Proc
--------------------	-----------------------------

Generate signal series or signal standard error series from an estimated sspace object.

Options allow you to choose to generate one-step ahead and smoothed values for the signals, and the signal standard errors.

Syntax

Sspace Proc: name.makesignals(options) [names]

Follow the object name with a period and the `makesignal` keyword, options to determine the output type, and a list of names or wildcard expression identifying the series to hold the output. If a list is used to identify the targets, the number of target series must match

the number of states implied in the sspace. If any signal variable contain expressions, you may not use wildcard expressions in the destination names.

Options

<code>t = output_type</code>	Defines output type: “pred” (one-step ahead signal predictions) (default). “predse” (RMSE of the one-step ahead signal predictions). “resid” (error in one-step ahead signal predictions). “residse” (RMSE of the one-step ahead signal prediction; same as predse). “stdresid” (standardized one-step ahead prediction residual). “smooth” (smoothed signals). “smoothse” (RMSE of the smoothed signals). “disturb” (estimate of the disturbances). “disturbse” (RMSE of the estimate of the disturbances). “stddisturb” (standardized estimate of the disturbances).
<code>n = group_name</code>	Name of group to hold newly created series.

Examples

```
ss1.makesignals(t=smooth) sm*
```

produces smoothed signals in the series with names beginning with “sm”, and ending with the name of the signal dependent variable.

```
ss2.makesignals(t=pred, n=pred_sigs) sig1 sig2 sig3
```

creates a group named PRED_SIGS which contains the one-step ahead signal predictions in series SIG1, SIG2, and SIG3.

Cross-references

See [Chapter 22](#) of the *User’s Guide* for details on state space models. For additional discussion of wildcards, see [Appendix C, “Wildcards”, on page 657](#) of the *User’s Guide*.

See also [forecast](#) (p. 214), [makefilter](#) (p. 252) and [makestates](#) (p. 262).

makestates**Sspace Proc**

Generate state series or state standard error series from an estimated sspace object.

Options allow you to choose to generate one-step ahead, filtered, and smoothed values for the states, and the state standard errors.

Syntax

Sspace Proc: name.**makestates**(*options*) [*names*]

Follow the object name with a period and the **makestate** keyword, options to determine the output type, and a list of names or wildcard expression identifying the series to hold the output. If a list is used to identify the targets, the number of target series must match the number of names implied by the keyword.

Options

t = *output_type* Defines output type:

“pred” (one-step ahead state predictions) (default).

“predse” (RMSE of the one-step ahead state predictions).

“resid” (error in one-step ahead state predictions).

“residse” (RMSE of the one-step ahead state prediction; same as predse).

“filt” (filtered states).

“filtse” (RMSE of the filtered states).

“stdresid” (standardized one-step ahead prediction residual).

“smooth” (smoothed states).

“smoothse” (RMSE of the smoothed states).

“disturb” (estimate of the disturbances).

“disturbse” (RMSE of the estimate of the disturbances).

“stddisturb” (standardized estimate of the disturbances).

n = *group_name* Name of group to hold newly created series.

Examples

```
ss1.makestates(t=smooth) sm*
```

produces smoothed states in the series with names beginning with “sm”, and ending with the name of the state dependent variable.

```
ss2.makestates(t=pred, n=pred_states) sig1 sig2 sig3
```

creates a group named PRED_STATES which contains the one-step ahead state predictions in series SIG1, SIG2, and SIG3.

Cross-references

See [Chapter 22](#) of the *User’s Guide* for details on state space models. For additional discussion of wildcards, see [Appendix C, “Wildcards”, on page 657](#) of the *User’s Guide*.

See also [forecast \(p. 214\)](#), [makefilter \(p. 252\)](#) and [makesignals \(p. 260\)](#).

makestats	Pool Proc
------------------	---------------------------

Creates and saves series of descriptive statistics computed from a pool object.

Syntax

Pool Proc: `pool_name.makestats(options) ser1? ser2? @ stat_name`

Follow the object name with a period, the makestat command, a list of series names, the “@”-sign, and a list of command names for the statistics to compute. The statistics series will have a name with the cross-section identifier “?” replaced by the statistic command.

Options

Options in parentheses specify the sample to use to compute the statistics

- | | |
|-------------|------------------------|
| i | Use individual sample. |
| c (default) | Use common sample. |
| b | Use balanced sample. |

Command names for the statistics to be computed

- | | |
|------|-------------------------|
| obs | Number of observations. |
| mean | Mean. |
| med | Median. |
| var | Variance. |

sd	Standard deviation.
skew	Skewness.
kurt	Kurtosis.
jarq	Jarque-Bera test statistic.
min	Minimum value.
max	Maximum value.

Examples

```
pool1.makestats gdp_? edu_? @ mean sd
```

computes the mean and standard deviation of the GDP_? and EDU_? series in each period across the cross-section members using the default common sample. The mean and standard deviation series will be named GDP_MEAN, EDU_MEAN, GDP_SD, EDU_SD.

```
pool1.makestats(b) gdp_? @ max min
```

Computes the maximum and minimum values of the GDP_? series in each period using the balanced sample. The max and min series will be named GDP_MAX and GDP_MIN.

Cross-references

See [Chapter 21](#) of the *User's Guide* for details on the computation of these statistics and a discussion of the use of individual, common, and balanced samples in pool.

See also [describe \(p. 190\)](#).

makesystem	Pool Proc Var Proc
-------------------	--

Create system from a pool object or var.

Syntax

Pool Proc: `pool_name.makesystem(options) ser1? ser2? @ ser3? @ ser4?`

Var Proc: `var_name.makesystem(options)`

The first usage creates a system out of the pool specification. You can specify the treatment of the constant term and the name of the system as options in parentheses. Follow the dependent variable with a list of regressors; first list the regressors with common coefficients, then list the regressors with cross-section specific coefficients, separating the two lists with an “@”- sign. You can use the cross-section identifier “?” in the series name. You may also list a set of instrumental variables after a second “@”-sign. Note that you can always modify the specification in the system specification window.

The second form is used to create a system out of the current var specification. You may specify to order by series (default) or by lags.

Options

Options for Pools:

n	No intercept.
c (default)	Common intercept term.
f	Cross-section specific intercepts (fixed effects).
name	Name for the system object.

Options for VARs

n = name	Specify name for the system object.
bylag	Specify system by lags (default is to order by variables).

Examples

```
pool1.makesystem(sys1) inv? cap? @ val?
```

creates a system named SYS1 with INV? as the dependent variable and a common intercept for each cross-section member. The regressor CAP? is restricted to have the same coefficient in each equation, while the VAL? regressor has a different coefficient in each equation.

```
pool1.makesystem(sys2,f) inv? @ cap? @ @trend inv?(-1)
```

This command creates a system named SYS2 with INV? as the dependent variable and a different intercept for each cross-section member equation. The regressor CAP? enters each equation with a different coefficient and each equation has two instrumental variables @TREND and INV? lagged.

Cross-references

See [Chapter 19](#) of the *User's Guide* for a discussion of system objects in EViews.

matrix	Object Declaration
---------------	------------------------------------

Declare and optionally initializes a matrix object.

Syntax

- | | |
|----------|--|
| Command: | <code>matrix(r, c) matrix_name</code> |
| Command: | <code>matrix(r, c) matrix_name = assignment</code> |

The `matrix` keyword is followed by the name you wish to give the matrix. `matrix` also takes an optional argument specifying the row *r* and column *c* dimension of the matrix.

You can combine matrix declaration and assignment. If there is no assignment statement, the matrix will initially be filled with zeros.

Once declared, matrices may be resized by repeating the `matrix` command using the original name.

You should use `sym` for symmetric matrices.

Examples

```
matrix mom
```

declares a matrix named MOM with one element, initialized to zero.

```
matrix(3,6) coefs
```

declares a 3 by 6 matrix named COEFS, filled with zeros.

Cross-references

See [Chapter 4, “Matrix Language”, beginning on page 55](#) of the *Command and Programming Reference* for further discussion.

See [“Matrix” \(p. 31\)](#), [“Rowvector” \(p. 36\)](#) and [“Vector” \(p. 52\)](#) and [“Sym” \(p. 44\)](#) for full descriptions of the various matrix objects.

means	Equation View
-------	-------------------------------

Descriptive statistics by category of dependent variable.

Computes and displays descriptive statistics of the explanatory variables (regressors) of an equation by categories/values of the dependent variable. `means` is currently available only for equations estimated by `binary`.

Syntax

Equation View: `eq_name.means(options)`

Options

p	Print the descriptive statistics table.
---	---

Examples

```
equation eq1.binary(d=1) work c edu faminc
```

```
eq1.means
```

estimates a logit and displays the descriptive statistics of the regressors C, EDU, FAMINC for WORK = 0 and WORK = 1.

Cross-references

See [Chapter 17](#) of the *User's Guide* for a discussion of binary dependent variable models.

merge	Graph Proc Model Proc
-------	---

Merge objects.

When used as a model procedure, merges equations from an estimated equation, model, pool, system, or var object. When used as a graph procedure, merges graph objects.

If you supply only the object's name, EViews first searches the current workfile for the object containing the equation. If it is not there, EViews looks in the default directory for an equation or pool file (.DBE). If you want to merge the equations from a system file (.DBS), a var file (.DBV), or a model file (.DBL), include the extension in the command. You may include a path when merging files. You must merge objects to a model one at a time; `merge` appends the object to the equations already existing in the model.

When used as a graph procedure, `merge` combines graph objects into a single graph object. The graph objects to merge must exist in the current workfile.

Syntax

Model Proc: `model_name.merge(options) object_name`

Graph Proc: `graph name.merge graph1 graph2`

For `merge` as a model procedure, follow the `merge` keyword with a name of an object containing estimated equation(s) to merge.

For `merge` as a graph procedure, follow the `graph` command with a name for the new merged graph, the `merge` keyword, and a list of existing graph object names to merge.

Options

t	Merge an ASCII text file (only for model merge).
---	--

Examples

```
eq1.makemodel(mod1)
mod1.merge eq2
mod1.merge(t) c:\data\test.txt
```

The first line makes a model named MOD1 from EQ1. The second line merges (appends) EQ2 to MOD1 and the third line further merges (appends) the text file TEST from the specified directory.

```
graph mygra.merge gra1 gra2 gra3 gra4  
show mygra.align(4,1,1)
```

The first line merges the four graphs GRA1, GRA2, GRA3, GRA4 into a graph named MYGRA. The second line displays the four graphs in MYGRA in a single row.

Cross-references

See [Chapter 10](#) of the *User’s Guide* for a discussion of graphs.

metafile	Graph Proc
-----------------	----------------------------

Save graph to disk as an enhanced or ordinary Windows metafile.

Syntax

Graph Proc: `graph_name.metafile(options) name`

Follow the `metafile` keyword with a name for the metafile. The graph will be saved with a .WMF or a .EMF extension.

Options

- `o` Save the graph as an ordinary Windows metafile (.WMF). Enhanced Windows metafiles (.EMF) are the default.
- `c` Save the graph in color. The default is in black and white.

Examples

```
mygra1.metafile c:\report\chap1_1
```

saves MYGRA1 as an enhanced Windows metafile CHAP1_1.EMF in the designated directory.

Cross-references

See [Chapter 10](#) of the *User’s Guide* for a discussion of using graphs in other Windows programs.

ml[Logl Method](#) | [SSpace Method](#)

Maximum likelihood estimation of logl and state space models.

Syntax

Object Method: `object_name.ml(options)`

Options

<code>b</code>	Berndt-Hall-Hausman (BHHH) algorithm (default is Marquardt).
<code>m = integer</code>	Set maximum number of iterations.
<code>c = scalar</code>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
<code>showopts / -showopts</code>	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
<code>p</code>	Print basic estimation results.

Examples

`bvar.ml`

estimates the sspace object BVAR by maximum likelihood.

Cross-references

See [Chapters 18](#) and [22](#) of the *User's Guide* for a discussion of user specified likelihood and state space models.

model[Object Declaration](#) || [Object Proc](#)

Declare or create a model.

Syntax

Command: `model model_name`

Object Proc: `object_name.model(name) assign_statement`

The keyword `model` should be followed by a name for the model. To fill the model, you can use the `append` command.

Examples

```
model macro  
macro.append cs=10+0.8*y(-1)  
macro.append i=0.7*(y(-1)-y(-2))  
macro.append y=cs+i+g
```

declares an empty model named MACRO and adds three lines to MACRO.

Cross-references

See [Chapter 23](#) of the *User’s Guide* for a discussion of specifying and solving models in EViews.

See also [append \(p. 143\)](#), [merge \(p. 267\)](#) and [solve \(p. 334\)](#).

msg	Model View
------------	----------------------------

Display model solution messages.

Show view containing messages generated by the most recent model solution.

Syntax

Model View: `model_name.msg(options)`

Options

p Print the model solution messages.

Cross-references

See [Chapter 23](#) of the *User’s Guide* for a discussion of specifying and solving models in EViews.

See also [solve \(p. 334\)](#) and [solveopt \(p. 335\)](#).

na	Expression
-----------	------------

Not available code. “NA” is used to represent missing observations.

Examples

```
smp1 if y >= 0  
series z = y
```

```
smp1 if y < 0
z = na
```

generates a series Z containing the contents of Y, but with all negative values of Y set to “NA”.

NA values will also be generated by mathematical operations that are undefined:

```
series y = nrnd
y = log(y)
```

will replace all positive value of Y with log(Y) and all negative values with “NA”.

```
series test = (yt <> na)
```

creates the series TEST which takes the value one for nonmissing observations of the series YT. A zero value of TEST indicates missing values of the series YT.

Note that the behavior of missing values has changed since EViews Version 2. Previously, NA values were coded as 1e-37. This implied that in EViews 2 you could use the expression

```
series z = (y>=0)*x + (y<0)*na
```

to return the value of Y for non-negative values of Y and “NA” for negative values of Y. This expression will now generate the value “NA” for all values of Y, since mathematical expressions involving missing values always return “NA”. You must now use the smp1 statement as in the first example above, or the @recode function.

Cross-references

See “[Missing Data](#)” on page 92 of the *User’s Guide* for a discussion of working with missing values in EViews.

name	Graph Proc
----------------------	----------------------------

Change the series name for legends or axis labels.

name allows you to provide an alternative to the series name for legends or axis labels. The name command is available only for single graphs and will be ignored in multiple graphs.

Syntax

Graph Proc: graph_name.name(*n*) text_for_legend

Provide a series number in parentheses and text for legend (or axis label) after the name keyword. If you do not provide text, the series name will be removed from the legend/axis label.

Examples

```
graph g1.line(d) unemp gdp
g1.name(1) Civilian unemployment rate
g1.name(2) Gross National Product
```

The first line creates a line graph named G1 with dual scale, no crossing. The second line replaces the legend of the first series UNEMP, and the third line replaces the legend of the second series GDP.

```
graph g2.scat id w h
g2.name(1)
g2.name(2) weight
g2.name(3) height
g2.legend(l)
```

The first line creates a scatter diagram named G2. The second line removes the legend of the horizontal axis, and the third and fourth lines replace the legends of the variables on the vertical axis. The last line moves the legend to the left side of the graph.

Cross-references

See [Chapter 10](#) of the *User's Guide* for a discussion of working with graphs.

See also [displayname \(p. 192\)](#).

nnfit	Group View
--------------	----------------------------

Nearest neighbor fit.

Displays the fit of the second series (vertical axis) on the first series (horizontal axis) in the group.

Syntax

Group View: `group_name.nnfit(options)`

Options

<code>b = fraction</code> (default = 0.3)	Bandwidth as a fraction of the total sample. The larger the fraction, the smoother the fit.
<code>d = integer</code> (default = 1)	Degree of polynomial to fit.
<code>b</code>	Bracket bandwidth span.
<code>integer</code> (default = 100)	Approximate number of data points at which to compute the fit (if performing subsampling).
<code>x</code>	Exact (full) sampling. Default is Cleveland subsampling.
<code>u</code>	No local weighting. Default is local weighting with tricube weights.
<code>m = integer</code>	Set number of robustness iterations.
<code>s</code>	Symmetric neighbors. Default is nearest neighbors.
<code>s = name</code>	Save fitted series.
<code>p</code>	Print the Loess fit.

Examples

```
group gr1 gdp90 gdp50
gr1.nnfit(x, m=3)
```

displays the nearest neighbor fit of GDP50 on GDP90 with exact (full) sampling and 3 robustness iterations. Each local regression fits the default linear regression with tricube weighting using a bandwidth of span 0.3.

```
group gro1 weight height
gro1.nnfit(50, d=2, m=3)
```

displays the nearest neighbor fit of HEIGHT on WEIGHT by fitting approximately 50 data points. Each local regression fits a quadratic using tricube robustness weights with 3 iterations.

Cross-references

See “Scatter with Nearest Neighbor Fit” on page 235 of the *User’s Guide* for a discussion of nearest neighbor regressions.

See also [linefit](#) (p. 242) and [kerfit](#) (p. 237).

nrnd	Expression
------	------------

Normal random number generator.

When used in a series expression, nrnd generates (pseudo) random draws from a normal distribution with zero mean and unit variance.

Examples

```
smp1 @first @first  
series y = 0  
smp1 @first+1 @last  
series y = .6*y(-1)+.5*nrnd
```

generates a Y series that follows an AR(1) process with initial value zero. The innovations are normally distributed with mean zero and standard deviation 0.5.

```
series u = 10+@sqr(3)*nrnd  
series z = u+.5*u(-1)
```

generates a Z series that follows an MA(1) process. The innovations are normally distributed with mean 10 and variance 3.

```
series x = nrnd^2+nrnd^2+nrnd^2
```

generates an X series as the sum of squares of three *independent* standard normal random variables, which has a $\chi^2(3)$ distribution. Note that adding the sum of the three series is not the same as issuing the command

```
series x=3*nrnd^2
```

since the latter involves the generation of a single random variable.

The command

```
series x=@qchisq(rnd,3)
```

provides an alternative method of simulating random draws from a $\chi^2(3)$ distribution.

Cross-references

See “[Statistical Distribution Functions](#)” on page 444 for a list of other random number generating functions from various distributions.

See also [rnd \(p. 302\)](#), [rndint \(p. 302\)](#) and [rndseed \(p. 303\)](#).

open	Command
-------------	-------------------------

Opens a workfile, database, program file, or ASCII text file.

You should provide the file name *including its extension*. File names with the extensions .WF and .WF1 are processed as workfiles, .PRG files are treated as program files, and anything else is considered a text file. By default, open will look in the current directory unless an explicit path is provided.

Syntax

Command: `open file_name`

Examples

`open finfile.wf1`

opens a workfile named FINFILE from the default directory.

`open c:\prog\test1.prg`

opens a program file named TEST1 from the specified directory.

`open a:\mymemo.tex`

opens a text file named MYMEMO.TEX from the A drive.

`open f:\mydata\dbase.edb`

opens an EViews database file named DBASE from F:\MYDATA.

Cross-references

See [Chapter 4](#) of the *User's Guide* for a discussion of basic file operations.

See also [load \(p. 244\)](#), [fetch \(p. 205\)](#), [read \(p. 291\)](#).

options	Graph Proc
----------------	----------------------------

Set options for a graph object.

Allows you to change the option settings of an existing graph object. When `options` is used with a multiple graph, the options are applied to all graphs.

Syntax

Graph Proc: `graph_name.options option_list`

Note: the syntax of the `options` proc has changed considerably from version 3.1 of EViews. While not documented here, the EViews 3.x options are still (for the most part) supported. However, we do not recommend using the old options, as future support is not guaranteed.

Options

<code>size(<i>h, w</i>)</code>	Specifies the size of the plotting frame
<code>inbox / -inbox</code>	[Enclose / Do not enclose] the plotting frame in a box.
<code>indent / -indent</code>	[Indent / Do not indent] lines/bars/spikes in the plotting frame.
<code>color / -color</code>	Specifies that lines/bars [use / do not use] color—solid lines will be used if “lineauto” is set. If color is turned off (see setelem (p. 323)) gray scales will be used for bars and pies and line patterns will be used if “lineauto” is set.
<code>linesolid</code>	Do not use line patterns (all lines will be solid).
<code>linepat</code>	Use line patterns.
<code>lineauto</code>	Use solid lines when drawing in color and use patterns when drawing in black and white.
<code>barlabelabove / -barlabelabove</code>	[Place / Do not place] text value of data above bar in bar graph.
<code>barlabelinside / -barlabelinside</code>	[Place / Do not place] text value of data inside bar in bar graph.
<code>barspace / -barspace</code>	[Put / Do not put] space between bars in bar graph.
<code>pielabel / -pielabel</code>	[Place / Do not place] text value of data in pie chart.

The options which support the “-” may be proceeded by a “+” or “-” indicating whether to turn on or off the option. The “+” is optional.

Examples

```
graph1.option size(4,4) +inbox color
```

sets GRAPH1 to use a 4×4 frame enclosed in a box. The graph will use color.

```
graph1.option linepat -color size(2,8) -inbox
```

sets GRAPH1 to use a 2×8 frame with no box. The graph does not use color, with the lines instead being displayed using patterns.

Cross-references

See [Chapter 10](#) of the *User's Guide* for a discussion of graph options in EViews.

See also [scale \(p. 309\)](#) and [setelem \(p. 323\)](#).

ordered	Command Equation Method
----------------	--

Estimation of ordered dependent variable models.

Syntax

Command: **ordered(*options*)** *y x1 x2 x3*

Equation Method: *equation name.ordered(*options*)* *y x1 x2 x3*

When used as an equation procedure, **ordered** estimates the model and saves the results as an equation object with the given name.

Options

d = n (default)	Maximize using standard normal likelihood (ordered probit).
d = l	Maximize using logistic likelihood (ordered logit).
d = x	Maximize using (Type I) extreme value likelihood (ordered Gompit).
q (default)	Use quadratic hill climbing as maximization algorithm.
r	Use Newton-Raphson as maximization algorithm.
b	Use Berndt-Hall-Hausman as maximization algorithm.
h	Quasi-maximum likelihood (QML) standard errors.
g	GLM standard errors.
m = <i>integer</i>	Set maximum number of iterations.
c = <i>scalar</i>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.

s	Use the current coefficient values in C as starting values.
s = <i>number</i>	Specify a number between zero and one to determine starting values as a fraction of preliminary EViews default values (out of range values are set to “s = 1”).
showopts / -showopts	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
deriv = <i>keyword</i>	Set derivative methods. The argument <i>keyword</i> should be a one- or two-letter string. The first letter should either be “f” or “a” corresponding to fast or accurate numeric derivatives (if used). The second letter should be either “n” (always use numeric) or “a” (use analytic if possible). If omitted, EViews will use the global defaults.
p	Print results.

If you choose to employ user specified starting values, the parameters corresponding to the limit points must be in ascending order.

Examples

```
ordered(d=1,h) y c wage edu kids
```

estimates an ordered logit model of Y on a constant, WAGE, EDU, and KIDS with QML standard errors. This command uses the default quadratic hill climbing algorithm.

```
param c(1) .1 c(2) .2 c(3) .3 c(4) .4 c(5) .5  
equation eq1.binary(s) y c x z  
coef betahat=eq1.@coefs  
eq1.makelimit gamma
```

estimates an ordered probit model of Y on a constant, X, and Z from the specified starting values. The estimated coefficients are then stored in the coefficient vector BETAHAT, and the estimated limit points are stored in the vector GAMMA.

Cross-references

See “[Ordered Dependent Variable Models](#)” on page 438 of the *User’s Guide* for additional discussion.

See [binary \(p. 152\)](#) for the estimation of binary dependent variable models. See also [makelimits \(p. 257\)](#).

output	Command Equation View Logl View Pool View Sspace View System View Var View
---------------	---

Redirect printer output or display estimation output.

When used as a command, `output` redirects printer output. You may specify that any procedure that would normally send output to the printer instead keep the output as a frozen table or graph, or put the output in an ASCII text file.

Syntax

Command:	<code>output(<i>options</i>) base_name</code>
Command:	<code>output off</code>

By default, the `output` command redirects the output into frozen objects. You should supply a base name after the `output` keyword. Each subsequent print command will create a new table or graph object in the current workfile, using the base name and an identifying number. For example, if you supply the base name of “OUT”, the first print command will generate a table or graph named OUT1, the second print command will generate OUT2, and so on.

You can also use the optional settings, described below, to redirect table and text output to an ascii test file.

When followed by the optional keyword `off`, the `output` command turns off output redirection. Subsequent print commands will be directed to the printer.

Options

Options for output command

<code>t</code>	Redirect table and text output to an ascii file in the default directory. Graphic output will continue to be sent to the printer.
----------------	---

Options for output view

<code>p</code>	Print estimation output.
----------------	--------------------------

Examples

```
output print_
```

causes the first print command to generate a table or graph object named PRINT_1, the second print command to generate an object named PRINT_2, and so on.

```
output(t) c:\data\results
equation eq1.ls(p) log(gdp) c log(k) log(l)
eq1.resids(g,p)
output off
```

The second line redirects printing to the RESULTS.TXT file, while the print option of the third line sends the graph output to the printer. The last line turns output redirection off and restores normal printer use.

Cross-references

See “[Output Redirection](#)” beginning on page 651 of the *User’s Guide* for further discussion.

See also [pon \(p. 427\)](#), [poff \(p. 427\)](#).

override	Model Proc
----------	----------------------------

Specifies (or merges) overridden exogenous variables and add factors in the active scenario.

Syntax

Model Proc: model_name.override(*options*) ser1 ser2 ...

Follow the `override` keyword with the argument list containing the exogenous variables or add factors you wish to override.

Options

m Merge into (instead of replace) the existing override list.

Examples

```
mod1.override fed1 add1
```

creates an override list containing the variables FED1 and ADD1.

If you then issue the command

```
mod1.override fed1
```

EViews will replace the original exclude list with one containing only FED1. To add overrides to an existing list, use the “m” option:

```
mod1.override(m) add1
```

The override list now contains both series.

Cross-references

See the discussion in “[Specifying Scenarios](#)” on page 625 of the *User’s Guide*. See also [Chapter 23](#) of the *User’s Guide* for a general discussion of models.

See also [model](#) (p. 269), [exclude](#) (p. 203) and [solveopt](#) (p. 335).

param	Command
--------------	-------------------------

Set parameter values.

Allows you to set the current values of coefficient vectors. The command may be used to provide starting values for the parameters in nonlinear least squares, nonlinear system estimation, and (optionally) ARMA estimation.

Syntax

Command: `param coef_name(1) n1 coef_name(2) n2 ...`

Simply list, in pairs, the names of the coefficient vector and its element number followed by the corresponding starting values. You can use `param` to change the starting value of all, or just a subset, of the parameters in your equation.

Examples

```
param c(1) .2 c(2) .1 c(3) .5
```

resets the first three values of the coefficient vector C.

```
coef(3) beta
param beta(2) .1 beta(3) .5
```

The first line declares a coefficient vector BETA of length 3 filled with zeros. The second line resets the second and third elements of BETA to 0.1 and 0.5, respectively.

Cross-references

See “[Starting Values](#)” on page 297 of the *User’s Guide* for a discussion of setting initial values in nonlinear estimation.

pcomp	Group View
--------------	----------------------------

Principal components analysis.

Syntax

Group View: `group_name.pcomp(options) ser_name1 ser_name2 ...`

Enter the name of the group followed by a period, the keyword `pcomp` and optionally a list of k names to store the first k principal components. Separate each name in the list with a space and do not list more names than the number of series in the group.

Options

<code>cor</code>	(default) Use sample correlation matrix.
<code>cov</code>	Use sample covariance matrix.
<code>dof</code>	Degrees of freedom adjustment if cov option used. Default is no adjustment (which divides by n rather than $n - 1$).
<code>eigval = vec_name</code>	Specify name of vector to save the eigenvalues in workfile.
<code>eigvec = mat_name</code>	Specify name of matrix to save the eigenvectors in workfile.
<code>p</code>	Print results.

Examples

```
group g1 x1 x2 x3 x4
freeze(tab1) g1.pcomp(cor, eigval=v1, eigvec=m1) pc1 pc2
```

The first line creates a group named G1 with four series X1, X2, X3, X4. The second line stores the first two principal components in series named PC1 and PC2 using the sample correlation matrix. The output view is stored in a table named TAB1, the eigenvalues in a vector named V1, and the eigenvectors in a matrix named M1.

Cross-references

See “[Principal Components](#)” on page 219 of the *User’s Guide* for further discussion.

<code>pdl</code>	Expression
------------------	------------

Polynomial distributed lag specification.

This expression allows you to estimate polynomial distributed lag specifications in `ls` or `tsls` estimation. `pdl` forces the coefficients of a distributed lag to lie on a polynomial. The expression can only be used in estimation by list.

Syntax

Expression: `ls y x1 x2 pdl(series_name, lags, order[,options])`

Expression: `tsls y x1 x2 pdl(series_name, lags, order[,options]) @ z1 z2`

Options

The PDL specification must be provided in parentheses after the keyword `pdl` in the following order: the name of the series to which to fit a polynomial lag, the number of lags to include, the order (degree) of polynomial to fit, and an option number to constrain the PDL. By default, EViews does not constrain the endpoints of the PDL.

The constraint options are:

- 1 Constrain the near end of the distribution to zero.
- 2 Constrain the far end of the distribution to zero.
- 3 Constrain both the near and far end of the distribution to zero.

Examples

```
ls sale c pdl(order,8,3) ar(1) ar(2)
```

fits a third degree polynomial to the coefficients of eight lags of the regressor ORDER.

```
tsls sale c pdl(order,12,3,2) @ c pdl(rain,12,6)
```

fits a third degree polynomial to the coefficients of twelve lags of ORDER, constraining the far end to be zero. Estimation is by two-stage least squares, using a constant and a sixth degree polynomial fit to twelve lags of RAIN.

```
tsls y c x1 x2 pdl(z,12,3,2) @ c pdl(*) z2 z3 z4
```

When the PDL variable is exogenous in 2SLS, you may use “`pdl(*)`” in the instrument list instead of repeating the full PDL specification.

Cross-references

See “[Polynomial Distributed Lags \(PDLs\)](#)” on page 323 of the *User’s Guide* for further discussion.

pieCommand || [Graph Proc](#) | [Group View](#) | [Matrix View](#) | [Sym View](#)

Pie graph.

The `pie` command creates an untitled graph object containing pie charts for any number of series. There will be one pie for each date or observation number, provided the values of the series are positive. Each series is shown as a wedge in a different color/pattern, where the width of the wedge equals the percentage contribution of the series to the total of all listed series.

Syntax

Command: `pie(options) ser1 ser2 ser3`

Object View: `object_name.pie(options)`

Graph Proc: `graph_name.pie(options)`

To use `pie` as a command, simply list the name of the series to include in the pie chart. You may also change the graph type by using `pie` as a method. Simply list the graph name, followed by a period, and the “`pie`” keyword.

Options

`o = graph_name` Use appearance options from the specified graph.

`t = graph_name` Use appearance options and copy text and shading from the specified graph.

`p` Print the pie chart.

Examples

```
smp1 1990 1995  
pie cons inv gov
```

shows six pie charts, each divided into CONS, INV, and GOV.

```
graph gr1.line cons inv gov  
gr1.pie
```

creates a line graph GR1 and then changes the graph to a pie chart.

Cross-references

See [Chapter 10](#) of the *User’s Guide* for a discussion of graphs and templates.

See also [graph \(p. 224\)](#).

plot	Command
-------------	---------

Line graph.

See [line \(p. 241\)](#).

pool	Object Declaration
-------------	------------------------------------

Declare pool object.

Syntax

Command: `pool name id1 id2 id3 ...`

Follow the `pool` keyword with a name for the pool object. You may optionally provide the identifiers for the cross-section members of the pool object. Pool identifiers may be added or removed at any time using the `add` and `drop` views of a pool.

Examples

```
pool ZOO1 dog cat pig owl ant
```

Declares a pool object named `ZOO1` with the listed cross-section identifiers.

Cross-references

[“Pool” on page 34](#) contains a complete description of the pool object. See [Chapter 21](#) of the *User’s Guide* for a discussion of working with pools in EViews.

See [ls \(p. 245\)](#) for details on estimation using a pool object.

predict	Equation Proc
----------------	-------------------------------

Prediction table for binary and ordered dependent variable models.

The prediction table displays the actual and estimated frequencies of each value of the discrete dependent variable.

Syntax

Equation Proc: `eq_name.predict(options)`

For binary models, you may optionally specify how large the estimated probability must be to be considered as a success ($y = 1$). Specify the cutoff level as an option in parentheses after the keyword `predict`.

Options

<code>n (probability)</code> (default = .5)	Cutoff probability for success prediction in binary models.
<code>p</code>	Print the prediction table.

Examples

```
equation eq1.binary(d=1) work c edu age race  
eq1.predict(0.7)
```

Estimates a logit and displays the expectation-prediction table using a cutoff probability of 0.7.

Cross-references

See “[Binary Dependent Variable Models](#)” on page 421 of the *User’s Guide* for a discussion of binary models, and “[Expectation-Prediction \(Classification\) Table](#)” on page 429 of the *User’s Guide* for examples of prediction tables.

<code>print</code>	Command
--------------------	---------

The `print` command sends views of objects to the default printer.

Syntax

Command: `print(options) name1 name2 name3 ...`

Command: `print(options) name1.view`

`print` should be followed by a list of object names or a view of an object to be printed. The list of names must be of the same object type. If you do not specify the view of the object, `print` will print the default view for each object.

