EViews® 8

Estimation · Forecasting · Statistical Analysis
Graphics · Data Management · Simulation

Command and Programming Reference

IHS

EViews 8 Command and Programming Reference

EViews 8 Command and Programming Reference

ISBN: 978-1-880411-17-9

Disclaimer

The authors and IHS Global Inc. 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

EViews® is a registered trademark of IHS Global Inc. Windows, Excel, PowerPoint, and Access are registered trademarks of Microsoft Corporation. PostScript is a trademark of Adobe Corporation. X11.2 and X12-ARIMA Version 0.2.7, and X-13ARIMA-SEATS are seasonal adjustment programs developed by the U. S. Census Bureau. Tramo/Seats is copyright by Agustin Maravall and Victor Gomez. Info-ZIP is provided by the persons listed in the infozip_license.txt file. Please refer to this file in the EViews directory for more information on Info-ZIP. Zlib was written by Jean-loup Gailly and Mark Adler. More information on zlib can be found in the zlib_license.txt file in the EViews directory. All other product names mentioned in this manual may be trademarks or registered trademarks of their respective companies.

IHS Global Inc.
4521 Campus Drive, #336
Irvine CA, 92612-2621
Telephone: (949) 856-3368
Fax: (949) 856-2044
e-mail: sales@eviews.com
web: www.eviews.com

March 4, 2013

Table of Contents

Preface

The EViews 8 *User's Guide* focuses primarily on interactive use of EViews using dialogs and other parts of the graphical user interface.

Alternatively, you may use EViews' powerful command and batch processing language to perform almost every operation that can be accomplished using the menus. You can enter and edit commands in the command window, or you can create and store the commands in programs that document your research project for later execution.

This text, the EViews 8 *Command and Programming Reference*, documents the use of commands in EViews, along with examples of commands for commonly performed operations. provide general information about the command, programming, and matrix languages:

The first chapter provides an overview of using commands in EViews:

- Chapter 1. "Object and Command Basics," on page 3 explains the basics of using commands to work with EViews objects, and provides examples of some commonly performed operations.

The next set of chapters discusses commands for working with specific EViews objects:

- Chapter 2. "Working with Graphs," on page 21 describes the use of commands to customize graph objects.
- Chapter 3. "Working with Tables and Spreadsheets," on page 45 documents the table object and describes the basics of working with tables in EViews.
- Chapter 4. "Working with Spools," on page 57 discusses commands for working with spools.

The EViews programming and matrix language are described in:

- Chapter 5. "Strings and Dates," on page 65 describes the syntax and functions available for manipulating text strings and dates.
- Chapter 6. "EViews Programming," on page 105 describes the basics of using programs for batch processing and documents the programming language.
- Chapter 7. "External Interfaces," on page 161 documents EViews features for interfacing with external applications through the OLEDB driver and various COM automation interfaces.
- Chapter 11. "Matrix Language," on page 239 describes the EViews matrix language.

The remaining chapters contain reference material:

- Chapter 12. “Command Reference,” on page 263 is the primary reference for commands to work with EViews objects, workfiles, databases, external interfaces, programs, as well as other auxiliary commands.
- Chapter 13. “Operator and Function Reference,” on page 503 offers a *categorical* list of element operators, numerical functions and descriptive statistics functions that may be used with series and (in some cases) matrix objects.
- Chapter 14. “Operator and Function Listing,” on page 535 contains an *alphabetical* list of the element operators, numerical functions and descriptive statistics functions that may be used with series and (in some cases) matrix objects.
- Chapter 15. “Workfile Functions,” on page 549 describes special functions for obtaining information about observations in the workfile.
- Chapter 16. “Special Expression Reference,” on page 557 describes special expressions that may be used in series assignment and generation, or as terms in estimation specifications.
- Chapter 17. “String and Date Function Reference,” on page 567 documents the library of string and date functions for use with alphanumeric and date values.
- Chapter 18. “Matrix Language Reference,” on page 605 describes the functions and commands used in the EViews matrix language.
- Chapter 19. “Programming Language Reference,” on page 649 documents the functions and keywords used in the EViews programming language.

There is additional material in the appendix:

- Appendix A. “Wildcards,” on page 683 describes the use of wildcards in different contexts in EViews.

Chapter 1. Object and Command Basics

This chapter provides an brief overview of the command method of working with EViews and EViews objects. 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 and assignment statements.
- object view and procedure commands.
- interactive commands for creating objects and displaying views and procedures.
- auxiliary commands.

The following sections provide an overview of each of the command types. But before discussing the various types, we offer a brief discussion of the interactive and batch methods of using commands in EViews.

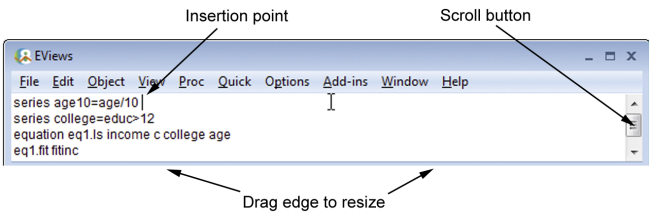
Using Commands

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

Interactive Use

The *command window* is located (by default) just below the main menu bar at the top of the EViews window. A blinking insertion cursor in the command window indicates that keyboard focus is in the command window and that keystrokes will be entered in the window at the insertion point. If no insertion cursor is present, simply click in the command window to change the focus.

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



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 that contains the insertion point.

The contents of the command area may also be saved directly into a text file for later use. First make certain that the command window is active by clicking anywhere in the window,

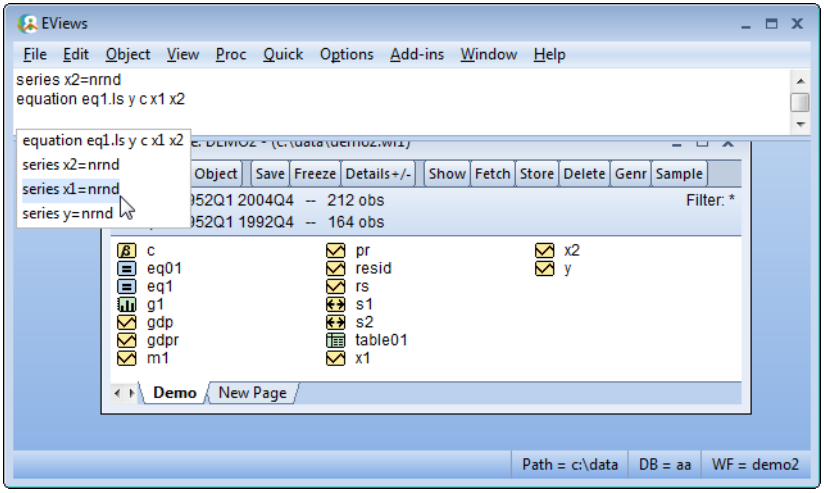
and then select **File/Save As…** from the main menu. EViews will prompt you to save an ASCII file in the default working directory (default name "commandlog.txt") containing the entire contents of the command window.

Command Window Editing

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

EViews offers a couple of specialized tools for displaying previous commands. First, to bring up previous commands in the order they were entered, press the Control key and the UP arrow (CTRL + UP). The last command will be entered into the command window. Holding down the CTRL key and pressing UP repeatedly will display the next prior commands. Repeat until the desired command is displayed.

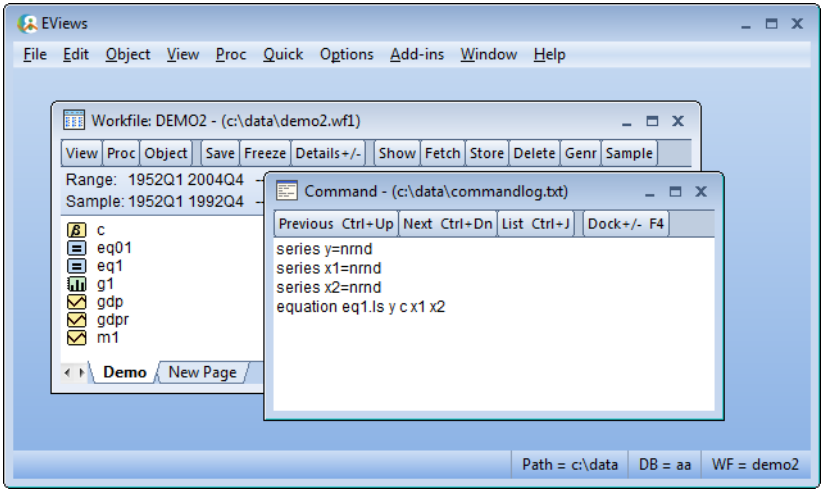
To look at a history of commands, press the Control Key and the J key (CTRL + J). This key combination displays a history window containing the last 30 commands executed. Use the UP and DOWN arrows until the desired command is selected and then press the ENTER key to add it to the command window, or simply double click on the desired command. To close the history window without selecting a command, click elsewhere in the command window or press the Escape (ESC) key.



To execute the retrieved command, simply press ENTER again. You may first edit the command if you wish to do so.

You may 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.

Command Window Docking



You may drag the command window to anywhere inside the EViews frame. Press F4 to toggle docking, or click on the command window, depress the right-mouse button and select **Toggle Command Docking**.

When undocked, the command window toolbar contains buttons for displaying commands in the list, and for redocking.

Keyboard Focus

We note that as you open and close object windows in EViews, the keyboard focus may change from the command window to the active window. If you then wish to enter a command, you will first need to click in the command window to set the focus. You can influence EViews' method of choosing keyboard focus by changing the global defaults—simply select **Options/General Options.../Window Behavior** in the main menu, and change the **Keyboard Focus** setting as desired.

Batch Program Use

You may 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. Batch program use of EViews is discussed in greater detail in Chapter 6. “EViews Programming,” on page 105.

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. It will prove convenient to name your file using the extension “.PRG”. The commands in this program may then be executed from within EViews.

You may also enter commands in the command window and then use **File/Save As...** to save the log for editing.

Object Declaration and Initialization

The simplest types of commands create an EViews object, or assign data to or initialize an existing object.

Object Declaration

A simple object declaration has the form

object_type(options) object_name

where *object_name* is the name you would like to give to the newly created object and *object_type* is one of the following object types:

 Alpha (p. 4)	 Pool (p. 406)	 Sym (p. 627)
 Coef (p. 16)	 Rowvector (p. 451)	 System (p. 651)
Equation (p. 31)	Sample (p. 466)	Table (p. 688)
Factor (p. 159)	Scalar (p. 473)	Text (p. 718)
 Graph (p. 208)	 Series (p. 478)	 User (p. 726)
 Group (p. 254)	 Spool (p. 596)	 Valmap (p. 735)
 Logl (p. 325)	 Sspace (p. 567)	 Var (p. 743)

Matrix (p. 340)	String (p. 617)	Vector (p. 781)
Model (p. 370)	Svector (p. 622)	

Details on each of the commands associated with each of these objects are provided in the section beginning on the specified page in the *Object Reference*.

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 as an option provided 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.

Simple declarations initialize the object with default values; in some cases, the defaults have a meaningful interpretation, while in other cases, the object will simply be left in an incomplete state. In our examples, the newly created LGDP will contain all NA values and X and RESULTS will be initialized to 0, while EQ1 will be simply be an uninitialized equation containing no estimates.

Note that in order to declare an object you must have a workfile currently open in EViews. You may open or create a workfile interactively from the File Menu or drag-and-dropping a file onto EViews (see Chapter 3. "Workfile Basics," on page 41 of *User's Guide I* for details), or you can may use the `wfopen` (p. 472) command to perform the same operations inside a program.

Object Assignment

Object assignment statements are commands which assign data to an EViews object using the " = " sign. 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. Note that not all objects per-

mit object assignment; for example, you may not perform assignment to an equation object. (You may, however, initialize the equation using a command method.)

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.

Similarly, if M1, M2, and M3 are matrices, we may use the assignment statement:

```
m1 = @inverse(m2) * m3
```

to postmultiply the matrix inverse of M2 by M3 and assign the result to M1. This statement presumes that M2 and M3 are suitably conformable.

Object Modification

In cases where direct assignment using the " = " operator is not allowed, one may initialize the object using one or more object commands. We will discuss object commands in greater detail in a moment (see "Object Commands," on page 9) but for now simply note that object commands may be used to modify the contents of an existing object.

For example:

```
eq1.ls log(cons) c x1 x2
```

uses an object command to estimate the linear regression of the LOG(CONS) on a constant, X1, and X2, and places the results in the equation object EQ1.

```
sys1.append y=c(1)+c(2)*x
sys1.append z=c(3)+c(4)*x
sys1.ls
```

adds two lines to the system specification, then estimates the specification using system least squares.

Similarly:

```
group01.add gdp cons inv g x
```

adds the series GDP, CONS, INV, G, and X to the group object GROUP01.

More on Object Declaration

Object declaration may often be combined with assignment or command initialization. For example:

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

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

```
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.

Lastly:

```
group group01 gdp cons inv g x
```

create the group GROUP01 containing the series GDP, CONS, INV, G, and X.

An object may 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:

```
smpl @first 1979
series dummy = 1
smpl 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.

Object Commands

Most of the commands that you will employ are *object commands*. An *object command* is a command which displays a view of or performs a procedure using a specific object. Object commands have two main parts: an *action* followed by a *view* or *procedure specification*. The (optional) display action determines what is to be done with the output from the view or procedure. The view or procedure specification may provide for options and arguments to modify the default behavior.

The complete syntax for an object command has the form:

action(*action_opt*) *object_name*.*view_or_proc(options_list) arg_list*

where:

action.....is one of the four verb commands (`do`, `freeze`, `print`, `show`).
action_opt .....an option that modifies the default behavior of the action.
object_name .....the name of the object to be acted upon.
view_or_proc .....the object view or procedure to be performed.
options_list .....an option that modifies the default behavior of the view or procedure.

arg_list ..... a list of view or procedure arguments.

Action Commands

There are four possible action commands:

- `show` displays the object view in a window.
- `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.

As noted above, in most cases, you need not specify an action explicitly. If no action is specified, the `show` action is assumed for views and the `do` action is assumed for most procedures (though some procedures will display newly created output in new windows unless the command was executed via a batch program).

For example, when using an object command to display the line graph series view, EViews implicitly adds a `show` command. Thus, the following two lines are equivalent:

```
gdp.line
show gdp.line
```

In this example, the *view_or_proc* argument is `line`, indicating that we wish to view a line graph of the GDP data. There are no additional options or arguments specified in the command.

Alternatively, for the equation method (procedure) `ls`, there is an implicit `do` action:

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

so that the two command lines describe equivalent behavior. In this case, the object command will not open the window for EQ1 to display the result. You may display the window by issuing an explicit `show` command after issuing the initial command:

```
show eq1
```

or by combining the two commands:

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

Similarly:

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

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

The following lines show a variety of object commands with modifiers:

```
show gdp.line
print(l) 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 noted 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 request that a view be simultaneously printed and displayed on your screen by including 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 recommend that the "p" option *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(l) gdp.correl(24)
```

Printer output can be redirected to a text file, frozen output, or a spool object. (See `output` (p. 387), and the discussion in "Print Setup" on page 779 of *User's Guide I* 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:

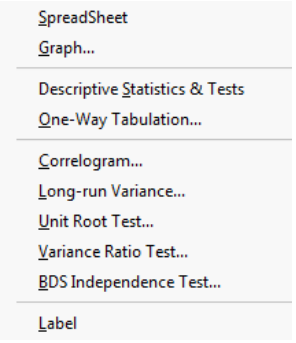
```
freeze(figure1) gdp.bar
```

names the frozen bar graph of GDP as "figure1".

View and Procedure Commands

Not surprisingly, the view or procedure commands correspond to elements of the views and procedures menus for the various objects.

For example, the *top level* of the view menu for the series object allows you to: display a spreadsheet view of the data, graph the data, perform a one-way tabulation, compute and display a correlogram or long-run variance, perform unit root or variance ratio tests, conduct a BDS independence test, or display or modify the label view.



Object commands exist for each of these views. Suppose for example, that you have the series object SER01. Then:

```
ser01.sheet
ser01.stats
```

display the spreadsheet and descriptive statistics views of the data in the series. There are a number of graph commands corresponding to the menu entry, so that you may enter:

```
ser01.line
ser01.bar
ser01.hist
```

which display a line graph, bar graph, and histogram, respectively, of the data in SER01. Similarly,

```
ser01.freq
```

performs a one-way tabulation of the data, and:

```
ser01.correl
ser01.lrvar
ser01.uroot
ser01.vratio 2 4 8
ser01.bdstest
```

display the correlogram and long-run variances, and conduct unit root, variance ratio, and independence testing for the data in the series. Lastly:

```
ser01.label(r) "this is the added series label"
```

appends the text "this is the added series label" to the end of the remarks field.

There are commands for all of the views and procedures of each EViews object. Details on the syntax of each of the object commands may be found in Chapter 1. "Object View and Procedure Reference," beginning on page 2 in the *Object Reference*.

Object Data Members

Every object type in EViews has a selection of data members. These members contain information about the object and can be retrieved from an object to be used as part of another command, or stored into the workfile as a new object.

Data members can be accessed by typing the object name followed by a period and then the data member name. Note that all data members' names start with an "@" symbol.

The following data members belong to every object type in EViews:

Data Member Name	Description
@name	Returns the name of the object
@displayname	Returns the display name of the object. If the object has no display name, the name is returned
@type	Returns the object type
@units	Returns the units of the object, (if available)
@source	Returns the source of the object (if available)
@description	Returns the description of the object (if available)
@remarks	Returns the remarks of the object (if available)
@updatetime	Returns the string representation of the time the object was last updated

Along with these global data members, each object type has a set of data members specific to that type. For example, equation objects have a data member, `@r2`, that returns a scalar containing the R-squared from that equation. Groups have an member, `@count`, that returns a scalar containing the number of series contained within that group. A full list of each object's data members can be found under the object's section in Chapter 1. "Object View and Procedure Reference," on page 2 of the *Object Reference*.

As an example of using data members, the commands:

```
equation eq1.ls y c x1 x2
table tab1
tab1(1,1) = eq1.@f
```

create an equation named EQ1 and a table named TAB1, and then set the first cell of the table equal to the *F*-statistic from the estimated equation.

Interactive Commands

There is also a set of auxiliary commands which are designed to facilitate interactive use. These commands perform the same operations as equivalent object commands, but do so on newly created, 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 untitled equation object to hold the results.

Similarly, the command:

```
scat x y
```

creates an untitled group object containing the series X and Y and then displays a scatterplot of the data in the two series.

Since these commands are designed primarily for interactive use, they are designed for carrying out simple tasks. Overuse of these interactive tools, or their use in programs, will make it difficult to manage your work since 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. In general, we recommend that you use named objects rather than untitled objects for your work. For example, we may replace the first auxiliary command above with the statement:

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

to create the named equation object EQ1. This example uses declaration of the object EQ1 and the equation method `ls` to perform the same task as the auxiliary command above.

Similarly,

```
group mygroup x y
mygroup.scat
```

displays the scatterplot of the series in the named group MYGROUP.

Auxiliary Commands

Auxiliary commands are commands which are unrelated to a particular object (*i.e.*, are not object views or procs), or act on an object in a way that is generally independent of the type or contents of the object. Many of the important auxiliary commands are used for managing objects, and object containers. A few of the more important commands are described below.

Auxiliary commands typically follow the syntax:

command(option_list) argument_list

where *command* is the name of the command, *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 is:

```
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.

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. "Workfile Basics," beginning on page 41 and Chapter 10. "EViews Databases," beginning on page 303 of *User's Guide I*.

Managing Workfiles

To declare and create a new workfile, you may use the `wfcreate` (p. 467) command. You may enter the keyword `wfcreate` followed by a name for the workfile, an option for the frequency of the workfile, and the start and end dates. The most commonly used 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/unstructured.

but there are additional options for multi-year, bimonthly, fortnight, ten-day, daily with custom week, intraday, integer date, and undated frequency workfiles.

For example:

```
wfcreate macro1 q 1965Q1 1995Q4
```

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

```
wfcreate cps88 u 1 1000
```

creates a new undated workfile named CPS88 with 1000 observations.

Alternately, you may use wfopen (p. 472) to read a foreign data source into a new workfile.

If you have multiple open workfiles, the wfselect (p. 488) command may be used to change the active workfile.

To save the active workfile, use the wfsave (p. 485) command by typing the keyword wfsave 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:

```
wfsave a:\mywork
```

If necessary, enclose the path name in quotations.

To close the workfile, use the close (p. 289) command. For example:

```
close mywork
```

closes the workfile window of MYWORK.

To open a previously saved workfile, use the wfopen (p. 472) command. You should follow the 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:

```
wfopen "c:\mywork\proj1"
```

Managing Databases

To create a new database, follow the dbcreate (p. 320) command keyword with a name for the new database. Alternatively, you could use the db (p. 317) command keyword followed by a name for the new database. The two commands differ only when the named database already exists.

If you use dbcreate and the named database already exists on disk, EViews will error indicating that the database already exits. 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 will become the default database.

For example:

```
dbcreate 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 may use dbopen (p. 322) 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 may use dbrename to rename an existing database. Follow the dbrename keyword by the current (old) name and a new name:

```
dbrename temp1 newmacro
```

To delete an existing database, use the dbdelete (p. 322) command. Follow the dbdelete keyword by the name of the database to delete:

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

dbcopy (p. 318) makes a copy of the existing database. Follow the dbcopy keyword with the name of the source file and the name of the destination file:

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

dbpack (p. 324) and dbrebuild (p. 325) are database maintenance commands. See also Chapter 10. "EViews Databases," beginning on page 303 of *User's Guide I* 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 most commonly used commands; a full description of these, and other commands is provided in Chapter 12. "Command Reference," on page 263.