Options

<code>p</code>	Override the default output orientation (set by Print Setup) and print in portrait.
<code>l</code>	Override the default output orientation (set by Print Setup) and print in landscape.

Examples

```
print gdp log(gdp) d(gdp) @pch(gdp)
```

sends a table of GDP, log of GDP, first difference of GDP, and percentage change of GDP to the printer.

```
print graph1 graph2 graph3
```

prints three graphs on a single page.

To merge the three graphs, realign them in one row, and print in landscape:

```
graph mygra.merge graph1 graph2 graph3
mygra.align(3,1,1)
print(l) mygra
```

To estimate the equation EQ1 and send the output view to the printer.

```
print eq1.ls gdp c gdp(-1)
```

Cross-references

See “[Print Setup](#) beginning on page 651” of the *User’s Guide* for a discussion of print options and the Print Setup dialog.

probit	Command
---------------	---------

Estimation of binary dependent variable models with normal errors.

Equivalent to “[binary \(d=n\)](#)”.

See [binary \(p. 152\)](#).

program	Command
----------------	-------------------------

Declare a program.

Syntax

Command: **program** prog_name

Enter a name for the program after the **program** keyword. If you do not provide a name, EViews will open an untitled program window. Programs are text files, not objects.

Examples

```
program runreg
```

opens a program window named RUNREG which is ready for program editing.

Cross-references

See [Chapter 6, “EViews Programming”, on page 85](#) of the *Command and Programming Reference* for further details and examples of writing EViews programs.

qqplot	Group View Series View
---------------	--

Quantile-quantile plots.

`qqplot` plots the (empirical) quantiles of a series against the quantiles of a theoretical distribution or the quantiles of another series. You may specify the theoretical distribution and/or the method used to compute the empirical quantiles as options.

Syntax

Object View: `object_name.qqplot(options)`

Options

<code>n</code>	Plot against the quantiles of a normal distribution.
<code>u</code>	Plot against the quantiles of a uniform distribution.
<code>e</code>	Plot against the quantiles of an exponential distribution.
<code>l</code>	Plot against the quantiles of a logistic distribution.
<code>x</code>	Plot against the quantiles of an extreme value distribution.
<code>s = series_name</code>	Plot against the (empirical) quantiles of the specified series.
<code>q = arg</code> (default = “r”)	Compute quantiles using the definition: “b” (Blom), “r” (Rankit-Cleveland), “o” (simple fraction), “t” (Tukey), “v” (van der Waerden).
<code>p</code>	Print the QQ-plot.

Examples

```
equation eq1.binary(d=1) work c edu age race
```

```
eq1.makeresid(o)  res1
res1.qqplot(1)
```

estimates a logit, retrieves the residuals, and plots the quantiles of the residuals against those from the logistic distribution. If the error distribution is correctly specified, the QQ-plot should lie on a straight line.

Cross-references

See “[Quantile-Quantile](#)” on page 227 of the *User’s Guide* for a discussion of QQ-plots.

See also [cdfplot](#) (p. 157).

qstats	Var View
---------------	--------------------------

Multivariate residual autocorrelation portmanteau tests.

Syntax

Var View: `var_name.qstats(h,options)`

You must specify the highest order of lag *h* to test for serial correlation. *h* must be larger than the VAR lag order.

Options

<code>name = <i>arg</i></code>	Save <i>Q</i> -statistics in named matrix object. The matrix has two columns, where the first column is the unmodified and the second column is the modified <i>Q</i> -statistics.
<code>p</code>	Print the portmanteau test results.

Examples

```
var var1.ls 1 6 lgdp lm1 lcpi
show var1.qstats(12,name=q)
```

The first line declares and estimates a VAR. The second line displays the portmanteau tests for lags up to 12 and stores the *Q*-statistics in a matrix named *Q*.

Cross-references

See “[Diagnostic Views](#)” on page 522 of the *User’s Guide* for a discussion of the portmanteau tests and other VAR diagnostics.

See also [ar1m](#) (p. 148) for a related multivariate residual serial correlation LM test.

range	Command
-------	---------

Reset the workfile range (resize the workfile).

`range` is more general than `expand`, since it also allows you to shrink the workfile.

Note that data in the workfile will be lost if you shrink the workfile. Sample objects contained in the workfile may also be modified when shrunk.

Syntax

Command: **range** start end

For workfiles with dates, follow the `range` keyword with the new starting and ending dates. To resize an undated workfile, list a pair of observation numbers to indicate the new range.

Examples

```
workfile mywork m 1957:1 1995:12
range 1945:1 1989:12
```

The first line creates a monthly workfile from January 1957 to December 1995. The second line resizes the workfile to start from January 1945 and end at December 1989.

```
workfile test a 1930 1999
sample s1 1930 1972
range 1940 1989
```

The first line creates an annual workfile named TEST with range from 1930 to 1999. The second line creates a sample object named S1 with sample range 1930 to 1972. The third line resizes the workfile range from 1940 to 1989. Note that the sample range of S1 will also be modified to 1940–1989.

Cross-references

See “[Workfile Basics](#)” on page 33 of the *User’s Guide* for a discussion of workfiles.

See also [workfile](#) (p. 381), [expand](#) (p. 204), [smpl](#) (p. 332).

read	Command Coef Proc Matrix Proc Pool Proc Sym Proc Vector Proc
------	---

Read data from a foreign disk file.

The “read” command may be used to read multiple series into a workfile from a file on disk. When used as a procedure, `read` imports data directly into pool and matrix objects.

Syntax

Command: `read(options) path\file_name name1 name2 name3`

Command: `read(options) path\file_name n`

Coef Proc: `coef_name.read(options) path\file_name`

Pool Proc: `pool_name.read(options) path\file_name n1? n2? n3?`

Matrix Proc: `matrix_name.read(options) path\file_name`

You must supply the name of the source file. If you do not include the optional path specification, EViews will look for the file in the default directory. The input specification follows the source file name. Path specifications may point to local or network drives. If the path specification contains a space, you may enclose the entire expression in double quotation marks.

In the command proc form of `read`, there are two ways to specify the input series. First, you may list the names of the series in the order they appear in the file. Second, if the data file contains a header line for the series names, you may specify the number *n* of the series in the file instead of a list of names; EViews will name the series as given in the header line. If you specify a number and the data file does not contain a header line, EViews will name the series as SER01, SER02, SER03, and so on.

For the pool proc form of `read`, you must provide a list of ordinary or pool series.

Options

File type options

<code>t = dat, txt</code>	ASCII (plain text) files.
---------------------------	---------------------------

<code>t = wk1, wk3</code>	Lotus spreadsheet files.
---------------------------	--------------------------

<code>t = xls</code>	Excel spreadsheet files.
----------------------	--------------------------

If you do not specify the “t” option, EViews uses the file name extension to determine the file type. If you do specify the “t” option, then the file name extension will not be used to determine the file type.

Options for ascii text files

na = <i>text</i>	Specify text for NAs. Default is “NA”.
byper	Panel data organized by date/period. Default is data organized by cross-section (only for pool read).
bycross (default)	Panel data organized by cross-section (only for pool read).
t	Read by series (or transpose the data for matrix objects). Default is to read by observation with series in columns.
d = t	Treat tab as delimiter.
d = c	Treat comma as delimiter.
d = s	Treat space as delimiter.
d = a	Treat alpha numeric characters as delimiter.
custom = <i>symbol</i>	Specify symbol/character to treat as delimiter.
mult	Treat multiple delimiters as one.
name	Series names in file.
label = <i>integer</i>	Number of lines between the header line and the data. Must be used with the “name” option.
rect(default)	Treat file layout as rectangular.
norect	Do not treat file layout as rectangular.
skipcol = <i>integer</i>	Number of columns to skip. Must be used with the “rect” option.
skiprow = <i>integer</i>	Number of rows to skip. Must be used with the “rect” option.
comment = <i>symbol</i>	Specify character/symbol to treat as comment sign. Everything to the right of the comment sign is ignored. Must be used with the “rect” option.
singlequote	Strings are in single quotes, not double quotes.
dropstrings	Do not treat strings as NA; simply drop them.
negparen	Treat numbers in parentheses as negative numbers.
allowcomma	Allow commas in numbers (note that using commas as a delimiter takes precedence over this option).
currency = <i>symbol</i>	Specify symbol/character for currency data.

Options for spreadsheet (Lotus, Excel) files

<i>letter_number</i> (default = b2)	Coordinate of the upper-left cell containing data.
<i>s = sheet_name</i>	Sheet name for Excel 5–8 Workbooks.
bypers	Panel data organized by date/period. Default is data organized by cross-section (only for pool read).
bycross (default)	Panel data organized by cross-section (only for pool read).
<i>t</i>	Read by series (or transpose the data for matrix objects). Default is to read by observation with each series in columns.

Examples

```
read(t=dat,na=.) a:\mydat.raw id lwage hrs
```

reads data from an ASCII file MYDAT.RAW in the A drive. The data file is listed by observation, NA is coded as a “.” (dot or period), and there are three series, which are to be named ID, LWAGE, HRS in this order from left to right.

```
read(a2,s=sheet3) cps88.xls 10
```

reads data from an Excel file CPS88 in the default directory. The data are organized by observation, the upper left data cell is A2, and 10 series are read from a sheet named SHEET3.

```
read(a2, s=sheet2) "\\network\dr 1\cps91.xls" 10
```

reads the Excel file CPS91 from the network drive specified in the path.

Cross-references

See “[Importing Data](#)” on page 64 of the *User’s Guide* for a discussion and examples of importing data from external files.

See also [write \(p. 383\)](#).

rename	Command
---------------	-------------------------

Rename an object in the active workfile or database.

Syntax

Command:	<code>rename old_name new_name</code>
----------	---------------------------------------

After the `rename` keyword, first list the old name, followed by the new name.

Examples

```
rename temp_u u2
```

renames an object named TEMP_U as U2.

```
rename aa::temp_u aa::u2
```

renames TEMP_U to U2 in database AA.

Cross-references

See “[Object Basics](#)” on page [41](#) of the *User’s Guide* for a discussion of working with objects in EViews.

representation	Equation View Pool View Var View
----------------	--

Display text of specification for equation, pool, and var objects.

Syntax

Object View: `object_name.representation(options)`

Options

p	Print the representation text.
---	--------------------------------

Examples

```
pool1.representation
```

displays the specifications of the estimation object POOL1.

Cross-references

See [Chapters 11, 20, and 21](#) of the *User’s Guide*, for basic details on the relevant estimation objects.

See also [spec](#) (p. 337).

resample

[Group Proc](#) | [Series Proc](#)

Resample from observations in a series or group.

Syntax

Object Proc: `object_name.resample(options) [output_names]`

You should follow the `resample` keyword and options with a list of names or wildcard expression identifying the series to hold the output. If a list is used to identify the targets, the number of target series must match the number of names implied by the keyword.

Options

`outsmpl = " "` Sample to fill the new series. Either provide the sample range in double quotes or specify a named sample object. The default is the current workfile sample.

`name = group_name` Name of group to hold created series

`permute` Draw from rows without replacement. Default is to draw with replacement.

`weight = series_name` Name of series to be used as weights. The weight series must be non-missing and non-negative in the current workfile sample. The default is equal weights.

`block = integer` Block length for each draw. Must be a positive integer. The default block length is 1.

`withna (default)` Draw from all rows in the current sample, including those with NAs.

`dropna` Do not draw from rows that contain missing values in the current workfile sample.

`fixna` Excludes NAs from draws but copies rows containing missing values to the output series.

- Since we append a suffix for the new name, no series in the group can be an auto-series. For example, a group containing series such as `X(-1)` or `LOG(X)` will error. You will have to generate new series, say by setting `XLAG = X(-1)` or `LOGX = LOG(X)`. Then create a new group consisting of `XLAG` and `LOGX` and call the bootstrap procedure on this new group.

- If the name you provided to group the resampled series already exists and if it is a group object, the group object will be overwritten with the resample series. If it already exists but is not a group object, EViews will error.
- Block bootstrap (block length larger than 1) requires a continuous output sample. Therefore a block length larger than 1 cannot be used together with the “fixna” option and the “outsmpl” should not contain any gaps.
- The “fixna” option will have an effect only if there are missing values in the overlapping sample of the input sample (current workfile sample) and the output sample specified by “outsmpl”.
- If you specify “fixna”, we first copy any missing values in the overlapping sample to the output series. Then the input sample is adjusted to drop the rows containing missing values and the output sample is adjusted not to overwrite the copied values.
- If you choose “dropna” and the block length is larger than 1, the input sample may shrink in order to ensure that there are no missing values in any of the drawn blocks.
- If you choose “permute”, the block option will be reset to 1, the “dropna” and “fixna” options will be ignored (reset to the default “withna” option), and the “weight” option will be ignored (reset to default equal weights).

Examples

```
group g1 x y  
g1.resample
```

creates new series X_B and Y_B by drawing with replacement from the rows of X and Y in the current workfile sample. If X_B or Y_B already exist in the workfile, they will be overwritten if they are series objects; otherwise EViews will error. Note that only values of X_B and Y_B in the output sample (in this case the current workfile sample) will be overwritten.

```
g1.resample(weight=wt,suffix=_2) g2
```

will append “_2” to the names for the new series and group them in a group object named G2. The rows in the sample will be drawn with probabilities proportional to the corresponding values in the series WT. WT must have non-missing non-negative values in the current workfile sample.

Cross-references

See [“Resampling” on page 175](#) of the *User’s Guide* for a discussion of the resampling procedure. For additional discussion of wildcards, see [Appendix C, “Wildcards”, on page 657](#) of the *User’s Guide*.

See also [@resample](#) (p. 412) and [@permute](#) (p. 412) for sampling from matrices.

reset	Command Equation View
--------------	--

Ramsey's regression specification error test.

Syntax

Command:	<code>reset(<i>n, options</i>)</code>
Equation View:	<code>eq_name.reset(<i>n, options</i>)</code>

You must provide the number of powers of fitted terms *n* to include in the test regression.

Options

<code>p</code>	Print the test result.
----------------	------------------------

Examples

```
equation eq1.ls lwage c edu race gender
eq1.reset(2)
```

carries out the RESET test by including two terms, the square and the cube of the fitted values.

Cross-references

See “[Ramsey's RESET Test](#)” on page 382 of the *User's Guide* for a discussion of the RESET test.

residcor	Pool View Sspace View System View Var View
-----------------	--

Residual correlation matrix.

Displays the correlations of the residuals from each equation in the system, sspace, or var object, or from each pool cross-section. The sspace object residuals are the standardized one-step ahead signal forecast errors.

Syntax

Object View:	<code>object_name.residcor(<i>options</i>)</code>
--------------	---

Options

<code>p</code>	Print the correlation matrix.
----------------	-------------------------------

Examples

```
sys1.residcor
```

displays the residual correlation matrix of SYS1.

Cross-references

See [Chapters 19, 20, 21, and 22](#) of the *User's Guide* for a discussion of systems, VARs, pools, and sspace models.

See also [residcov \(p. 298\)](#) and [makeresids \(p. 259\)](#).

residcov	Pool View SSpace View System View Var View
----------	--

Residual covariance matrix.

Displays the covariances of the residuals from each equation in the system, sspace, or var object, or each pool cross-section. The sspace object residuals are the standardized one-step ahead forecast errors.

Syntax

Object View: object_name.residcov(*options*)

Options

p	Print the covariance matrix.
---	------------------------------

Examples

```
var1.residcov
```

displays the residual covariance matrix of VAR1.

Cross-references

See [Chapters 19, 20, 21, and 22](#) of the *User's Guide* for a discussion of systems, vars, pools, and sspace models.

See also [residcor \(p. 297\)](#) and [makeresids \(p. 259\)](#).

resids[Equation View](#) | [Pool View](#) | [Sspace View](#) | [System View](#) | [Var View](#)

Display residuals.

For equation and pool objects, `resids` allows you to display the actual, fitted values and residuals in either tabular or graphical form.

For sspace objects, `resids` allows you to display the actual-fitted-residual graph.

For system, var or pool objects, `resids` displays multiple graphs of the residuals. Each graph will contain the residuals for each equation in the VAR, or for each cross-section in the pool.

Syntax

Object View: `object_name.resids(options)`

Options

`g` Display graph(s) of residuals (default).

`t` Display table(s) of residuals (not available for system, pool, sspace or var objects).

`p` Print the table/graph.

Examples

```
equation eq1.ls m1 c inc tb3 ar(1)
eq1.resids
```

regresses M1 on a constant, INC, and TB3 correcting for first order serial correlation, and displays a table of actual, fitted, and residual series.

```
eq1.resids(g)
```

displays a graph of the actual, fitted, and residual series.

Cross-references

See also [graph \(p. 224\)](#).

results

[Equation View](#) | [Logl View](#) | [Pool View](#) | [SSpace View](#) | [System View](#) | [Var View](#)

Displays the results view of objects containing estimated equations.

Syntax

Object View: object_name.results(*options*)

Options

p Print the view.

Examples

```
equation eq1.ls m1 c inc tb3 ar(1)  
eq1.results(p)
```

estimates an equation using least squares, and displays and prints the results.

```
var mvar.ls 1 4 8 8 m1 gdp tb3 @ @trend(70.4)  
mvar.results(p)
```

prints the estimation results from the estimated VAR.

Cross-references

See the various chapters of the *User's Guide* for a description of the results view and data members for each of the objects.

rls

[Equation View](#)

Recursive least squares regression.

The **rls** view of an equation displays the results of recursive least squares (rolling) regression. This view is only available for equations estimated by ordinary least squares without ARMA terms.

You can plot various statistics from **rls** by choosing an option.

Syntax

Equation View: eq_name.rls(*options*) *c(1)* *c(2)* ...

Options

r	Plot the recursive residuals about the zero line with plus and minus two standard errors.
r,s	Plot the recursive residuals and save the residual series and their standard errors as series named R_RES and R_RESSE, respectively.
c	Plot the recursive coefficient estimates with two standard error bands. Follow the “rls(c)” command with a list of coefficients to be displayed.
c,s	Plot the listed recursive coefficients and save all coefficients and their standard errors as series named R_C1, R_C1SE, R_C2, R_C2SE, and so on.
o	Plot the <i>p</i> -values of recursive one-step Chow forecast tests.
n	Plot the <i>p</i> -values of recursive <i>n</i> -step Chow forecast tests.
q	Plot the CUSUM (standardized cumulative recursive residual) and 5 percent critical lines.
v	Plot the CUSUMSQ (CUSUM of squares) statistic and 5 percent critical lines.
p	Print the view.

Examples

```
equation eq1.ls m1 c tb3 gdp
eq1.rls(r,s)
eq1.rls(c) c(2) c(3)
```

plots and saves the recursive residual series and their standard errors from EQ1 as R_RES and R_RESSE. The third line plots the recursive slope coefficients of EQ1.

```
equation eq2.ls m1 c pdl(tb3,12,3) pdl(gdp,12,3)
eq2.rls(c) c(3)
eq2.rls(q)
```

The second command plots the recursive coefficient estimates of PDL02, the linear term in the polynomial of TB3 coefficients. The third line plots the CUSUM test statistic and the 5% critical lines.

Cross-references

See “[Recursive Least Squares](#)” on page 384 of the *User’s Guide*.

rnd	Expression
------------	------------

Uniform random number generator.

Generates (pseudo) random draws from a uniform distribution on (0,1). The expression may be included in a series expression or in an equation to be used in `solve`.

Examples

```
series u=5+(12-5)*rnd
```

generates a U series drawn from a uniform distribution on (5, 12).

Cross-references

See the list of available random number generators in [Appendix A, “Operator and Function Reference”, beginning on page 435](#).

See also [nrnd \(p. 274\)](#), [rndint \(p. 302\)](#) and [rndseed \(p. 303\)](#).

rndint	Command
---------------	---------

Uniform random integer generator.

The `rndint` command fills series, vector, and matrix objects with (pseudo) random integers drawn uniformly from zero to a user specified maximum. The `rndint` command ignores the current sample and fills the entire object with random integers.

Syntax

Command: `rndint(object_name, n)`

Type the name of the series, vector, or matrix object to fill followed by an integer value representing the maximum value *n* of the random integers. *n* should a positive integer.

Examples

```
series index  
rndint(index, 10)
```

fills the entire series INDEX randomly with integers from 0 to 10. Note that, unlike `genr`, `rndint` ignores the current sample and fills the series for the entire workfile range.

```
sym(3) var3
rndint(var3, 5)
```

fills the entire symmetric matrix VAR3 with random integers ranging from 0 to 5.

Cross-references

See the list of available random number generators in [Appendix A, “Operator and Function Reference”, beginning on page 435](#).

See also [nrnd \(p. 274\)](#), [rnd \(p. 302\)](#) and [rndseed \(p. 303\)](#).

rndseed	Command
----------------	---------

Seed the random number generator.

Use `rndseed` when you wish to generate a repeatable sequence of random numbers or to select the generator to be used.

EViews 4.0 now supports three types of uniform random number generators. Note that many of the non-uniform random numbers are generated as transformations of the uniform random number.

Syntax

Command: `rndseed(options) integer`

Options

<code>type = arg</code>	Type of random number generator: Knuth's (1997) lagged Fibonacci generator (“ <code>type = kn</code> ”), L'Ecuyer's (1999) combined multiple recursive generator (“ <code>type = le</code> ”), or Matsumoto and Nishimura's (1998) Mersenne Twister (“ <code>type = mt</code> ”).
-------------------------	---

Follow the `rndseed` keyword with the optional generator type, and then an integer for the seed.

- *Important backward compatibility note:* while the default generator type in EViews 4.0 remains the same as in EViews 3.0, setting the seed to the same number will no longer generate the same sequence. The reason for this discrepancy is due to the modification we introduced in the seeding of Knuth's generator. After reseeding the generator, we now discard a few draws as burn-in. The reason for this "wasteful" step is to avoid the repeats we observed in a short run of draws right after reseeding the Knuth generator.

When EViews starts up, the default generator type is set to the Knuth lagged Fibonacci generator. Unless changed using `rndseed`, Knuth's generator will be used for subsequent pseudo-random number generation.

	Knuth	L'Ecuyer	Mersenne Twister
Period	2^{129}	2^{319}	2^{19937}
Time (for 10^7 draws)	27.3 secs	15.7 secs	1.76 secs
Cases failed Diehard test	0	0	0

Examples

```
rndseed 123456  
genr t3=@qtdist(rnd,3)  
rndseed 123456  
genr t30=@qtdist(rnd,30)
```

generates random draws from a *t*-distribution with 3 and 30 degrees of freedom using the same seed.

Cross-references

See the list of available random number generators in [Appendix A, “Operator and Function Reference”, beginning on page 435](#).

See also [nrnd \(p. 274\)](#), [rnd \(p. 302\)](#) and [rndint \(p. 302\)](#).

rowvector	Object Declaration
------------------	------------------------------------

Declare a `rowvector` object.

The `rowvector` command declares and optionally initializes a (row) vector object.

Syntax

Command: `rowvector(n1) vector_name`
Command: `rowvector vector_name = assignment`

You may optionally specify the size (number of columns) of the row vector in parentheses after the `rowvector` keyword. If you do not specify the size, EViews creates a `rowvector` of size 1 unless the declaration is combined with an assignment.

By default, all elements of the vector are set to 0, unless an assignment statement is provided. EViews will automatically resize new rowvectors, if appropriate.

Examples

```
rowvector rvec1
rowvector(20) coefvec=2
rowvector newcoef=coefvec
```

RVEC1 is a row vector of size one with element 0. COEFVEC is a row vector of size 20 with all elements equal to 2. NEWCOEF is also a row vector of size 20 with all elements equal to 2.

Cross-references

See “[Rowvector](#)” on page 36 for a complete description of the rowvector object.

See also [coef](#) (p. 164) and [vector](#) (p. 377).

run	Command
------------	-------------------------

Run a program.

The `run` command executes a program. The program may be located in memory or stored in a program file on disk.

Syntax

Command: `run(options) path\program_name %0 %1 ...`

If the program has arguments, you should list them after the filename. EViews first sees if the specified program is in memory. If not, it looks for the program on disk in the current working directory, or in the specified path. EViews expects that program files will have a .PRG extension.

Options

<code>v</code> (default)	Verbose mode in which messages will be sent to the status line at the bottom of the EViews window (slow).
<code>q</code>	Quiet mode suppresses workfile display updates (faster execution).
<code>integer</code> (default = 1)	Set maximum errors allowed before halting the program.

Examples

```
run(q) simul x xhat
```

quietly runs a program named SIMUL from the default directory using arguments X and XHAT.

Since `run` is a command, it may also be placed in a program file. You should note that if you put the `run` command in a program file and then execute the program, EViews will stop after executing the program referred to by the `run` command. For example, if you have a program containing

```
run simul  
print x
```

the `print` statement will not be executed since execution will stop after executing the commands in `SIMUL.PRG`. If this behavior is not intended, you should consider using the `include` statement ([p. 425](#)).

Cross-references

See “[Executing a Program](#)” on page 86 for further details.

See also [include](#) (p. 425).

sample	Object Declaration
--------	------------------------------------

Declare a sample object.

The `sample` statement declares, and optionally defines, a sample object. If no `sample` statement is provided, the sample object will be set to the current workfile sample.

To reset the sample dates in a sample object, you must use the `set` procedure.

Syntax

Command: `sample name start end if_statement`

Follow the `sample` keyword with a name for the sample object and a sample range, possibly with an if condition.

Examples

```
sample ss
```

declares a sample object named SS and sets it to the current workfile sample.

```
sample s2 1974:1 1995:4
```

declares a sample object named S2 and sets it from 1974:1 to 1995:4.

```
sample fe_b1 @all if gender=1 and race=3
smp1 fe_b1
```

The first line declares a sample FE_BL with observations such that GENDER = 1 and RACE = 3 in the current workfile range. The second line sets the current sample to FE_BL.

```
sample sf @last-10 @last
```

declares a sample object named SF and sets it to the last 10 observations of the current workfile range.

```
sample s1 @first 1973:1
s1.set 1973:2 @last
```

The first line declares a sample object named S1 and sets it from the beginning of the workfile range to 1973:1. The second line resets S1 from 1973:2 to the end of the workfile range.

Cross-references

See “[Samples](#)” on page 60, and [Appendix B](#) of the *User’s Guide* for a discussion of using dates and samples in EViews.

See also [set](#) (p. 319) and [smp1](#) (p. 332).

sar	Expression
------------	------------

Seasonal autoregressive error specification.

sar can be included in **ls** or **tsls** specification to specify a multiplicative seasonal autoregressive term. A **sar(p)** term can be included in your equation specification to represent a seasonal autoregressive term with lag p . The lag polynomial used in estimation is the product of that specified by the **ar** terms and that specified by the **sar** terms. The purpose of the **sar** expression is to allow you to form the product of lag polynomials.

Examples

```
ls tb3 c ar(1) ar(2) sar(4)
```

TB3 is modeled as a second order autoregressive process with a multiplicative seasonal autoregressive term at lag four.

```
tsls sale c adv ar(1) sar(12) sar(24) @ c gdp
```

In this two-stage least squares specification, the error term is a first order autoregressive process with multiplicative seasonal autoregressive terms at lags 12 and 24.

Cross-references

See “[ARIMA Theory](#)” beginning on page 311 of the *User’s Guide* for details on ARMA and seasonal ARMA modeling.

See also [sma \(p. 330\)](#), [ar \(p. 144\)](#), and [ma \(p. 249\)](#).

save	Command
-------------	-------------------------

Save the current workfile to disk.

Syntax

Command: **save** *file_name*

You may supply a name to be given the file and the name may include a path designation.

Examples

The command

```
save
```

saves the current workfile in the default directory using the current name.

```
load macro1  
save a:\macro2
```

loads a workfile named MACRO1 from the default directory and saves it as MACRO2 in the A drive. The current workfiles will change to MACRO2. The original MACRO1 workfile will be unaltered.

Cross-references

See “[Workfile Basics](#)” on page 33 of the *User’s Guide* for a discussion of workfile operations.

See also [load \(p. 244\)](#) and [open \(p. 275\)](#).

scalar

Object Declaration

Declare a scalar object.

The **scalar** command declares a scalar object and optionally assigns a value.

Syntax

Command: **scalar scalar_name**

Command: **scalar scalar_name = assignment**

The **scalar** keyword should be followed by a valid name, and optionally, by an assignment. If there is no explicit assignment, the scalar will be assigned a value of zero.

Examples

```
scalar alpha
```

declares a scalar object named ALPHA with value zero.

```
equation eq1.ls res c res(-1 to -4) x1 x2
scalar lm=eq1.@regobs*eq1.@r2
show lm
```

runs a regression, saves the nR^2 as a scalar named LM, and displays its value in the status line at the bottom of the EViews window.

Cross-references

See “[Scalar](#)” on page 38 for a summary of the features of scalar objects in EViews.

scale[Graph Proc](#)

Sets axis and data scaling characteristics for the graph.

By default, EViews optimally chooses the axes to fit the graph data.

Syntax

Graph Proc: **graph_name.scale(axis) options_list**

Note: the syntax of the **scale** proc has changed considerably from version 3.1 of EViews. While not documented here, the EViews 3 options are still (for the most part) supported. However, we do not recommend using the old options as future support is not guaranteed.

Options

The *axis* parameter identifies which of the axes the `scale` proc modifies. If no option is specified, the `scale` proc will modify all of the axes. *axis* may take on one of the following values:

left, l	Left vertical axis.
right, r	Right vertical axis (for dual scale graphs).
bottom, b	Bottom axis (for xyline and scatter graphs).
top, t	Top axis.
all, al	All axes.

The options list may include any of the following options:

Data scaling options

linear	Linear data scaling (default).
linearzero	Linear data scaling (include zero when auto range selection is employed).
log	Logarithmic scaling.
norm	Norm (standardize) the data prior to plotting.

Axes scaling options

range(<i>arg</i>)	Specifies the endpoints for the scale: <i>arg</i> = “auto” (automatic choice), <i>arg</i> = “minmax” (use the maximum and minimum values of the data), <i>arg</i> = “ <i>n1,n2</i> ” (set minimum to <i>n1</i> and maximum to <i>n2</i> , e.g. “range(3,9)”).
overlap / -overlap	[Overlap / Do not overlap] scales on dual scale graphs.
dual / -dual	[Label / Do not label] both left and right axes (dual or single scale graphs).
grid / -grid	[Draw / Do not draw] grid lines.
zeroline / -zeroline	Draw a horizontal line at zero.
ticksout	Draw tickmarks outside the graph axes.
ticksin	Draw tickmarks inside the graph axes.
ticksboth	Draw tickmarks both outside and inside the graph axes.

ticksnone	Do not draw tickmarks.
label / - label	[Place / Do not place] numeric labels on the axes.
font(<i>arg</i>)	Set font size of labels.

Examples

To set the right scale to logarithmic with manual range you can enter

```
graph1.scale(right) log range(10, 30)
```

Alternatively,

```
graph1.scale zeroline ticksnone range(minmax)
```

draws a horizontal zero line and suppresses the tick marks on the axes which are defined to match the data range.

Cross-references

See [Chapter 10](#) of the *User’s Guide* for a discussion of graph options.

See also [dates \(p. 178\)](#), [options \(p. 275\)](#) and [setelem \(p. 323\)](#).

scat	Command Graph Proc Group View Matrix View Sym View
------	---

Scatter diagram.

The `scat` command produces an untitled graph object containing a scatter diagram of two or more series. When used as a view, `scat` displays a scatter diagram view of the series in the group or columns of a matrix.

By default, the first series or column of data will be located along the horizontal axis and the remaining data on the vertical axis. You may optionally choose to plot the data in pairs, where the first two series or columns are plotted against each other, the second two series or columns are plotted against each other, and so forth.

When used as a group or matrix view, there must be at least series or columns in the object.

Syntax

Command: `scat(options) ser1 ser2 ser3 ...`

Object View: `object_name.scat(options)`

Graph Proc: `graph_name.scat(options)`

For `scat` used as a command, list the names of series or groups. The first series is used for the horizontal axis and the remaining series are used for the vertical axis.

Options

Template and printing options

<code>o = graph_name</code>	Use appearance options from the specified graph object.
<code>t = graph_name</code>	Use appearance options and copy text and shading from the specified graph.
<code>p</code>	Print the scatter plot.

Scale options

<code>b</code>	Plot X and Y series in pairs.
<code>m</code>	Display scatter plots as multiple graphs.

Examples

```
scat unemp inf want
```

produces an untitled graph object containing a scatter plot with UNEMP on the horizontal and INF and WANT on the vertical axis.

```
group med age height weight
med.scat (t=scat2)
```

produces a scatter plot view of the group object MED using the graph object SCAT2 as a template.

```
group pairs age height weight length
pairs.scat (b)
```

produces a scatter plot view with AGE plotted against HEIGHT, and WEIGHT plotted against LENGTH.

Cross-references

See [Chapter 10](#) of the *User's Guide* for a discussion of graphs and templates.

See also [xylines](#) (p. 394) and [graph](#) (p. 224).

scatmat	Group View
----------------	----------------------------

Matrix of scatter plots.

The `scatmat` view displays a matrix of scatter plots for all pairs of series in a group.

Syntax

Group View: `group_name.scatmat(options)`

Options

p	Print the scatter plot matrix.
---	--------------------------------

Examples

```
group g1 weight height age
g1.scatmat
```

displays a 3×3 matrix of scatter plots for all pairs of the three series in group G1.