Copying Objects

You may create a duplicate copy of one or more objects using the copy (p. 306) 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 features is 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 patterns:

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

conform to each other. 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 A. "Wildcards," on page 683.

Renaming Objects

You can give an object a different name using the `rename` (p. 421) 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 or a database 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 A. "Wildcards," on page 683 describes further the use of wildcards.

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 `wfsave` (p. 485) and `wfopen` (p. 472) 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` (p. 444) 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 10. "EViews Databases," on page 303 of *User's Guide I* 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` (p. 332) 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`.

Chapter 2. Working with Graphs

EViews provides an extensive set of commands to generate and customize graphs from the command line or using programs. A summary of the graph commands described below may be found under "Graph" on page 208 of the *Object Reference*.

In addition, Chapter 15. "Graph Objects," on page 657 of *User's Guide I* describes graph customization in detail, focusing on the interactive method of working with graphs.

Creating a Graph

There are three types of graphs in EViews: graphs that are views of other objects, and named or unnamed graph objects. The commands provided for customizing the appearance of your graphs are available for use with named graph objects. You may use the dialogs interactively to modify the appearance of all types of graphs.

Displaying graphs using commands

The simplest way to display a graph view is to use one of the basic graph commands. ("Graph Creation Commands" on page 267 provides a convenient listing.)

Where possible EViews will simply open the object and display the appropriate graph view. For example, to display a line or bar graph of the series INCOME and CONS, you may simply issue the commands:

```
line income
bar cons
```

In other cases, EViews must first create an unnamed object and then will display the desired view of that object. For example:

```
scat x y z
```

first creates an unnamed group object containing the three series and then, using the `scat` view of a group, displays scatterplots of Y on X and Z on X in a single frame.

As with other EViews commands, graph creation commands allow you to specify a variety of options and arguments to modify the default graph settings. You may, for example, rotate the bar graph using the "rotate" option,

```
bar(rotate) cons
```

or you may display boxplots along the borders of your scatter plot using:

```
scat(ab=boxplot) x y z
```

Note that while using graph commands interactively may be quite convenient, these commands are not recommended for program use since you will not be able to use the resulting unnamed objects in your program.

The next section describes a more flexible approach to displaying graphs.

Displaying graphs as object views

You may display a graph of an existing object using a graph view command. For example, you may use the following two commands to display graph views of a series and a group:

```
ser2.area(n)
grp6.xypair
```

The first command plots the series SER2 as an area graph with normalized scaling. The second command provides an XY line graph view of the group GRP6, with the series plotted in pairs.

To display graphs for multiple series, we may first create a group containing the series and then display the appropriate view:

```
group g1 x y z
g1.scat
```

shows the scatterplot of the series in the newly created group G1.

There are a wide range of sophisticated graph views that you may display using commands. See Chapter . “,” beginning on page 799 of the *Object Reference* for a detailed listing along with numerous examples.

Before proceeding, it is important to note that *graph views* of objects differ from *graph objects* in important ways:

- First, graph views of objects may not be customized using commands after they are first created. The graph commands for customizing an existing graph are designed for use with graph objects.
- Second, while you may use interactive dialogs to customize an existing object’s graph view, we caution you that there is no guarantee that the customization will be permanent. In many cases, the customized settings will not be saved with the object and will be discarded when the view changes or if the object is closed and then reopened.

  In contrast, graph objects may be customized extensively after they are created. Any customization of a graph object is permanent, and will be saved with the object.

Since construction of a graph view is described in detail elsewhere (Chapter . “,” beginning on page 799 of the *Object Reference*), we focus the remainder of our attention on the creation and customization of graph objects.

Creating graph objects from object views

If you wish to create a graph object from another object, you should combine the object view command with the `freeze` command. Simply follow the `freeze` keyword with an optional name for the graph object, and the object view to be frozen. For example,

```
freeze grp6.xypair(m)
```

creates and displays an unnamed graph object of the GRP6 view showing an XY line graph with the series plotted in pairs in multiple graph frames. Be sure to specify any desired graph options (*e.g.*, "m"). Note that freezing an object view will not necessarily copy the existing custom appearance settings such as line color, axis assignment, *etc.* For this reason that we recommend that you create a graph object before performing extensive customization of a view.

You should avoid creating unnamed graphs when using commands in programs since you will be unable to refer to, or work with the resulting object in a program. Instead, you should tell EViews to create a named object, as in:

```
freeze(graph1) grp6.line
```

which creates a graph object GRAPH1 containing a line graph of the data in GRP6. By default, the frozen graph will have updating turned off, but in most cases you may use the `Graph::setupdate` graph proc to turn updating on.

Note that using the `freeze` command with a name for the graph will create the graph object and store it in the workfile without showing it. Furthermore, since we have frozen a graph type (line) that is different from our current XY line view, existing custom appearance settings will not be copied to the new graph.

Once you have created a named graph object, you may use the various graph object procs to further customize the appearance of your graph. See "Customizing a Graph," beginning on page 26.

Creating named graph objects

There are three direct methods for creating a named graph object. First, you may use the `freeze` command as described above. Alternatively, you may declare a graph object using the `graph` command. The `graph` command may be used to create graph objects with a specific graph type or to merge existing graph objects.

Specifying a graph by type

To specify a graph by type you should use the `graph` keyword, followed by a name for the graph, the type of graph you wish to create, and a list of series (see "Graph Type Commands" on page 208 of the *Object Reference* for a list of types). If a type is not specified, a line graph will be created.

For example, both:

```
graph gr1 ser1 ser2
graph gr2.line ser1 ser2
```

create graph objects containing the line graph view of SER1 and SER2, respectively.

Similarly:

```
graph gr3.xyline group3
```

creates a graph object GR3 containing the XY line graph view of the series in GROUP3.

Each graph type provides additional options, which may be included when declaring the graph. Among the most important options are those for controlling scaling or graph type.

The scaling options include:

- Automatic scaling ("a"), in which series are graphed using the default single scale. The default is left scale for most graphs, or left and bottom for XY graphs.
- Dual scaling without crossing ("d") scales the first series on the left and all other series on the right. The left and right scales will not overlap.
- Dual scaling with possible crossing ("x") is the same as the "d" option, but will allow the left and right scales to overlap.
- Normalized scaling ("n"), scales using zero mean and unit standard deviation.

For example, the commands:

```
graph g1.xyline(d) unemp gdp inv
show g1
```

create and display an XY line graph of the specified series with dual scales and no crossing.

The graph type options include:

- Mixed graph ("l") creates a single graph in which the first series is the selected graph type (bar, area, or spike) and all remaining series are line graphs.
- Multiple graph ("m") plots each graph in a separate frame.
- Stacked graph ("s") plots the cumulative addition of the series, so the value of a series is represented as the difference between the lines, bars, or areas.

For example, the commands:

```
group grp1 sales1 sales2
graph grsales.bar(s) grp1
show grsales
```

create a group GRP1 containing the series SALES1 and SALES2, then create and display a stacked bar graph GRSALES of the series in the group.

You should consult the command reference entry for each graph type for additional information, including a list of the available options (*i.e.*, see `bar` for complete details on bar graphs, and `line` for details on line graphs).

Merging graph objects

The `graph` command may also be used to merge existing named graph objects into a named multiple graph object. For example:

```
graph gr2.merge gr1 grsales
```

creates a multiple graph object GR2, combining two graph objects previously created.

Creating unnamed graph objects

There are two ways of creating an unnamed graph object. First, you may use the `freeze` command as described in "Creating graph objects from object views" on page 23.

As we have seen earlier you may also use any of the graph type keywords as a command ("Displaying graphs using commands" on page 21). Follow the keyword with any available options for that type, and a list of the objects to graph. EViews will create an unnamed graph of the specified type that is not stored in the workfile. For instance:

```
line(x) ser1 ser2 ser3
```

creates a line graph with series SER1 scaled on the left axis and series SER2 and SER3 scaled on the right axis.

If you later decide to name this graph, you may do so interactively by clicking on the **Name** button in the graph button bar. Alternatively, EViews will prompt you to name or delete any unnamed objects before closing the workfile.

Note that there is no way to name an unnamed graph object in a program. We recommend that you avoid creating unnamed graphs in programs since you will be unable to use the resulting object.

Changing Graph Types

You may change the graph type of a named graph object by following the object name with the desired graph type keyword and any options for that type. For example:

```
grsales.bar(l)
```

converts the bar graph GRSALES, created above, into a mixed bar-line graph, where SALES1 is plotted as a bar graph and SALES2 is plotted as a line graph within a single graph.

Note that specialized graphs, such as boxplots, place limitations on your ability to change the graph type. In general, your ability to customize the graph settings is more limited when changing graph types than when generating the graph from the original data.

Graph options are generally preserved when changing graph types. This includes attributes such as line color and axis assignment, as well as objects added to the graph, such as text labels, lines and shading. Commands to modify the appearance of named graph objects are described in "Customizing a Graph" on page 26.

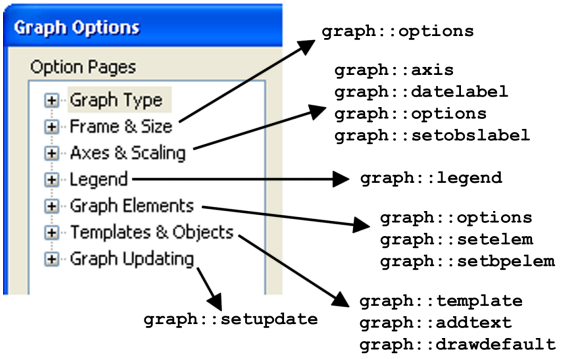
Note, however, that the line and fill graph settings are set independently. Line attributes apply to line and spike graphs, while fill attributes apply to bar, area, and pie graphs. For example, if you have modified the color of a line in a spike graph, this color will not be used for the fill area if the graph is changed to an area graph.

Customizing a Graph

EViews provides a wide range of tools for customizing the appearance of a named graph object. Nearly every display characteristic of the graph may be modified, including the appearance of lines and filled areas, legend characteristics and placement, frame size and attributes, and axis settings. In addition, you may add text labels, lines, and shading to the graph.

You may modify the appearance of a graph using dialogs or via the set of commands described below. Note that the commands are only available for graph objects since they take the form of graph procedures.

An overview of the relationship between the tabs of the graph dialog and the associated graph commands is illustrated below:



Line characteristics

For each data line in a graph, you may modify color, width, pattern and symbol using the `Graph::setelem` command. Follow the command keyword with an integer representing the data element in the graph you would like to modify, and one or more keywords for the characteristic you wish to change. List of symbol and pattern keywords, color keywords, and RGB settings are provided in `Graph::setelem`.

To modify line color and width you should use the `lcolor` and `lwidth` keywords:

```
graph gr1.line ser1 ser2 ser3
gr1.setelem(3) lcolor(orange) lwidth(2)
gr1.setelem(3) lcolor(255, 128, 0) lwidth(2)
```

The first command creates a line graph GR1 with colors and widths taken from the global defaults, while the latter two commands equivalently change the graph element for the third series to an orange line 2 points wide.

Each data line in a graph may be drawn with a line, symbols, or both line and symbols. The drawing default is given by the global options, but you may elect to add lines or symbols using the `lpattern` or `symbol` keywords.

To add circular symbols to the line for element 3, you may enter:

```
gr1.setelem(3) symbol(circle)
```

Note that this operation modifies the existing options for the symbols, but that the line type, color and width settings from the original graph will remain. To return to line only or symbol only in a graph in which both lines and symbols are displayed, you may turn off either symbols or patterns, respectively, by using the “none” type:

```
gr1.setelem(3) lpat(none)
```

or

```
gr1.setelem(3) symbol(none)
```

The first example removes the line from the drawing for the third series, so only the circular symbol is used. The second example removes the symbol, so only the line is used.

If you attempt to remove the lines or symbols from a graph element that contains only lines or symbols, respectively, the graph will change to show the opposite type. For example:

```
gr1.setelem(3) lpat(dash2) symbol(circle)
gr1.setelem(3) symbol(none)
gr1.setelem(3) lpat(none)
```

initially represents element 3 with both lines and symbols, then turns off symbols for element 3 so that it is displayed as lines only, and finally shows element 3 as symbols only, since the final command turns off lines in a line-only graph.

The examples above describe customization of the basic elements common to most graph types. "Modifying Boxplots" on page 40 provides additional discussion of `Graph::setelem` options for customizing boxplot data elements.

Use of color with lines and filled areas

By default, EViews automatically formats graphs to accommodate output in either color or black and white. When a graph is sent to a printer or saved to a file in black and white, EViews translates the colored lines and fills seen on the screen into an appropriate black and white representation. The black and white lines are drawn with line patterns, and fills are drawn with gray shading. Thus, the appearance of lines and fills on the screen may differ from what is printed in black and white (this color translation does not apply to boxplots).

You may override this auto choice display method by changing the global defaults for graphs. You may choose, for example, to display all lines and fills as patterns and gray shades, respectively, whether the graph uses color or not. All subsequently created graphs will use the new settings.

Alternatively, if you would like to override the color, line pattern, and fill settings for a given graph object, you may use the `Graph::options` graph proc.

Color

To change the color setting for an existing graph object, you should use `options` with the `color` keyword. If you wish to turn off color altogether for all lines and filled areas, you should precede the keyword with a negative sign, as in:

```
gr1.options -color
```

To turn on color, you may use the same command with the "-" omitted.

Lines and patterns

To always display solid lines in your graph, irrespective of the color setting, you should use `options` with the `linesolid` keyword. For example:

```
gr1.options linesolid
```

sets graph GR1 to use solid lines when rendering on the screen in color and when printing, even if the graph is printed in black and white. Note that this setting may make identification of individual lines difficult in a printed black and white graph, unless you change the widths or symbols associated with individual lines (see "Line characteristics" on page 27).

Conversely, you may use the `linepat` option to use patterned lines regardless of the color setting:

```
gr1.options linepat
```

One advantage of using the `linepat` option is that it allows you to see the pattern types that will be used in black and white printing without turning off color in your graph. For example, using the `Graph::setelem` command again, change the line pattern of the second series in GR1 to a dashed line:

```
gr1.setelem(2) lpat(dash1)
```

This command will not change the appearance of the colored lines on the screen if color is turned on and auto choice of line and fill type is set. Thus, the line will remain solid, and the pattern will not be visible until the graph is printed in black and white. To view the corresponding patterns, either turn off color so all lines are drawn as black patterned lines, or use the `linepat` setting to force patterns.

To reset the graph or to override modified global settings so that the graph uses auto choice, you may use the `lineauto` keyword:

```
gr1.options lineauto
```

This setting instructs the graph to use solid lines when drawing in color, and use line patterns and gray shades when drawing in black and white.

Note that regardless of the color or line pattern settings, you may always view the selected line patterns in the **Lines & Symbols** section of the graph options dialog. The dialog can be brought up interactively by double clicking anywhere in the graph.

Filled area characteristics

You can modify the color, gray shade, and hatch pattern of each filled area in a bar, area, or pie graph.

To modify these settings, use `Graph::setelem`, followed by an integer representing the data element in the graph you would like to modify, and a keyword for the characteristic you wish to change. For example, consider the commands:

```
graph mygraph.area(s) series1 series2 series3
mygraph.setelem(1) fcolor(blue) hatch(fdiagonal) gray(6)
mygraph.setelem(1) fcolor(0, 0, 255) hatch(fdiagonal) gray(6)
```

The first command creates MYGRAPH, a stacked area graph of SERIES1, SERIES2, and SERIES3. The latter two commands are equivalent, modifying the first series by setting its fill color to blue with a forward diagonal hatch. If MYGRAPH is viewed without color, the area will appear with a hatched gray shade of index 6.

See `Graph::setelem` for a list of available color keywords, and for gray shade indexes and available hatch keywords. Note that changes to gray shades will not be visible in the graph unless color is turned off.

Using preset lines and fills

For your convenience, EViews provides you with a collection of preset line and fill characteristics. Each line preset defines a color, width, pattern, and symbol for a line, and each fill preset defines a color, gray shade, and hatch pattern for a fill. There are thirty line and thirty fill presets.

The global graph options are initially set to use the EViews preset settings. These global options are used when you first create a graph, providing a different appearance for each line or fill. The first line preset is applied to the first data line, the second preset is applied to the second data line, and so on. If your graph contains more than thirty lines or fills, the presets are simply reused in order.

You may customize the graph defaults in the global **Graph Options** dialog. Your settings will replace the existing EViews defaults, and will be applied to all graphs created in the future.

EViews allows you to use either the original EViews presets, or those you have specified in the global **Graph Options** dialog when setting the characteristics of an existing graph. The keyword `preset` is used to indicate that you should use the set of options from the corresponding EViews preset; the keyword `default` is used to indicate that you should use the set of options from the corresponding global graph element defaults.

For example:

```
mygraph.setelem(2) preset(3)
```

allows the second fill area in MYGRAPH to use the original EViews presets for a third fill area. In current versions of EViews, these settings include a green fill, a medium gray shade of 8, and no hatch.

Alternatively:

```
mygraph.setelem(2) default(3)
```

also changes the second area of MYGRAPH, but uses the third set of user-defined presets. If you have not yet modified your global graph defaults, the two commands will yield identical results.

When using the `preset` or `default` keywords with boxplots, the line color of the specified preset will be applied to all boxes, whiskers, and staples in the graph. See "Modifying Boxplots" on page 40 for additional information.

Scaling and axes

There are four commands that may be used to modify the axis and scaling characteristics of your graphs:

- First, the `Graph::setelem` command with the `axis` keyword may be used to assign data elements to different axes.

- Second, the `Graph::axis` command can be used to customize the appearance of any axes in the graph object. You may employ the `axis` command to modify the scaling of the data itself, for example, as when you use a logarithmic scale, or to alter the scaling of the axis, as when you enable dual scaling. The `axis` command may also be used to change the appearance of axes, such as to modify tick marks, change the font size of axis labels, turn on grid or zero lines, or duplicate axes.
- Third, the `Graph::datelabel` command modifies the labeling of the bottom date/time axis in time plots. Use this command to change the way date labels are formatted or to specify label frequency.
- Finally, the `Graph::setobslabel` command may be used to create custom axis labels for the observation scale of a graph.

Assigning data to an axis

In most cases, when a graph is created, all data elements are initially assigned to the left axis. XY graphs differ slightly in that data elements are initially assigned to either the left or bottom axis.

Once a graph is created, individual elements may generally be assigned to either the left or right axis. In XY graphs, you may reassign individual elements to either the left, right, top, or bottom axis, while in boxplots or stacked time/observation graphs all data elements must be assigned to the same vertical axis.

To assign a data element to a different axis, use the `setelem` command with the `axis` keyword. For example, the commands:

```
graph graph02.line ser1 ser2
graph02.setelem(2) axis(right)
```

first create GRAPH02, a line graph of SER1 and SER2, and then turn GRAPH02 into a dual scaled graph by assigning the second data element, SER2, to the right axis.

In this example, GRAPH02 uses the default setting for dual scale graphs by disallowing crossing, so that the left and right scales do not overlap. To allow the scales to overlap, use the `axis` command with the `overlap` keyword, as in:

```
graph02.axis overlap
```

The left and right scales now span the entire axes, allowing the data lines to cross. To reverse this action and disallow crossing, use `-overlap`, (the `overlap` keyword preceded by a minus sign, "–").

For XY graphs without pairing, the first series is generally plotted along the bottom axis, and the remaining series are plotted on the left axis. XY graphs allow more manipulation than time/observation plots, because the top and bottom axes may also be assigned to an element. For example:

```
graph graph03.xyline s1 s2 s3 s4
graph03.setelem(1) axis(top)
graph03.setelem(2) axis(right)
```

first creates an XY line graph GRAPH03 of the series S1, S2, S3, and S4. The first series is then assigned to the top axis, and the second series is moved to the right axis. Note that the graph now uses three axes: top, left, and right.

Note that the element index in the `setelem` command is not necessary for boxplots and stacked time/observation graphs, since all data elements must be assigned to the same vertical axis.

While EViews allows dual scaling for the vertical axes in most graph types, the horizontal axes must use a single scale on either the top or bottom axis. When a new element is moved to or from one of the horizontal axes, EViews will, if necessary, reassign elements as required so that there is a single horizontal scale.

For example, using the graph created above, the command:

```
graph03.setelem(3) axis(bottom)
```

moves the third series to the bottom axis, forcing the first series to be reassigned from the top to the left axis. If you then issue the command:

```
graph03.setelem(3) axis(right)
```

EViews will assign the third series to the right axis as directed, with the first (next available element, starting with the first) series taking its place on the horizontal bottom axis. If the first element is subsequently moved to a vertical axis, the second element will take its place on the horizontal axis, and so forth. Note that series will never be reassigned to the right or top axis, so that series that placed on the top or right axis and subsequently reassigned will not be replaced automatically.

For XY graphs with pairing, the same principles apply. However, since the elements are graphed in pairs, there is a set of elements that should be assigned to the same horizontal axis. You can switch which set is assigned to the horizontal using the `axis` keyword. For example:

```
graph graph04.xypair s1 s2 s3 s4
graph03.setelem(1) axis(left)
```

creates an XY graph that plots the series S1 against S2, and S3 against S4. Usually, the default settings assign the first and third series to the bottom axis, and the second and fourth series to the left axis. The second command line moves the first series (S1) from the bottom to the left axis. Since S1 and S3 are tied to the same axis, the S3 series will also be assigned to the left axis. The second and fourth series (S2 and S4) will take their place on the bottom axis.