Cross-references

See “[Scatter](#)” on page [211](#) of the *User’s Guide* for a discussion of scatter plot matrices.

scenario	Model Proc
-----------------	----------------------------

Manage the model scenarios.

The scenario procedure is used to set the active and comparison scenarios for a model, to create new scenarios, to initialize one scenario with settings from another scenario, to delete scenarios, and to change the variable aliasing associated with a scenario.

Syntax

Model Proc: `model_name.scenario(options) "name"`

performs scenario options on a scenario given by the “name”. By default the scenario procedure also sets the active scenario to the specified name.

Options

c	Set the comparison scenario to the named scenario.
n	Create a new scenario with the specified name.
i = "name"	Copy the Excludes and Overrides from the named scenario.
d	Delete the named scenario.
a = <i>string</i>	Set the scenario alias string to be used when creating aliased variables (<i>string</i> is a 1 to 3 alphanumeric string to be used in creating aliased variables). If an underscore is not specified, one will be added to the beginning of the string. Examples: “_5”, “_T”, “S2”. The string “A” may not be used since it may conflict with add factor specifications.

Examples

The command string

```
mod1.scenario "baseline"
```

sets the active scenario to the baseline, while

```
mod1.scenario(c) "actuals"
```

sets the comparison scenario to the actuals (warning: this will overwrite any historical data in the solution period).

A newly created scenario will become the active scenario. Thus,

```
mod1(n) "Peace Scenario"
```

creates a scenario called "Peace Scenario" and makes it the active scenario. The scenario will automatically be assigned a unique numeric alias. To change the alias, simply use the "a = " option:

```
mod1(a=_ps) "Peace Scenario"
```

changes the alias for "Peace Scenario" to "_PS" and makes this scenario the active scenario.

The command:

```
mod1.scenario(n, a=w, i="Peace Scenario", c) "War Scenario"
```

creates a scenario called "War Scenario", initializes it with the Excludes and Overrides contained in "Peace Scenario", associates it with the alias "_W", and makes this scenario the comparison scenario.

```
mod1.scenario(i="Scenario 1") "Scenario 2"
```

copies the Excludes and Overrides in "Scenario 1" to "Scenario 2" and makes "Scenario 2" the active scenario.

Compatibility Notes

For backward compatibility with EViews 4.0, the option "a" may be used to set the comparison scenario, but is method not guaranteed to be supported in the future.)

In all of the arguments above the quotation marks around scenario name are currently optional. Support for the non-quoted names is provided for backward compatibility, but may be dropped in the future, thus

```
mod1.scenario Scenario 1
```

is currently valid, but may not be in future versions of EViews.

Cross-references

Scenarios are described in detail beginning on [page 616](#) of the *User's Guide*. [Chapter 23](#) of the *User's Guide* documents EViews models in depth.

See also [solve \(p. 334\)](#) in the *Command and Programming Reference*.

seas	Command Series Proc
------	--

Seasonal adjustment.

The `seas` command carries out seasonal adjustment using either the ratio to moving average, or the difference from moving average technique.

EViews also performs Census X11 and X12 seasonal adjustment. For details, see [x11 \(p. 387\)](#) and [x12 \(p. 388\)](#).

Syntax

Command:	<code>seas(options) series_name name_adjust name_fac</code>
Series Proc:	<code>series_name.seas(options) name_adjust name_fac</code>

To use `seas` as a command, list the name of the original series and the name to be given to the seasonally adjusted series. You may optionally include a third series name for the

seasonal factors. `seas` will display the seasonal factors using the convention of the Census X11 program.

`seas` used as a series procedure applies seasonal adjustment to a series.

Options

m	Multiplicative (ratio to moving average) method.
a	Additive (difference from moving average) method.

Examples

```
seas(a) pass pass_adj pass_fac
```

seasonally adjusts the series PASS using the additive method and saves the adjusted series as PASS_ADJ and the seasonal factors as PASS_FAC.

```
sales.seas(m) adj_sales
```

seasonally adjusts the series SALES using the multiplicative method and saves the adjusted series as ADJ_SALES.

Cross-references

See “[Seasonal Adjustment](#)” on page 177 of the *User’s Guide* for a discussion of seasonal adjustment methods.

See also [seasplot](#) (p. 316), [x11](#) (p. 387) and [x12](#) (p. 388).

seasplot	Series View
--------------------------	-----------------------------

Seasonal line graph.

`seasplot` displays a line graph view of a series ordered by season. Available only for quarterly and monthly frequencies.

Syntax

Series View: `series_name.seasplot(options)`

Options

m	Plot series split by season. Default is to plot series stacked by season.
---	---

Examples

```
freeze(gra_ip) ipnsa.seasplot
```

creates a graph object named GAR_IP that contains the stacked seasonal line graph view of the series IPNSA.

Cross-references

See “[Spreadsheet and Graph Views](#)” on page 151 of the *User’s Guide* for a brief discussion of seasonal line graphs.

See also [seas](#) (p. 315), [x11](#) (p. 387) and [x12](#) (p. 388).

series	Object Declaration
---------------	------------------------------------

Declare a series object.

The **series** command creates and optionally initializes a series, or modifies an existing series.

Syntax

Command:	series ser_name
Command:	series ser_name = <i>formula</i>

The **series** command should be followed by either the name of a new series, or an explicit or implicit expression for generating a series. If you create a series and do not initialize it, the series will be filled with NAs. Rules for composing a formula are given in “[Using Expressions](#)” on page 87 of the *User’s Guide*.

Examples

```
series x
```

creates a series named X filled with NAs.

Once a series is declared, you do not need to include the **series** keyword prior to entering the formula. The following example generates a series named LOW that takes value 1 if either INC is less than or equal to 5000 or EDU is less than 13 and 0 otherwise.

```
series low
low=inc<=5000 or edu<13
```

This example solves for the implicit relation and generates a series named Z which is the double log of Y so that Z = log(log(Y)).

```
series exp(exp(z))=y
```

The command

```
series z=(x+y)/2
```

creates a series named Z which is the average of series X and Y.

```
series cwage=wage*(hrs>5)
```

generates a series named CWAGE which is equal to WAGE if HRS exceeds 5 and zero otherwise.

```
series 10^z=y
```

generates a series named Z which is the base 10 log of Y.

The commands

```
series y_t=y  
smp1 if y<0  
y_t=na  
smp1 @all
```

generates a series named Y_T which replaces negative values of Y with NAs.

```
series z=@movav(x(+2),5)
```

creates a series named Z which is the *centered* moving average of the series X with two leads and two lags. This works only for centered moving averages over an odd number of periods.

```
series z=(.5*x(6)+@movsum(x(5),11)+.5*x(-6))/12
```

generates a series named Z which is the *centered* moving average of the series X over twelve periods.

```
genr y=2+(5-2)*rnd
```

creates series named Y which is a random draw from a uniform distribution between 2 and 5.

```
series y=3+@sqr(5)*nrnd
```

generates a series named Y which is a random draw from a normal distribution with mean 3 and variance 5.

Cross-references

A full listing and description of series functions and expressions is provided in [Appendix A, “Operator and Function Reference”, on page 435](#).

See [“Using Expressions” on page 87](#) of the *User’s Guide* for a discussion of rules for forming EViews expressions.

set	Sample Proc
------------	-----------------------------

Set the sample in a `sample` object.

The `set` procedure resets the sample of an existing sample object.

Syntax

Sample Proc: `sample_name.set sample_description`

Follow the `set` command with a sample description. See `sample` for instructions on describing a sample.

Examples

```
sample s1 @first 1973
s1.set 1974 @last
```

The first line declares and defines a sample object named S1 from the beginning of the workfile range to 1973. The second line resets S1 from 1974 to the end of the workfile range.

Cross-references

See [“Samples” on page 60](#) of the *User’s Guide* for a discussion of samples in EViews.

See also `sample` (p. 306) and `smp1` (p. 332).

setcell	Command
----------------	-------------------------

Insert contents into cell of a table.

The `setcell` command puts a string or number into a cell of a table.

Syntax

Command: `setcell(table_name, r, c, content[, "options"])`

Options

Provide the following information in parentheses in the following order: the name of the table object, the row number, *r*, of the cell, the column number, *c*, of the cell, a number or string to put in the cell, and optionally, a justification and/or numerical format code. A string of text must be enclosed in double quotes.

The justification options are:

c	Center the text/number in the cell.
r	Right-justify the text/number in cell.
l	Left-justify the text/number in cell.

The numerical format code determines the format with which a number in a cell is displayed; cells containing strings will be unaffected. The format code can either be a positive integer, in which case it specifies the number of decimal places to be displayed after the decimal point, or a negative integer, in which case it specifies the total number of characters to be used to display the number. These two cases correspond to the **fixed decimal** and **fixed character** fields in the number format dialog.

Note that when using a negative format code, one character is always reserved at the start of a number to indicate its sign, and that if the number contains a decimal point, that will also be counted as a character. The remaining characters will be used to display digits. If the number is too large or too small to display in the available space, EViews will attempt to use scientific notation. If there is insufficient space for scientific notation (six characters or less), the cell will contain asterisks to indicate an error.

Examples

```
setcell(tab1, 2, 1, "Subtotal")
```

puts the string "Subtotal" in row 2, column 1 of the table object named TAB1.

```
setcell(tab1, 1, 1, "Price and cost", "r")
```

puts the a right-justify string "Price and cost" in row 1, column 1 of the table object named TAB1.

Cross-references

[Chapter 5](#) describes table formatting using commands. See [Chapter 10](#) of the *User's Guide* for a discussion and examples of table formatting in EViews.

See also [setcolwidth \(p. 321\)](#).

setcolwidth	Command
--------------------	-------------------------

Set width of a column of a table.

The `setcolwidth` command determines the width of a column in a table. By default, each column is approximately 10 characters wide.

Syntax

Command: `setcolwidth(table_name, c, width)`

Options

To change the width of a column, provide the following information in parentheses in the following order: the name of the table, the column number *c*, and the number of characters *width* for the new width. EViews measures units in terms of the width of a numeric character. Because different characters have different widths, the actual number of characters that will fit may differ slightly from the number you specify.

Examples

```
setcolwidth (mytab, 2, 20)
```

sets the second column of table MYTAB to fit approximately 20 characters.

Cross-references

[Chapter 5](#) describes table formatting using commands. See also [Chapter 10](#) of the *User's Guide* for a discussion and examples of table formatting in EViews.

See also [setcell \(p. 319\)](#).

setconvert	Series View
-------------------	-----------------------------

Set frequency conversion method.

Determines the default frequency conversion method for a series when moved to different frequency workfiles (using `copy` or `fetch`).

You can override this default conversion method by specifying a frequency conversion method as an option in the [fetch \(p. 205\)](#) command.

If you do not set a conversion method and if you do not specify a conversion method as an option in the `fetch` command, EViews will use the conversion method set in the global option.

Syntax

Series View: `ser_name.setconvert up_method [down_method]`

Follow the series name with a period, the word `setconvert`, and option letters to specify the frequency conversion method. You may specify an up-conversion method, a down-conversion method, or both. If either the up-conversion or down-conversion method is omitted, EViews will set the method to the “use EViews default”.

Options

The following options control the frequency conversion method when copying series and group objects to a workfile, converting from *low* to *high* frequency:

r	Conversion by constant match average.
d	Conversion by constant match sum.
q	Conversion by quadratic match average.
t	Conversion by quadratic match sum.
i	Conversion by linear match last
c	Conversion by cubic match last.

The following options control the frequency conversion method when copying series and group objects to a workfile, converting from *high* to *low* frequency:

a	Conversion by taking the average of the nonmissing observations.
s	Conversion by taking the sum of the nonmissing observations.
f	Conversion by taking the first nonmissing observation.
l	Conversion by taking the last nonmissing observation.
x	Conversion by taking the maximum nonmissing observation.
m	Conversion by taking the minimum nonmissing observation.
an, na	Conversion by taking the average, propagating missing values.
sn, ns	Conversion by taking the sum, propagating missing values.

fn, nf	Conversion by taking the first observation, propagating missing values.
ln, nl	Conversion by taking the last observation, propagating missing values.
xn, nx	Conversion by taking the maximum observation, propagating missing values.
mn, nm	Conversion by taking the minimum observation, propagating missing values.

Examples

```
unemp.setconvert a
```

sets the default down-conversion method of the series UNEMP to take the average of non-missing observations, and resets the up-conversion method to use the global default.

```
ibm_hi.setconvert xn d
```

sets the default down-conversion method for IBM_HI to take the largest observation of the higher frequency observations, propagating missing values, and the default up-conversion method to constant, match sum.

```
consump.setconvert
```

resets both methods to the global default.

Cross-references

See “[Frequency Conversion](#)” on page 72 of the *User’s Guide* for a discussion of frequency conversions and the treatment of missing values.

See also [copy \(p. 168\)](#) and [fetch \(p. 205\)](#).

setelem	Graph Proc
----------------	----------------------------

Set individual line, bar and legend options for each series in the graph.

Syntax

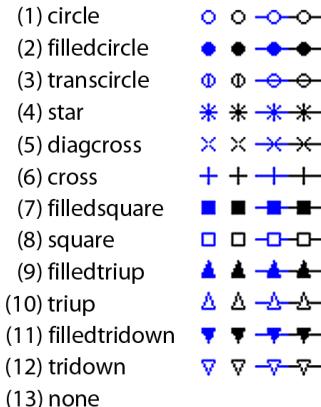
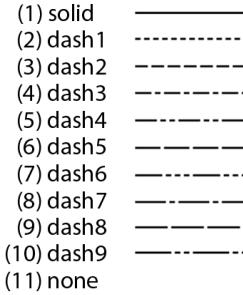
Graph Proc: `graph_name.setelem(n1) option_list`

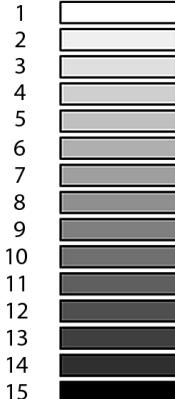
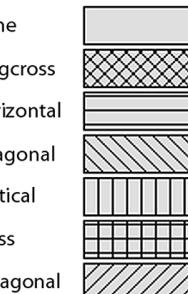
where *n1* is the identifier for the graph element whose options you wish to modify. For example, if *n1* = “2”, EViews will modify the second line in the graph.

Options

The option list for `setelem` may contain one or more of the following:

<code>lcolor(args)</code>	Sets the line and spike color.
<code>lpat(arg)</code>	<p>Sets the line pattern to the type: <i>arg</i> can be an integer from 1–11 or one of the matching keywords.</p> <p>Note that the “<i>lpat</i>” option interacts with the global options “color”, “lineauto”, “linesolid”, “linepat”. <i>In particular, you may need to set the global option “linepat” to enable the display of line patterns.</i> See options (p. 275).</p>
<code>lwidth(n1)</code>	Sets the line width: <i>n1</i> should be a number between “.25” and “5”, indicating the line width in points.
<code>symbol(arg)</code>	<p>Sets the drawing symbol: <i>arg</i> can be an integer from 1–13, or one of the matching keywords.</p> <p>Selecting a symbol automatically turns on symbol use. The “none” option turns off symbol use.</p>
<code>fcolor(args)</code>	Sets the fill color for symbols, bars, and pies.



gray(<i>n1</i>)	Sets the gray scale for bars and pies: <i>n1</i> should be an integer from 1–15 corresponding to one of the predefined gray scale settings (from lightest to darkest).	 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
hatch(<i>arg</i>)	Sets the hatch characteristics for bars and pies: <i>arg</i> can be an integer from 1–7, or one of the matching keywords.	 (1) none (2) diagcross (3) horizontal (4) fdiagonal (5) vertical (6) cross (7) bdiagonal
preset(<i>n1</i>)	Sets line and bar characteristics to EViews defaults: <i>n1</i> should be an integer from 1–10 representing settings for “lcolor”, “lpat”, “symbol”, “fcolor”, “gray”, and “hatch” specified in predefined definitions.	
default(<i>n1</i>)	Sets line and bar characteristics to user defaults: <i>n1</i> is an integer from 1–10 representing settings for “lcolor”, “lpat”, “symbol”, “fcolor”, “gray”, and “hatch” specified in the user default definitions.	
axis(<i>arg</i>)	Assigns the element to an axis: left (“l”), right (“r”), bottom (“b”), top (“t”).	
legend(<i>str</i>)	Assigns legend text for the element. <i>str</i> will be used in the legend to label the element.	

Examples

```
graph1.setelem(2) lcolor(blue) lwidth(2) symbol(circle)
```

sets the second line of GRAPH1 to be a blue line of width 2 with circle symbols.

```
graph1.setelem(1) lcolor(blue)
```

```
graph1.setelem(1) linecolor(0, 0, 255)
```

are equivalent methods of setting the linecolor to blue.

```
graph1.setelem(1) fillgray(6)
```

sets the gray-scale color for the first graph element.

The lines

```
graph1.setelem(1) scale(1)
```

```
graph1.setelem(2) scale(1)
```

```
graph1.setelem(3) scale(r)
```

create a dual scale graph where the first two series are scaled together and labeled on the left axis and the third series is scaled and labeled on the right axis.

```
graph1.setelem(2) legend("gross domestic product")
```

sets the legend for the second graph element.

Cross-references

See [Chapter 10](#) of the *User's Guide* for a discussion of graph options in EViews.

See also [scale \(p. 309\)](#) and [options \(p. 275\)](#)

setline	Command
----------------	-------------------------

Place a horizontal line in a table.

The **setline** command places a double horizontal line as a separator in a table.

Syntax

Command: **setline(table_name, r)**

Options

Specify the name of the table and the row number *r* in which to place the horizontal line.

Examples

```
setline(tab3,8)
```

places a (double) horizontal line in the eighth row of the table object TAB3.

Cross-references

[Chapter 5](#) describes table formatting using commands. See also [Chapter 10](#) of the *User's Guide* for a discussion and examples of table formatting in EViews.

sheet	Coef View Group View Matrix View Pool View Series View Table View Sym View Vector View
-------	--

Spreadsheet view.

The `sheet` view displays the spreadsheet view of the named object. For table objects, `sheet` simply displays the table.

Syntax

Object View: `object_name.sheet(options)`

Pool View: `pool_name.sheet(options) ser1? ser2? ...`

The `sheet` view of pool objects displays the spreadsheet view of the series in the pool. Follow the word `sheet` by a list of series to display; you may use the cross section identifier “?” in the series name.

Options

p

Print the spreadsheet view.

Examples

```
tab1.sheet
```

displays the spreadsheet view of TAB1.

Cross-references

See [Chapter 4](#) of the *User's Guide* for a discussion of the spreadsheet view of series and groups, and [Chapter 21](#) of the *User's Guide* for a discussion of pools. See also the chapters for the other individual objects.

show	Command
------	---------

Display objects.

The `show` command displays series or other objects on your screen. A scalar object is displayed in the status line at the bottom of the EViews window.

Syntax

Command: `show object_name.view`

Command: `show name1 name2 name3`

The command `show` should be followed by the name of an object, or an object name with an attached view.

For series and graph objects, `show` can operate on a list of names. The list of names must be of the same type. `show` creates and displays an untitled group or multiple graph object.

Examples

```
genr x=nrnd  
show x.hist  
close x
```

generates a series X of random draws from a standard normal distribution, displays the histogram view of X, and closes the series window.

```
show wage log(wage)
```

opens an untitled group window with the spreadsheet view of the two series.

```
freeze(gra1) wage.hist  
genr lwage=log(wage)  
freeze(gra2) lwage.hist  
show gra1 gra2
```

opens an untitled graph object with two histograms.

Cross-references

See [Chapter 3](#) for a complete listing of the views of the various objects.

See also [close \(p. 163\)](#).

signalgraphs[Sspace View](#)

Graph signal series.

Display graphs of a set of signal series computed using the Kalman filter.

Syntax

Sspace View: `object_name.signalgraphs(options)`

Options

`t = output_type` Defines output type:

“`t = pred`” (one-step ahead signal predictions) (default).

“`t = predse`” (RMSE of the one-step ahead signal predictions).

“`t = resid`” (error in one-step ahead signal predictions).

“`t = residse`” (RMSE of the one-step ahead prediction; same as `predse`).

“`t = stdresid`” (standardized one-step ahead prediction residual).

“`t = smooth`” (smoothed signals).

“`t = smoothse`” (RMSE of the smoothed signals).

“`t = disturb`” (estimate of the disturbances).

“`t = disturbse`” (RMSE of the estimate of the disturbances).

“`t = stddisturb`” (standardized estimate of the disturbances).

Examples

```
ss1.signalgraphs(t=smooth)
```

```
ss1.signalgraphs(t=smoothse)
```

displays a graph view containing the smoothed signal values, and then displays a graph view containing the root MSE of the smoothed states.

Cross-references

See [Chapter 22](#) of the *User’s Guide* for a discussion of state space models.

See also [stategraphs](#) (p. 342), [makesignals](#) (p. 260) and [makestates](#) (p. 262).

sma	Expression
-----	------------

Seasonal moving average error specification.

`sma` can be included in a `ls` or `tsls` specification to specify a multiplicative seasonal moving average term. A `sma(p)` term can be included in your equation specification to represent a seasonal moving average term of order p . The lag polynomial used in estimation is the product of that specified by the `ma` terms and that specified by the `sma` terms. The purpose of the `sma` expression is to allow you to form the product of lag polynomials.

Examples

```
ls tb3 c ma(1) ma(2) sma(4)
```

`TB3` is modeled as a second order moving average process with a multiplicative seasonal moving average term at lag four.

```
tsls(z) sale c adv ma(1) sma(12) sma(24) @ c gdp
```

In this two-stage least squares specification, the error term is a first order moving average process with multiplicative seasonal moving average terms at lags 12 and 24. The “z” option turns off backcasting.

Cross-references

See “[ARIMA Theory](#)” beginning on page 311 of the *User’s Guide* for details on ARMA and seasonal ARMA modeling.

See also [sar \(p. 307\)](#), [ar \(p. 144\)](#), and [ma \(p. 249\)](#).

smooth	Command Series Proc
--------	--

Exponential smoothing.

`smooth` forecasts a series using one of a number of exponential smoothing techniques. By default, `smooth` estimates the damping parameters of the smoothing model to minimize the sum of squared forecast errors, but you may specify your own values for the damping parameters.

`smooth` automatically calculates in-sample forecast errors and puts them into the series `RESID`.

Syntax

Command: **smooth(*method*)** *series_name smooth_name freq*

Series Proc: *series_name.smooth(*method*)* *smooth_name freq*

You should follow the **smooth** keyword with the name of the series and a name for the smoothed series. You must specify the smoothing method in parentheses as an option. The optional *freq* may be used to override the default for the number of periods in the seasonal cycle. By default, this value is set to the workfile frequency (e.g. — 4 for quarterly data). For undated data, the default is 5.

Options

Smoothing method options

s,x	Single exponential smoothing for series with no trend. You may optionally specify a number “x” between zero and one for the mean parameter.
d,x	Double exponential smoothing for series with a trend. You may optionally specify a number “x” between zero and one for the mean parameter.
n,x,y	Holt-Winters without seasonal component. You may optionally specify numbers “x”, “y” between zero and one for the mean and trend parameters, respectively.
a,x,y,z	Holt-Winters with additive seasonal component. You may optionally specify numbers “x”, “y”, “z” between zero and one for the mean, trend, and seasonal parameters, respectively.
m,x,y,z	Holt-Winters with multiplicative seasonal component. You may optionally specify numbers “x”, “y”, “z” between zero and one for the mean, trend, and seasonal parameters, respectively.

Other Options:

p	Print a table of forecast statistics.
---	---------------------------------------

If you wish to set only some of the damping parameters and let EViews estimate the other parameters, enter the letter “e” where you wish the parameter to be estimated.

If the number of seasons is different from the frequency of the workfile (an unusual case that arises primarily if you are using an undated workfile for data that are not monthly or

quarterly), you should enter the number of seasons after the smoothed series name. This optional input will have no effect on forecasts without seasonal components.

Examples

```
smooth(s) sales sales_f
```

smooths the SALES series by a single exponential smoothing method and saves the smoothed series as SALES_F. EViews estimates the damping (smoothing) parameter and displays it with other forecast statistics in the SALES series window.

```
smooth(n,e,.3) tb3 tb3_hw
```

smooths the TB3 series by a Holt-Winters no seasonal method and saves the smoothed series as TB3_HW. The mean damping parameter is estimated while the trend damping parameter is set to 0.3.

```
smp1 @first @last-10  
smooth(m,.1,.1,.1) order order_hw  
smp1 @all  
graph gra1.line order order_hw  
show gra1
```

smooths the ORDER series by a Holt-Winters multiplicative seasonal method leaving the last 10 observations. The damping parameters are all set to 0.1. The last three lines plot and display the actual and smoothed series over the full sample.

Cross-references

See “[Exponential Smoothing](#)” on page 190 of the *User’s Guide* for a discussion of exponential smoothing methods.

smpl	Command
------	---------

Set sample range.

The smpl command sets the workfile sample to use for statistical operations and series assignment expressions.

Syntax

Command: `smpl start1 end1 start2 end2 ... if_condition`

Command: `smpl sample_name`

List the date or number of the first observation and the date or number of the last observation for the sample. Rules for specifying dates are given in [Appendix B, “Date Formats”, on](#)

[page 653](#) of the *User's Guide*. `smpl` may contain more than one pair of beginning and ending observations.

The `smpl` command also allows you to select observations on the basis of conditions specified in an `if` statement. This enables you to use logical operators to specify what observations to include in EViews' procedures. Put the `if` statement after the pairs of dates.

You can also use `smpl` to set the current observations to the contents of a named sample object; put the name of the sample object after the command `smpl`.

Special keywords for `smpl`

The following “@-keywords” can be used in a `smpl` command:

<code>@all</code>	The whole workfile range.
<code>@first</code>	The first observation in the workfile.
<code>@last</code>	The last observation in the workfile.

Examples

```
smpl 1955:1 1972:12
```

sets the workfile sample from 1955:1 to 1972:12

```
smpl @first 1940 1946 1972 1975 @last
```

excludes observations (or years) 1941–1945 and 1973–1974 from the workfile sample.

```
smpl if union=1 and edu<=15
```

sets the sample to those observations where UNION takes the value 1 and EDU is less than or equal to 15.

```
sample half @first @first+@obs(x)/2
smpl half
smpl if x>0
smpl @all if x>0
```

The first line declares a sample object named HALF which includes the first half of the series X. The second line sets the sample to HALF and the third line sets the sample to those observations in HALF where X is positive. The last line sets the sample to those observations where X is positive over the full sample.

Cross-references

See “[Samples](#)” on page 60 of the *User's Guide* for a discussion of samples in EViews.

See also [set](#) (p. 319) and [sample](#) (p. 306).

solve	Command Model Proc
--------------	---------------------------------------

Solve the model.

`solve` finds the solution to a simultaneous equation model for the set of observations specified in the current workfile sample.

Syntax

Command: `solve(options)`
Model Proc: `model_name.solve(options)`

Note: When `solve` is used in a program (batch mode) models are always solved over the workfile sample. If the model contains a solution sample, it will be ignored in favor of the workfile sample.

You should follow the name of the model after the `solve` command or use `solve` as a procedure of a named model object. The default solution method is dynamic simulation. You may modify the solution method as an option.

`solve` first looks for the specified model in the current workfile. If it is not present, `solve` attempts to `fetch` a model file (.DBL) from the default directory or, if provided, the path specified with the model name.

Options

`solve` can take any of the options available in [solveopt](#) (p. 335).

Examples

```
solve mod1
```

solves the model MOD1 using the default solution method.

```
nonlin2.solve(m=500,e)
```

solves the model NONLIN2 with an extended search of up to 500 iterations.

Cross-references

See [Chapter 23](#) of the *User's Guide* for a discussion of models.

See also [model](#) (p. 269), [msg](#) (p. 270) and [solveopt](#) (p. 335).

solveopt	Model Proc
----------	----------------------------

Solve options for models.

`solveopt` sets options for model solution but does not solve the model. The same options can be set directly in a `solve` procedure.

Syntax

Model Proc: `model_name.solveopt(options)`

Options

<code>s = arg</code> (default = d)	Solution type: “d” (deterministic), “m” (stochastic – collect means only), “s” (stochastic – collect means and s.d.), “b” (stochastic – collect means and confidence bounds), “a” (stochastic – collect all; means, s.d. and confidence bounds).
<code>d = arg</code> (default = d)	Model solution dynamics: “d” (dynamic solution), “s” (static solution), “f” (fitted values – single equation solution).
<code>m = integer</code> (default = 5000)	Maximum number of iterations for solution (maximum 100,000).
<code>c = number</code> (default = 1e-8)	Convergence criterion. Based upon the maximum change in any of the endogenous variables in the model. You may set a number between 1e-15 and 0.01.
<code>r = integer</code> (default = 1000)	Number of stochastic repetitions (used with stochastic “s = ” options).
<code>b = number</code> (default = .95)	Size of stochastic confidence intervals (used with stochastic “s = ” options).
<code>a = arg</code> (default = f)	Alternate scenario solution: “t” (true - solve both active and alternate scenario and collect deviations for stochastic), “f” (false - solve only the active scenario).
<code>o = arg</code> (default = g)	Solution method: “g” (Gauss-Seidel), “e” (Gauss-Seidel with extended search/reduced step size), “n” (Newton), “m” (Newton with extended search/reduced step size).

i = arg	Set initial (starting) solution values: “a” (actuals), “p” (values in period prior to start of solution period).
n = arg (default = t)	NA behavior: “t” (true - stop on “NA” values), “f” (false - do not stop when encountering “NA” values). Only applies to deterministic solution; EViews will always stop on “NA” values in stochastic solution.
e = arg	Excluded variables initialized from actuals: “t” (true), “f” (false).
t = arg	Terminal condition for forward solution: “u” (user supplied), “l” (constant level), “d” (constant difference), “g” (constant growth rate).
g = arg (default = 7)	Number of digits to round solution: an integer value (number of digits), “n” (do not roundoff).
z = arg (default = 1e-7)	Zero value: a positive number below which the solution (absolute value) is set to zero), “n” (do not set to zero).

Cross-references

See [Chapter 23](#) of the *User’s Guide* for a discussion of models.

See also [model \(p. 269\)](#), [msg \(p. 270\)](#) and [solve \(p. 334\)](#).

sort	Command
------	-------------------------

Sort the workfile.

The `sort` command sorts *all* series in the workfile on the basis of the values of one or more of the series. For purposes of sorting, NAs are considered to be smaller than any other value.

By default, EViews will sort the series in ascending order. You may use options to override the default behavior.

Syntax

Command: `sort(options) ser1 ser2 ...`

List the name of the series by which you wish to sort the workfile. If you list two or more series, `sort` uses the values of the second series to resolve ties from the first series, and values of the third series to resolve ties from the second, and so on.

Options

d sort in descending order.

Examples

`sort(d) inc`

sorts all series in the workfile in order of the INC series with the highest value of INC first. NAs in INC (if any) will be placed at the bottom.

`sort gender race wage`

sorts all series in the workfile in order of the values of GENDER from low to high, with ties resolved by ordering on the basis of RACE, with further ties resolved by ordering on the basis of WAGE.

Cross-references

See “[Sorting Workfiles](#)” on page 39 of the *User’s Guide*.

spec	Logl View Model View SSpace View System View
------	--

Display the text specification view for logl, model, sspace, system objects.

Syntax

Object View: `object_name.spec(options)`

Options

p Print the specification text.

Examples

`model1.spec`

displays the specification of the object MODEL1.

Cross-references

See also [append](#) (p. 143), [merge](#) (p. 267), [text](#) (p. 363).

spike

[Coef View](#) | [Graph Proc](#) | [Group View](#) | [Matrix View](#) | [Series View](#) | [Sym View](#) | [Vector View](#)

Spike graph.

The spike graph view of a group creates spike graphs for all series in the group. The spike graph view of a matrix plots spikes for each column in the matrix.

Syntax

Object View: object_name.spike(*options*)
Graph Proc: graph_name.spike(*options*)

Options

Template and printing options

<i>o = graph_name</i>	Use appearance options from the specified graph.
<i>t = graph_name</i>	Use appearance options and copy text and shading from the specified graph.
<i>p</i>	Print the spike graph.

Scale options

<i>a</i> (default)	Automatic scaling. The series are graphed in their original units and the range of the graph is chosen to accommodate the highest and lowest values of the series.
<i>d</i>	Dual scaling. The first series is scaled on the left and all other series are scaled on the right.
<i>s</i>	Stacked spike graph. Each segment represents the cumulative total of the series listed (may not be used with the “ <i>l</i> ” option).
<i>l</i>	Spike graph for the first series listed and a line graph for all subsequent series (may not be used with the “ <i>s</i> ” option).
<i>x</i>	Same as the “ <i>d</i> ” option (dual scaling).
<i>m</i>	Plot spikes in multiple graphs.

Examples

```
group g1 gdp cons m1
```

```
g1.spike(d)
```

plots line graphs of the three series in group G1 using dual scaling.

```
matrix1.spike(t=mygra)
```

displays spike graphs of the columns of MATRIX1 using the graph object MYGRA as a template.

```
graph1.spike(m)
```

changes GRAPH1 so that it contains spike graphs of each of the series in the original graph, with each graph plotted on separate axes.

Cross-references

See [Chapter 10](#) of the *User's Guide* for a detailed discussion of graphs in EViews.

See also [graph \(p. 224\)](#) for additional graph types.

sspace	Object Declaration
---------------	------------------------------------

Declare state space object.

Syntax

Command: **sspace** name

Follow the **sspace** keyword with a name to be given the **sspace** object. The **append** keyword may be used to add lines to an existing **sspace**.

Examples

```
sspace stsp1
```

declares a **sspace** object named STSP1.

```
sspace tvp
tvp.append cs = c(1) + sv1*inc
tvp.append @state sv1 = sv1(-1) + [var=c(2)]
tvp.ml
```

declares a **sspace** object named TVP, specifies a time varying coefficient model, and estimates the model by maximum likelihood.

Cross-references

The **sspace** object is documented in greater detail beginning on [page 40](#).

See [Chapter 22](#) of the *User’s Guide* for a discussion of state space models.

See also [m1](#) (p. 269) for estimation of state space models.

statby	Series View
---------------	-----------------------------

Basic statistics by classification.

The **statby** view displays descriptive statistics for the elements of a series classified into categories by one or more other series.

Syntax

Series View: `series_name.statby(options) classifier_name`

Follow the series name with a period, the **statby** keyword, and a name (or a list of names) for the series or group by which to classify. The options control which statistics to display and in what form. By default, **statby** displays the means, standard deviations, and counts for the series.

Options

Options to control statistics to be displayed

sum	Display sums.
med	Display medians.
max	Display maxima.
min	Display minima.
quant = <i>arg</i> (default = .5)	Display quantile with value given by the argument.
q = <i>arg</i> (default = “r”)	Compute quantiles using the definition: “b” (Blom), “r” (Rankit-Cleveland), “o” (simple fraction), “t” (Tukey), “v” (van der Waerden).
skew	Display skewness.
kurt	Display kurtosis.
na	Display counts of NAs.
nomean	Do not display means.
nostd	Do not display standard deviations.
nocount	Do not display counts.

Options to control layout

l	Display in list mode (for more than one classifier).
nor	Do not display row margin statistics.
noc	Do not display column margin statistics.
nom	Do not display table margin statistics (unconditional tables); for more than two classifier series.
nos	Do not display sub-margin totals in list mode; only used with “l” option and more than two classifier series.
sp	Display sparse labels; only with list mode option, “l”.

Options to control binning

dropna (default), keepna	[Drop/Keep] NA as a category.
v = <i>integer</i> (default = 100)	Bin categories if classification series take on more than the specified number of distinct values.
nov	Do not bin based on the number of values of the classification series.
a = <i>number</i> (default = 2)	Bin categories if average cell count is less than the specified number.
noa	Do not bin based on the average cell count.
b = <i>integer</i> (default = 5)	Set maximum number of binned categories.

Other options

p	Print the descriptive statistics table.
---	---

Examples

```
wage.statby(max,min) sex race
```

displays the mean, standard deviation, max, and min of the series WAGE by (possibly binned) values of SEX and RACE.

Cross-references

See “[Stats by Classification](#)” on page 154 and “[Descriptive Statistics](#)” on page 214 of the *User’s Guide*.

See also [hist](#) (p. 229).

stategraphs	SSpace View
--------------------	-----------------------------

Display graphs of a set of state series computed using the Kalman filter.

Syntax

SSpace View: `sspace_name.stategraph(options)`

Options

<code>t = arg</code>	Defines output type: “pred” (one-step ahead state predictions). “predse” (RMSE of the one-step ahead state predictions). “resid” (error in one-step ahead state predictions). “residse” (RMSE of the one-step ahead state prediction; same as predse). “filt” (filtered states). “filtse” (RMSE of the filtered states). “stdresid” (standardized one-step ahead prediction residual). “smooth” (smoothed states). “smoothse” (RMSE of the smoothed states). “disturb” (estimate of the disturbances). “disturbse” (RMSE of the estimate of the disturbances). “stddisturb” (standardized estimate of the disturbances).
----------------------	--

Other options

<code>p</code>	Print the view.
----------------	-----------------

Examples

```
ss1.stategraphs(t=filt)
```

displays a graph view containing the filtered state values.

Cross-references

See [Chapter 22](#) of the *User's Guide* for a discussion of state space models.

See also [signalgraphs \(p. 329\)](#), [makesignals \(p. 260\)](#) and [makestates \(p. 262\)](#).

statefinal	Sspace View
-------------------	-----------------------------

Display final state values.

Show the one-step ahead state predictions or the state prediction covariance matrix at the final values ($T + 1|T$) where T is the last observation in the estimation sample. By default, EViews shows the state predictions.

Syntax

Sspace View: `sspace_name.statefinal(options)`

Options

- | | |
|---|---|
| c | Display the state prediction covariance matrix. |
| p | Print the view. |

Examples

```
ss1.statefinal(c)
```

displays a view containing the final state covariances (the one-step ahead covariances for the first out-of-(estimation) sample period).

Cross-references

See [Chapter 22](#) of the *User's Guide* for a discussion of state space models.

See also [stateinit \(p. 343\)](#).

stateinit	Sspace View
------------------	-----------------------------

Display initial state values.

Show the state initial values or the state covariance initial values used to initialize the Kalman Filter ($1|0$). By default, EViews shows the state values.

Syntax

Sspace View: `sspace_name.stateinit(options)`

Options

c Display the covariance matrix.

p Print the view.

Examples

```
ss1.stateinit
```

displays a view containing the initial state values (the one-step ahead predictions for the first period).

Cross-references

See [Chapter 22](#) of the *User’s Guide* for a discussion of state space models.

See also [statefinal \(p. 343\)](#).

stats	Command Coef View Group View Matrix View Sym View Vector View
--------------	--

Descriptive statistics.

Computes and displays a table of means, medians, maximum and minimum values, standard deviations, and other descriptive statistics of one or more series or a group of series.

When used as a command, **stats** creates an untitled group containing all of the specified series and opens a statistics view of the group. By default, if more than one series is given, the statistics are calculated for the common sample.

When used for coef or matrix objects, **stats** computes the statistics for each column of data.

Syntax

Command: `stats(options) ser1 ser2 ser3 ...`

Object View: `group_name.stats(options)`

Options

i Individual sample for each series.

p Print the descriptive statistics table.

Examples

```
stats height weight age
```

opens an untitled group window displaying the histogram and descriptive statistics for the common sample of the three series.

```
group group1 wage hrs edu
group1.stats(i)
```

displays the descriptive statistics view of GROUP1 for the individual samples.

Cross-references

See “[Descriptive Statistics](#)” on page 152 and [page 214](#) of the *User’s Guide* for a discussion of the descriptive statistics views of series and groups.

See also [hist \(p. 229\)](#).

statusline	Command
-------------------	----------------

Send text to the status line.

Displays a message in the status line at the bottom of the EViews main window. The message may include text, control variables, and string variables.

Syntax

Command: **statusline** *message*

Examples

```
statusline Iteration Number: !t
```

Displays the message “Iteration Number: !t” in the status line replacing “!t” with the current value of the control variable in the program.

Cross-references

See [Chapter 6, “EViews Programming”, on page 85](#) for a discussion and examples of programs, control variables and string variables.

stom	Group Proc Series Proc
-------------	---------------------------------

Convert a series or group to a vector or matrix.

Fills a vector or matrix with the data from a series or group.

Syntax

Series Proc: `stom(series, vector[, sample])`
Group Proc: `stom(group, matrix[, sample])`

Include the series or group name in parentheses followed by a comma and the vector or matrix name. By default, the series values in the current workfile sample are used to fill the vector or matrix; you may optionally provide an alternative sample.

There are two important features of `stom` that you should keep in mind:

- If any of the series contain NAs, those observations will be dropped from the vector/matrix (for alternative behavior, see `stomna`, below).
- If the vector or matrix already exists in the workfile, EViews automatically resizes the vector/matrix to fit the series/group.

Examples

```
series lwage=log(wage)
stom(lwage,vec1)
```

converts the series LWAGE into a vector named VEC1 using the current workfile sample. Any NAs in LWAGE will be dropped from VEC1.

```
group rhs x1 x2 x3
sample s1 1951 1990
stom(rhs,x,s1)
```

converts a group of three series named X1, X2 and X3 to a matrix named X using sample S1. The matrix X will have 40 rows and 3 columns (provided there are no NAs).

Cross-references

See [Chapter 4, “Matrix Language”, on page 55](#) of the *Command and Programming Reference* for further discussion and examples of matrices.

See also [stomna](#) (p. 346) and [mtos](#) (p. 409).

stomna	Group Proc Series Proc
------------------------	--

Convert a series or group to a vector or matrix without dropping NAs.

Fills a vector or matrix with the data from a series or group without dropping observations with missing values.

Works in identical fashion to `stom`, above, but does not drop observations containing NAs.

Syntax

Series Proc: `stomna(series, vector[, sample])`
 Group Proc: `stomna(group, matrix[, sample])`

Include the series or group name in parentheses followed by a comma and the vector or matrix name. By default, the series values in the current workfile sample are used to fill the vector or matrix; you may optionally provide an alternative sample.

Examples

```
series lwage=log(wage)
stomna(lwage,vec1)
```

converts the series LWAGE into a vector named VEC1 using the current workfile sample. Any NAs in LWAGE will be placed in VEC1.

```
group rhs x1 x2 x3
sample s1 1951 1990
stom(rhs,x,s1)
```

converts a group of three series RHS to a matrix named X using sample S1. The matrix X will always have 40 rows and 3 columns.

Cross-references

See [Chapter 4, “Matrix Language”, on page 55](#) of the *Command and Programming Reference* for further discussion and examples of matrices.

See also [stom \(p. 345\)](#) and [mtos \(p. 409\)](#).

store	Command Pool Proc
--------------	--

Store objects in databases and databank files.

Stores one or more objects in the current workfile in EViews databases or individual databank files on disk. The objects are stored under the name that appears in the workfile.

When used as a pool proc, EViews will first expand the list of series using the pool operator, and then perform the fetch.

Syntax

Command: `store(options) object_name_list`
 Pool Proc: `pool_name.store(options) ser1? ser2?`

Follow the `store` command with a list of the object names (each separated by a space) that you wish to store. The default is to store the objects in the default database. (*This behavior is a change from EViews Version 2 and earlier where the default was to store objects in individual databank files*).

You can precede the object name with a database name and the double colon “::” to indicate a specific database. You can also specify the database name as an option in parentheses, in which case all objects without an explicit database name will be stored in the specified database.

When used as a command, you may use wild card characters “?” (to match any single character) or “*” (to match zero or more characters) in the object name list. All objects with names matching the pattern will be stored. You may not use “?” as a wildcard character if `store` is being used as a pool proc since this conflicts with the pool identifier.

You can optionally choose to store the listed objects in individual databank files. To store in files other than the default path, you should include a path designation before the object name.

Options

<code>d = db_name</code>	Store to the specified database.
<code>i</code>	Store to individual databank files.
<code>1</code>	Store series in single precision to save space.
<code>2</code>	Store series in double precision.
<code>o</code>	Overwrite object in database (default is to merge data, where possible).
<code>g = s</code>	For group objects, store group definition and series as separate objects.
<code>g = t</code>	For group objects, store group definition and series as one object.
<code>g = d</code>	For group objects, store only the series (as separate objects).
<code>g = l</code>	For group objects, store only the group definition.

If you do not specify the precision option (1 or 2), the global option setting will be used. See “[Data Registry / Database Default Storage Options](#)” on page 648 of the *User’s Guide*.

Examples

```
store m1 gdp unemp
```

stores the three objects M1, GDP, UNEMP in the default database.

```
store(d=us1) m1 gdp macro::unemp
```

stores M1 and GDP in the US1 database and UNEMP in the MACRO database.

```
store usdat::gdp macro::gdp
```

stores the same object GDP in two different databases USDAT and MACRO.

```
store(1) cons*
```

stores all objects with names starting with CONS in the default database. The 1 option uses single precision to save space.

```
store(i) m1 c:\data\unemp
```

stores M1 and UNEMP in individual databank files.

Cross-references

[“Basic Data Handling” on page 55](#) of the *User’s Guide* discusses exporting data in other file formats. See [Chapter 6](#) of the *User’s Guide* for a discussion of EViews databases and databank files. For additional discussion of wildcards, see [Appendix C, “Wildcards”, on page 657](#) of the *User’s Guide*.

See also [fetch \(p. 205\)](#) and [copy \(p. 168\)](#).

structure	SSpace View
------------------	-----------------------------

Display summary of sspace specification.

Show view which summarizes the system transition matrices or the covariance structure of the state space specification. EViews can display either the formulae or the values of the system transition matrices or covariance. By default, EViews will display the formulae for the system matrices.

Syntax

SSpace View: `sspace_name.structure(options) [argument]`

If you choose to display the values for a time-varying system using the “v” option, you should use the optional *[argument]* to specify a single date at which to evaluate the matrices. If none is provided, EViews will use the first date in the current sample.

Options

v	Display the values of the system transition or covariance matrices.
c	Display the system covariance matrix.
p	Print the view.

Examples

```
ss1.structure
```

displays a system transition matrices.

```
ss1.structure 1993:4
```

displays the transition matrices evaluated at 1993:4.

Cross-references

See [Chapter 22](#) of the *User's Guide* for a discussion of state space models.

sur	System Method
------------	-------------------------------

Estimate a system using seemingly unrelated regression (SUR).

Note that the EViews procedure is more general than textbook versions of SUR since the system of equations may contain cross-equation restrictions on parameters.

Syntax

System Method: system_name.sur(*options*)

Options

i	Iterate on the weighting matrix and coefficient vector simultaneously.
s	Iterate on the weighting matrix and coefficient vector sequentially.
o (default)	Iterate only on the coefficient vector with one step of the weighting matrix.
c	One step iteration on the coefficient vector after one step of the weighting matrix.
m = <i>integer</i>	Maximum number of iterations.

<code>c = number</code>	Set convergence criterion.
<code>l = number</code>	Set maximum number of iterations on the first-stage iteration to get one-step weighting matrix.
<code>showopts / -showopts</code>	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
<code>deriv = keyword</code>	Set derivative methods. The argument <i>keyword</i> should be a one- or two-letter string. The first letter should either be “f” or “a” corresponding to fast or accurate numeric derivatives (if used). The second letter should be either “n” (always use numeric) or “a” (use analytic if possible). If omitted, EViews will use the global defaults.
<code>p</code>	Print estimation results.

Examples

```
sys1.sur(i)
```

estimates SYS1 by SUR, iterating simultaneously on the weighting matrix and coefficient vector.

```
nlsys.sur(d,m=500)
```

estimates NLSYS by SUR with up to 500 iterations. The “d” option displays the starting values.

Cross-references

See [Chapter 19](#) of the *User’s Guide* for a discussion of system estimation.

svar	Var Proc
-------------	--------------------------

Estimate factorization matrix for structural innovations.

Syntax

Var Proc: `var_name.svar(options)`

You must specify the identifying restrictions either in text form by the append proc or by a pattern matrix option. See [“Specifying the Identifying Restrictions” on page 531](#) of the *User’s Guide* for details on specifying restrictions.

Options

You must specify one of the following restriction type:

<code>rtype = text</code>	Text form restrictions. The restrictions must be specified by the <code>append</code> command to use this option.
<code>rtype = patsr</code>	Short-run pattern restrictions. You must provide the names of the patterned matrices by the “ <code>namea =</code> ” and “ <code>nameb =</code> ” options as described below.
<code>rtype = patlr</code>	Long-run pattern restrictions. You must provide the name of the patterned matrix by the “ <code>namelr =</code> ” option as described below.

Other Options:

<code>namea = arg</code> , <code>nameb = arg</code>	Names of the pattern matrices for A and B matrices. Must be used with “ <code>rtype = patsr</code> ”.
<code>namelr = arg</code>	Name of the pattern matrix for long-run impulse responses. Must be used with “ <code>rtype = patlr</code> ”.
<code>fsign</code>	Do not apply the sign normalization rule. Default is to apply the sign normalization rule whenever applicable. See “ Sign Indeterminacy ” on page 536 of the <i>User’s Guide</i> for a discussion of the sign normalization rule.
<code>f0 = arg</code> (default “ <code>f0 = 0.1</code> ”)	Starting values for the free parameters: “ <code>f0 = scalar</code> ” (specify fixed value for starting values), “ <code>f0 = s</code> ” (user specified starting values are taken from the C coefficient vector), “ <code>f0 = u</code> ” (draw starting values for free parameters from a uniform distribution on [0,1]), “ <code>f0 = n</code> ” (draw starting values for free parameters from standard normal).
<code>maxiter = integer</code>	Maximum number of iterations. Default is taken from global option setting.
<code>conv = number</code>	Convergence criterion. Default is taken from global option setting.
<code>trace = integer</code>	Trace iterations process every <code>integer</code> iterations (displays an untitled text object containing summary information).
<code>nostop</code>	Suppress “Near Singular Matrix” error message even if Hessian is singular at final parameter estimates.

Examples

```
var var1.ls 1 4 m1 gdp cpi
matrix(3,3) pata
'fill matrix in row major order
pata.fill(by=r) 1,0,0, na,1,0, na,na,1
matrix(3,3) patb
pata.fill(by=r) na,0,0, 0,na,0, 0,0,na
var1.svar(rtype=patsr, namea=pata, nameb=patb)
```

The first line declares and estimates a VAR with three variables. Then we create the short-run pattern matrices and estimate the factorization matrix.

```
var var1.ls 1 8 dy u @
var1.append(svar) @lr1(@u1)=0
freeze(out1) var1.svar(rtype=text)
```

The first line declares and estimates a VAR with two variables without a constant. The next two lines specify a long-run restriction in text form and stores the estimation output in a table object named OUT1.

Cross-references

See “[Structural \(Identified\) VARs](#)” on page 531 of the *User’s Guide* for a discussion of structural VARs.

sym	Object Declaration
-----	------------------------------------

Declare a symmetric matrix object.

The `sym` command declares and optionally initializes a matrix object.

Syntax

Command:	<code>sym(<i>n</i>) sym_name</code>
Command:	<code>sym(<i>n</i>) sym_name = <i>assignment</i></code>

`sym` takes an optional argument *n* specifying the row and column dimension of the matrix and is followed by the name you wish to give the matrix.

You may also include an assignment in the `sym` command. The symmetric matrix will be resized, if necessary. Once declared, symmetric matrices may be resized by repeating the `sym` command for a given matrix name.

You may use `sym` instead of `matrix` for working with symmetric matrices.

Examples

```
sym mom
```

declares a symmetric matrix named MOM with one zero element.

```
sym y=@inner(x)
```

declares a symmetric matrix Y and assigns to it the inner product of the matrix X.

Cross-references

See “[Matrix Language](#)” on page 55 of the *Command and Programming Reference* for a discussion of matrix objects in EViews.

See also [matrix \(p. 265\)](#).

system	Object Declaration
---------------	------------------------------------

Declare system of equations.

The **system** command declares an object as a system for estimation by system methods such as three-stage least squares.

Syntax

Command: **system** system_name

Follow the **system** keyword by a name for the system. If you do not provide a name, EViews will open an untitled system object (if in interactive mode).

Examples

```
system mysys
```

creates a system named MYSYS.

Cross-references

[Chapter 19](#) of the *User’s Guide* provides a full discussion of system objects.

See [ls \(p. 245\)](#), [wls \(p. 380\)](#), [tsls \(p. 368\)](#), [wtsls \(p. 385\)](#), [3sls \(p. 136\)](#), and [gmm \(p. 221\)](#) for various system estimation methods.

template	Graph Proc
-----------------	----------------------------

Applies templates to graph objects.

If you apply template to a multiple graph object, the template options will be applied to each graph in the multiple graph. If the template graph is a multiple graph, the options of the first graph will be used.

Syntax

Graph Proc: `graph_name.template(options) template_graph_name`

Follow the name of the graph you want to apply the template options with a period, the keyword `template`, and the name of the graph object to use as a template.

Options

- | | |
|---|--|
| t | Copy any text labels and shading in the template graph in addition to its option settings. |
|---|--|

Examples

```
gra_cs.template gra_gdp
```

applies the option settings in the graph object GRA_GDP to the graph GRA_CS. Text and shadings in GRA_GDP will not be applied to GRA_CS.

```
g1.template(t) mygraph1
```

applies the option settings of MYGRAPH1 and all text and shadings in the graph to the graph G1.

Cross-references

See “[Graph Templates](#)” on page 249 of the *User’s Guide* for additional discussion.

testadd	Command Equation View
----------------	--

Test whether to add regressors.

Tests the hypothesis that the listed variables were incorrectly omitted from the (default) equation. The test displays the Wald and LR test statistics and the test regression.

Syntax

Command: **testadd** ser1 ser2 ser3

Equation View: eq_name.**testadd** ser1 ser2 ser3

List the names of the series or groups of series to test for omission after the `add` keyword. The command form applies the test to the default equation.

Options

p	Print output from the test.
---	-----------------------------

Examples

```
ls sales c adver lsales ar(1)  
testadd gdp gdp(-1)
```

tests whether GDP and GDP(-1) belong in the specification for SALES. The commands

```
equation oldeq.ls sales c adver lsales ar(1)  
oldeq.testadd gdp gdp(-1)
```

perform the same test using a named equation object.

Cross-references

See “[Coefficient Tests](#)” on page 368 of the *User’s Guide* for further discussion.

See also [testdrop \(p. 358\)](#) and [wald \(p. 378\)](#).

testbtw	Group View
----------------	----------------------------

Test equality of the mean, median or variance between (among) series in a group.

Syntax

Group View: group_name.**testbtw**(options)

Specify the type of test as an option.

Options

mean (default)	Test equality of mean.
----------------	------------------------

med	Test equality of median.
-----	--------------------------

var	Test equality of variance.
-----	----------------------------

c	Use common sample.
i (default)	Use individual sample.
p	Print the test results.

Examples

```
group g1 wage_m wage_f
g1.testbtw
g1.testbtw(var,c)
```

tests the equality of means between the two series WAGE_M and WAGE_F.

Cross-references

See “[Tests of Equality](#)” on page 214 of the *User’s Guide* for further discussion of these tests.

See also [testby](#) (p. 357), [teststat](#) (p. 362).

testby	Series View
---------------	-----------------------------

Test equality of the mean, median, or variance of a series across categories classified by a list of series or a group.

Syntax

Series View: `series_name.testby(options) ser1 ser2 ser3 ...`

Follow the `testby` keyword by a list of the names of the series to use as classifiers. Specify the type of test as an option.

Options

mean (default)	Test equality of mean.
med	Test equality of median.
var	Test equality of variance.
dropna (default), keepna	[Drop /Keep] NAs as a category.
v = <i>integer</i> (default = 100)	Bin categories if classification series take more than the specified number of distinct values.
nov	Do not bin based on the number of values of the classification series.

a = <i>number</i> (default = 2)	Bin categories if average cell count is less than the specified number.
noa	Do not bin on the basis of average cell count.
b = <i>integer</i> (default = 5)	Set maximum number of binned categories.
p	Print the test results.

Examples

```
wage.testby(med) race
```

Tests equality of medians of WAGE across groups classified by RACE.

Cross-references

See “[Equality Tests by Classification](#)” on page 159 of the *User’s Guide* for a discussion of equality tests.

See also [testbtw](#) (p. 356), [teststat](#) (p. 362).

testdrop	Command Equation View
-----------------	--

Test whether to drop regressors from a regression.

Tests the hypothesis that the listed variables were incorrectly included in the (default) equation. The test displays the *F* and LR test statistics and the test regression.

Syntax

Command: **testdrop** ser1 ser2 ser3

Equation View: eq_name.**testdrop** ser1 ser2 ser3

List the names of the series or groups of series to test for omission after the `add` keyword. The command form applies the test to the default equation.

Options

p	Print output from the test.
---	-----------------------------

Examples

```
ls sales c adver lsales ar(1)  
testdrop adver
```

tests whether ADVER should be excluded from the specification for SALES. The commands

```
equation oldeq.ls sales c adver lsales ar(1)
oldeq.testdrop adver
```

perform the same test using a named equation object.

Cross-references

See “[Coefficient Tests](#)” on page 368 of the *User’s Guide* for further discussion of testing coefficients.

See also [testadd](#) (p. 355) and [wald](#) (p. 378).

testexog	Var View
-----------------	--------------------------

Exogeneity (Granger causality) tests.

Syntax

Var View: `var_name.testexog(options)`

Options

<code>name = arg</code>	Save the Wald test statistics in named matrix object. See below for a description of the statistics stored in the matrix.
-------------------------	---

<code>p</code>	Print output from the test.
----------------	-----------------------------

The `name=` option stores a $(k + 1) \times k$ matrix, where k is the number of endogenous variables in the VAR. For the first k rows, the i -th row, j -th column entry is the Wald statistic for the joint significance of lags of the i -th endogenous variable in the j -th equation. (Note that the entries in the main diagonal are not reported in the table view). The degrees of freedom of the Wald statistics for the first k rows is the number of lags you included in the VAR.

The j -th column of the last row contains the Wald statistic for the joint significance of all lagged endogenous variables (excluding lags of the dependent variable) in the j -th equation. The degrees of freedom of the Wald statistics in the last row is $(k - 1)$ times the number of lags you included in the VAR.

Examples

```
var var1.ls 1 6 lgdp lm1 lcpi
freeze(tab1) var1.testexog(name=exog)
```

The first line declares and estimates a VAR. The second line stores the exclusion test results in a named table TAB1 and stores the Wald statistics in a matrix named EXOG.

Cross-references

See “[Diagnostic Views](#)” on page 522 of the *User’s Guide* for a discussion other VAR diagnostics.

See also [testlags \(p. 361\)](#).

testfit	Equation View
----------------	-------------------------------

Carry out the Hosmer-Lemeshow and/or Andrews goodness-of-fit tests for binary models.

Syntax

Equation View: binary_equation.**testfit**(*options*)

Options

h	Group by the predicted values of the estimated equation.
s = <i>series name</i>	Group by the specified series.
integer (default = 10)	Specify the number of quantile groups in which to classify observations.
u	Unbalanced grouping. Default is to randomize ties to balance the number of observations in each group.
v	Group according to the values of the reference series.
l = <i>integer</i> (default = 100)	Limit the number of values to use for grouping. Should be used with the “v” option.
p	Print the result of the test.

Examples

```
equation eq1.binary work c age edu
eq1.testfit(h,5,u)
```

estimates a probit specification, and tests goodness-of-fit by comparing five unbalanced groups of actual data to those estimated by the model.

Cross-references

See “[Goodness-of-Fit Tests](#)” on page 431 of the *User’s Guide* for a discussion of the Andrews and Hosmer-Lemeshow tests.

testlags	Var View
-----------------	--------------------------

Lag exclusion (Wald) tests.

Syntax

Var View: `var_name.testlags(options)`

Options

`name = arg` Save the Wald test statistics in named matrix object. See below for a description of the statistics contained in the stored matrix.

`p` Print the result of the test.

The “`name =`” option stores an $m \times (k + 1)$ matrix, where m is the number of lagged terms and k is the number of endogenous variables in the VAR. For the first k columns, the i -th row, j -th column entry is the Wald statistic for the joint significance of all i -th lagged endogenous variables in the j -th equation. These Wald statistics have a χ^2 distribution with k degrees of freedom under the exclusion null.

The i -th row of the last column contains the system Wald statistic for testing the joint significance of all i -th lagged endogenous variables in the VAR system. The system Wald statistics has a chi-square distribution with k^2 degrees of freedom under the exclusion null.

Examples

```
var var1.ls 1 6 lgdp lm1 lcpi
freeze(tab1) var1.testlags(name=lags)
```

The first line declares and estimates a VAR. The second line stores the lag exclusion test results in a table named TAB1 and stores the Wald statistics in a matrix named LAGS.

Cross-references

See “[Diagnostic Views](#)” on page 522 of the *User’s Guide* for a discussion other VAR diagnostics.

See also [laglen](#) (p. 239) and [texexog](#) (p. 359).

teststat[Series View](#)

Test simple hypotheses of whether the mean, median, or variance of a series is equal to a specified value.

Syntax

Series View: `series_name.teststat(options)`

Specify the type of test and the value under the null hypothesis as an option.

Options

`mean = number` Test the null hypothesis that the mean equals the specified number.

`med = number` Test the null hypothesis that the median equals the specified number.

`var = number` Test the null hypothesis that the variance equals the specified number. The number must be positive.

`std = number` Test equality of mean conditional on the specified standard deviation. The standard deviation must be positive.

`p` Print the test results.

Examples

```
smp1 if race=1  
lwage.teststat(var=4)
```

tests the null hypothesis that the variance of LWAGE is equal to 4 for the subsample with RACE = 1.

Cross-references

See “[Tests for Descriptive Stats](#)” on page 156 of the *User’s Guide* for a discussion of simple hypothesis tests.

See also [testbtw](#) (p. 356), [testby](#) (p. 357).

text	Object Declaration Model View
-------------	--

Declare a text object when used as a command, or display text representation of the model specification.

Syntax

Command: `text text_name`

Model View: `model_name.text(options)`

Follow the `text` keyword with a name for the text object. When used as a model view, `text` is equivalent to [spec \(p. 337\)](#).

Options

<code>p</code>	Print the model text specification.
----------------	-------------------------------------

Examples

```
text notes1
```

declares a text object named NOTES1.

Cross-references

See [Chapter 23](#) of the *User's Guide* for further details on models. See [Chapter 10](#) of the *User's Guide* for a discussion of text objects in EViews.

See also [spec \(p. 337\)](#).

tic	Command
------------	-------------------------

Reset the timer.

Syntax

Command: `tic`

Examples

The sequence of commands

```
tic
[some commands]
toc
```

resets the timer, executes commands, and then displays the elapsed time in the statusline.
Alternatively,

```
tic  
[some commands]  
!elapsed = @toc
```

resets the time, executes commands, and saves the elapsed time in the control variable
!ELAPSED.

Cross-references

See also [toc \(p. 364\)](#) and [@toc \(p. 432\)](#).

toc	Command
------------	---------

Display elapsed time (since timer reset) in seconds.

Syntax

Command: **toc**

Examples

The sequence of commands

```
tic  
[some commands]  
toc
```

resets the timer, executes commands, and then displays the elapsed time in the statusline,
while the set of commands

```
tic  
[some commands]  
!elapsed = @toc
```

resets the time, executes commands, and saves the elapsed time in the control variable
!ELAPSED.

Cross-references

See also [tic \(p. 363\)](#) and [@toc \(p. 432\)](#).

trace	Model View
-------	----------------------------

Display trace view of a model showing iteration history for selected solved variables.

Syntax

Model View: `model_name.trace(options)`

Options

p	Print the block structure view.
---	---------------------------------

Cross-references

See “[Diagnostics](#)” on page 637 of the *User’s Guide* for further details on tracing model solutions.

See also [msg](#) (p. 270), [solve](#) (p. 334) and [solveopt](#) (p. 335).

tramoseats	Series Proc
------------	-----------------------------

Run the external seasonal adjustment program Tramo/Seats using the data in the series.

`tramoseats` is available for annual, semi-annual, quarterly, and monthly series. The procedure requires at least n observations and can adjust up to 600 observations where

$$\underline{n} = \begin{cases} 36 & \text{for monthly data} \\ \max\{12, 4s\} & \text{for other seasonal data} \end{cases} \quad (8.2)$$

Syntax

Series Proc: `series_name.tramoseats(options) base_name`

Enter the name of the original series followed by a dot, the keyword `tramoseats`, and optionally provide a base name (no more than 20 characters long) to name the saved series. The default base name is the original series name. The saved series will have post-fixes appended to the base name.

Options

runtype = ts (default)	Run Tramo followed by Seats. The “opt = ” options are passed to Tramo, and Seats is run with the input file returned from Tramo.
runtype = t	Run only Tramo.
runtype = s	Run only Seats.
save = <i>arg</i>	<p>Specify series to save in workfile. You must use one or more from the following key word list:</p> <p>“hat” for forecasts of original series</p> <p>“lin” for linearized series from Tramo</p> <p>“pol” for interpolated series from Tramo</p> <p>“sa” for seasonally adjusted series from Seats</p> <p>“trd” for final trend component from Seats</p> <p>“ir” or final irregular component from Seats</p> <p>“sf” for final seasonal factor from Seats</p> <p>“cyc” for final cyclical component from Seats</p> <p>To save more than one series, separate the list of key words with a space. <i>Do not use commas</i> within the list. The special key word “save = *” will save all series in the key word list. The five key words “sa”, “trd”, “ir”, “sf”, “cyc” will be ignored if “runtype = t”.</p>
opt = <i>arg</i>	A space delimited list of input namelist. <i>Do not use commas within the list</i> . The syntax for the input namelist is explained in the pdf documentation file. See also “Notes” below.
reg = <i>arg</i>	A space delimited list for one line of reg namelist. <i>Do not use commas within the list</i> . This option must be used in pairs, either with another “reg = ” option or “regname = ” option. The reg namelist is available only for Tramo and its syntax is explained in the pdf documentation file. See also “Notes” below.
regname = <i>arg</i>	Name of a series or group in the current workfile that contains the exogenous regressors specified in the previous “reg = ” option. See “Notes” below.
p	Print the results of the Tramo/Seats procedure.

Notes

The command line interface to Tramo/Seats does very little error checking of the command syntax. EViews simply passes on the options you provide “as is” to Tramo/Seats. If the syntax contains an error, you will most likely see the EViews error message “output file not found”. If you see this error message, check the input files produced by EViews for possible syntax errors as described in “[Trouble Shooting](#) on page 189 of the *User’s Guide*.