Modifying the data axis

The Graph::axis command may be used to change the way data is scaled on an axis. To rescale the data, specify the axis you wish to change and use one of the following keywords: linear, linearzero (linear with zero included in axis), log (logarithmic), norm (standardized). For example:

```
graph graph05.line ser1 ser2
graph05.axis(left) log
```

creates a line graph GRAPH05 of the series SER1 and SER2, and changes the left axis scaling method to logarithmic.

The interaction of the data scales (these are the left and right axes for non-XY graphs) can be controlled using axis with the overlap keyword. The overlap keyword controls the overlap of vertical scales, where each scale has at least one series assigned to it. For instance:

```
graph graph06.line s1 s2
graph06.setelem(2) axis(right)
graph06.axis overlap
```

first creates GRAPH06, a line graph of series S1 and S2, and assigns the second series to the right axis. The last command allows the vertical scales to overlap.

The axis command may also be used to change or invert the endpoints of the data scale, using the range or invert keywords:

```
graph05.axis(left) -invert range(minmax)
```

inverts the left scale of GRAPH05 ("-" indicates an inverted scale) and sets its endpoints to the minimum and maximum values of the data.

Modifying the date/time axis

EViews automatically determines an optimal set of labels for the bottom axis of time plots. If you wish to modify the frequency or date format of the labels, you should use the Graph::datelabel command. Alternately, to create editable labels on the observation scale, use the Graph::setobslabel command.

To control the number of observations between labels, use datelabel with the interval keyword to specify a desired step size. The stand-alone step size keywords include: auto (use EViews' default method for determining step size), ends (label first and last observations), and all (label every observation). For example,

```
mygraph.datelabel interval(ends)
```

labels only the endpoints of MYGRAPH. You may also use a step size keyword in conjunction with a step number to further control the labeling. These step size keywords include:

`obs` (one observation), `year` (one year), `m` (one month), and `q` (one quarter), where each keyword determines the units of the number specified in the step keyword. For example, to label every ten years, you may specify:

```
mygraph.datelabel interval(year, 10)
```

In addition to specifying the space between labels, you may indicate a specific observation to receive a label. The step increment will then center around this observation. For example:

```
mygraph.datelabel interval(obs, 10, 25)
```

labels every tenth observation, centered around the twenty-fifth observation.

You may also use `datelabel` to modify the format of the dates or change their placement on the axis. Using the `format` or `span` keywords,

```
mygraph02.datelabel format(yy) -span
```

formats the labels so that they display as two digit years, and disables interval spanning. If interval spanning is enabled, labels will be centered between the applicable tick marks. If spanning is disabled, labels are placed directly on the tick marks. For instance, in a plot of monthly data with annual labeling, the labels may be centered over the twelve monthly ticks (spanning enabled) or placed on the annual tick marks (spanning disabled).

If your axis labels require further customization, you may use the `setobslabel` command to create a set of custom labels.

```
mygraph.setobslabel(current) "CA" "OR" "WA"
```

creates a set of axis labels, initializing each with the date or observation number and assigns the labels “CA”, “OR”, and “WA” to the first three observations.

To return to EViews automatic labeling, you may use the `clear` option:

```
mygraph.setobslabel(clear)
```

Customizing axis appearance

You may customize the appearance of tick marks, modify label font size, add grid lines, or duplicate axes labeling in your graph using `Graph::axis`.

Follow the `axis` keyword with a descriptor of the axis you wish to modify and one or more arguments. For instance, using the `ticksin`, `minor`, and `font` keywords:

```
mygraph.axis(left) ticksin -minor font(10)
```

The left axis of MYGRAPH is now drawn with the tick marks inside the graph, no minor ticks, and a label font size of 10 point.

To add lines to a graph, use the `grid` or `zeroline` keywords:

```
mygraph01.axis(left) -label grid zeroline
```

MYGRAPH01 hides the labels on its left axis, draws horizontal grid lines at the major ticks, and draws a line through zero on the left scale.

In single scale graphs, it is sometimes desirable to display the axis labels on both the left and right hand sides of the graph. The `mirror` keyword may be used to turn on or off the display of duplicate axes. For example:

```
graph graph06.line s1 s2
graph06.axis mirror
```

creates a line graph with both series assigned to the left axis (the default assignment), then turns on mirroring of the left axis to the right axis of the graph. Note that in the latter command, you need not specify an axis to modify, since mirroring sets both the left and right axes to be the same.

If dual scaling is enabled, mirroring will be overridden. In our example, assigning a data element to the right axis:

```
graph06.setelem(1) axis(right)
```

will override axis mirroring. Note that if element 1 is subsequently reassigned to the left scale, mirroring will again be enabled. To turn off mirroring entirely, simply precede the mirror keyword with a minus sign. The command:

```
graph06.axis -mirror
```

turns off axis mirroring.

Customizing the graph frame

The graph frame is used to set the basic graph proportions and display characteristics that are not part of the main portion of the graph.

Graph size

The graph frame size and proportions may be modified using the `Graph::options` command. Simply specify a width and height using the `size` keyword. For example:

```
testgraph.options size(5,4)
```

resizes the frame of TESTGRAPH to 5×4 virtual inches.

Other frame characteristics

The `Graph::options` command may also be used to modify the appearance of the graph area and the graph frame. A variety of modifications are possible.

First, you may change the background colors in your graph, by using the “fillcolor” and “backcolor” keywords to change the frame fill color and the graph background color, respectively. The graph proc command:

```
testgraph.options fillcolor(gray) backcolor(white)
```

fills the graph frame with gray, and sets the graph area background color to white. Here we use the predefined color settings ("blue," "red," "green," "black," "white," "purple," "orange," "yellow," "gray," "ltgray"); alternately, you may specify "color" with three arguments corresponding to the respective RGB settings.

You may control display of axes frames. To select which axes should have a frame, you should use the "frameaxes" keyword:

```
testgraph.options frameaxes(labeled)
```

which turns off the frame on any axis which is not associated with data. Similarly:

```
testgraph.options frameaxes(lb)
```

draws a frame on the left and bottom axes only.

By default, EViews uses the entire width of the graph for plotting data. If you wish to indent the data display from the edges of the graph frame, you should use the "indenth" (indent horizontal) or "indentv" (indent vertical) keywords:

```
testgraph.options indenth(.05) indentv(0.1)
```

indents the data by 0.05 inches horizontally, and 0.10 inches vertically from the edge of the graph frame.

The options command also allows you to add and modify grid lines in your graph. For example:

```
testgraph.options gridb -gridl gridpat(dash2) gridcolor(red)
```

turns on dashed, red, vertical gridlines from the bottom axis, while turning off left scale gridlines.

Labeling data values

Bar and pie graphs allow you to label the value of your data within the graph. Use the `Graph::options` command with one of the following keywords: `barlabelabove`, `barlabelinside`, or `pielabel`. For example:

```
mybargraph.options barlabelabove
```

places a label above each bar in the graph indicating its data value. Note that the label will be visible only when there is sufficient space in the graph.

Outlining and spacing filled areas

EViews draws a black outline around each bar or area in a bar or area graph, respectively. To disable the outline, use `options` with the `outlinebars` or `outlineareas` keyword:

```
mybargraph.options -outlinebars
```

Disabling the outline is useful for graphs whose bars are spaced closely together, enabling you to see the fill color instead of an abundance of black outlines.

EViews attempts to place a space between each bar in a bar graph. This space disappears as the number of bars increases. You may remove the space between bars by using the `barspace` keyword:

```
mybargraph.options -barspace
```

Modifying the Legend

A legend's location, text, and appearance may be customized. Note that single series graphs and special graph types such as boxplots and histograms use text objects for labeling instead of a legend. These text objects may only be modified interactively by double-clicking on the object to bring up the text edit dialog.

To change the text string of a data element for use in the legend, use the `Graph::name` command:

```
graph graph06.line ser1 ser2
graph06.name(1) Unemployment
graph06.name(2) DMR
```

The first line creates a line graph GRAPH06 of the series SER1 and SER2. Initially, the legend shows “SER1” and “SER2”. The second and third command lines change the text in the legend to “Unemployment” and “DMR”.

Note that the `name` command is equivalent to using the `Graph::setelem` command with the `legend` keyword. For instance,

```
graph06.setelem(1) legend(Unemployment)
graph06.setelem(2) legend(DMR)
```

produces the same results.

To remove a label from the legend, you may use `name` without providing a text string:

```
graph06.name(2)
```

removes the second label “DMR” from the legend.

For an XY graph, the `name` command modifies any data elements that appear as axis labels, in addition to legend text. For example:

```
graph xygraph.xy ser1 ser2 ser3 ser4
xygraph.name(1) Age
xygraph.name(2) Height
```

creates an XY graph named XYGRAPH of the four series SER1, SER2, SER3, and SER4. “SER1” appears as a horizontal axis label, while “SER2,” “SER3,” and “SER4” appear in the

legend. The second command line changes the horizontal label of the first series to "Age". The third line changes the second series label in the legend to "Height".

To modify characteristics of the legend itself, use `Graph::legend`. Some of the primary options may be set using the `inbox`, `position` and `columns` keywords. Consider, for example, the commands:

```
graph graph07.line s1 s2 s3 s4
graph07.legend -inbox position(botleft) columns(4)
```

The first line creates a line graph of the four series S1, S2, S3, and S4. The second line removes the box around the legend, positions the legend in the bottom left corner of the graph window, and specifies that four columns should be used for the text strings of the legend.

When a graph is created, EViews automatically determines a suitable number of columns for the legend. A graph with four series, such as the one created above, would likely display two columns of two labels each. The `columns` command above, with an argument of four, creates a long and slender legend, with each of the four series in its own column.

You may also use the `legend` command to change the font size or to disable the legend completely:

```
graph07.legend font(10)
graph07.legend -display
```

Note that if the legend is hidden, any changes to the text or position of the legend remain, and will reappear if the legend is displayed again.

Adding text to the graph

Text strings can be placed anywhere within the graph window. Using the `Graph::addtext` command:

```
graph07.addtext(t) Fig 1: Monthly GDP
```

adds the text "Fig 1: Monthly GDP" to the top of the GRAPH07 window. You can also use specific coordinates to specify the position of the upper left corner of the text. For example:

```
graph08.addtext(.2, .1, x) Figure 1
```

adds the text string "Figure 1" to GRAPH08. The text is placed 0.2 virtual inches in, and 0.1 virtual inches down from the top left corner of the graph frame. The "x" option instructs EViews to place the text inside a box.

An existing text object can be edited interactively by double-clicking on the object to bring up a text edit dialog. The object may be repositioned by specifying new coordinates in the dialog, or by simply dragging the object to its desired location.