Additionally, here are some of the more commonly encountered syntax errors.

- To replicate the dialog options from the command line, use the following input options in the “opt = ” list. See the pdf documentation file for a description of each option.
 1. data frequency: “mq = ”.
 2. forecast horizon: “npred = ” for Tramo and “fh = ” for Seats .
 3. transformation: “lam = ”.
 4. ARIMA order search: “inic = ” and “idif = ”.
 5. Easter adjustment: “ieast = ”.
 6. trading day adjustment: “itrad = ”.
 7. outlier detection: “iatip = ” and “aio = ”.
- The command option input string list must be space delimited. *Do not use commas*. Options containing an equals sign should not contain any spaces around the equals; the space will be interpreted as a delimiter by Tramo/Seats.
- If you set “rtype = ts”, you are responsible for supplying either “seats = 1” or “seats = 2” in the “opt = ” option list. EViews will issue the error message “Seats.itr not found” if the option is omitted. Note that the dialog option **Run Seats after Tramo** sets “seats = 2”.
- Each “reg = ” or “regname = ” option is passed to the input file as a separate line in the order that they appear in the option argument list. Note that these options must come in pairs. A “reg = ” option must be followed by another “reg = ” option that specifies the outlier or intervention series or by a “regname = ” option that provides the name for an exogenous series or group in the current workfile. See the sample programs in the EXAMPLE FILES directory.
- If you specify exogenous regressors with the “reg = ” option, you must set the appropriate “ireg = ” option (for the total number of exogenous series) in the “opt = ” list.
- To use the “regname = ” option, the preceding “reg = ” list must contain the “user = -1” option and the appropriate “ilong = ” option. *Do not use “user = 1”* since EViews will always write data in a separate external file. The “ilong = ” option must be at

least the number of observations in the current workfile sample *plus* the number of forecasts. The exogenous series should not contain any missing values in this range. Note that Tramo may increase the forecast horizon, in which case the exogenous series is extended by appending zeros at the end.

Examples

```
freeze(tab1) show x.tramoseats(runtype=t, opt="lam=-1 iatip=1  
aio=2 va=3.3 noadmiss=1 seats=2", save=*) x
```

replicates the example file EXAMPLE.1 in Tramo. The output file from Tramo is stored in a text object named tab1. This command returns three series named X_HAT, X_LIN, X_POL.

```
show x.TramoSeats(runtype=t, opt="NPRED=36 LAM=1 IREG=3  
INTERP=2 IMEAN=0 P=1 Q=0 D=0", reg="ISEQ=1 DELTA=1.0",  
reg="61 1", reg="ISEQ=8 DELTAS=1.0", reg="138 5 150 5 162 5  
174 5 186 5 198 5 210 5 222 5", reg="ISEQ=8 DELTAS=1.0",  
reg="143 7 155 7 167 7 179 7 191 7 203 7 215 7 227 7") x
```

replicates the example file EXAMPLE.2 in Tramo. This command produces an input file that looks as follows:

```
$INPUT NPRED=36 LAM=1 IREG=3 INTERP=2 IMEAN=0 P=1 Q=0 D=0, $  
$REG ISEQ=1 DELTA=1.0$  
61 1  
$REG ISEQ=8 DELTAS=1.0$  
138 5 150 5 162 5 174 5 186 5 198 5 210 5 222 5  
$REG ISEQ=8 DELTAS=1.0$  
143 7 155 7 167 7 179 7 191 7 203 7 215 7 227 7
```

Further examples that replicate many of the example files provided by Tramo/Seats can be found in the EXAMPLE FILES directory. You will also find example files that compare seasonal adjustments from Census X12 and Tramo/Seats.

Cross-references

See also the Tramo/Seats documentation that accompanied your EViews distribution.

See also [seas](#) (p. 315) and [x12](#) (p. 388).

tsls	Command Equation Method System Method
-------------	--

Two-stage least squares.

Carries out estimation for equations or systems using two-stage least squares.

Syntax

Command: `tsls(options) y x1 x2 @ z1 z2 z3`

`tsls(options) formula @ z1 z2 z3`

Equation Method: `eq_name.tsls(options) y x1 x2 @ z1 z2 z3`

`eq_name.tsls(options) formula @ z1 z2 z3`

System Method: `system_name.tsls(options)`

To use `tsls` as a command or equation method, list the dependent variable first, followed by the regressors, then any AR or MA error specifications, then an “@”-sign, and finally, a list of exogenous instruments. There must be at least as many instrumental variables as there are independent variables. All exogenous variables included in the regressor list should also be included in the instrument list. A constant is included in the list of instrumental variables even if not explicitly specified. You can estimate nonlinear equations or equations specified with formulas; always list the instrumental variables after an “@”-sign.

Options

General options

<code>m = integer</code>	Set maximum number of iterations.
<code>c = number</code>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
<code>deriv = keyword</code>	Set derivative methods. The argument <i>keyword</i> should be a one- or two-letter string. The first letter should either be “f” or “a” corresponding to fast or accurate numeric derivatives (if used). The second letter should be either “n” (always use numeric) or “a” (use analytic if possible). If omitted, EViews will use the global defaults.
<code>showopts / -showopts</code>	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
<code>p</code>	Print estimation results.

Additional options for equations

w = <i>series_name</i>	Weighted TSLS. Each observation will be weighted by multiplying by the specified series.
h	White's heteroskedasticity consistent standard errors.
n	Newey-West heteroskedasticity and autocorrelation consistent (HAC) standard errors.
s	Use the current coefficient values in C as starting values for equations with AR or MA terms (see PARAM).
s = <i>number</i>	Specify a number between zero and one to determine starting values for equations with AR or MA terms as a fraction of preliminary LS or TSLS estimates made without including AR or MA terms (out of range values are set to "s = 1").
z	Turn off backcasting in ARMA models.

Additional options for systems

i	Iterate on the weighting matrix and coefficient vector simultaneously.
s	Iterate on the weighting matrix and coefficient vector sequentially.
o (default)	Iterate only on the coefficient vector with one step of the weighting matrix.
c	One step iteration of the coefficient vector after one step of the weighting matrix.
l = <i>number</i>	Set maximum number of iterations on the first-stage iteration to get one-step weighting matrix.

Examples

```
eq1.tsls y_d c cpi inc ar(1) @ lw(-1 to -3)
```

estimates EQ1 using TSLS regression of Y_D on a constant, CPI, INC with AR(1) using a constant, LW(-1), LW(-2), LW(-3) as instruments.

```
param c(1) .1 c(2) .1  
eq1.tsls(s,m=500) y_d=c(1)+inc^c(2) @ cpi
```

estimates a nonlinear TSLS model using a constant and CPI as instruments. The first line sets the starting values for the nonlinear iteration algorithm.

`sys1.tsls`

estimates the system object using TSLS.

Cross-references

See “[Two-stage Least Squares](#)” on page 283 and [page 497](#) of the *User’s Guide* for details on two-stage least squares estimation in single equations and systems.

See also [ls \(p. 245\)](#). For estimation of weighted TSLS in systems, see [wtsls \(p. 385\)](#).

unlink	Model Proc
---------------	----------------------------

Break model links.

Breaks equation links in the model. Follow the name of the model object by a period, the keyword `unlink`, and a specification for the variables to unlink.

Syntax

Model Proc: `object.unlink spec`

where *spec* is

`@all` Unlinks all equations in the model.

`list of
endogenous vars` Unlink equations for the listed endogenous variables.

Note: If a link is to another Model or a System object, then more than one endogenous variable may be associated with the link. If the *spec* contains any of the endogenous variables in a linked Model or System, EViews will break the link for all of the variables found in the link.

Examples

```
mod1.unlink @all
mod2.unlink z1 z2
```

unlinks all of equations in MOD1, and all of the variables associated with the links for Z1 and Z2 in MOD2.

Cross-references

See [Chapter 23](#) of the *User’s Guide* for a discussion of specifying and solving models in EViews. See also [append \(p. 143\)](#), [merge \(p. 267\)](#) and [solve \(p. 334\)](#).

update[Model Proc](#)

Update model specification.

Recompiles the model and updates all links.

Syntax

Model Proc: **model.update**

Follow the name of the model object by a period and the keyword **update**. See [Chapter 23](#) of the *User's Guide* for a discussion of specifying and solving models in EViews.

Examples

```
mod1.update
```

recompiles and updates all of the links in MOD1.

See also [append \(p. 143\)](#), [merge \(p. 267\)](#) and [solve \(p. 334\)](#).

updatecoefs[Equation Proc](#) | [Logl Proc](#) | [Pool Proc](#) | [Sspace Proc](#) | [System Proc](#)

Update coefficients.

Copies coefficients from the estimation object into the appropriate coefficient vector or vectors.

Syntax

Object Proc: **object.updatecoef**

Follow the name of the estimation object by a period and the keyword **updatecoef**.

Examples

```
equation eq1.ls y c x1 x2 x3  
equation eq2.ls z c z1 z2 z3  
eq1.updatecoef
```

places the coefficients from EQ1 in the default coefficient vector C.

```
coef(3) a  
equation eq3.ls y=a(1)+z1^c(1)+log(z2+a(2))+exp(c(4)+z3/a(3))  
equation eq2.ls z c z1 z2 z3  
eq3.updatecoef
```

updates the coefficient vector A and the default vector C so that both contain the coefficients from EQ3.

Cross-references

See also [coef](#) (p. 164).

uroot	Command Series View
--------------	--

Unit root tests.

Carries out the Augmented Dickey-Fuller (ADF), GLS detrended Dickey-Fuller (DFGLS), Phillips-Perron (PP), Kwiatkowski, *et. al.* (KPSS), Elliot, Rothenberg, and Stock (ERS) Point Optimal, or Ng and Perron (NP) tests for a unit root in the series (or its first or second difference).

Syntax

Command: `uroot(options) series_name`

Series View: `series_name.uroot(options)`

You should enter the keyword `uroot` followed by the series name, or the series name followed by a period and the keyword `uroot`.

Options

Specify the test type using one of the following keywords:

adf (default)	Augmented Dickey-Fuller.
dfglss	GLS detrended Dickey-Fuller (Elliot, Rothenberg, and Stock).
pp	Phillips-Perron.
kpss	Kwiatkowski, Phillips, Schmidt, and Shin.
ers	Elliot, Rothenberg, and Stock (Point Optimal).
np	Ng and Perron.

Specify the exogenous variables in the test equation from the following:

const (default)	Include a constant in the test equation.
trend	Include a constant and a linear time trend in the test equation.
none	Do not include a constant or time trend (only available for the ADF and PP tests).

For backward compatibility, the shortened forms “c”, “t”, and “n” are presently supported. However for future compatibility we recommend that you use the longer forms.

Other Options:

dif = <i>integer</i> (default = 0)	Order of differencing of the series prior to running the test. Valid values are {0, 1, 2}.
hac = <i>arg</i>	Method of estimating the frequency zero spectrum: “bt” (Bartlett kernel), “pr” (Parzen kernel), “qs” (Quadratic Spectral kernel), “ar” (AR spectral), “ardt” (AR spectral - OLS detrended data), “argls” (AR spectral - GLS detrended data). <i>The default settings are test specific:</i> “bt” for PP and KPSS, “ar” for ERS, “argls” for NP. Applicable to PP, KPSS, ERS and NP tests.
band = <i>arg</i> , b = <i>arg</i> (default = “nw”)	Method of selecting the bandwidth: “nw” (Newey-West automatic variable bandwidth selection), “a” (Andrews automatic selection), “number” (user specified bandwidth). Applicable to PP, KPSS, ERS, ERS and NP tests when using kernel sums-of-covariances estimators (where “hac = ” is one of {bt, pz, qs}).
lag = <i>arg</i> (default = “a”)	Method of selecting lag length (number of first difference terms) to be included in the regression: “a” (automatic information criterion based selection), “integer” (user specified lag length) Applicable to ADF and DFGLS tests, and for the other tests when using AR spectral density estimators (where “hac = ” is one of {ar, ardt, argls}).

<code>info = arg</code> (default = "maic")	Information criterion to use when computing automatic lag length selection: "aic" (Akaike), "sic" (Schwarz), "hqc" (Hannan-Quinn), "msaic" (Modified Akaike), "msic" (Modified Schwarz), "mhqc" (Modified Hannan-Quinn). Applicable to ADF and DFGLS tests, and for other tests when using AR spectral density estimators (where "hac = " is one of {ar, ardt, argls}).
<code>maxlag = integer</code>	Maximum lag length to consider when performing automatic lag length selection (default = $\text{int}(12(T/100)^{1/4})$). Applicable to ADF and DFGLS tests, and for other tests when using AR spectral density estimators (where "hac = " is one of {ar, ardt, argls}).
<code>p</code>	Print output from the test.

Examples

The command

```
gnp.uroot(adf,const,lag=3,save=mout)
```

performs an ADF test on the series GDP with the test equation including a constant term and three lagged first-difference terms. Intermediate results are stored in the matrix MOUT.

```
ip.uroot(dfgrls,trend,info=sic)
```

runs the DFGLS unit root test on the series IP with a constant and a trend. The number of lagged difference terms is selected automatically using the Schwarz criterion.

```
unemp.uroot(kpss,const,hac=pr,b=2.3)
```

runs the KPSS test on the series UNEMP. The null hypothesis is that the series is stationary around a constant mean. The frequency zero spectrum is estimated using kernel methods (with a Parzen kernel), and a bandwidth of 2.3.

```
sp500.uroot(np,hac=ardt,info=maic)
```

runs the NP test on the series SP500. The frequency zero spectrum is estimated using the OLS AR spectral estimator with the lag length automatically selected using the modified AIC.

Cross-references

See “[Unit Root Tests](#)” on page 329 of the *User’s Guide* for further discussion.

var	Object Declaration
-----	--------------------

Declare a var (Vector Autoregression) object.

Syntax

Command: `var var_name`

Command: `var var_name.ls lag_pairs y1 y2 @ x1 x2`

Command: `var var_name.ec(options) specification`

Declare the var as a name, or a name followed by an estimation method (either `ls` or `ec`).

The `ls` method estimates an unrestricted VAR using equation-by-equation OLS. You must specify the order of the VAR (using one or more pairs of lag intervals), and then provide a list of endogenous variables. You may include exogenous variables such as trends and seasonal dummies in the VAR by including an “@-sign” followed by a list of series. A constant is always included as an exogenous variable.

See [ec \(p. 196\)](#) for the error correction specification of a VAR.

Options

p	Print the estimation result if the estimation procedure is specified.
---	---

Examples

```
var mvar.ls 1 4 8 8 m1 gdp tb3 @ @trend(70.4)
```

declares and estimates an unrestricted VAR named MVAR with three endogenous variables (M1, GDP, TB3), five lagged terms (lags 1 through 4 and 8), a constant, and a linear trend.

```
var jvar.ec(c,2) 1 4 m1 gdp tb3
```

declares and estimates an error correction model named JVAR with three endogenous variables (M1, GDP, TB3), four lagged terms (lags 1 through 4) including two cointegrating relations. The “c” option assumes a linear trend in data but only a constant in the cointegrating relations.

Cross-references

See [Chapter 20](#) of the *User’s Guide* for a discussion of vector autoregressions.

See [ec \(p. 196\)](#) for error correction models in VARs.

vars	Model View
-------------	----------------------------

View of model organized by variable.

Display the model in variable form with identification of endogenous, exogenous, and identity variables, as well as dependency tracking.

Syntax

Model View: `model_name.vars`

Cross-references

See “[Variable View](#)” on page 622 of the *User’s Guide* for details. See [Chapter 23](#) of the *User’s Guide* for a general discussion of models.

See also [block](#) (p. 154), [text](#) (p. 363), and [eqs](#) (p. 201) for alternative representations of the model.

vector	Object Declaration
---------------	------------------------------------

Declare a vector object.

The `vector` command declares and optionally initializes a (column) vector object.

Syntax

Command: `vector(size) name`

Command: `vector(size) name = assignment`

The `vector` keyword should be followed by the name you wish to give the vector. `vector` also takes an optional argument specifying the size of the vector. Once declared, vectors may be resized by repeating the `vector` command.

You may combine vector declaration and assignment. If there is no assignment statement, the vector will initially be filled with zeros.

Examples

```
vector vec1
vector(20) vec2=nrnd
rowvector(10) row3=3
vector vec3=row3
```

VEC1 is declared as a column vector of size one with element 0. VEC2 is declared as a column vector of size 20 containing a simulated random draw from the standard normal distribution. Although declared as a column vector, VEC3 is reassigned as a row vector of size 10 with all elements equal to 3.

Cross-references

See [Chapter 4, “Matrix Language”, on page 55](#) of the *Command and Programming Reference* for a discussion of matrices and vectors in EViews.

See also [coef \(p. 164\)](#) and [rowvector \(p. 304\)](#).

wald	Equation View Logl View Pool View Sspace View System View
------	---

Wald coefficient restriction test.

The `wald` view carries out a Wald test of coefficient restrictions for an estimation object.

Syntax

Object View: `object_name.wald restrictions`

Enter the object name, followed by a period, and the keyword `wald`. This should be followed by a list of the coefficient restrictions. Joint (multiple) coefficient restrictions should be separated by commas.

Options

`p` Print the test results.

Examples

`eq1.wald c(2)=0, c(3)=0`

tests the null hypothesis that the second and third coefficients in equation EQ1 are jointly zero.

`sys1.wald c(2)=c(3)*c(4)`

tests the non-linear restriction that the second coefficient is equal to the product of the third and fourth coefficients in SYS1.

```
pool panel us uk jp  
panel.ls cons? inc? @ ar(1)  
panel.wald c(3)=c(4)=c(5)
```

declares a pool object with three cross section members (US, UK, JP), estimates a pooled OLS regression with separate AR(1) coefficients, and tests the null hypothesis that all AR(1) coefficients are equal.

Cross-references

See “[Wald Test \(Coefficient Restrictions\)](#)” on page 368 of the *User’s Guide* for a discussion of Wald tests.

See also [testdrop \(p. 358\)](#), [testadd \(p. 355\)](#).

white	Equation View Var View
--------------	--

White’s test for heteroskedasticity.

Carries out White’s test for heteroskedasticity to the residuals of the specified equation. By default, the test is computed without the cross-product terms (using only the terms involving the original variables and squares of the original variables). You may elect to compute the original form of the White test including the cross-products.

White’s test is not available for equations estimated by `binary`, `ordered`, `censored`, or `count`. For a `var` object, the `white` command computes the multivariate version of the test.

Syntax

Equation View: `eq_name.white(options)`

Var View: `var_name.white(options)`

Options

`c` Include all possible nonredundant cross-product terms in the test regression.

`p` Print the test results.

Options for Var View

`name = arg` Save test statistics in named matrix object. See below for a description of the statistics stored in the matrix.

For `var` views, the “`name =`” option stores a $(r + 1) \times 5$ matrix, where r is the number of unique residual cross-product terms. For a VAR with k endogenous variables, $r = k(k + 1)/2$. The first r rows contain statistics for each individual test equation, where the first column is the regression R-squared, the second column is the F -statistic,

the third column is the p -value of F -statistic, the 4th column is the $T \times R^2$ chi-square statistic, and the fifth column is the p -value of the chi-square statistic. The numerator and denominator degrees of freedom of the F -statistic are stored in the $(r + 1)$ -st row, third and fourth columns, while the chi-square degrees of freedom is stored in the $(r + 1)$ -st row, fifth column.

In the $(r + 1)$ -st row and first column contains the joint (system) LM chi-square statistic and the second column contains the degrees of freedom of this χ^2 statistic.

Examples

```
eq1.white(c)
```

carries out the White test of heteroskedasticity including all possible cross-product terms.

Cross-references

See “[White's Heteroskedasticity Test](#)” on page 378 for a discussion of White’s test. For the multivariate version of this test, see [page 526](#) of the *User’s Guide*.

wls	System Method
-----	-------------------------------

Weighted least squares.

wls estimates a system of equations using weighted least squares. To perform weighted least squares in single equation estimation, see [1s \(p. 245\)](#).

Syntax

System Method: `system_name.wls(options)`

Options

i	Iterate simultaneously over the weighting matrix and coefficient vector.
s	Iterate sequentially over the computation of the weighting matrix and the estimation of the coefficient vector.
o (default)	Iterate the estimate of the coefficient vector to convergence following one-iteration of the weighting matrix.
c	One step (iteration) of the coefficient vector estimates following one iteration of the weighting matrix.
m = <i>integer</i>	Maximum number of iterations.

<i>c = number</i>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
<i>l = number</i>	Set maximum number of iterations on the first-stage coefficient estimation to get one-step weighting matrix.
<i>showopts / -showopts</i>	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
<i>deriv = keyword</i>	Set derivative methods. The argument <i>keyword</i> should be a one- or two-letter string. The first letter should either be “f” or “a” corresponding to fast or accurate numeric derivatives (if used). The second letter should be either “n” (always use numeric) or “a” (use analytic if possible). If omitted, EViews will use the global defaults.
<i>p</i>	Print the estimation results.

Examples

`sys1.wls`

estimates the system of equations in SYS1 by weighted least squares.

Cross-references

See [Chapter 19](#) of the *User’s Guide* for a discussion of system estimation.

See also the available options for weighted least squares in [1s \(p. 245\)](#).

workfile	Command
----------	---------

Create or change workfiles.

Syntax

Command: `workfile name`

Command: `workfile name frequency start end`

Typing `workfile` without arguments is equivalent to selecting **File/New/Workfile**. A dialog box will open prompting you for additional information concerning the frequency and dates for the workfile (see [“Creating a Workfile” on page 34](#) of the *User’s Guide* for the rules of specifying dates). Once you specify the information, EViews will create an untitled workfile of the specified type.

You may also use the keyword `workfile` followed by a name for a workfile. If a workfile with that name is already loaded into memory, the workfile will become active.

If you specify `workfile` with a name, followed by a frequency code and a start and end date or observation, a new workfile will be created.

Options

The frequency codes for workfiles are:

a	Annual.
s	Semi-annual.
q	Quarterly.
m	Monthly.
w	Weekly.
d	Daily (5 day week).
7	Daily (7 day week).
u	Undated or irregular.

Examples

```
workfile test
```

either makes the loaded workfile named TEST active or brings up a dialog to create a new workfile named TEST.

```
workfile macro q 45.1 2000
```

creates a quarterly workfile named MACRO from 1945.1 to 2000.4.

```
workfile psid u 1 20000
```

creates an undated workfile named PSID with 20000 observations.

Cross-references

See “[Creating a Workfile](#)” on page 34 of the *User’s Guide* and [Appendix B, “Date Formats”](#), on page 653 of the *User’s Guide* for a discussion of frequencies and rules for composing dates in EViews.

See also [create \(p. 176\)](#), [load \(p. 244\)](#) and [open \(p. 275\)](#).

write	Command Coef Proc Matrix Proc Pool Proc Sym Proc Vector Proc
--------------	---

Write series to a disk file.

The `write` command creates a foreign format disk file containing any number of series. You should use `write` when you wish to export EViews data to another program.

Syntax

Command: `write(options) path\file name1 name2 name3 ...`

Pool Proc: `pool_name.write(options) path\file n1? n2? n3? ...`

Coef Proc: `coef_name.write(options) path\file`

Matrix Proc: `matrix_name.write(options) path\file`

Follow the `write` keyword by a name for the output file and list the series to be written. The optional path name may be on the local machine, or may point to a network drive. If the path name contains spaces, enclose the entire expression in double quotation marks. To write matrix objects, simply provide a filename; the entire matrix will be exported.

Note that EViews cannot, at present, write into an existing file. The file that you select will, if necessary, be replaced.

Options

Options are specified in parentheses after the `write` keyword and are used to specify the format of the output file.

File type

<code>t = dat, txt</code>	ASCII (plain text) files.
---------------------------	---------------------------

<code>t = wk1, wk3</code>	Lotus spreadsheet files.
---------------------------	--------------------------

<code>t = xls</code>	Excel spreadsheet files.
----------------------	--------------------------

If you omit the “`t =`” option, EViews will determine the type based on the file extension. Unrecognized extensions will be treated as ASCII files. For Lotus and Excel spreadsheet files specified without the “`t =`” option, EViews will automatically append the appropriate extension if it is not otherwise specified.

ASCII text files

na = text	Specify text for NAs. Default is “NA”.
dates	Write dates/obs and (for pool) cross section identifiers.
nodates (default)	Do not write dates/obs and (for pool) cross-section identifiers.
names (default)	Write series names.
nonames	Do not write series names.
id	Write cross-section identifier.
d = s	Single space delimiter (default is tab).
d = c	Comma delimiter (default is tab).
byper	Panel data organized by date/period. Default is data organized by cross-section (only for pool write).
bycross (default)	Panel data organized by cross-section (only for pools).
t	Write by series (or transpose the data for matrix objects). Default is to read by obs with series in columns.

Spreadsheet (Lotus, Excel) files

letter_number	Coordinate of the upper-left cell containing data.
dates (default)	Write dates/obs and (for pool) cross-section identifiers.
dates = first	Write date in Excel date format converting to the first day of the corresponding observation if necessary (only for Excel files).
dates = last	Write date in Excel date format converting to the last day of the corresponding observation if necessary (only for Excel files).
nodates	Do not write dates/obs and (pool) cross-section identifiers.
names (default)	Write series names.
nonames	Do not write series names.

byper	Panel data organized by date/period. Default is data organized by cross-section (only for pool write).
bycross (default)	Panel data organized by cross-section (only for pools).
t	Write by series (or transpose the data for matrix objects). Default is to write by obs with each series in columns.

Examples

```
write(t=txt,na=.,d=c,dates) a:\dat1.csv hat1 hat_se1
```

Writes the two series HAT1 and HAT_SE1 into an ASCII file named DAT1.CSV on the A drive. The data file is listed by observations, NAs are coded as “.” (dot), each series is separated by a comma, and the date/observation numbers are written together with the series names.

```
write(t=txt,na=.,d=c,dates) dat1.csv hat1 hat_se1
```

writes the same file in the default directory.

```
mypool.write(t=xls,per) "\\\network\drive a\growth" gdp? edu?
```

writes an Excel file GROWTH.XLS in the specified directory. The data are organized by observations and are listed by period/time.

Cross-references

See “[Exporting to a Spreadsheet or Text File](#)” on page 71 of the *User’s Guide* for a discussion.

See also [read \(p. 291\)](#).

wtls	System Method
------	-------------------------------

Weighted two-stage least squares.

wtls estimates a system of equations using weighted two-stage least squares. To perform weighted two-stage least squares in a single equation, see [ls \(p. 245\)](#).

Syntax

System Method: system_name.wtls(*options*)

Options

i	Iterate simultaneously over the weighting matrix and coefficient vector.
s	Iterate sequentially over the computation of the weighting matrix and the estimation of the coefficient vector.
o (default)	Iterate the coefficient vector to convergence following one-iteration of the weighting matrix.
c	One step (iteration) of the coefficient vector following one iteration of the weighting matrix.
m = <i>integer</i>	Maximum number of iterations.
c = <i>number</i>	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients.
l = <i>number</i>	Set maximum number of iterations on the first-stage iteration to get the one-step weighting matrix.
showopts / -showopts	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
deriv = <i>keyword</i>	Set derivative methods. The argument <i>keyword</i> should be a one- or two-letter string. The first letter should either be “f” or “a” corresponding to fast or accurate numeric derivatives (if used). The second letter should be either “n” (always use numeric) or “a” (use analytic if possible). If omitted, EViews will use the global defaults.
p	Print estimation results.

Examples

```
sys1.wtsls
```

estimates the system of equations in SYS1 by weighted two-stage least squares.

Cross-references

See “[Weighted Two-Stage Least Squares](#)” on page 497 of the *User’s Guide* for further discussion.

See also [tsls](#) (p. 368) for both unweighted and weighted single equation 2SLS.

x11	Series Proc
-----	-------------

Seasonally adjust series using the Census X11.2 method.

Syntax

Series Proc: *series_name.x11(options) adj_name fac_name*

The X11 procedure carries out Census X11.2 seasonal adjustment. Enter the name of the original series followed by a period, the keyword `x11` and then provide a name for the seasonally adjusted series. You may optionally list a second series name for the seasonal factors. The seasonal adjustment method is specified as an option in parentheses after the `x11` keyword.

The X11 procedure is available only for quarterly and monthly series. The procedure requires at least four full years of data, and can adjust up to 20 years of monthly data and 30 years of quarterly data.

Options

m	Multiplicative seasonals.
a	Additive seasonals.
s	Use sliding spans.
h	Adjustment for all holidays (only for monthly data specified with the <code>m</code> option).
i	Adjustment for holidays if significant (only for monthly data specified with the “m” option).
t	Adjustment for all trading days (only for monthly data).
q	Adjustment for trading days if significant (only for monthly data).
p	Print the X11 results.

Examples

```
sales.x11(m,h) salesx11 salesfac
```

seasonally adjusts the SALES series and saves the adjusted series as SALESX11 and the seasonal factors as SALESFAC. The adjustment assumes multiplicative seasonals and makes adjustment for all holidays.

Cross-references

See “[Census X11 \(Historical\)](#)” on page 184 of the *User’s Guide* for a discussion of Census X11 seasonal adjustment method.

Note that the X11 routines are separate programs provided by the Census and are installed in the EViews directory in the files X11Q2.EXE and X11SS.EXE. The documentation for these programs can also be found in your EViews directory as text files X11DOC1.TXT through X11DOC3.TXT.

See also [seas](#) (p. 315), [seasplot](#) (p. 316) and [x12](#) (p. 388).

x12	Series Proc
------------	-----------------------------

Seasonally adjust series using the Census X12 method.

`x12` is available only for quarterly and monthly series. The procedure requires at least 3 full years of data and can adjust up to 600 observations (50 years of monthly data or 150 years of quarterly data).

Syntax

Series Proc: `series_name.x12(options) adj_base_name`

Enter the name of the original series followed by a dot, the `x12` keyword, and a base name (no more than the maximum length of a series name minus 4) for the saved series. If you do not provide a base name, the original series name will be used as a base name. See the description in “`save =`” option below for the naming convention used to save series.

Options

Commonly Used Options

<code>mode = m</code> (default)	Multiplicative seasonal adjustment. <i>Series must take non-negative values.</i>
<code>mode = a</code>	Additive seasonal adjustment.
<code>mode = p</code>	Pseudo-additive seasonal adjustment.
<code>mode = l</code>	Log-additive seasonal adjustment. <i>Series must take non-negative values.</i>
<code>filter = msr</code> (default)	Automatic (moving seasonality ratio) seasonal filter.
<code>filter = x11</code>	X11 default seasonal filter.
<code>filter = stable</code>	Stable seasonal filter.

filter = s3x1	3x1 moving average seasonal filter.
filter = s3x3	3x3 moving average seasonal filter.
filter = s3x5	3x5 moving average seasonal filter.
filter = s3x9	3x9 moving average seasonal filter.
filter = s3x15	3x15 moving average seasonal filter. <i>Series must have at least 20 years of data.</i>
save = "arg"	<p>Optionally saved series enclosed in quotes. List the extension (given in Table 6-8, p.71 of the <i>X12-ARIMA Reference Manual</i>) for the series you want to save. Commonly used options and its stored name (in parentheses) are given below:</p> <ul style="list-style-type: none"> "d10" final seasonal factors (basename_sf). "d11" final seasonally adjusted series (basename_sa). "d12" final trend-cycle component (basename_tc). "d13" final irregular component (basename_ir). <p>All other options are named using the option symbol. For example ‘save = “d16”’ will store a series named <u>BASENAME_D16</u>.</p> <p>To save more than two series, separate the list with a space. For example, ‘save = “d10 d12”’ saves the seasonal factors and the trend-cycle series.</p>
tf = logit	Logit transformation for regARIMA.
tf = auto	Automatically choose between no transformation and log transformation for regARIMA.
tf = number	Box-Cox power transformation using specified parameter for regARIMA. Use “tf = 0” for log transformation.
sspan	Sliding spans stability analysis. <i>Cannot be used along with the “h” option.</i>
history	Historical record of seasonal adjustment revisions. <i>Cannot be used along with the “sspan” option.</i>
check	Check residuals of regARIMA.
outlier	Outlier analysis of regARIMA.

x11reg =	Regressors to model the irregular component in seasonal adjustment. Regressors must be chosen from the predefined list in Table 6-14, p. 88 of the <i>X12-ARIMA Reference Manual</i> . To specify more than one regressor, separate by a space within the double quotes.
reg = <i>arg_list</i>	Regressors for the regARIMA model. Regressors must be chosen from the predefined list in Table 6-17, pp. 100-101 of the <i>X12-ARIMA Reference Manual</i> . To specify more than one regressor, separate by a space within the double quotes.
arima = <i>arg</i>	ARIMA spec of the regARIMA model. Must follow the X12 ARIMA specification syntax. <i>Cannot be used together with the “amdl =” option</i> .
amdl = f	Automatically choose the ARIMA spec. Use forecasts from the chosen model in seasonal adjustment. <i>Cannot be used together with the arima = option and must be used together with the mfile = option</i> .
amdl = b	Automatically choose the ARIMA spec. Use forecasts and backcasts from the chosen model in seasonal adjustment. <i>Cannot be used together with the “arima =” option and must be used together with the “mfile =” option</i> .
best	Sets the method option of the auto model spec to best (default is first). Also sets the identify option of the auto model spec to all (default is first). <i>Must be used together with the “amdl =” option</i> .
mod- elsmpl = <i>arg</i>	Sets the subsample for fitting the ARIMA model. Either specify a sample object name or a sample range. <i>The model sample must be a subsample of the current workfile sample and should not contain any breaks</i> .
mfile = <i>arg</i>	Specifies the file name (include the extension, if any) that contains a list of ARIMA specifications to choose from. <i>Must be used together with the “amdl =” option</i> . The default is the X12A.MDL file provided by the Census.

outsmpl	Use out-of-sample forecasts for automatic model selection. Default is in-sample forecasts. <i>Must be used together with the “amdl = ” option.</i>
plotspectra	Save graph of spectra for differenced seasonally adjusted series and outlier modified irregular series. The saved graph will be named GR_ <i>seriesname</i> _SP.
p	Print X12 procedure results.