Adding lines and shading

You may wish to highlight or separate specific areas of your graph by adding a line or shaded area to the interior of the graph using the `Graph::draw` command. Specify the type of line or shade option (`line` or `shade`), which axis it should be attached to (`left`, `right`, `bottom`, `top`) and its position. For example:

```
graph09.draw(line, left) 5.2
```

draws a horizontal line at the value 5.2 on the left axis. Alternately:

```
graph09.draw(shade, left) 4.8 5.6
```

draws a shaded horizontal area bounded by the values 4.8 and 5.6 on the left axis. You can also specify `color`, line `width`, and line `pattern`:

```
graph09.draw(line, bottom, color(blue), width(2), pattern(3))
    1985:1
```

draws a vertical blue dashed line of width two points at the date "1985:1" on the bottom axis. Color may be specified using one or more of the following options: `color(`*n1*, *n2*, *n3*`)`, where the arguments correspond to RGB settings, or `color(`*keyword*`)`, where *keyword* is one of the predefined color keywords ("blue", "red", "green", "black", "white", "purple", "orange", "yellow", "gray", "ltgray").

Using graphs as templates

After customizing a graph as described above, you may wish to use your custom settings in another graph. Using a graph template allows you to copy the graph type, line and fill settings, axis scaling, legend attributes, and frame settings of one graph into another. This enables a graph to adopt all characteristics of another graph—everything but the data itself. To copy custom line or fill settings from the global graph options, use the `preset` or `default` keywords of the `Graph::setelem` command (as described in "Using preset lines and fills" on page 30).

Modifying an existing graph

To modify a named graph object, use the `template` command:

```
graph10.template customgraph
```

This command copies all the appearance attributes of CUSTOMGRAPH into GRAPH10.

To copy text labels, lines and shading in the template graph in addition to all other option settings, use the "t" option:

```
graph10.template(t) customgraph
```

This command copies any text or shading objects that were added to the template graph using the `Graph::addtext` or `Graph::draw` commands or the equivalent steps using dia-

logs. Note that using the "t" option overwrites any existing text and shading objects in the target graph.

Using a template during graph creation

All graph type commands also provide a template option for use when creating a new graph. For instance:

```
graph mygraph.line(o = customgraph) ser1 ser2
```

creates the graph MYGRAPH of the series SER1 and SER2, using CUSTOMGRAPH as a template. The "o" option instructs EViews to copy all but the text, lines, and shading of the template graph. To include these elements in the copy, use the "t" option in place of the "o" option.

When used as a graph procedure, this method is equivalent to the one described above for an existing graph, so that:

```
graph10.template(t) customgraph
graph10.bar(t = customgraph)
```

produce the same results.

Arranging multiple graphs

When you create a multiple graph, EViews automatically arranges the graphs within the graph window. (See "Creating a Graph" on page 21 for information on how to create a multiple graph.) You may use either the "m" option during graph creation or the `merge` command.

To change the placement of the graphs, use the `Graph::align` command. Specify the number of columns in which to place the graphs and the horizontal and vertical space between graphs, measured in virtual inches. For example:

```
graph graph11.merge graph01 graph02 graph03
graph11.align(2, 1, 1.5)
```

creates a multiple graph GRAPH11 of the graphs GRAPH01, GRAPH02, and GRAPH03. By default, the graphs are stacked in one column. The second command realigns the graphs in two columns, with 1 virtual inch between the graphs horizontally and 1.5 virtual inches between the graphs vertically.

Modifying Boxplots

The appearance of boxplots can be customized using many of the commands described above. A few special cases and additional commands are described below.

Customizing lines and symbols

As with other graph types, the `setelem` command can be used with boxplots to modify line and symbol attributes, assign the boxes to an axis, and use preset and default settings. To use the `Graph::setelem` command with boxplots, use a box element keyword after the command. For example:

```
boxgraph01.setelem(mean) symbol(circle)
```

changes the means in the boxplot BOXGRAPH01 to circles. Note that all boxes within a single graph have the same attributes, and changes to appearance are applied to all boxes. For instance:

```
boxgraph01.setelem(box) lcolor(orange) lpat(dash1) lwidth(2)
```

plots all boxes in BOXGRAPH01 with an orange dashed line of width 2 points. Also note that when shaded confidence intervals are used, a lightened version of the box color will be used for the shading. In this way, the above command also changes the confidence interval shading to a light orange.

Each element in a boxplot is represented by either a line or symbol. EViews will warn you if you attempt to modify an inappropriate option (*e.g.*, modifying the symbol of the box).

Assigning boxes to an axis

The `setelem` command may also be used to assign the boxes to another axis:

```
boxgraph01.setelem axis(right)
```

Note that since all boxes are assigned to the same axis, the index argument specifying a graph element is not necessary.

Using preset line colors

During general graph creation, lines and fills take on the characteristics of the user-defined presets. When a boxplot is created, the first user-defined line color is applied to the boxes, whiskers, and staples. Similarly, when you use the `preset` or `default` keywords of the `setelem` command with a boxplot, the line color of the preset is applied to the boxes, whiskers, and staples. (See "Using preset lines and fills" on page 30 for a description of presets.)

The `preset` and `default` methods work just as they do for other graph types, although only the line color is applied to the graph. For example:

```
boxgraph01.setelem default(3)
```

applies the line color of the third user-defined line preset to the boxes, whiskers, and staples of BOXGRAPH01. Note again that `setelem` does not require an argument specifying an index, since the selected preset will apply to all boxes.

There are a number of `setelem` arguments that do not apply to boxplots. The `fillcolor`, `fillgray`, and `fillhatch` option keywords are not available, as there are no custom areas to be filled. The `legend` keyword is also not applicable, as boxplots use axis text labels in place of a legend.

Hiding boxplot elements

In addition to the `setelem` command, boxplots provide a `Graph::setbpelem` command for use in enabling or disabling specific box elements. Any element of the boxplot can be hidden, except the box itself. Use the command with a list of box elements to show or hide. For example:

```
boxgraph01.setbpelem -mean far
```

hides the means and confirms that the far outliers are shown in BOXGRAPH01.

Modifying box width and confidence intervals

The width of the individual boxes in a boxplot can be drawn in three ways: fixed width over all boxes, proportional to the sample size, or proportional to the square root of the sample size. To specify one of these methods, use the `setbpelem` command with the `width` keyword, and one of the supported types (`fixed`, `rootn`, `n`). For example:

```
boxgraph01.setbpelem width(rootn)
```

draws the boxes in BOXGRAPH01 with widths proportional to the square root of their sample size.

There are three methods for displaying the confidence intervals in boxplots. They may be notched, shaded, or not drawn at all, which you may specify using one of the supported keywords (`notch`, `shade`, `none`). For example:

```
boxgraph01.setbpelem ci(notch)
```

draws the confidence intervals in BOXGRAPH01 as notches.

Labeling Graphs

As with all EViews objects, graphs have a label view to display and edit information such as the graph name, last modified date, and remarks. To modify or view the label information, use the `Graph::label` command:

```
graph12.label(r) Data from CPS 1988 March File
```

This command shows the label view, and the “r” option appends the text “Data from CPS 1988 March File” to the remarks field of GRAPH12.

To return to the graph view, use the `graph` keyword:

```
graph12.graph
```

All changes made in label view will be saved when the graph is saved.

Printing Graphs

A graph may be printed using the `print` (p. 414) command. For example:

```
print graph11 graph12
```

prints GRAPH11 and GRAPH12 on a single page.

In addition, many graph commands and graph views of objects include a print option. For example, you can create and simultaneously print a line graph GRA1 of SER1 using the "p" option:

```
graph gra1.line(p) ser1
```

You should check the individual commands for availability of this option.

Exporting Graphs to Files

You may use the `Graph::save` proc of a graph object to save the graph as a Windows metafile (.wmf), Enhanced Windows metafile (.emf), PostScript file (.eps), bitmap (.bmp), Graphics Interchange Format (.gif), Joint Photographics Exchange Group (.jpg), or Portable Network Graphics (.png) file.

You must specify a file name and file type, and may also provide the file height, width, units of measurement, and color use. PostScript files also allow you to save the graph with or without a bounding box and to specify portrait or landscape orientation. For instance:

```
graph11.save(t=postscript, u=cm, w=12, -box) MyGraph1
```

saves GRAPH11 in the default directory as a PostScript file "MyGraph1.EPS", with a width of 12 cm and no bounding box. The height is determined by holding the aspect ratio of the graph constant. Similarly:

```
graph11.save(t=emf, u=pts, w=300, h=300, -c) c:\data\MyGraph2
```

saves GRAPH11 as an Enhanced Windows metafile "Mygraph2.EMF". The graph is saved in black and white, and scaled to 300×300 points.

```
graph11.save(t=png, u=in, w=5, d=300) MyGraph3
```

saves GRAPH11 in the default directory as a PNG file "Mygra3.PNG". The image will be 5 inches wide at 300 dpi.

```
graph11.save(t=gif, u=pixels, w=500) MyGraph4
```

saves GRAPH11 in a 500 pixel wide GIF file, "Mygraph4.GIF".

Graph Summary

See "Graph" on page 208 of the *Object Reference* for a full listing of procs that may be used to customize graph objects, and for a list of the graph type commands.

Graph commands are documented in "Graph Creation Command Summary" on page 799 of the *Object Reference*.

Chapter 3. Working with Tables and Spreadsheets

There are three types of tables in EViews: tabular views, which are tables used in the display of views of other objects, named table objects, and unnamed table objects. The main portion of this discussion focuses on the use of commands to customize the appearance of named table objects. The latter portion of the chapter describes the set of tools that may be used to customize the display characteristics of spreadsheet views of objects (see "Customizing Spreadsheet Views," beginning on page 55).

You may use EViews commands to generate custom tables of formatted output from your programs. A *table object* 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.

Chapter 16. "Table and Text Objects," on page 691 of the *Object Reference* describes various interactive tools for customizing table views and objects.

Creating a Table

There are two basic ways to create a table object: by freezing an object view, or by issuing a table declaration.

Creating Tables from an Object View

You may create a table object from another object, by combining an object view command with the `freeze` (p. 339) command. Simply follow the `freeze` keyword with an optional name for the table object, and the tabular view to be frozen. For example, since the command

```
grp6.stats
```

displays the statistics view of the group GRP6, the command

```
freeze(mytab) grp6.stats
```

creates and displays a table object MYTAB containing the contents of the previous view.

You should avoid creating unnamed tables when using commands in programs since you will be unable to refer to, or work with the resulting object using commands. If the MYTAB option were omitted in the previous example, EViews would create and display an untitled table object. This table object may be customized interactively, but may not be referred to in programs. You may, of course, assign a name to the table interactively.

Once you have created a named table object, you may use the various table object procs to further customize the appearance of your table. See "Customizing Tables," beginning on page 48.

Declaring Tables

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. Re-declaring the table 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 matrix objects which issue an error when an out-of-range element is accessed.

Assigning Table Values

You may modify the contents of cells in a table using assignment statements. Each cell of the table can be assigned either a string or a numeric value.

Assigning Strings

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

For example:

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

creates the table BESTRES and places various string values into cells of the table.

Assigning Numbers

Numbers can be entered directly into cells, or they can be converted to strings before being placed in the table.

Unless there is a good reason to do otherwise, we recommend that numbers be entered directly into table cells. 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 to a different precision.

For example:

```
table tab1
tab1(3,4) = 15.345
tab1(4,2) = 1e-5
!ev = 10
tab1(5,1) = !ev
scalar f = 12345.67
tab1(6,2) = f
```

creates the table TAB1 and assigns numbers to various cells.

Assignment with Formatting

The `setcell` (p. 429) command is like direct cell assignment in that it allows you to set the contents of a cell, but `setcell` also allows you to provide a set of simple formatting options for the cell. If you desire even greater control over formatting, or if you wish to alter the format of a cell without altering its contents, you should use the tools outlined in "Customizing Tables," beginning on page 48.

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
- (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 sci-

entific 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.

Customizing Tables

EViews provides considerable control over the appearance of table objects, providing a variety of table procedures allowing you specify row heights and column widths, content formatting, justification, font face, size, and color, cell background color and borders. Cell merging and annotation are also supported.

Column Width and Row Height

We begin by noting that if the contents of a cell are wider or taller than the display width or height of the cell, part of the cell contents may not be visible. You may use the `Table::setwidth` and `Table::setheight` table procedures to change the dimensions of a column or row of table cells.

To change the column widths for a set of columns in a table, use the `setwidth` keyword followed by a column range specification in parentheses, and a desired width.

The column range should be either a single column number or letter (*e.g.*, “5”, “E”), a colon delimited range of columns (from low to high, *e.g.*, “3:5”, “C:E”), or the keyword “@ALL”. The width unit is computed from representative characters in the default font for the current table (the EViews table default font at the time the table was created), and corresponds

roughly to a single character. Width values may be non-integer values with resolution up to 1/10 of a unit. The default width value for columns in an unmodified table is 10.

For example, both commands

```
tab1.setwidth(2) 12
tab1.setwidth(B) 12
```

set the width of column 2 to 12 width units, while the command

```
tab1.setwidth(2:10) 20
```

sets the widths for columns 2 through 10 to 20 width units. To set all of the column widths, use the "@ALL" keyword.

```
tab1.setwidth(@all) 20
```

Similarly, you may specify row heights using the `setheight` keyword, followed by a row specification in parentheses, and a desired row height.

Rows are specified either as a single row number (*e.g.*, "5"), as a colon delimited range of rows (from low to high, *e.g.*, "3:5"), or using the keyword "@ALL". Row heights are given in height unit values, where height units are in character heights. The character height is given by the font-specific sum of the units above and below the baseline and the leading in the default font for the current table. Height values may be non-integer values with resolution up to 1/10 of a height unit. The default row height value is 1.

For example,

```
tab1.setheight(2) 1
```

sets the height of row 2 to match the table default font character height, while

```
tab1.setheight(2) 3.5
```

increases the row height to 3-1/2 character heights.

Similarly, the command:

```
tab1.setheight(2:7) 1.5
```

sets the heights for rows 2 through 7 to 1-1/2 character heights.

```
tab1.setheight(@all) 2
```

sets all row heights to twice the default height.

Earlier versions of EViews supported the setting of column widths using the `setcolwidth` command. This command, which is provided for backward compatibility, offers only a subset of the capabilities of `Table::setwidth`.

Cell Formatting

A host of cell characteristics may be set using table procedures. Each procedure is designed to work on individual cells, ranges of cells, or the entire table.

Content Formatting

Cell content formatting allows you to alter the appearance of the data in a table cell without changing the contents of the cell. Using the table proc `Table::setformat`, you may, for example, instruct EViews to change the format of a number to scientific or fixed decimal, or to display a date number in a different date format. These changes in display format do not alter the cell values.

To format the contents of table cells, simply follow the table name with a period and the `setformat` proc keyword, followed by a cell range specification in parentheses, and then a valid numeric or date format string. The cell range may be specified in a number of ways, including individual cells, cell rectangles, row or column ranges or the entire table. See `Table::setformat` for a description of cell range specification and numeric and date format string syntax.

For example, to set the format for the fifth column of a matrix to fixed 5-digit precision, you may provide the format specification:

```
tab1.setformat(e) f.5
```

To set a format for the cell in the third row of the fifth column to scientific notation with 5 digits of precision, specify the individual cell, as in:

```
tab1.setformat(3,e) e.5
tab1.setformat(e3) e.5
```

To specify the format for a rectangle of cells, specify the upper left and lower right cells in the rectangle. The following commands set cells in the same region to show 3-significant digits, with negative numbers in parentheses:

```
tab1.setformat(2,B,10,D) (g.3)
tab1.setformat(r2c2:r10c4) (g.3)
tab1.setformat(b2:d10) (g.3)
```

The rectangle of cells is delimited by row 2, column 2, and row 10, column 4.

Alternately you may provide a date format for the table cells. The command:

```
tab1.setformat(@all) "dd/MM/YY HH:MI:SS.SSS"
```

will display numeric values in the entire table using formatted date strings containing days followed by months, years, hours, minutes and seconds, to a resolution of thousandths of a second.

Note that changing the display format of a cell that contains a string will have no effect unless the cell is later changed to contain a numerical value.

Justification and Indentation

The cell justification and indentation control the position of the table cell contents within the table cell itself.

You may use the `Table::setjust` proc to position the cell contents in the cell. Simply use the `setjust` keyword, followed by a cell range specification in parentheses, and one or more keywords describing a vertical or horizontal position for the cell contents. You may use the keywords `auto`, `left`, `right`, and `center` to control horizontal positioning, and `top`, `middle`, and `bottom` to control vertical positioning. You may use the `auto` keyword to specify left justification for string cells and right justification for numeric cells.

For example,

```
tab1.setjust(@all) top left
```

sets the justification for all cells in the table to top left, while

```
tab1.setjust(2,B,10,D) center
```

horizontally centers the cell contents in the rectangle from B2 to D10, while leaving the vertical justification unchanged.

In addition, you may use `Table::setindent` to specify a left or right indentation from the edge of the cell for cells that are left or right justified, respectively. You should use the `setindent` keyword followed by a cell range in parentheses, and an indentation unit, specified in 1/5 of a width unit. Indentation is only relevant for non-center justified cells.

For example:

```
tab1.setjust(2,B,10,D) left
tab1.indent(2,B,10,D) 2
```

left-justifies, then indents the specified cells by 2/5 of a width unit from the left-hand side of the cell.

Alternatively,

```
tab2.setjust(@all) center
tab2.indent(@all) 3
```

will set the indentation for all cells in the table to 3/5 of a width unit, but this will have no effect on the center justified cells. If the cells are later modified to be left or right justified, the indentation will be used. If you subsequently issue the command

```
tab2.indent(@all) right
```

the cells will be indented 3/5 of a width unit from the right-hand edges.

Fonts

You may specify font face and characteristics, and the font color for table cells using the Table::setfont and Table::settextcolor table procs.

The setfont proc should be used to set the font face, size, boldface, italic, strikethrough and underline characteristics for table cells. You should provide a cell range specification, and one or more font arguments corresponding to font characteristics that you wish to modify. For example:

```
tab1.setfont(3,B,10,D) "Times New Roman" +u 8pt
```

changes the text in the specified cells to Times New Roman, 8 point, underline. Similarly,

```
tab1.setfont(4,B) -b +i -s
```

adds the italic to and removes boldface and strikethrough from the B4 cell.

To set the color of your text, use settextcolor with a cell range specification and a color specification. Color specifications may be provided using the @RGB settings, or using one of the EViews predefined colors keywords:

```
tab1.settextcolor(f2:g10) @rgb(255, 128, 0)
tab1.settextcolor(f2:g10) orange
```

sets the text color for the specified cells to orange. See Table::setfillcolor for a complete description of color specifications.

Background and Borders

You may set the background color for cells using the Table::setfillcolor table procedure. Specify the cell range and provide a color specification using @RGB settings or one of the predefined color keywords. The commands:

```
tab1.setfillcolor(R2C3:R3C6) ltgray
tab1.setfillcolor(2,C,3,F) @rgb(192, 192, 192)
```

both set the background color of the specified cells to light gray.

The Table::setlines table proc may be used to draw borders or lines around specified table cells. If a single cell is specified, you may draw borders around the cell or a double line through the center of the cell. If multiple columns or rows is selected, you may, in addition, add borders between cells.

Follow the name of the table object with a period, the setlines keyword, a cell range specification, and one or more line arguments describing the lines and borders you wish to draw. For example:

```
tab1.setlines(b2:d6) +a -h -v
```

first adds all borders ("a") to the cells in the rectangle defined by B2 and D6, then removes the inner horizontal ("h"), and inner vertical ("v") borders. The command

```
tab1.setlines(2,b) +o
```

adds borders to the outside ("o"), all four borders, of the B2 cell.

You may also use the `setlines` command to place double horizontal separator lines in the table. Enter the `setlines` keyword, followed by the name of the table, and a row number, both in parentheses. For example,

```
bestres.setlines(8) +d
```

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

```
bestres.setlines(8) -d
```

removes the double separator lines from all of the cells in the eighth row of the table.

Cell Annotation and Merging

Each cell in a table object is capable of containing a comment. Cell comments contain text that is hidden until the mouse cursor is placed over the cell containing the comment. Comments are useful for adding notes to a table without changing the appearance of the table.

To add a comment with the `Table::comment` table proc, follow the name of the table object with a period, a single cell identifier (in parentheses), and the comment text enclosed in double quotes. If no comment text is provided, a previously defined comment will be removed.

To add a comment "hello world" to the cell in the second row, fourth column, you may use the command:

```
tab1.comment(d2) "hello world"
```

To remove the comment simply repeat the command, omitting the text:

```
tab1.comment(d2)
```

In addition, EViews permits you to merge cells horizontally in a table object. To merge multiple cells in a row or to un-merge previously merged cells, you should use the `Table::setmerge` table proc. Enter the name of the table object, a period, followed by a cell range describing the cells in a single row that are to be merged.

If the first specified column is less than the last specified column (left specified before right), the cells in the row will be merged left to right, otherwise, the cells will be merged from right to left. The contents of the merged cell will be taken from the first cell in the merged region. If merging from left to right, the leftmost cell contents will be used; if merging from right to left, the rightmost cell contents will be displayed.

For example,

```
tab1.setmerge(a2:d2)
```

merges the cells in row 2, columns 1 to 4, from left to right, while

```
tab2.setmerge(d2:a2)
```

merges the cells in row 2, columns 2 to 5, from right to left. The cell display will use the leftmost cell in the first example, and the rightmost in the second.