Other Options

hma = <i>integer</i>	Specifies the Henderson moving average to estimate the final trend-cycle. The X12 default is automatically selected based on the data. To override this default, specify an <i>odd integer between 1 and 101</i> .
sigl = <i>arg</i>	Specifies the lower sigma limit used to downweight extreme irregulars in the seasonal adjustment. The default is 1.5 and you can specify any positive real number.
sigh = <i>arg</i>	Specifies the upper sigma limit used to downweight extreme irregulars in the seasonal adjustment. The default is 2.5 and you can specify any positive real number less than the lower sigma limit.
ea	Nonparametric Easter holiday adjustment (x11easter). <i>Cannot be used together with the “easter[w]” regressor in the “reg = ” or “x11reg = ” options.</i>
f	Appends forecasts up to one year to some optionally saved series. Forecasts are appended only to the following series specified in the “save = ” option: “b1” (original series, adjusted for prior effects), “d10” (final seasonal factors), “d16” (combined seasonal and trading day factors).
flead = <i>integer</i>	Specifies the number of periods to forecast (to be used in the seasonal adjustment procedure). The default is one year and you can specify an integer up to 60.
fback = <i>integer</i>	Specifies the number of periods to backcast (to be used in the seasonal adjustment procedure). The default is 0 and you can specify an integer up to 60. No backcasts are produced for series more than 15 years long.

aicx11	Test (based on AIC) whether to retain the regressors specified in “x11reg =”. <i>Must be used together with the “x11reg =” option.</i>
aicreg	Test (based on AIC) whether to retain the regressors specified in “reg =”. <i>Must be used together with the “reg =” option.</i>
sfile = <i>arg</i>	Path/name (including extension, if any) of user provided specification file. The file must follow a specific format; see the discussion below.

User provided spec file

The `x12` command in EViews provides most of the basic options in the X12 program. For users who need to access the full set of options, we have provided an option to pass your own X12 specification file from EViews. The advantage of using this option (as opposed to running the X12 program in DOS) is that EViews will automatically handle the data in the input and output series.

To provide your own specification file, specify the path/name of your file using the “`sfile =`” option in the `x12` proc. Your specification file should follow the format of an X12 specification file as described in the *X12-ARIMA Reference Manual*, except

- the specification file should not have a series spec nor a composite spec.
- the `x11` spec must have a `save` option for `D11` (the final seasonally adjusted series) and any other extensions you want to store. EViews will always look for `D11` and will error if not found.
- to read back data for a `save` option other than `D11`, you must use the “`save =`” option in the `x12` proc. For example, to get back the final trend-cycle series (`D12`) into EViews, you must have a “`save =`” option for `D12` (and `D11`) in the `x11` spec of your specification file and a “`save = d12`” option in the EViews `x12` proc.

Note that when you use an “`sfile =`” option, EViews will ignore any other options in the `x12` proc, except for the “`save =`” option.

Difference between the dialog and command line

The options corresponding to the **Trading Day/Holiday** and **Outliers** tab in the X12 dialog should be specified by listing the appropriate regressors in the “`x11reg =`” and “`reg =`” options.

Examples

The command

```
sales.x12(mode=m, save="d10 d12") salesx12
```

seasonally adjusts the SALES series in multiplicative mode. The seasonal factors (d10) are stored as SALESX12_SF and the trend-cycles series is stored as SALESX12_TC.

```
sales.x12(tf=0, arima="(0 0 1)", reg="const td")
```

specifies a regARIMA model with a constant, trading day effect variables, and MA(1) using a log transformation. This command does not store any series.

```
freeze(x12out) sales.x12(tf=auto, amdl=f, mfile=
    "c:\eviews\mymdl.txt")
```

stores the output from X12 in a text object named X12OUT. The options specify an automatic transformation and an automatic model selection from the file MYMDL.TXT.

```
revenue.x12(tf=auto, sfile="c:\eviews\spec1.txt", save="d12
    d13")
```

adjusts the series REVENUE using the options given in the SPEC1.TXT file. Note the following: (1) the "tf = auto" option will be ignored (you should specify this in your specification file) and (2) EVViews will save two series REVENUE_TC and REVENUE_IR which will be filled with NAs unless you provided the save option for D12 and D13 in your specification file.

```
freeze(x12out) sales.x12(tf=auto, amdl=f, mfile=
    "c:\eviews\mymdl.txt")
```

stores the output from X12 in a text object named X12OUT. The options specify an automatic transformation and an automatic model selection from the file MYMDL.TXT. The seasonally adjusted series is stored as SALES_SA by default.

```
revenue.x12(tf=auto, sfile="c:\eviews\spec1.txt", save="d12
    d13")
```

adjusts the series REVENUE using the options given in the SPEC1.TXT file. Note the following: (1) the “tf = auto” option will be ignored (you should specify this in your specification file) and (2) EVViews will error if you did not specify a save option for D11, D12, and D13 in your specification file.

Cross-references

See “[Census X12](#)” on page 177 of the *User’s Guide* for a discussion of the Census X12 program. The documentation for X12, *X12-ARIMA Reference Manual*, can be found in the DOCS subdirectory of your EVViews directory in the PDF files FINALPT1.PDF and FINALPT2.PDF.

See also [seas](#) (p. 315) and [x11](#) (p. 387).

xyline[Graph Proc](#) | [Group View](#) | [Matrix View](#) | [Sym View](#)

XY line graph.

The `xyline` view of a graph object changes the graph to XY line (if possible). The `xyline` view of a group displays an XY plot.

By default, the first series or column of data will be located along the horizontal axis and the remaining data on the vertical axis. You may optionally choose to plot the data in pairs, where the first two series or columns are plotted against each other, the second two series or columns are plotted against each other, and so forth.

When used as a group or matrix view, there must be at least series or columns in the object.

Syntax

Graph Proc: `graph_name.xyline(options)`

Group View: `group_name.xyline(options)`

Options

Options may be specified in parentheses after the `xyline` keyword.

Template and printing options

`o = graph_name` Use appearance options from the specified graph.

`t = graph_name` Use appearance options and copy text and shading from the specified graph.

`p` Print the XY line graph.

Scale options

`a` (default) Automatic single scale.

`b` Plot X and Y series in pairs.

`n` Normalized scale (zero mean and unit standard deviation).

`d` Dual scaling with no crossing.

x	Dual scaling with possible crossing.
s	Stack so that the difference between lines corresponds to the value of the associated series (not available if “b” option is set)..
m	Display XY plots as multiple graphs.

Examples

```
group g1 inf unemp gdp inv  
g1.xyline
```

plots INF on the horizontal axis and UNEMP, GDP and INV on the vertical axis.

```
g1.xyline(b)  
g1.xyline(b,m)
```

plots INF against UNEMP and GDP against INV in first in a single graph, and then in multiple graphs.

Cross-references

See “[XY Line](#)” on page 210 of the *User’s Guide* for additional details.

See also [graph \(p. 224\)](#) for additional graph types. See also [options \(p. 275\)](#) for graph options.

Chapter 9. Matrix and String Reference

The following is an alphabetical listing of the functions and commands used when working with the EViews matrix language. For a description of the EViews matrix language, see [Chapter 4, “Matrix Language”, on page 55](#). For a quick summary of these entries, see [“Matrix Function and Command Summary” on page 76](#).

@cholesky	Matrix Algebra Function
------------------	---

Syntax: `@cholesky(s)`

Argument: `sym, s`

Return: `matrix`

Returns a matrix containing the Cholesky factorization of s . The Cholesky factorization finds the lower triangular matrix L such that LL' is equal to the symmetric source matrix. Example:

```
matrix fact = @cholesky(s1)
matrix orig = fact*@transpose(fact)
```

colplace	Matrix Utility Command
-----------------	--

Syntax: `colplace(m, v, n)`

Argument 1: `matrix, m`

Argument 2: `vector, v`

Argument 3: `integer, n`

Places the column vector v into the matrix m at column n . The number of rows of m and v must match, and the destination column must already exist within m . Example:

```
colplace(m1,v1,3)
```

The third column of M1 will be set equal to the vector V1.

See also [rowplace \(p. 414\)](#).

@columnextract	Matrix Utility Function
-----------------------	---

Syntax: `@columnextract(m, n)`
Argument 1: matrix or sym, *m*
Argument 2: integer, *n*
Return: vector

Extract a vector from column *n* of the matrix object *m*, where *m* is a matrix or sym.

Example:

```
vector v1 = @columnextract(m1, 3)
```

Note that while you may extract the first column of a column vector, or any column of a row vector, the command is more easily executed using simple element or vector assignment in these cases.

See also [@rowextract \(p. 414\)](#).

@columns	Matrix Utility Function
-----------------	---

Syntax: `@columns(o)`
Argument: matrix, vector, rowvector, sym, scalar, or series, *o*
Return: integer

Returns the number of columns in the matrix object *o*. Example:

```
scalar sc2 = @columns(m1)
vec1(2) = @columns(s1)
```

See also [@rows \(p. 414\)](#).

@cond	Matrix Algebra Function
--------------	---

Syntax: `@cond(o, n)`
Argument 1: matrix or sym, *o*
Argument 2: (optional) scalar *n*
Return: scalar

Returns the condition number of a square matrix or sym, o . If no norm is specified, the infinity norm is used to determine the condition number. The condition number is the product of the norm of the matrix divided by the norm of the inverse. Possible norms are “-1” for the infinity norm, “0” for the Frobenius norm, and an integer “ n ” for the L^n norm.

Example:

```
scalar scl = @cond(m1)
mat1(1, 4) = @cond(s1, 2)
```

@convert	Matrix Utility Function
-----------------	-------------------------

Syntax:	@convert(o , <i>smp</i>)
Argument 1:	series or group, o
Argument 2:	(optional) sample, <i>smp</i>
Return:	vector or matrix

If o is a series, @convert returns a vector from the values of o using the optional sample *smp* or the current workfile sample. If any observation has the value “NA”, the observation will be omitted from the vector. Examples:

```
vector v2 = @convert(ser1)
vector v3 = @convert(ser2, smp1)
```

If o is a group, @convert returns a matrix from the values of o using the optional sample object *smp* or the current workfile sample. The series in o are placed in the columns of the resulting matrix in the order they appear in the group spreadsheet. If any of the series for a given observation, has the value “NA”, the observation will be omitted for all series. For example,

```
matrix m1=@convert(grp1)
matrix m2=@convert(grp1, smp1)
```

For a conversion method that preserves NAs, see [stomna](#) (p. 416).

@COR	Matrix Descriptive Statistic
-------------	------------------------------

Syntax: $\text{@cor}(v1, v2)$
Argument 1: vector, rowvector, or series, $v1$
Argument 2: vector, rowvector, or series, $v2$
Return: scalar

Syntax: $\text{@cor}(o)$
Argument: matrix object or group, o
Return: sym

If used with two vector or series objects, $v1$ and $v2$, `@cor` returns the correlation between the two vectors or series. Examples:

```
scalar sc1 = @cor(v1,v2)
s1(1,2) = @cor(v1,r1)
```

If used with a matrix object or group, o , `@cor` calculates the correlation matrix between the columns of the matrix object.

```
scalar sc2 = @cor(v1,v2)
mat3(4,2) = 100*@cor(r1,v1)
```

For series and group calculations, EViews will use the current workfile sample. See also [@cov \(p. 400\)](#).

@COV	Matrix Descriptive Statistic
-------------	------------------------------

Syntax: $\text{@cov}(v1, v2)$
Argument 1: vector, rowvector, or series, $v1$
Argument 2: vector, rowvector, or series, $v2$
Return: scalar

Syntax: $\text{@cov}(o)$
Argument: matrix object or group, o
Return: sym

If used with two vector or series objects, $v1$ and $v2$, `@cov` returns the covariance between the two vectors or series. Examples:

```
scalar scl = @cov(v1, v2)
s1(1,2) = @cov(r1, v1)
```

If used with a matrix object or group, *o*, @cov calculates the covariance matrix between the columns of the matrix object.

```
!1 = @cov(v1, v2)
mat3(4,2) = 100*@cov(r1, v1)
```

For series and group calculations, EViews will use the current workfile sample. See also [@cor \(p. 400\)](#).

@det	Matrix Algebra Function
----------------------	---

Syntax:	<code>@det(<i>m</i>)</code>
Argument:	matrix or sym, <i>m</i>
Return:	scalar

Calculate the determinant of the square matrix or sym, *m*. The determinant is nonzero for a nonsingular matrix and 0 for a singular matrix. Example:

```
scalar scl = @det(m1)
vec4(2) = @det(s2)
```

See also [@rank \(p. 412\)](#).

@dtoo	String Function
-----------------------	---------------------------------

Syntax:	<code>@dtoo(<i>str</i>)</code>
Argument:	string, <i>str</i>
Return:	integer

Date-TO-Observation. Returns the observation number corresponding to the date contained in the string. The observation number is relative to the start of the current workfile range, not the current sample. Observation numbers may be used to select a particular element in a vector that has been converted from a series, provided that NAs are preserved (see *stomna*). Examples:

```
scalar obnum = @dtoo("1994:01")
vec1(1) = gdp(@dtoo("1955:01") + 10)
```

Suppose that the workfile contains quarterly data. Then the second example places the 1957:02 value of the GDP series in the first element of VEC1.

See also [@otod](#).

@eigenvalues	Matrix Algebra Function
---------------------	---

Syntax: [@eigenvalues\(s\)](#)

Argument: `sym, s`

Return: `vector`

Returns a vector containing the eigenvalues of the `sym`. The eigenvalues are those scalars λ that satisfy $Sx = \lambda x$, where S is the `sym` associated with the argument s . Associated with each eigenvalue is an eigenvector (see [@eigenvectors \(p. 402\)](#)). The eigenvalues are arranged in ascending order.

Example:

```
vector v1 = @eigenvalues(s1)
```

@eigenvectors	Matrix Algebra Function
----------------------	---

Syntax: [@eigenvectors\(s\)](#)

Argument: `sym, s`

Return: `matrix`

Returns a square matrix, of the same dimension as the `sym`, whose columns are the eigenvectors of the source matrix. Each eigenvector v satisfies $Sv = nv$, where S is the symmetric matrix given by s , and where n is the eigenvalue associated with the eigenvector v . The eigenvalues are arranged in ascending order, and the columns of the eigenvector matrix correspond to the sorted eigenvalues. Example:

```
matrix m2 = @eigenvectors(s1)
```

See also the function [@eigenvalues \(p. 402\)](#).

@explode	Matrix Utility Function
-----------------	---

Syntax: [@explode\(s\)](#)

Argument: `sym, s`

Return: `matrix`

Creates a square matrix from a `sym`, s , by duplicating the lower triangle elements into the upper triangle. Example:

```
matrix m2 = @explode(s1)
```

See also [@implode \(p. 405\)](#).

@filledmatrix	Matrix Utility Function
----------------------	---

Syntax:	<code>@filledmatrix(n1, n2, n3)</code>
Argument 1:	integer, $n1$
Argument 2:	integer, $n2$
Argument 3:	scalar, $n3$
Return:	matrix

Returns a matrix with $n1$ rows and $n2$ columns, where each element contains the value $n3$.

Example:

```
matrix m2 = @filledmatrix(3, 2, 7)
```

creates a 3×2 matrix where each element is set to 7. See also [fill \(p. 208\)](#).

@filledrowvector	Matrix Utility Function
-------------------------	---

Syntax:	<code>@filledrowvector(n1, n2)</code>
Argument 1:	integer, $n1$
Argument 2:	scalar, $n2$
Return:	rowvector

Returns a rowvector of length $n1$, where each element contains the value $n2$. Example:

```
rowvector r1 = @filledrowvector(3, 1)
```

creates a 3 element rowvector where each element is set to 1. See also [fill \(p. 208\)](#).

@filledsym	Matrix Utility Function
-------------------	---

Syntax:	<code>@filledsym(n1, n2)</code>
Argument 1:	integer, $n1$
Argument 2:	scalar, $n2$
Return:	sym

Returns an $n1 \times n1$ sym, where each element contains $n2$. Example:

```
sym s2= @filledsym(3, 9)
```

creates a 3×3 sym where each element is set to 9. See also [fill \(p. 208\)](#).

@filledvector	Matrix Utility Function
----------------------	---

Syntax: `@filledvector(n1, n2)`

Argument 1: integer, *n1*

Argument 2: scalar, *n2*

Return: vector

Returns a vector of length *n1*, where each element contains the value *n2*. Example:

```
vector r1 = @filledvector(5, 6)
```

creates a 5 element column vector where each element is set to 6. See also [fill \(p. 208\)](#).

@getmaindiagonal	Matrix Utility Function
-------------------------	---

Syntax: `@getmaindiagonal(m)`

Argument: matrix or sym, *m*

Return: vector

Returns a vector created from the main diagonal of the matrix or sym object. Example:

```
vector v1 = @getmaindiagonal(m1)
```

```
vector v2 = @getmaindiagonal(s1)
```

@identity	Matrix Utility Function
------------------	---

Syntax: `@identity(n)`

Argument: integer, *n*

Return: matrix

Returns a square $n \times n$ identity matrix. Example:

```
matrix i1 = @identity(4)
```

@implode	Matrix Utility Function
-----------------	---

Syntax: `@implode(m)`

Argument: square matrix, *m*

Return: *sym*

Forms a *sym* by copying the lower triangle of a square input matrix, *m*. Where possible, you should use a *sym* in place of a matrix to take advantage of computational efficiencies. Be sure you know what you are doing if the original matrix is not symmetric—this function does not check for symmetry. Example:

```
sym s2 = @implode(m1)
```

See also [@explode \(p. 402\)](#).

@inner	Matrix Algebra Function
---------------	---

Syntax: `@inner(v1, v2, smp)`

Argument 1: vector, rowvector, or series, *v1*

Argument 2: vector, rowvector, or series, *v2*

Argument 3: (optional) sample, *smp*

Return: scalar

Syntax: `@inner(o, smp)`

Argument 1: matrix object or group, *o*

Argument 2: (optional) sample, *smp*

Return: scalar

If used with two vectors, *v1* and *v2*, `@inner` returns the dot or inner product of the two vectors. Examples:

```
scalar sc1 = @inner(v1, v2)
s1(1,2) = @inner(v1, r1)
```

If used with two series, `@inner` returns the inner product of the series using observations in the workfile sample. You may provide an optional sample.

If used with a matrix or sym, o , $@inner$ returns the inner product, or moment matrix, $o' o$. Each element of the result is the vector inner product of two columns of the source matrix. The size of the resulting sym is the number of columns in o . Examples:

```
scalar sc1 = @inner(v1)
sym sym1 = @inner(m1)
```

If used with a group, $@inner$ returns the uncentered second moment matrix of the data in the group, g , using the observations in the sample, smp . If no sample is provided, the workfile sample is used. Examples:

```
sym s2 = @inner(gr1)
sym s3 = @inner(gr1, smp1)
```

See also [@outer \(p. 411\)](#).

@inverse	Matrix Algebra Function
-----------------	---

Syntax: $@inverse(m)$
Argument: square matrix or sym, m
Return: matrix or sym

Returns the inverse of a square matrix object or sym. The inverse has the property that the product of the source matrix and its inverse is the identity matrix. The inverse of a matrix returns a matrix, while the inverse of a sym returns a sym. Note that inverting a sym is much faster than inverting a matrix.

Examples:

```
matrix m2 = @inverse(m1)
sym s2 = @inverse(s1)
sym s3 = @inverse(@implode(m2))
```

See [@solvesystem \(p. 415\)](#).

@issingular	Matrix Algebra Function
--------------------	---

Syntax: $@issingular(o)$
Argument: matrix or sym, o
Return: integer

Returns “1” if the square matrix or sym, o , is singular, and “0” otherwise. A singular matrix has a determinant of 0, and cannot be inverted.

Example:

```
scalar scl = @issingular(m1)
```

@kronecker	Matrix Algebra Function
-------------------	---

Syntax: `@kronecker(o1, o2)`

Argument 1: matrix object, *o1*

Argument 2: matrix object, *o2*

Return: matrix

Calculates the Kronecker product of the two matrix objects, *o1* and *o2*. The resulting matrix has a number of rows equal to the product of the numbers of rows of the two matrix objects and a number of columns equal to the product of the numbers of columns of the two matrix objects. The elements of the resulting matrix consist of submatrices consisting of one element of the first matrix object multiplied by the entire second matrix object.

Example:

```
matrix m3 = @kronecker(m1, m2)
```

@left	String Function
--------------	---------------------------------

Syntax: `@left(str, n)`

Argument 1: string, *str*

Argument 2: integer, *n*

Return: string

Returns a string containing *n* characters from the left end of *str*. If the string is shorter than *n* characters, this function returns all of the characters in the source string.

Example:

```
scalar scl = @left("I did not do it", 5)
```

returns "I did".

See also [@right \(p. 413\)](#), [@mid \(p. 409\)](#).

@makediagonal	Matrix Utility Function
----------------------	---

Syntax: `@makediagonal(v, k)`

Argument 1: vector or rowvector, *v*

Argument 2: (optional) integer, *k*

Return: matrix

Creates a square matrix with the specified vector or rowvector, *v*, in the *k*-th diagonal relative to the main diagonal, and zeroes off the diagonal. If no *k* value is provided or if *k* is set to 0, the resulting matrix will have the same number of rows and columns as the length of *v*, and will have *v* in the main diagonal. If a value for *k* is provided, the matrix has the same number of rows and columns as the number of elements in the vector plus *k*, and will place *v* in the diagonal offset from the main by *k*. Example:

```
matrix m1 = @makediagonal(v1)
matrix m2 = @makediagonal(v1, 1)
matrix m4 = @makediagonal(r1, -3)
```

M1 will contain V1 in the main diagonal; M2 will contain V1 in the diagonal immediately above the main diagonal; M4 will contain R1 in the diagonal 3 positions below the main diagonal. Using the optional *k* parameter may be useful in creating covariance matrices for AR models. For example, you can create an AR(1) correlation matrix by issuing the commands:

```
matrix(10,10) m1
vector(9) rho = .3
m1 = @makediagonal(rho, -1) + @makediagonal(rho, +1)
m1 = m1 + @identity(10)
```

matplace	Matrix Utility Command
-----------------	--

Syntax: `matplace(m1, m2, n1, n2)`

Argument 1: matrix, *m1*

Argument 2: matrix, *m2*

Argument 3: integer, *n1*

Argument 4: integer, *n2*

Places the matrix object *m2* into *m1* at row *n1* and column *n2*. The sizes of the two matrices do not matter, as long as *m1* is large enough to contain all of *m2* with the upper left cell of *m2* placed at row *n1* and column *n2*.

Example:

```
matrix(100,5) m1
matrix(100,2) m2
matplace(m1,m2,1,1)
```

@mid	String Function
-------------	---------------------------------

Syntax:	<code>@mid(str, n1, n2)</code>
Argument 1:	string, <i>str</i>
Argument 2:	integer, <i>n1</i>
Argument 3:	integer, <i>n2</i>
Return:	string

Returns *n2* characters from *str*, starting at location *n1* and continuing to the right. If you omit *n2*, it will return all of the remaining characters in the string.

Example:

```
%1 = @mid("I doubt it", 9, 2)
%2 = @mid("I doubt it", 9)
```

See also [@left \(p. 407\)](#) and [@right \(p. 413\)](#).

mtos	Matrix Utility Command
-------------	--

Convert matrix to a series or group. Fills a series or group with the data from a vector or matrix.

Syntax

Vector Proc:	<code>mtos(vector, series[, sample])</code>
Matrix Proc:	<code>mtos(matrix, group[, sample])</code>

Matrix-TO-Series Object. Include the vector or matrix name in parentheses, followed by a comma and then the series or group name. The number of included observations in the sample must match the row size of the matrix to be converted. If no sample is provided, the matrix is written into the series using the current workfile sample.

Examples

The command

```
mtos (mom,gr1)
```

converts the first column of the matrix MOM to the first series in the group GR1, the second column of MOM to the second series in GR1, and so on. The current workfile sample length must match the row length of the matrix MOM. If GR1 is an existing group object, the number of series in GR1 must match the number of columns of MOM. If a group object named GR1 does not exist, EViews creates GR1 with the first series named SER1, the second series named SER2, and so on.

```
series col1
series col2
group g1 col1 col2
sample s1 1951 1990
mtos (m1,g1,s1)
```

The first two lines declare series objects, the third line declares a group object, the fourth line declares a sample object, and the fifth line converts the columns of the matrix M1 to series in group G1 using sample S1. This command will generate an error if M1 is not a 40×2 matrix.

Cross-references

See [Chapter 4, “Matrix Language”, on page 55](#) for further discussions and examples of matrices.

See also [stom \(p. 415\)](#) and [stomna \(p. 416\)](#).

@norm	Matrix Algebra Function
--------------	---

Syntax: `@norm(o, n)`

Argument 1: matrix, vector, rowvector, sym, scalar, or series, *o*

Argument 2: (optional) integer, *n*

Return: scalar

Returns the value of the norm of any matrix object, *o*. Possible choices for the norm type *n* include “-1” for the infinity norm, “0” for the Frobenius norm, and an integer “*n*” for the L^n norm. If no norm type is provided, this function returns the infinity norm.

Examples:

```
scalar sc1 = @norm(m1)
scalar sc2 = @norm(v1, 1)
```

@otod	String Function
--------------	---------------------------------

Syntax: `@otod(n)`

Argument: integer, *n*

Return: string

Observation-TO-Date. Returns a string containing the date or observation corresponding to observation number *n*, counted from the beginning of the current workfile. For example, consider the string assignment

```
%1 = @otod(5)
```

For an annual workfile dated 1991–2000, %1 will contain the string “1995”. For a quarterly workfile dated 1970:1–2000:4, %1 will contain the string “1971:1”. Note that `@otod(1)` returns the date or observation label for the start of the workfile.

See also `@dtoo`.

@outer	Matrix Algebra Function
---------------	---

Syntax: `@outer(v1, v2)`

Argument 1: vector, rowvector, or series, *v1*

Argument 2: vector, rowvector, or series, *v2*

Return: matrix

Calculates the cross product of *v1* and *v2*. Vectors may be either row or column vectors. The outer product is the product of *v1* (treated as a column vector) and *v2* (treated as a row vector), and is a square matrix of every possible product of the elements of the two inputs. Example:

```
matrix m1=@outer(v1,v2)
matrix m4=@outer(r1,r2)
```

See also [@inner \(p. 405\)](#).

@permute**Matrix Utility Function**Syntax: `@permute(m1)`Input: matrix *m1*

Return: matrix

This function returns a matrix whose rows are randomly drawn without replacement from rows of the input matrix *m1*. The output matrix has the same size as the input matrix.

```
matrix xp = @permute(x)
```

To draw with replacement from rows of a matrix, use [@resample \(p. 412\)](#).

@rank**Matrix Algebra Function**Syntax: `@rank(o, n)`Argument 1: vector, rowvector, matrix, sym, or series, *o*Argument 2: (optional) integer, *n*

Return: integer

Returns the rank of the matrix object *o*. The rank is calculated by counting the number of singular values of the matrix which are smaller in absolute value than the tolerance, which is given by the argument *n*. If *n* is not provided, EViews uses the value given by the largest dimension of the matrix multiplied by the norm of the matrix multiplied by machine epsilon (the smallest representable number).

```
scalar rank1 = @rank(m1)
scalar rank2 = @rank(s1)
```

See also [@svd \(p. 418\)](#).

@resample**Matrix Utility Function**Syntax: `@resample(m1, n2, n3, v4)`Input 1: matrix *m1*Input 2: (optional) integer *n2*Input 3: (optional) positive integer *n3*Input 4: (optional) vector *v4*

Output: matrix

This function returns a matrix whose rows are randomly drawn with replacement from rows of the input matrix.

$n2$ represents the number of “extra” rows to be drawn from the matrix. If the input matrix has r rows and c columns, the output matrix will have $r + n2$ rows and c columns. By default, $n2 = 0$.

$n3$ represents the block size for the resample procedure. If you specify $n3 > 1$, then blocks of consecutive rows of length $n3$ will be drawn with replacement from the first $r - n3 + 1$ rows of the input matrix.

You may provide a name for the vector $v4$ to be used for weighted resampling. The weighting vector must have length r and all elements must be non-missing and non-negative. If you provide a weighting vector, each row of the input matrix will be drawn with probability proportional to the weights in the corresponding row of the weighting vector. (The weights need not sum to 1. EViews will automatically normalize the weights).

```
matrix xb = @bootstrap(x)
```

To draw without replacement from rows of a matrix, use [@permute \(p. 412\)](#).

@right	String Function
--------	---------------------------------

Syntax: `@right(str, n)`

Argument 1: string, str

Argument 2: integer, n

Return: same as source

Returns a string containing n characters from the right end of str . If the source is shorter than n , the entire string is returned. Example:

```
%1 = @right("I doubt it", 8)
```

returns the string “doubt it”.

See also [@left \(p. 407\)](#), [@mid \(p. 409\)](#).

@rowextract[Matrix Utility Function](#)Syntax: `@rowextract(m, n)`Argument 1: matrix or sym, *m*Argument 2: integer, *n*

Return: rowvector

Extracts a rowvector from row *n* of the matrix object *m*. Example:

```
rowvector r1 = @rowextract(m1, 3)
```

See also [@columnextract \(p. 398\)](#).

rowplace[Matrix Utility Command](#)Syntax: `rowplace(m, r, n)`Argument 1: matrix, *m*Argument 2: rowvector, *r*

Argument 3: integer

Places the rowvector *r* into the matrix *m* at row *n*. The number of columns in *m* and *r* must match, and row *n* must exist within *m*. Example:

```
rowplace(m1, r1, 4)
```

See also [colplace \(p. 397\)](#).

@rows[Matrix Utility Function](#)Syntax: `@rows(o)`Argument: matrix, vector, rowvector, sym, series, or group, *o*

Return: scalar

Returns the number of rows in the matrix object, *o*.

Example:

```
scalar scl=@rows(m1)
scalar size=@rows(m1)*@columns(m1)
```

For series and groups [@rows \(p. 414\)](#) returns the number of observations in the workfile range. See also [@columns \(p. 398\)](#).

@solvesystem	Matrix Algebra Function
---------------------	---

Syntax: `@solvesystem(o, v)`

Argument 1: matrix or sym, *o*

Argument 2: vector, *v*

Return: vector

Returns the vector *x* that solves the equation $Mx = p$ where the matrix or sym *M* is given by the argument *o*. Example:

```
vector v2 = @solvesystem(m1, v1)
```

See also [@inverse \(p. 406\)](#).

stom	Matrix Utility Command
-------------	--

Syntax: `stom(o1, o2, smp)`

Argument 1: series or group, *o1*

Argument 2: vector or matrix, *o2*

Argument 3: (optional) sample *smp*

Series-TO-Matrix Object. If *o1* is a series, stom fills the vector *o2* with data from the *o1* using the optional sample object *smp* or the workfile sample. *o2* will be resized accordingly. If any observation has the value “NA”, the observation will be omitted from the vector. Example:

```
stom(ser1, v1)
stom(ser1, v2, smp1)
```

If *o1* is a group, stom fills the matrix *o2* with data from *o1* using the optional sample object *smp* or the workfile sample. *o2* will be resized accordingly. The series in *o1* are placed in the columns of *o2* in the order they appear in the group spreadsheet. If any of the series in the group has the value “NA” for a given observation, the observation will be omitted for all series. Example:

```
stom(grp1, m1)
stom(grp1, m2, smp1)
```

For a conversion method that preserves NAs, see [stomna \(p. 416\)](#).

stomna	Matrix Utility Command
--------	------------------------

- Syntax: `stomna(o1, o2, smp)`
- Argument 1: series or group, *o1*
- Argument 2: vector or matrix, *o2*
- Argument 3: (optional) sample *smp*

Series-TO-Matrix Object with NAs. If *o1* is a series, `stom` fills the vector *o2* with data from *o1* using the optional sample object *smp* or the workfile sample. *o2* will be resized accordingly. All “NA” values in the series will be assigned to the corresponding vector elements.

Example:

```
stom(serial,v1)  
stom(serial,v2,smp1)
```

If *o1* is a group, `stom` fills the matrix *o2* with data from *o1* using the optional sample object *smp* or the workfile sample. *o2* will be resized accordingly. The series in *o1* are placed in the columns of *o2* in the order they appear in the group spreadsheet. All NAs will be assigned to the corresponding matrix elements. Example:

```
stomna(grp1,m1)  
stomna(grp1,m2,smp1)
```

For conversion methods that automatically remove observations with NAs, see [@convert \(p. 399\)](#) and [stom \(p. 415\)](#).

@str	String Function
------	-----------------

- Syntax: `@str(n)`
- Argument: scalar, *n*
- Return: string

Returns a string representing the given number. Example:

```
%1 = @str(15)
```

See also [@val \(p. 419\)](#).

@strlen	String Function
----------------	---------------------------------

Syntax: `@strlen(s)`

Argument: string, *s*

Return: number *n*

Returns the length of a string. Example:

```
!1 = @strlen("Hi mom")
```

!1 will contain the value “6”. See also [@val \(p. 419\)](#).

@subextract	Matrix Utility Function
--------------------	---

Syntax: `@subextract(o, n1, n2, n3, n4)`

Argument 1: vector, rowvector, matrix or sym, *o*

Argument 2: integer, *n1*

Argument 3: integer, *n2*

Argument 4: (optional) integer, *n3*

Argument 5: (optional) integer, *n4*

Return: matrix

Returns a submatrix of a specified matrix, *o*. *n1* is the row and *n2* is the column of the upper left element to be extracted. The optional arguments *n3* and *n4* provide the row and column location of the lower right corner of the matrix. Unless *n3* and *n4* are provided this function returns a matrix containing all of the elements below and to the right of the starting element.

Examples:

```
matrix m2 = @subextract(m1, 5, 9, 6, 11)
```

```
matrix m2 = @subextract(m1, 5, 9)
```

@svd	Matrix Algebra Function
-------------	---

Syntax: `@svd(m1, v1, m2)`

Argument 1: matrix or sym, $m1$

Argument 2: vector, $v1$

Argument 3: matrix or sym, $m2$

Return: matrix

Performs a singular value decomposition of the matrix $m1$. The matrix U is returned by the function, the vector $v1$ will be filled (resized if necessary) with the singular values and the matrix $m2$ will be assigned (resized if necessary) the other matrix, V , of the decomposition. The singular value decomposition satisfies

$$\begin{aligned}m1 &= UWV' \\ U'U &= VV = I\end{aligned}\tag{9.1}$$

where W is a diagonal matrix with the singular values along the diagonal. Singular values close to zero indicate that the matrix may not be of full rank. See the [@rank \(p. 412\)](#) function for a related discussion.

Examples:

```
matrix m2
vector v1
matrix m3 = @svd(m1,v1,m2)
```

@trace	Matrix Algebra Function
---------------	---

Syntax: `@trace(m)`

Argument: matrix or sym, m

Return: scalar

Returns the trace (the sum of the diagonal elements) of a square matrix or sym, m . Example:

```
scalar scl = @trace(m1)
```

@transpose	Matrix Algebra Function
-------------------	-------------------------

Syntax: `@transpose(o)`

Argument: matrix, vector, rowvector, or sym, *o*

Return: matrix, rowvector, vector, or sym

Forms the transpose of a matrix object, *o*. *o* may be a vector, rowvector, matrix, or a sym. The result is a matrix object with a number of rows equal to the number of columns in the original matrix and number of columns equal to the number of rows in the original matrix. This function is an identity function for a sym, since a sym by definition is equal to its transpose. Example:

```
matrix m2 = @transpose(m1)
rowvector r2 = @transpose(v1)
```

@unitvector	Matrix Utility Function
--------------------	-------------------------

Syntax: `@unitvector(n1, n2)`

Argument 1: integer, *n1*

Argument 2: integer, *n2*

Return: vector

Creates an *n1* element vector with a “1” in the *n2*-th element, and “0” elsewhere. Example:

```
vec v1 = @unitvector(8, 5)
```

creates an 8 element vector with a “1” in the fifth element and “0” for the other 7 elements. Note: if you wish to create an *n1* element vector of ones, you should use a declaration statement of the form

```
vector(n1) v1=1
```

@val	String Function
-------------	-----------------

Syntax: `@val(str)`

Argument: string, *str*

Return: scalar

Returns the number that a string *str* represents. The number is terminated by the first non-numeric character. If the string begins with a non-numeric character (that is not a plus or a minus sign), the function returns “NA”. This function is useful for extracting numbers from tables. Example:

```
scalar sc1 = @val("17.4648")
scalar sc2 = @val(tab1(3,4))
scalar sc3 = @val("-234.35")
```

See also [@str \(p. 416\)](#).

@vec	Matrix Utility Function
-------------	---

Syntax: **@vec(*o*)**
Argument: matrix, sym, *o*
Return: vector

Creates a vector from the columns of the given matrix stacked one on top of each other. The vector will have the same number of elements as the source matrix. Example:

```
vector v1 = @vec(m1)
```

@vech	Matrix Utility Function
--------------	---

Syntax: **@vech(*o*)**
Argument: matrix, sym, *o*
Return: vector

Creates a vector from the columns of the lower triangle of the source square matrix *o* stacked on top of each another. The vector has the same number of elements as the source matrix has in its lower triangle. Example:

```
vector v1 = @vech(m1)
```

Chapter 10. Programming Language Reference

The following reference is an alphabetical listing of the program statements and support functions used by the EViews programming language.

For details on the EViews programming language, see [Chapter 6, “EViews Programming”, on page 85](#). For a quick summary of these entries, see [“Programming Summary” on page 114](#).

call	Program Statement
------	-----------------------------------

Call a subroutine within a program.

The call statement is used to call a subroutine within a program.

Cross-references

See [“Calling Subroutines” on page 109](#). See also [subroutine \(p. 430\)](#), [endsub \(p. 422\)](#).

@date	Support Function
-------	----------------------------------

Syntax: @date

Return: string

Returns a string containing the current date in “mm/dd/yy” format.

Examples

```
%y = @date
```

assigns a string of the form “10/10/00”.

Cross-references

See also [@time \(p. 431\)](#).

else	Program Statement
------	-------------------

ELSE clause of IF statement in a program.

Starts a sequence of commands to be executed when the IF condition is false. The `else` keyword must be terminated with an `endif`.

Syntax

```
if [condition] then  
    [commands to be executed if condition is true]  
else  
    [commands to be executed if condition is false]  
endif
```

Cross-references

See “[IF Statements](#)” on page 98. See also, [if](#) (p. 424), [endif](#) (p. 422), [then](#) (p. 431).

endif	Program Statement
-------	-------------------

End of IF statement. Marks the end of an IF, or an IF-ELSE statement.

Syntax

```
if [condition] then  
    [commands if condition true]  
endif
```

Cross-references

See “[IF Statements](#)” on page 98. See also, [if](#) (p. 424), [else](#) (p. 422), [then](#) (p. 431).

endsub	Program Statement
--------	-------------------

Mark the end of a subroutine.

Syntax

```
subroutine name(arguments)  
    commands
```

```
endsub
```

Cross-references

See “[Defining Subroutines](#)” beginning on page 107. See also, [subroutine \(p. 430\)](#), [return \(p. 428\)](#).

@errorcount	Support Function
-----------------------------	----------------------------------

Syntax: [@errorcount](#)

Argument: none

Return: integer

Number of errors encountered. Returns a scalar containing the number of errors encountered during program execution.

@evpath	Support Function
-------------------------	----------------------------------

Syntax: [@evpath](#)

Return: string

Returns a string containing the directory path for the EViews executable.

Examples

If your currently executing copy of EViews is installed in “D:\\EVIEWS”, then

```
%y = @evpath
```

assigns a string of the form “D:\\EVIEWS”.

Cross-references

See also [cd](#), [chdir \(p. 156\)](#) and [@temppath \(p. 430\)](#).

exitloop	Program Statement
--------------------------	-----------------------------------

Exit from current loop in programs.

`exitloop` causes the program to break out of the current FOR or WHILE loop.

Syntax

Command: `exitloop`

Examples

```
for !i=1 to 107  
    if !i>6 then exitloop  
next
```

Cross-references

See “[The FOR Loop](#)” on page 100. See also, [stop](#) (p. 430), [return](#) (p. 428), [for](#) (p. 424), [next](#) (p. 425), [step](#) (p. 429).

for	Program Statement
------------	-----------------------------------

FOR loop in a program.

The **for** statement is the beginning of a FOR...NEXT loop in a program.

Syntax

```
for counter = start to end [step stepsize]  
    commands  
next
```

Cross-references

See “[The FOR Loop](#)” on page 100. See also, [exitloop](#) (p. 423), [next](#) (p. 425), [step](#) (p. 429).

if	Program Statement
-----------	-----------------------------------

IF statement in a program.

The **if** statement marks the beginning of a condition and commands to be executed if the statement is true. The statement must be terminated with the beginning of an ELSE clause, or an **endif**.

Syntax

```
if [condition] then  
    [commands if condition true]  
endif
```

Cross-references

See “[IF Statements](#)” on page 98. See also [else \(p. 422\)](#), [endif \(p. 422\)](#), [then \(p. 431\)](#).

include	Program Statement
---------	-----------------------------------

Include another file in a program.

The `include` statement is used to include the contents of another file in a program file.

Syntax

`include filename`

Cross-references

See “[Multiple Program Files](#)” on page 106. See also [call \(p. 421\)](#).

@isobject	Support Function
-----------	----------------------------------

Syntax: `@isobject(str)`

Argument: `string, str`

Return: `integer`

Check for an object’s existence. Returns a “1” if the object exists in the current workfile, and a “0” if it does not exist.

next	Program Statement
------	-----------------------------------

End of FOR loop. `next` marks the end of a FOR loop in a program.

Syntax

`for [conditions of the FOR loop]`

`[commands]`

`next`

Cross-references

See “[The FOR Loop](#)” beginning on page 100. See also, [exitloop \(p. 423\)](#), [for \(p. 424\)](#), [step \(p. 429\)](#).

@obsrange**Support Function**Syntax: **@obsrange**

Return: number

returns number of observations in the current active workfile range (0 if no workfile in memory)

Examples

```
!z = @obsrange
```

will place the number of observations in the workfile range in the replacement variable !Z“.

Cross-references

See also [@obssmpl \(p. 426\)](#).

@obssmpl**Support Function**Syntax: **@obssmpl**

Return: number

returns number of observations in the current active workfile sample (0 if no workfile in memory)

Examples

```
!z = @obssmpl
```

will place the number of observations in the sample in the replacement variable !Z“.

Cross-references

See also [@obsrange \(p. 426\)](#).

open**Command**

Open a file. Opens a workfile, database, program file, or ASCII text file.

See [open \(p. 275\)](#).

output	Command
---------------	-------------------------

Redirects printer output or display estimation output.

See [output \(p. 279\)](#).

poff	Program Statement
-------------	-----------------------------------

Turn off automatic printing in programs.

`poff` turns off automatic printing of all output. In programs, `poff` is used in conjunction with `pon` to control automatic printing; these commands have no effect in interactive use.

Syntax

Command: **poff**

Cross-references

See “[Print Setup](#)” on page 651 for a discussion of printer control.

See also [pon \(p. 427\)](#).

pon	Program Statement
------------	-----------------------------------

Turn on automatic printing in programs.

`pon` instructs EViews to send all statistical and data display output to the printer (or the redirected printer destination; see [output \(p. 279\)](#)). It is equivalent to including the “p” option in all commands that generate output. `pon` and `poff` only work in programs; they have no effect in interactive use.

Syntax

Command: **pon**

Cross-references

See “[Print Setup](#)” on page 651 of the *User’s Guide* for a discussion of printer control.

See also [poff \(p. 427\)](#).

program	Command
---------	---------

Create a program.

See [program \(p. 287\)](#).

return	Program Statement
--------	-------------------

Exit subroutine.

The `return` statement forces an exit from a subroutine within a program. A common use of `return` is to exit from the subroutine if an unanticipated error has occurred.

Syntax

```
if [condition] then
    return
endif
```

Cross-references

See “[Subroutines](#)” beginning on page 107. See also [exitloop \(p. 423\)](#), [stop \(p. 430\)](#).

run	Command
-----	---------

Run a program. The `run` command executes a program. The program may be located in memory or stored in a program file on disk.

See [run \(p. 305\)](#).

statusline	Program Statement
------------	-------------------

Send a text message to the EViews statusline.

Syntax

```
for !i = 1 to 10
    statusline Iteration !
next
```

step	Program Statement
------	-----------------------------------

Step size of a FOR loop.

Syntax

```
for !i=a to b step n
    [commands]
next
```

`step` may be used in a FOR loop to specify the size of the step in the looping variable. If no step is provided, it assumes a step of “+1”.

If a given step exceeds the end value *b* in the FOR loop specification, the contents of the loop will not be executed.

Examples

```
for !j=5 to 1 step -1
    series x = nrnd*!j
next
```

repeatedly executes the commands in the loop with the control variable !J set to “5”, “4”, “3”, “2”, “1”.

```
for !j=0 to 10 step 3
    series z = z/!j
next
```

Loops the commands with the control variable !J set to “0”, “3”, “6”, and “9”.

You should take care when using non-integer values for the stepsize since round-off error may yield unanticipated results. For example:

```
for !j=0 to 1 step .01
    series w = !j
next
```

may stop before executing the loop for the value !J = 1 due to round-off error.

Cross-references

See “[The FOR Loop](#)” beginning on page 100. See also [exitloop](#) (p. 423), [for](#) (p. 424), [next](#) (p. 425).

stop	Program Statement
------	-------------------

Break out of program.

The `stop` command halts execution of a program. It has the same effect as hitting the F1 (break) key.

Syntax

Command: stop

Cross-references

See also, [exitloop \(p. 423\)](#), [return \(p. 428\)](#).

subroutine	Program Statement
------------	-------------------

Declare a subroutine within a program.

The `subroutine` statement marks the start of a subroutine.

Syntax

subroutine name(*arguments*)

[*commands*]

endsub

Cross-references

See “[Subroutines](#)” beginning on page 107. See also [endsub \(p. 422\)](#).

@temppath	Support Function
-----------	------------------

Syntax: @temppath

Return: string

Returns a string containing the directory path for the EViews temporary files as specified in the global options **File Locations.....** menu.

Examples

If your currently executing copy of EViews puts temporary files in “D:\EVIEWS”, then

```
%y = @temppath
```

assigns a string of the form “D:\EVIEWS”.

Cross-references

See also [cd](#), [chdir](#) (p. 156) and [@evpath](#) (p. 423).

then	Program Statement
-------------	-----------------------------------

Part of IF statement.

then marks the beginning of commands to be executed if the condition given in the IF statement is satisfied.

Syntax

```
if [condition] then
    [commands if condition true]
endif
```

Cross-references

See “[IF Statements](#)” on page 98. See also, [else](#) (p. 422), [endif](#) (p. 422), [if](#) (p. 424).

@time	Support Function
--------------	----------------------------------

Syntax: **@time**

Return: string

Returns a string containing the current time in “hh:mm” format.

Examples

```
%y = @time
```

assigns a string of the form “15:35”.

Cross-references

See also [@date](#) (p. 421).

to	Expression Program Statement
----	---------------------------------

Upper limit of for loop OR lag range specifier.

to is required in the specification of a FOR loop to specify the upper limit of the control variable; see “[The FOR Loop](#)” on page 100.

When used as a lag specifier, to may be used to specify a range of lags to be used in estimation.

Syntax

Used in a FOR loop:

```
for !i=n to m  
    [commands]  
next
```

Used as a Lag specifier:

```
series_name(n to m)
```

Examples

```
ls cs c gdp(0 to -12)
```

Runs an OLS regression of CS on a constant, and the variables GDP, GDP(-1), GDP(-2), ..., GDP(-11), GDP(-12).

Cross-references

See “[The FOR Loop](#)” beginning on page 100. See also, [exitloop](#) (p. 423), [for](#) (p. 424), [next](#) (p. 425).

@toc	Support Function
------	------------------

Syntax: [@toc](#)

Return: integer

Compute elapsed time (since timer reset) in seconds.

Examples

```
tic  
[some commands]  
!elapsed = @toc
```

resets the timer, executes commands, and saves the elapsed time in the control variable !ELAPSED.

Cross-references

See also [tic \(p. 363\)](#) and [toc \(p. 364\)](#).

wend	Program Statement
-------------	-----------------------------------

End of WHILE clause.

wend marks the end of a set of program commands that are executed under the control of a WHILE statement.

Syntax

```
while [condition]
    [commands while condition true]
wend
```

Cross-references

See “[The WHILE Loop](#)” on page 104. See also [while \(p. 433\)](#).

while	Program Statement
--------------	-----------------------------------

Conditional control statement. The while statement marks the beginning of a WHILE loop.

The commands between the while keyword and the wend keyword will be executed repeatedly until the condition in the while statement is false.

Syntax

```
while [condition]
    [commands while condition true]
wend
```

Cross-references

See “[The WHILE Loop](#)” on page 104. See also [wend \(p. 433\)](#).

Appendix A. Operator and Function Reference

The reference material in this section describes the operators and functions that may be used with series and (in some cases) matrix objects. A general description of the use of these operators and functions may be found in [Chapter 5, “Working with Data”, beginning on page 87](#) of the *User’s Guide*.

This material is divided into several topics:

- [Operators](#).
- [Date and observation functions](#).
- [Basic mathematical functions](#).
- [Time series functions](#).
- [Descriptive statistics](#).
- [Additional and special functions](#).
- [Trigonometric functions](#).
- [Statistical distribution functions](#).

For a list of functions specific to matrices, see [“Matrix Function and Command Summary” on page 76](#).

Operators

All of the operators described below may be used in expressions involving series and scalar values. When applied to a series expression, the operation is performed for each observation in the current sample. The precedence of evaluation is listed in [“Operators” on page 87](#) of the *User’s Guide*. Note that you can enforce order-of-evaluation using parentheses.

Expression	Operator	Description
+	add	$x+y$ adds the contents of X and Y.
-	subtract	$x-y$ subtracts the contents of Y from X.
*	multiply	$x*y$ multiplies the contents of X by Y.
/	divide	x/y divides the contents of X by Y.
\wedge	raise to the power	x^y raises X to the power of Y.

>	greater than	$x>y$ takes the value 1 if X exceeds Y, and 0 otherwise.
<	less than	$x<y$ takes the value 1 if Y exceeds X, and 0 otherwise.
=	equal to	$x=y$ takes the value 1 if X and Y are equal, and 0 otherwise.
<>	not equal to	$x<>y$ takes the value 1 if X and Y are not equal, and 0 if they are equal.
<=	less than or equal to	$x<=y$ takes the value 1 if X does not exceed Y, and 0 otherwise.
>=	greater than or equal to	$x>=y$ takes the value 1 if Y does not exceed X, and 0 otherwise.
and	logical and	x and y takes the value 1 if both X and Y are nonzero, and 0 otherwise.
or	logical or	x or y takes the value 1 if either X or Y is non-zero, and 0 otherwise.

Date and Observation Functions

These functions allow you to identify the period associated with a given observation, or the value associated with a given date/observation.

Function	Name	Description
@day	observation day	for daily or weekly workfiles, returns the observation day in the month for each observation.
@elem(x, "d") , @elem(x, s)	element	returns the value of the series x , at observation or date, d , or string s .
@month	observation month	returns the month of observation (for monthly, daily, and weekly data) for each observation.
@quarter	observation quarter	returns the quarter of observation (except for annual, semi-annual, and undated data) for each observation.
@year	observation year	returns the year associated with each observation (except for undated data) for each observation.

For example, if you create the series

```
series y = @month
```

with a monthly workfile, Y will contain a numeric indicator for each month (1 through 12). If you create the series

```
series z = @quarter
```

for the same workfile, EViews will fill Z with the numeric quarter indicator (1 through 4) associated with each observation.

You may only use these functions in a workfile of at least as high frequency. Thus, while you can use the @month function in a daily or annual workfile, you cannot use the @day function in a monthly or annual workfile, nor can you use the @month function in an annual workfile.

Basic Mathematical Functions

These functions perform basic mathematical operations. When applied to a series, they return a value for every observation in the current sample. When applied to a matrix object, they return a value for every element of the matrix object. The functions will return NA values for observations where the input values are NAs, or where the input values are not valid. For example, the square-root function @sqrt, will return NAs for all observations less than zero.

Name	Function	Examples/Description
@abs(x), abs(x)	absolute value	@abs(-3) = 3 .
@ceiling(x)	smallest integer not less than	@ceiling(2.34) = 3 ; @ceiling(4) = 4 .
@exp(x), exp(x)	exponential, e^x	@exp(1) ≈ 2.71813 .
@fact(x)	factorial, $x!$	@fact(3) = 6 ; @fact(0) = 1 .
@factlog(x)	natural logarithm of the factorial, $\log_e(x!)$	@factlog(3) ≈ 1.7918 ; @factlog(0) = 0 .
@floor(x)	largest integer not greater than	@floor(1.23) = 1 ; @floor(-3.1) = -4 .
@inv(x)	reciprocal, $1/x$	@inv = 0.5 .
@mod(x, y)	floating point remainder	returns the remainder of x/y with the same sign as x . If $y = 0$ the result is 0.
@log(x), log(x)	natural logarithm, $\log_e(x)$	@log(2) ≈ 0.693 ; log(2.71813) ≈ 1 .

<code>@log10(x)</code>	base-10 logarithm, $\log_{10}(x)$	$\text{@log10}(100) = 2$
<code>@logx(x, b)</code>	base- b logarithm, $\log_b(x)$	$\text{@log}(256, 2) = 8$
<code>@nan(x, y)</code>	recode NAs in X to Y	returns x if $x < > \text{NA}$, and y if $x = \text{NA}$.
<code>@recode(s, x, y)</code>	recode by condition	returns x if condition s is true; otherwise returns y : $\text{@recode}(y > 0, x, 2)$.
<code>@round(x)</code>	round to the nearest integer	$\text{@round}(-97.5) = -98$; $\text{@round}(3.5) = 4$.
<code>@sqrt(x), sqr(x)</code>	square root	$\text{@sqrt}(9) = 3$.

Time Series Functions

The following functions facilitate working with time series data. Note that NAs will be returned for observations for which lagged values are not available. For example, `d(x)` returns a missing value for the first observation in the workfile, since the lagged value is not available.

Name	Function	Description
<code>d(x)</code>	first difference	$(1 - L)X = X - X(-1)$ where L is the lag operator.
<code>d(x, n)</code>	n -th order difference	$(1 - L)^n X$.
<code>d(x, n, s)</code>	n -th order difference with a seasonal difference at s	$(1 - L)^n(1 - L^s)X$.
<code>dlog(x)</code>	first difference of the logarithm	$(1 - L)\log(X)$ $= \log(X) - \log(X(-1))$
<code>dlog(x, n)</code>	n -th order difference of the logarithm	$(1 - L)^n \log(X)$.
<code>dlog(x, n, s)</code>	n -th order difference of the logarithm with a seasonal difference at s	$(1 - L)^n(1 - L^s)\log(X)$.
<code>@movav(x, n)</code>	n -period backward moving average	$\text{@movav}(x, 3)$ $= (X + X(-1) + X(-2))/3$

<code>@movsum(x, n)</code>	n -period backward moving sum	$\begin{aligned} @movsum(x, 3) \\ = (X + X(-1) + X(-2)) \end{aligned}$
<code>@pc(x)</code>	one-period percentage change (in percent)	equals $\text{@pch}(x) * 100$
<code>@pch(x)</code>	one-period percentage change (in decimal)	$(X - X(-1)) / X(-1)$
<code>@pca(x)</code>	one-period percentage change—annualized (in percent)	equals $\text{@pcha}(x) * 100$
<code>@pcha(x)</code>	one-period percentage change—annualized (in decimal)	$\begin{aligned} @pcha(x) \\ = (1 + \text{@pch}(x))^n - 1 \end{aligned}$
		where n is the lag associated with one-year ($n = 4$) for quarterly data, etc.).
<code>@pcy(x)</code>	one-year percentage change (in percent)	equals $\text{@pchy}(x) * 100$
<code>@pchy(x)</code>	one-year percentage change (in decimal)	$(X - X(-n)) / X(-n)$, where n is the lag associated with one-year ($n = 12$) for annual data, etc.).
<code>@seas(n)</code>	seasonal dummy	returns 1 when the quarter or month equals n and 0 otherwise.
<code>@trend,</code> <code>@trend(n)</code>	time trend	returns a trend series, normalized to 0 in period n , where n is a date or observation number; if n is omitted, then the series is normalized at the first observation in the workfile.

Descriptive Statistics

These functions compute descriptive statistics for a specified sample, excluding missing values if necessary. The default sample is the current workfile sample. If you are performing these computations on a series and placing the results into a series, you can specify a sample as the last argument of the descriptive statistic function, either as a string (in double quotes) or using the name of a sample object. For example,

```
series z = @mean(x, "1945:01 1979:12")
```

or

```
w = @var(y, s2)
```

where S2 is the name of a sample object and W and X are series. Note that you may not use a sample argument if the results are assigned into a matrix, vector, or scalar object. For example, the following assignment

```
vector(2) a  
series x  
a(1) = @mean(x, "1945:01 1979:12")
```

is not valid since the target A(1) is a vector element. To perform this latter computation you must explicitly set the global sample prior to performing the calculation performing the assignment:

```
smp1 1945:01 1979:12  
a(1) = @mean(x)
```

To determine the number of observations available for a given series, use the @obs function. Note that where appropriate, EViews will perform casewise exclusion of data with missing values. For example, @cov(x,y) and @cor(x,y) will use only observations for which data on both X and Y are valid.

In the following table, arguments in square brackets [] are optional arguments:

- [s]: sample expression in double quotes or name of a sample object. *The optional sample argument may only be used if the result is assigned to a series.* For @quantile, you must provide the method option argument in order to include the optional sample argument.

Function	Name	Description
@cor(x,y[,s])	correlation	the correlation between X and Y.
@cov(x,y[,s])	covariance	the covariance between X and Y.
@inner(x,y[,s])	inner product	the inner product of X and Y.
@obs(x[,s])	number of observations	the number of non-missing observations for X in the current sample.
@mean(x[,s])	mean	average of the values in X.

<code>@median(x[, s])</code>	median	computes the median of the X (uses the average of middle two observations if the number of observations is even).
<code>@min(x[, s])</code>	minimum	minimum of the values in X.
<code>@max(x[, s])</code>	maximum	maximum of the values in X.
<code>@quantile(x, q[, m, s])</code>	quantile	the q -th quantile of the series X. m is an optional integer argument for specifying the quantile method: 1 (rankit - default), 2 (ordinary), 3 (van der Waerden), 4 (Blom), 5 (Tukey).
<code>@stdev(x[, s])</code>	standard deviation	square root of the unbiased sample variance (sum-of-squared residuals divided by $n - 1$).
<code>@sum(x[, s])</code>	sum	the sum of X.
<code>@sumsq(x[, s])</code>	sum-of-squares	sum of the squares of X.
<code>@var(x[, s])</code>	variance	variance of the values in X (division by n).

Additional and Special Functions

EViews provides a number of utility and “special” functions used in evaluating the properties of various statistical distributions or for returning special mathematical values such as Euler’s constant. For further details on special functions, see the extensive discussions in Temme (1996), Abramowitz and Stegun (1964), and Press, *et al.* (1992).

Function	Description
<code>@beta(a, b)</code>	beta integral (Euler integral of the second kind) $B(a, b) = \int_0^1 t^{a-1} (1-t)^{b-1} dt = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$ for $a, b > 0$.
<code>@betainc(x, a, b)</code>	incomplete beta integral $\frac{1}{B(a, b)} \int_0^x t^{a-1} (1-t)^{b-1} dt$ for $0 \leq x \leq 1$ and $a, b > 0$.

<code>@betaincder(x, a, b, s)</code>	derivative of the incomplete beta integral: Evaluates the derivatives of the incomplete beta integral $B(x, a, b)$, where s is an integer from 1 to 9 corresponding to the desired derivative:
	$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} \frac{\partial B}{\partial x} & \frac{\partial B}{\partial a} & \frac{\partial B}{\partial b} \\ \frac{\partial^2 B}{\partial x^2} & \frac{\partial^2 B}{\partial x \partial a} & \frac{\partial^2 B}{\partial x \partial b} \\ \frac{\partial^2 B}{\partial a^2} & \frac{\partial^2 B}{\partial a \partial b} & \frac{\partial^2 B}{\partial b^2} \end{bmatrix}$
<code>@betaincinv(p, a, b)</code>	inverse of the incomplete beta integral: returns an x satisfying: $p = \frac{1}{B(a, b)} \int_0^x t^{a-1} (1-t)^{b-1} dt$ for $0 \leq p \leq 1$ and $a, b > 0$.
<code>@betalog(a, b)</code>	natural logarithm of the beta integral: $\log B(a, b) = \log \Gamma(a) + \log \Gamma(b) - \log \Gamma(a+b)$
<code>@binom(n, x)</code>	binomial coefficient $\binom{n}{x} = \frac{n!}{x!(n-x)!}$ for n and x positive integers, $0 \leq x \leq n$.
<code>@binomlog(n, x)</code>	natural logarithm of the binomial coefficient: $\log(n!) - \log(x!) - \log((n-x)!)$
<code>@cloglog(x)</code>	complementary log-log function: $\log(-\log(1-x))$
See also <code>@qextreme</code> .	
<code>@digamma(x), @psi(x)</code>	first derivative of the log gamma function: $(x) = \frac{d \log \Gamma(x)}{dx} = \frac{1}{\Gamma(x)} \frac{d \Gamma(x)}{dx}$

<code>@erf(x)</code>	error function: $\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$ for $x \geq 0$.
<code>@erfc(x)</code>	complementary error function: $\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt = 1 - \text{erf}(x).$ for $x \geq 0$.
<code>@gamma(x)</code>	(complete) gamma function: $\Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt$ for $x \geq 0$.
<code>@gammader(x)</code>	first derivative of the gamma function: $\Gamma'(x) = d\Gamma(x)/(dx)$ Note: Euler's constant, $\gamma \approx 0.5772$, may be evaluated as $\gamma = -@gammader(1)$. See also <code>@digamma</code> and <code>@trigamma</code> .
<code>@gammainc(x, a)</code>	incomplete gamma function: $G(x, a) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$ for $x \geq 0$ and $a > 0$.
<code>@gammaincder(x, a, n)</code>	derivative of the incomplete gamma function: Evaluates the derivatives of the incomplete gamma integral $G(x, a)$, where n is an integer from 1 to 5 corresponding to the desired derivative: $\begin{bmatrix} 1 & 2 & - \\ 3 & 4 & 5 \end{bmatrix} = \begin{bmatrix} \frac{\partial G}{\partial x} & \frac{\partial G}{\partial a} & - \\ \frac{\partial^2 G}{\partial x^2} & \frac{\partial^2 G}{\partial x \partial a} & \frac{\partial^2 G}{\partial a^2} \end{bmatrix}$

@gammaincinv(p, a)	inverse of the incomplete gamma function: find the value of x satisfying
	$p = G(x, a) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$ for $0 \leq p < 1$ and $a > 0$.
@gammalog(x)	logarithm of the gamma function: $\log \Gamma(x)$. For derivatives of this function see @digamma and @trigamma.
@logit(x)	logistic transform:
	$\frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x}$
@psi(x)	see @digamma.
@trigamma(x)	second derivative of the log gamma function:
	$\psi'(x) = \frac{d^2 \log \Gamma(x)}{dx^2}$

Trigonometric Functions

When applied to a series, all of the trigonometric functions operate on every observation in the current sample and return a value for every observation. Where relevant, the input and results should/will be expressed in radians. All results are real valued—complex values will return NAs.

Function	Name	Examples/Description
@acos(x)	arc cosine (real results in radians)	$\@acos(-1) = \pi$
@asin(x)	arc sine (real results in radians)	$\@asin(-1) = -\pi/2$
@atan(x)	arc tangent (results in radians)	$\@atan(1) = \pi/4$
@cos(x)	cosine (argument in radians)	$\@cos(3.14159) \approx -1$
@sin(x)	sine (argument in radians)	$\@sin(3.14159) \approx 0$
@tan(x)	tangent (argument in radians)	$\@tan(1) \approx 1.5574$

Statistical Distribution Functions

The following functions provide access to the density or probability functions, cumulative distribution, quantile functions, and random number generators for a number of standard statistical distributions.

There are four functions associated with each distribution. The first character of each function name identifies the type of function:

Function Type	Beginning of Name
Cumulative distribution (CDF)	@c
Density or probability	@d
Quantile (inverse CDF)	@q
Random number generator	@r

The remainder of the function name identifies the distribution. For example, the functions for the beta distribution are @cbeta, @dbeta, @qbeta and @rbeta.

When used with series arguments, EViews will evaluate the function for each observation in the current sample. As with other functions, NA or invalid inputs will yield NA values. For values outside of the support, the functions will return zero.

Note that the CDFs are assumed to be right-continuous: $F_X(k) = \Pr(X \leq k)$. The quantile functions will return the smallest value where the CDF evaluated at the value equals or exceeds the probability of interest: $q_X(p) = q^*$, where $F_X(q^*) \geq p$. The inequalities are only relevant for discrete distributions.

The information provided below should be sufficient to identify the meaning of the parameters for each distribution. For further details, see the *Command and Programming Reference*.

Distribution	Functions	Density/Probability Function
Beta	@cbeta(x, a, b), @dbeta(x, a, b), @qbeta(p, a, b), @rbeta(a, b)	$f(x, a, b) = \frac{x^{a-1}(1-x)^{b-1}}{B(a, b)}$ for $0 \leq p \leq 1$ and for $a, b > 0$, where B is the @beta function.
Binomial	@cbinom(x, n, p), @dbinom(x, n, p), @qbinom(s, n, p), @rbinom(n, p)	$\Pr(x, n, p) = \binom{n}{x} p^x (1-p)^{n-x}$ if $x = 0, 1, \dots, n, \dots$, and 0 otherwise, for $0 \leq p \leq 1$.

Chi-square	<code>@cchisq(x,v), @dchisq(x,v), @qchisq(p,v), @rchisq(v)</code>	$f(x, v) = \frac{1}{2^{v/2} \Gamma(v/2)} x^{v/2-1} e^{-x/2}$ where $x \geq 0$, and $v > 0$. Note that the degrees-of-freedom parameter v need not be an integer.
Exponential	<code>@cexp(x,m), @dexp(x,m), @qexp(p,m), @rexp(m)</code>	$f(x, m) = \frac{1}{m} e^{-x/m}$ for $x \geq 0$, and $m > 0$.
Extreme Value (Type I-minimum)	<code>@cextreme(x), @dextreme(x), @qextreme(p), @cloglog(p), @rextreme</code>	$f(x) = \exp(x - e^x)$ for $-\infty < x < \infty$.
F-distribution	<code>@cfdist(x,v1,v2), @dfdist(x,v1,v2), @qfdist(p,v1,v2), @rfdist(v1,v1)</code>	$f(x, v_1, v_2) = \frac{v_1^{v_1/2} v_2^{v_2/2}}{B(v_1/2, v_2/2)} x^{(v_1-2)/2} (v_2 + v_1 x)^{-(v_1+v_2)/2}$ where $x \geq 0$, and $v_1, v_2 > 0$. Note that the functions allow for fractional degrees-of-freedom parameters v_1 and v_2 .
Gamma	<code>@cgamma(x,b,r), @dgamma(x,b,r), @qgamma(p,b,r), @rgamma(b,r)</code>	$f(x, b, r) = b^{-r} x^{r-1} e^{-x/b} / \Gamma(r)$ where $x \geq 0$, and $b, r > 0$.
Generalized Error	<code>@cged(x,r), @dged(x,r), @qged(p,r), @rged(r)</code>	$f(x, r) = \frac{r \Gamma\left(\frac{3}{r}\right)^{1/2}}{2r \Gamma\left(\frac{1}{r}\right)^{3/2}} \exp - x ^r \left(\frac{\Gamma(3/r)}{\Gamma(1/r)}\right)^{1/2}$ where $-\infty < x < \infty$, and $r > 0$.
Laplace	<code>@claplace(x), @dlaplace(x), @qlaplace(x), @rlaplace</code>	$f(x) = \frac{1}{2} e^{- x }$ for $-\infty < x < \infty$.

Logistic	<code>@clogistic(x), @dlogistic(x), @qlogistic(p), @rlogistic</code>	$f(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x}$ for $-\infty < x < \infty$.
Log-normal	<code>@clognorm(x,m,s), @dlognorm(x,m,s), @qlognorm(p,m,s), @rlognorm(m,s)</code>	$f(x, m, s) = \frac{1}{x \sqrt{2\pi s^2}} e^{-(\log x - m)^2 / (2s^2)}$ $x > 0, -\infty < m < \infty, \text{ and } s > 0.$
Negative Binomial	<code>@cnegbin(x,n,p), @dnegbin(x,n,p), @qnegbin(s,n,p), @rnegbin(n,p)</code>	$\Pr(x, n, p) = \frac{\Gamma(x+n)}{\Gamma(x+1)\Gamma(n)} p^n (1-p)^x$ if $x = 0, 1, \dots, n, \dots$, and 0 otherwise, for $0 \leq x \leq 1.$
Normal (Gaussian)	<code>@cnorm(x), @dnorm(x), @qnorm(p), @rnorm, rnrn</code>	$f(x) = (2\pi)^{-1/2} e^{-x^2/2}$ for $-\infty < x < \infty.$
Poisson	<code>@cpoisson(x,m), @dpoisson(x,m), @qpoisson(p,m), @rpoisson(m)</code>	$\Pr(x, m) = m^x e^{-m} / x!$ if $x = 0, 1, \dots, n, \dots$, and 0 otherwise, for $m > 0.$
Pareto	<code>@cpareto(x,a,k), @dpareto(x,a,k), @qpareto(p,a,k), @rpareto(a,k)</code>	$f(x, a, k) = (ak^a) / x^{a+1}$ for $a > 0$, and $0 \leq k \leq x.$

Student's t-distribution	<code>@ctdist(x,v),</code> <code>@dtdist(x,v),</code> <code>@qtdist(p,v),</code> <code>@rtdist(v)</code>	$f(x, v) = \frac{I\left(\frac{v+1}{2}\right)}{(v\pi)^{1/2} I\left(\frac{v}{2}\right)} \left(1 + \left(\frac{x^2}{v}\right)\right)^{\frac{-(v+1)}{2}}$
for $-\infty < x < \infty$, and $v > 0$. Note that $v = 1$, yields the Cauchy distribution.		
Uniform	<code>@cunif(x,a,b),</code> <code>@dunif(x,a,b),</code> <code>@qunif(p,a,b),</code> <code>@runif(a,b), rnd</code>	$f(x) = \frac{1}{b-a}$
Weibull	<code>@cweib(x,m,a),</code> <code>@dweib(x,m,a),</code> <code>@qweib(p,m,a),</code> <code>@rweib(m,a)</code>	$f(x, m, a) = am^{-a} x^{a-1} e^{-(x/m)^a}$
where $-\infty < x < \infty$, and $m, a > 0$.		

Additional Distribution Related Functions

The following utility functions were designed to facilitate the computation of *p*-values for common statistical tests. While these results may be derived using the distributional functions above, they are retained for convenience and backward compatibility.

Function	Distribution	Description
<code>@chisq(x,v)</code>	Chi-square	Returns the probability that a Chi-squared statistic with v degrees of freedom exceeds x : $@chisq(x,v) = 1 - @cchisq(x,d)$
<code>@fdist(x,v1,v2)</code>	<i>F</i> -distribution	Probability that an <i>F</i> -statistic with v_1 numerator degrees of freedom and v_2 denominator degrees of freedom exceeds x : $@fdist(x,v1,v2) = 1 - @cfdist(x,v1,v2)$
<code>@tdist(x,v)</code>	<i>t</i> -distribution	Probability that a <i>t</i> -statistic with v degrees of freedom exceeds x in absolute value (two-sided <i>p</i> -value): $@tdist(x,v) = 2 * (1 - @ctdist(x,v))$

Index

Symbols

- ! (exclamation) control variable [90](#)
- % (percent sign)
 - program arguments [96](#)
 - string variable [90](#)
- * (asterisk) multiplication [67](#)
- + (plus)
 - addition [67](#)
 - string concatenation [92](#)
- / (slash) division [68](#)
- @abs [437](#)
- @beta [441](#)
- @betainc [441](#)
- @betaincder [442](#)
- @betaincinv [442](#)
- @betalog [442](#)
- @binom [442](#)
- @binomlog [442](#)
- @ceiling [437](#)
- @cholesky [397](#)
- @cloglog [442](#)
- @columnextract [398](#)
- @columns [398](#)
- @cond [398](#)
- @convert [399](#)
- @cor [400, 440](#)
- @cov [400, 440](#)
- @date [421](#)
- @day [436](#)
- @det [401](#)
- @digamma [442](#)
- @dtoc [401](#)
- @eigenvalues [402](#)
- @eigenvectors [402](#)
- @elem [436](#)
- @erf [443](#)
- @erfc [443](#)
- @errorcount [423](#)
- @evpath [423](#)
- @exp [437](#)
- @explode [402](#)
- @fact [437](#)
- @factlog [437](#)
- @filledmatrix [403](#)
- @filledrowvector [403](#)
- @filledsym [403](#)
- @filledvector [404](#)
- @floor [437](#)
- @-functions
 - date functions [436](#)
 - descriptive statistics [439](#)
 - integrals and other special functions [441](#)
 - mathematical functions [437](#)
 - special p-value functions [448](#)
 - statistical distribution functions [444](#)
 - string manipulation [92](#)
 - time series functions [438](#)
 - trigonometric functions [444](#)
- @gamma [443](#)
- @gammader [443](#)
- @gammainc [443](#)
- @gammaincder [443](#)
- @gammaincinv [444](#)
- @ gammalog [444](#)
- @getmaindiagonal [404](#)
- @identity [404](#)
- @implode [405](#)
- @inner [405, 440](#)
- @inv [437](#)
- @inverse [406](#)
- @isobject [425](#)
- @issingular [406](#)
- @kronecker [407](#)
- @left [407](#)
- @log [437](#)
- @log10 [438](#)
- @logit [444](#)
- @logx [438](#)
- @makediagonal [408](#)
- @max [441](#)
- @median [441](#)

@mid 409
@min 441
@mod 437
@month 436
@movav 438
@movsum 439
@nan 438
@norm 410
@obs 440
@obssrange 426
@obssmpl 426
@otod 411
@outer 411
@permute 412
@psi 442
@quantile 441
@quarter 436
@rank 412
@recode 438
@resample 412
@round 438
@rowextract 414
@rows 414
@seas 439
@solvesystem 415
@sqrt 438
@stdev 441
@str 416
@strlen 417
@subextract 417
@sum 441
@sumsq 441
@svd 418
@temppath 430
@time 431
@toc 432
@trace 418
@transpose 419
@trend 439
@trigamma 444
@unitvector 419
@val 419
@var 441
@vec 420
@vech 420
@year 436
_ (continuation character) 86
- (dash)
negation 66
subtraction 67

Numerics

2sls (two-stage least squares) 368
3sls (three-stage least squares) 136

A

abs 437
add 137
Add factor
 initialization 139
addassign 138
addinit 139
Addition operator (+) 67
addtext 140
align 142
Align multiple graphs 142
And operator 436
Anderson-Darling test 198
Andrews test 360
append 143
AR
 autoregressive error 144
 inverse roots of polynomial 148
 seasonal 307
ar 144
Arc cosine 444
Arc sine 444
Arc tangent 444
ARCH
 residual LM test for ARCH 147
 see also GARCH. 145
arch 145
archtest 147
Arguments
 in programs 96
 in subroutines 108
arlm 148
arroots 148
Assign values to matrix objects 56

-
- by element 56
 converting series or group 62
 copy 59
 copy submatrix 61
 fill procedure 57
 Augmented Dickey-Fuller test 373
 auto 149
 Autocorrelation
 compute and display 172
 multivariate VAR residual test 289
 Autoregressive error. See AR.
 Autoregressive conditional heteroskedasticity. See
 ARCH and GARCH.
 Autowrap text 85
 Auxiliary commands 10
 summary 17
 Axis
 rename label 271
 scale 309
- B**
- bar 150
 Bar graph 150
 Batch mode
 See Program.
 BDS test 152
 bdstest 152
 Beta
 distribution 445
 integral 441
 Beta integral
 logarithm 442
 binary 152
 Binary dependent variables 152
 Binary models
 prediction table 285
 Binomial
 coefficient function 442
 distribution 445
 Binomial coefficients 442
 logarithm 442
 block 154
 Bootstrap rows of matrix 412
 Breusch-Godfrey test
 See also Serial correlation.
- C**
- call 421
 Call subroutine 109
 Causality test 154
 cause 154
 ccopy 156
 cd 156
 cdfplot 157
 censored 158
 Censored models 158
 cfetch 160
 Change default directory 156
 chdir 156
 checkderivs 160
 Chi-square distribution 446
 Cholesky factor
 function to compute 397
 chow 161
 Chow test 161
 xlabel 162
 cleartext 163
 close 163
 Close EViews application 203
 Close window 163
 coef 164
 Coef (object) 20
 data members 20
 declare 164
 fill values 208
 procs 20
 views 20
 coefcov 165
 Coefficient
 covariance matrix of estimates 165
 See Coef (object).
 update default coef vector 372
 coint 166
 Cointegration
 make cointegrating relations from VEC 250
 Cointegration test 166
 colplace 397
 Column
 extract from matrix 398
 number of columns in matrix 398
 place in matrix 397

stack matrix 420
stack matrix (lower triangle) 420
Column width of table 321
Commands
auxiliary 10, 17
basic command summary 17
batch mode 2
execute without opening window 192
interactive mode 1
object assignment 9
object command 6
object declaration 5, 9
save record of 2
window 1
Comparison operators 68
for strings 99
with missing values 100
Complementary log-log function 442
Condition number of matrix 398
Conditional standard deviation
display graph of 219
Conditional variance
make series from ARCH 252
Container (object) 11
database 12
workfile 11
Continuation character in programs 86
control 168
Control variable 90
as replacement variable 94
in while statement 104
save value to scalar 90
Convert
date to observation number 93, 401
matrix object to series or group 409
matrix objects 62, 74, 399
matrix to sym 405
observation number to date 93, 411
scalar to string 416
series or group to matrix (drop NAs) 415
series or group to matrix (keep NAs) 416
string to scalar 91, 93, 419
sym to matrix 402
Coordinates
for legend in graph 240
Copy
database 181
copy 168
Copy objects 14, 168
cor 171
correl 172
Correlation 171, 400, 440
cross 177
Correlogram 172
squared residuals 173
correlsq 173
Cosine 444
count 173
Count models 173
cov 175
Covariance 175, 400, 440
Cramer-von Mises test 198
create 176
Create database 181
Create workfile 176
cross 177
Cross correlations 177
Cross product 411
Cross section member
add to pool 137
define list of 188
Current date function 421
Current time function 431
CUSUM test 301
of squares 301

D

Data
entering from keyboard 178
data 178

Data members
coef 20
equation 23
group 28
matrix 32
pool 35
rowvector 37
series 40
sspace 41
sym 45
system 46
table 48
var 50

vector 52
Database
 copy 13, 181
 create 12, 181
 delete 13, 182
 fetch 205
 Haver Analytics 227, 230
 open 13, 183
 open or create 180
 pack 184
 rebuild 184
 rename 13, 185
 repair 185
Date
 @-functions 93
 convert from observation number 411
 convert to observation number 93, 401
 functions 436
Dated data report table 196
Dated data table 255
dates 178
db 180
 dbcopy 181
 dbcreate 181
 dbdelete 182
 dbopen 183
 dbpack 184
 dbrebuild 184
 dbrename 185
 dbrepair 185
Declare
 matrix 55
 object 5, 9
decomp 186
define 188
Delete
 database 182
 object 16
 objects or pool identifiers 188
delete 188
Derivatives
 make series or group containing 251
Derivatives of equation specification 189
derivs 189
describe 190
Descriptive statistics 344
@-functions 439
 by category of dependent variable 266
 by classification 340
 make series 263
 matrix functions 77
 pool 190
Determinant 401
Diagonal matrix 408
Dickey-Fuller test 373
Difference 438
Directory
 change working 156
 EViews executable 423
Display
 action 6
 and print 8
Display numbers 327
Display object 328
displayname 192
Distribution function
 empirical cumulative, survivor and quantiles 157
 Quantile-quantile plot 288
Division operator (/) 68
do 192
Double exponential smoothing 330
draw 193
Draw lines in graph 193
DRI database
 convert to EViews database 194
 copy from 156
 fetch series 160
 read series description 162
driconvert 194
Drop
 group series or cross-section from pool definition 195
drop 195
dtable 196
Dummy variable
 seasonal 439
Durbin's h 107
Dynamic forecast 214
E
ec 196

edftest 198
Eigenvalues 402
Eigenvectors 402
Elapsed time 364
elapsed time 432
Element
 assign in matrix 56
 matrix functions 77
else 422
Else clause in if statement 98, 422
Empirical distribution functions 157
Empirical distribution test 198
Empty string 99
endif 422
endog 200
Endogenous variables 200
 make series or group 251
endsub 422
eqs 201
equals comparison 68
equation 200
Equation (object) 21
 data members 23
 declare 200
 methods 21
 procs 23
 views 22
errbar 201
Error bar graph 201
Error correction model
 See VEC and VAR.Vector error correction model
Error count in programs 423
Error function 443
 complementary 443
Error handling 105
Estimation methods
 (single) equation 21
 pool 248
 state space 29, 41
 system (of equations) 46
 VAR 49
Euler's constant 443
Excel file
 reading data from 291
 writing data to 383
exclude 203
Exclude variables from model solution 203
Execute program 86
 abort 88
 quiet mode 87
 verbose mode 87
 with arguments 96
Exit
 from EViews 203
 loop 106, 423
 subroutine 428
exit 203
exitloop 423
exp 437
expand 204
Expand workfile 204
Exponential
 distribution 446
 function 437
Exponential smoothing 330
Export data
 matrix 73
Exporting data to file 383
Extract
 main diagonal of matrix 404
 row vector 414
 submatrix from matrix 417
Extreme value distribution 446

F

Factorial 437
F-distribution 446
fetch 205
Fetch object 16, 205
Files
 temporary location 430
Fill
 values of matrix 57
 values of object 208
fill 208
Filled
 matrix 403
 row vector 403
 symmetric matrix 403
 vector 404
fiml 210
fit 212

Fixed effects 248
 for 424
 For loop
 accessing elements of a series 101
 accessing elements of a vector 101
 changing samples within 101
 define using control variables 100
 define using scalars 102
 define using string variables 102
 exit loop 106
 mark end 425
 nesting 102
 roundoff error in 429
 start loop 424
 step size 429
 upper limit 432
 Forecast
 dynamic (multi-period) 214
 static (one-period ahead) 212
 forecast 214
 Format number 81
 freeze 216
 Freeze view 216
 freq 217
 Frequency conversion
 set method 321
 Frequency table
 one-way 217
 Full information maximum likelihood 210

G

Gamma
 distribution 446
 Gamma function 443
 derivative 442, 443
 incomplete 443
 incomplete derivative 443
 incomplete inverse 444
 logarithm 444
 second derivative 444
 GARCH
 display conditional standard deviation 219
 estimate model 145
 generate conditional variance series 252
 garch 219
 Gaussian distribution 447
 Generalized autoregressive conditional heteroske-
 dasticity. See ARCH and GARCH.
 Generalized error distribution 446
 Generalized method of moments 221
 Generate series
 for pool 220
 genr 220
 See also series.
 Global
 subroutine 110
 variable 110
 GMM
 estimate 221
 gmm 221
 Gompit models 152
 Goodness of fit (for binary models) 360
 Gradients
 display 223
 saving in series 253
 grads 223
 Granger causality test 154, 359
 Graph
 align multiple graphs 142
 axis labeling 178
 change legend or axis name 271
 drawing lines and shaded areas 193
 error bar 201
 high-low-open-close 228
 place text 140
 set axis scale 309
 set individual graph options 323
 set options 275
 spike 338
 templates 355
 XY line graph 394
 graph 224
 Graph (object) 25
 creating 224
 procs 25
 greater than comparison 68
 greater than or equal to comparison 68
 Group
 convert to matrix 345, 346, 409
 convert to matrix (with NAs) 416
 group 226
 Group (object) 26
 add series 137
 data members 28

- declare 226
procs 27
views 26
- H**
- Haver Analytics Database
 convert to EViews database 226
hconvert 226
Heteroskedasticity test (White) 379
hfetch 227
High-Low (Open-Close) graphs 228
hilo 228
hist 229
Histogram 229
hlabel 230
Hodrick-Prescott filter 231
Holt-Winters 330
Hosmer-Lemeshow test 360
hpf 231
- I**
- Identity matrix 404
 extract column 419
if 424
If statement 98
 else clause 98, 422
 end of condition 422
 start of condition 424
 then 431
Import data
 matrix 72
Import data from file 291
impulse 232
Impulse response function 232
Include
 file in a program file 425
 program file 107
include 425
Incomplete beta
 derivative 442
 integral 441
 inverse 442
Incomplete beta integral 441
Incomplete gamma 443
Independence test 152
- Initial parameter values 281
Inner product 405, 440
Insertion point in command line 2
Integer random number 302
Interactive mode 1
Inverse of matrix 406
- J**
- Jarque-Bera
 multivariate normality test 234
jbera 234
Johansen cointegration test 166
- K**
- Kalman filter 252
kdensity 236
kerfit 237
Kernel
 bivariate regression 237
 density 236
Kolmogorov-Smirnov test 198
Kronecker product 407
- L**
- label 238
Label object 192, 238
Lag
 specify as range 432
 VAR lag order selection 239
Lag exclusion test 361
laglen 239
Lagrange multiplier
 test for ARCH in residuals 147
Landscape printing 8
Laplace distribution 446
Least squares estimation 245
Legend
 appearance and placement 240
 rename 271
legend 240
less than comparison 68
less than or equal to comparison 68
Lilliefors test 198
line 241
Line drawing 193

Line graph 241
 Line pattern 193
 Line style 193
 linefit 242
 Load
 workfile 244
 load 244
 Local
 subroutine 112
 variable 110
 log
 arbitrary base 438
 base 10 438
 natural 437
 Log difference 438
 Logistic
 logit function 444
 Logistic distribution 447
 logit 244
 Logit models 152
 logl 245
 Logl (object) 29
 check user-supplied derivatives 160
 data members 30
 declare 245
 method 29
 procs 29
 statements 29
 views 29
 Log-normal distribution 447
 Loop
 exit loop 106, 423
 for (control variables) 100
 for (scalars) 102
 for (string variables) 102
 nest 102
 over matrix elements 73, 101
 while 104
 ls 245

M

MA
 seasonal 330
 ma 249
 Main diagonal of matrix 404
 Make model object 257
 Make residuals 259
 makecoint 250
 makederivs 251
 makeendog 251
 makefilter 252
 makegarch 252
 makegraph 254
 makegroup 255
 makelimits 257
 makemodel 257
 makeregs 258
 makeresids 259
 makesignals 260
 makestates 262
 makestats 263
 makesystem 264
 Mathematical functions 437
 matplace 408
 Matrix
 assign values 56
 convert to other matrix objects 74
 convert to series or group 62
 copy 59
 copy submatrix 61
 declare 55
 export data 73
 filled 403
 import data 72
 main diagonal 404
 objects 55
 permute rows of 412
 place submatrix 408
 resample rows from 412
 singular value decomposition 418
 stack columns 420
 stack lower triangular columns 420
 matrix 265
 Matrix (object)
 data members 32
 declare 265
 fill values 208
 procs 31
 views 31, 72
 Matrix commands and functions
 commands 70
 descriptive statistics 69, 77
 difference 70

element 69, 77
functions 70
matrix algebra 69, 77
missing values 71
utility 69, 76

Matrix operators
addition (+) 67
and loop operators 73
comparison operators 68
division (/) 68
multiplication (*) 67
negation (-) 66
order of evaluation 66
subtraction (-) 67

Maximum 441
Maximum likelihood estimation 269
 Logl (object) 29
 Sspace (object) 41

Mean 440
Mean test 356, 357
means 266

Median 441
Median test 356, 357

merge 267
Messages
 suppress during program execution 87

metafile 268

Minimum 441
Missing value code 270

Missing values 71
 inequality comparison 100
 mathematical functions 437
 recoding 438
 test 99

ml 269
model 269

Model (object) 32
 append specification line 143
 break all model links 371
 declare 269
 equation view 201
 procs 33
 update specification 372
 variable view 377
 views 33

Models
 add factor assignment and removal 138

add factor initialization 139
block structure 154
exclude variables from solution 203
make from estimation object 257
make graph of model series 254
make group of model series 255
options for solving 335
overrides in model solution 280
scenarios 313
solution messages 270
solve 334
solve to match target 168
text representation 363
trace iteration history 365

modulus 437

Moving average 249, 438

Moving sum 439

msg 270

mtos 409

Multiplication operator (*) 67

N

NA
 inequality comparison 100
 recode 438
 test 99

na 270

name 271
Nearest neighbor regression 272

Negation operator (-) 66

Negative binomial count model 173

Negative binomial distribution 447

next 425

nnfit 272
Nonlinear least squares 245

Norm of a matrix 410

Normal distribution 447

Normal random number 274

not equal to comparison 68

nrnd 274

Number
 evaluate a string 419
 formatting in tables 81

Number of observations 440

O

Object
 assignment 9
 command 6
 containers 11
 copy 14
 create using freeze 216
 declaration 5, 9
 delete 16
 fetch 16, 205
 merge 267
 rename 15, 293
 save 16
 store 16
 test for existence 425

Observations
 number in workfile range 426
 OLS (ordinary least squares) 245
 Omitted variables test 355
 One-way frequency table 217

Open
 database 183
 files 275
 workfile 244

open 275
 Operator 435
 options 275
 Or operator 436
 ordered 277

Ordered dependent variable
 estimating models with 277
 make vector of limit points from equation 257

Outer product 411

Output
 display estimation results 279
 extracting results from views 124
 printing 8

output 279

Output redirection 427

override 280

Override variables in model solution 280

P

Pack database 184
 param 281

Parameters 281
 Pareto distribution 447
 Partial autocorrelation 172
 Partial correlation 172
 pdl 282
 PDL (polynomial distributed lag) 282
 Percent change 439
 Percentage change 439
 Permute rows of matrix 412
 Phillips-Perron test 373
 Pi 444
 Pi (constant) 444
 pie 284
 Pie graph 284
 poff 427
 Poisson count model 173
 Poisson distribution 447
 Polynomial distributed lags 282
 pon 427
 Pool
 generate series using identifiers 220
 make group of pool series 255
 pool 285
 Pool (object) 34
 add cross section member 137
 data members 35
 declare 285
 delete identifiers 188
 fixed effects 248
 members 34
 procs 34
 random effects 248
 views 34
 Portrait (print orientation) 8
 Power (raise to) 435
 Precedence of evaluation 66
 predict 285
 Prediction table 285
 Presentation table 196
 Principal components 281
 Print
 and display 8
 automatic printing 427
 landscape 8
 portrait 8
 turn off in program 427

print 286
Printing
 automatic printing 427
probit 287
Probit models 152
Program 85
 abort 88
 arguments 96
 call subroutine 109, 421
 counting execution errors 423
 create 85
 declare 287
 entering text 85
 exit loop 106
 if statement 98
 include file 107, 425
 line continuation character 86
 open 86
 place subroutine 109
 quiet mode 87
 run 305
 running 86
 save 86
 stop 105
 stop execution 430
 verbose mode 87
program 287
P-value functions 448

Q

QQ-plot
 See Distribution function.
qqplot (quantile-quantile) 288
Q-statistic 172
qstas 289
Quantile function 441
Quantile-Quantile plot. See QQ-plot
Quiet mode 87

R

Random effects 248
Random number
 integer 302
 seed 303
 uniform 302
Random number generator
 normal 274
range 290
Rank 412
Read
 data from foreign file 291
read 291
Recode values 438
Recursive least squares 300
 CUSUM 301
 CUSUM of squares 301
Redirect output to file 8, 279
Redundant variables test 358
Regressors
 make group containing 258
Rename
 database 185
 object 15, 293
rename 293
Repair database 185
Replacement variable 93
 naming objects 95
Resample
 observations 295
 rows from matrix 412
resample 295
reset 297
RESET test 297
Reset timer 363
residcor 297
residcov 298
Residuals
 correlation matrix of 297
 covariance matrix of 298
 make series or group containing 259
Resize workfile 290
Restricted VAR text 163
Results
 display or retrieve 300
results 300
return 428
rls 300
rnd 302
rndint 302
rndseed 303
Roots of the AR polynomial 148

Roundoff error in for loops 429
 Row
 numbers 414
 place in matrix 414
 rowplace 414
 rowvector 304
 Rowvector (object)
 data members 37
 declare 304
 extract 414
 filled rowvector function 403
 views 37
 run 305
 Run program 305
 multiple files 106

S

Sample
 change using for loop 101
 number of observations 426
 set 319, 332
 sample 306
 Sample (object)
 declare 306
 procs 38
 sar 307
 Save
 commands in file 2
 objects to disk 16
 workfile 308
 save 308
 scalar 309
 Scalar (object) 38
 declare 309
 scale 309
 scat 311
 scatmat 313
 Scatter diagrams 311
 matrix of 313
 with bivariate fit 242
 scenario 313
 seas 315
 Seasonal adjustment
 moving average 315
 Tramo/Seats 365
 X11 387
 X12 388
 Seasonal autoregressive error 307
 Seasonal dummies 439
 Seasonal dummy variable 439
 Seasonal graphs 316
 seasplot 316
 Second moment matrix 405
 Seed random number generator 303
 Seemingly unrelated regression. See SUR
 Sequential LR tests 106
 Serial correlation
 Breusch-Godfrey LM test 149
 multivariate VAR LM test 148
 Series
 convert to matrix 345, 346, 409, 415
 convert to matrix (with NAs) 416
 extract observation 436
 series 317
 Series (object) 39
 data members 40
 declare 317
 element function 40
 fill values 208
 views 39
 set 319
 Set graph date labeling formats 178
 setcell 319
 setcolwidth 321
 setconvert 321
 setelem 323
 setline 326
 Shade region of graph 193
 sheet 327
 show 328
 Show object view 328
 Signal variables
 display graphs 329
 saving 260
 signalgraph 329
 Sine 444
 Singular matrix
 test for 406
 Singular value decomposition 418
 sma 330
 smooth 330

Smoothing
 exponential smooth series 330
 signal series 260
 state series 262
smpl 332
Solve
 linear system 415
 simultaneous equations model 334
solve 334
Solve. See Models.
solveopt 335
sort 336
Sort workflow 336
spec 337
Specification view 337
spike 338
Spike graph 338
Spreadsheet view 327
sqrt 438
sspace 339
Sspace (object)
 append specification line 143
 data members 41
 declare 339
 display signal graphs 329
 make Kalman filter from 252
 method 41
 procs 41
 state graphs 342
 views 41
Stack matrix by column 420
 lower triangle 420
Standard deviation 441
Starting values 281
statby 340
State variables
 display graphs of 342
 final one-step ahead predictions 343
 initial values 343
 smoothed series 262
statefinal 343
stategraph 342
stateinit 343
Static forecast 212
Statistical distribution functions 444
Statistics 190
 compute for subgroups 340
stats 344
Status line 345
statusline 345, 428
step 429
stom 345, 415
stomna 346, 416
stop 430
Stop program execution 105, 430
store 347
Store object 16, 347
String 90
 assign to table cell 80
 comparison 99
 convert to a scalar 419
 empty 99
 from a number 416
 length of 417
String variable 90
 @-functions 92
 as replacement variable 94
 comparison 99
 convert to a scalar 91, 93
 in for loop 102
 program arguments 96
 test for empty string 99
Subroutine 107
 arguments 108
 call 109, 421
 declare 430
 define 107
 global 110
 local 112
 mark end 422
 placement 109
 return from 107, 428
 subroutine 430
Substring 407, 409
Subtraction operator (-) 67
Sum 441
Sum of squares 441
SUR
 estimating 350
sur 350
svar 351
sym 353

-
- Sym (object) 44
 create from lower triangle of square matrix 405
 create from scalar function 403
 create square matrix from 402
 data members 45
 declare 353
 procs 45
 stack columns 420
 views 44
- Symmetric matrix
 See Sym.
- system 354
- System (object) 45
 append specification line 143
 create from pool or var 264
 data members 46
 declare 354
 methods 46
 procs 46
 views 46
- T**
- Table
 decimal format code 81
 declare 79
 example 82
 fill cell with number 80
 fill cell with string 80
 format cell 81
 horizontal line 80, 326
 justification code 81
 set and format cell contents 319
 set column width 79, 321
- Table (object) 48
 data members 48
 views 48
- Tangent 444
- t-distribution 448
- template 355
- Test
 Chow 161
 CUSUM 301
 CUSUM of squares 301
 exogeneity 359
 for ARCH 147
 for serial correlation 148, 149
 Goodness of fit 360
- Granger causality 154
 heteroskedasticity (White) 379
 Johansen cointegration 166
 lag exclusion (Wald) 361
 mean, median, variance equality 356, 362
 mean, median, variance equality by classification 357
 omitted variables 355
 redundant variables 358
 RESET 297
 unit root 373
 Wald 378
- testadd 355
 testbtw 356
 testby 357
 testdrop 358
 testexog 359
 testfit 360
 testlags 361
 teststat 362
 text 363
- Text (object)
 declare 363
- Then 431
- Three stage least squares 136
- tic 363
- Time trend 439
- Timer 363, 364, 432
- to 432
- Tobit models 158
- toc 364
- trace 365
- Trace of a matrix 418
- Tramo/Seats 365
- tramoseats 365
- Transpose 419
- Trend series 439
- Trigonometric functions 444
- Truncated dependent variable models 158
- tsls 368
- Two-stage least squares
 see 2sls

U

Uniform distribution 448
Uniform random number generator 302
Unit vector 419
unlink 371
Untitled objects 11
update 372
updatecoefs 372
uroot 373

V

VAR
estimate factorization matrix 351
impulse response 232
lag exclusion test 361
lag length test 239
multivariate autocorrelation test 289
variance decomposition 186

var 376

VAR (object) 49
clear restrictions 163
data members 50
declare 376
methods 49
procs 49
views 49

Variance 441

Variance decomposition 186

Variance test 356, 357

vars 377

VEC

estimating 196

vector 377

Vector (object) 52
data members 52
declare 377
procs 52
return filled 404
views 52

Vector autoregression
See VAR.

Verbose mode 87

Views

extracting results from 124

W

wald 378
Wald test 378
Watson test 198
Weibull distribution 448
Weighted least squares 380
Weighted two-stage least squares 385
wend 433
while 433
While loop 104
abort 104
end of 433
exit loop 106
start of 433
white 379
Wildcards 14
wls 380
Workfile 11
close 12
creating 176
creating or changing active 381
expand 204
frequency 11
open 12
open existing 244
resize 290
save 12
save to disk 308
sort observations 336
workfile 381

Write
data to file 383
write 383
wtsls 385

X

X11 387
x11 387
X12 388
x12 388
xyline 394