If you specify a merge involving previously merged cells, EViews will unmerge all cells within the specified range. We may then unmerge cells by issuing the `Table::setmerge` command using any of the previously merged cells. The command:

```
tab2.setmerge(r2c4)
```

unmerges the previously merged cells.

Labeling Tables

Tables have a label view to display and edit information such as the graph name, last modified date, and remarks. To modify or view the label information, use the `Table::label` command:

```
table11.label(r) Results from GMM estimation
```

This command shows the label view, and the "r" option appends the text "Results from GMM estimation" to the remarks field of TABLE11.

To return to the basic table view, use the `table` keyword:

```
table11.table
```

All changes made in label view will be saved with the table.

Printing Tables

To print a table, use the `print` (p. 414) command, followed by the table object name. For example:

```
print table11
```

The `print` destination is taken from the EViews global print settings.

Exporting Tables to Files

You may use the table `Table::save` procedure to save the table to disk as a CSV (comma separated file), tab-delimited ASCII text, RTF (Rich text format), or HTML file.

You must specify a file name and an optional file type, and may also provide options to specify the cells to be saved, text to be written for NA values, and precision with which numbers

should be written. RTF and HTML files also allow you to save the table in a different size than the current display. If a file type is not provided, EViews will write a CSV file.

For example:

```
tab1.save(t=csv, n="NAN") mytable
```

saves TAB1 in the default directory as a CSV file "Mytable.CSV", with NA values translated to the text "NAN".

Alternately, the command:

```
tab1.save(r=B2:C10, t=html, s=.5) c:\data\MyTab2
```

saves the specified cells in TAB1 as an HTML file to "Mytab2.HTM" in the directory "c:\data". The table is saved at half of the display size.

Customizing Spreadsheet Views

Several of the table procs for customizing table display may also be used for customizing spreadsheet views of objects. You may use `Series::setformat`, `Series::setindent`, `Series::setjust`, and `Series::setwidth` to modify the spreadsheet view of a series. Similar procs are available for other objects with table views (*e.g.*, alpha, group, and matrix objects).

Suppose, for example, that you wish to set the format of the spreadsheet view for series SER1. Then the commands:

```
ser1.setformat f.5
ser1.setjust right center
ser1.setindent 3
ser1.setwidth 10
ser1.sheet
```

sets the spreadsheet display format for SER1 and then displays the view.

Similarly, you may set the characteristics for a matrix object using the commands:

```
mat1.setformat f.6
mat1.setwidth 8
mat1.sheet
```

For group spreadsheet formatting, you must specify a column range specification. For example:

```
group1.setformat(2) (f.7)
group1.setwidth(2) 10
group1.setindent(b) 6
group1.sheet
```

set the formats for the second series in the group, then displays the spreadsheet view.

```
group1.setwidth(@all) 10
```

sets the width for all columns in the group spreadsheet to 10.

Note that the group specified formats are used only to display series in the group and are not exported to the underlying series. Thus, if MYSER is the second series in GROUP1, the spreadsheet view of MYSER will use the original series settings, not those specified using the group procs.

Table Summary

See "Table," on page 688 of the *Object Reference* for a full listing of formatting procs that may be used with table objects.

Chapter 4. Working with Spools

The EViews spool object allows you to create sets of output comprised of tables, graphs, text, and other spool objects. Spools allow you to organize EViews results, allowing you to generate a log of output for a project, or perhaps to collect output for a presentation.

The following discussion focuses on command methods for working with a spool object. A general description of the spool object, featuring a discussion of interactive approaches to working with your spool, may be found in Chapter 17. "Spool Objects," on page 703 of *User's Guide I*.

Creating a Spool

There are two methods you may use to create a spool. You may declare a spool using the `spool` command, or you may print an object to a new spool.

To declare an empty spool, use the keyword `spool` followed by a name for the new spool:

```
spool myNewSpool
```

creates a new, empty spool object MYNEWSPOOL.

A new spool may also be created by printing from an object to a non-existent spool. To print to a spool you must redirect the output of print jobs to the spool using the `output` command. For example, the command:

```
output(s) myNewSpool
```

instructs EViews to send all subsequent print jobs to the MYNEWSPOOL spool (see `output` (p. 387)).

Once you redirect your output, you may create a spool using the print command or the "p" option of an object view or procedure.

```
tab1.print
```

creates the spool object MYNEWSPOOL and appends a copy of TAB1. Alternately,

```
eq1.output(p)
```

appends the EQ1 equation output to the newly created spool object.

To turn off redirection, simply issue the command

```
output off
```

Working with a Spool

Spool objects provide easy-to-use tools for working with the objects in the spool. Among other things, you may manage (add, delete, extract, rearrange, hide) or customize (resize, space and indent, title and comment, and edit) the spool and the individual objects in a spool.

Adding Objects

You may add objects to a spool by printing to the spool, or by using the `Spool::append` and `Spool::insert` procs.

Printing to a Spool

Earlier, we saw how one may redirect subsequent print jobs to the spool object using the `output` (p. 387) command to change the default print destination. Once redirection is in place, simply use the `print` command or the "p" option to send view or procedure output to the spool. The following command lines:

```
output(s) myOldSpool
ser01.line(p)
grp01.scat(p)
eq1.wald(p) c(1)=c(2)
```

redirect output to the existing spool object MYOLDSPOOL, then adds a line graph of SER01, a scatterplot of the series in GRP01, and the table output of a Wald test for equation EQ1 to the spool, in that order.

Note that the three output objects in the spool will be named UNTITLED01, UNTITLED02, and UNTITLD03.

To turn off redirection, simply issue the command:

```
output off
```

Appending and Inserting

You may use the `Spool::append` procedure to add output objects to the end of an existing spool object. You may insert any valid EViews object view into a spool. For example,

```
spool01.append ser01.line
```

appends a line graph of SER01 to the end of SPOOL01.

The name of the object in the spool will be the next available name beginning with "UNTITLED". For example, if two objects have already been appended to SPOOL01, named UNTITLED01 and UNTITLED02, then the line graph of SER01 will be named UNTITLED03.

You may append multiple EViews objects using a single `append` command:

```
spool03.append ser02.line ser03
```

appends a line graph of SER02 and the default spreadsheet view of SER03 to the end of SPOOL03.

The `Spool::insert` proc offers additional control over the location of the added object by allowing you to specifying an integer position for the inserted object. If a position is not specified or the specified position is greater than the number of objects already in the spool, the object will be appended to the end. The command:

```
spool01.insert(loc=3) series01
```

inserts the default spreadsheet view of SERIES01 into SPOOL01 at position three. All existing objects in the spool from position three and higher are pushed down in the list to accommodate the new object.

You may include more than one object view using a single `insert` command:

```
spool01.insert(loc=5) "eq1.wald c(1)=c(2)" series01.uroot
```

inserts both the results for a Wald test on EQ1, and the results for a unit root test for SERIES01 into the spool in the fifth and sixth positions. Existing objects from the fifth position onward will be moved down to the seventh position so that they follow the unit root table. Note that since the Wald test command contains spaces, we require the use of double quotes to delimit the expression.

Alternately, `insert` accepts an object name for the location and an optional offset keyword. The command:

```
spool01.insert(loc=obj3) mycity.line
```

adds the line graph view of MYCITY to SPOOL01, placing it before OBJ3. You may modify the default positioning by adding the “offset = after” option,

```
spool01.insert(loc=obj3, offset=after) mycity.line
```

so that the line graph is inserted after OBJ3.

You may use `insert` or `append` to add spool objects to a spool. Suppose that we have the spool objects SPOOL01 and STATESPOOL. Then

```
spool01.insert statespool
```

adds STATESPOOL to the end of SPOOL01.

Subsequent `insert` commands may be used to place objects before, after, or inside of the spool object. The commands

```
spool01.insert(loc=state) mycity.line
spool01.insert(loc=state, offset=after) mytown.hist
```

inserts a line graph view of MYCITY before, and the histogram view of MYTOWN after the STATE spool. You may also use the "offset = " option to instruct EViews to place the new output object inside of an embedded spool:

```
spool01.insert(loc=state, offset=first) mycity.boxplot
spool01.insert(loc=state, offset=last) mystate.stats
```

places a boxplot view of MYCITY and a descriptive statistics view of MYSTATE inside of the STATE spool object. The boxplot view is inserted at the beginning of STATE, while the descriptive statistics view is appended to the end of STATE.

Objects within a embedded spool should be referred to using the full path description. For example, suppose we have a spool object COUNTY which we wish to add to the end of the previously embedded spool STATE. Then,

```
spool01.insert(loc=state, offset=last) county
```

inserts COUNTY as the last member of the spool STATE, and:

```
spool01.insert(loc=state/county, offset=first) mycity.bar
```

inserts a bar graph of MYCITY into the first position of the COUNTY spool.

Naming Objects

The default name of an object when it is inserted into a spool is UNTITLED followed by the next available number *(e.g.* UNTITLED03). When using the `Spool::append` or the `Spool::insert` procs may use the "name = " option to specify a name.

Alternately, you may use the `Spool::name` command to change the name of an object in the spool. For example,

```
spool01.name untitled03 losangeles
```

renames the UNTITLED03 object to LOSANGELES. Note that names are not case-sensitive, and that they must follow EViews' standard naming conventions for objects. Names must also uniquely identify objects in the spool.

To rename an object contained in an embedded spool, you should provide the full path description of the object. For example, the command:

```
spool01.name untitled01/untitled02 newyork
```

renames the object UNTITLED02 which is contained in the spool UNITITLED01 to NEWYORK.

Object Displaynames

The `Spool::displayname` proc may also be used to alter the display name of an object. The default display name of an object is simply the uppercase version of the object name. Display names, which are case-sensitive, not restricted to be valid object names, and need

not be unique, allow you to provide a more descriptive label in the tree pane view when displaying object names.

For example,

```
spool01.displayname untitled03 "Los Angeles"
```

sets the display name for UNTITLED03 object to the text “Los Angeles”. Note that since the desired display name has spaces, we have enclosed the text in double-quotes.

Similarly,

```
spool01.displayname untitled01/untitled02 "New York"
```

sets the display name for UNTITLED02 in the spool UNITITLED01 to “New York”.

Object Comments

The `Spool::displayname` may be used to assign a comment to an object in the spool. Setting a comment for an object is similar to setting the display name. Comments can be multiline; you may use “\n” to indicate the start of a new line in a comment.

```
Spool01.comment untitled01 "The state population of Alabama as
    found\nfrom http://www.census.gov/popest/states/NST-ann-
    est.html."
```

assigns the following comment to object UNTITLED01:

> “The state population of Alabama as found
> from http://www.census.gov/popest/states/NST-ann-est.html.”

Removing Objects

Use the `Spool::remove` proc to delete objects from a spool. Follow the `remove` keyword with names of the objects to be deleted. The unique object name should be used; the display name cannot be used as a valid argument for the `remove` command.

```
spool01.remove untitled02 untitled01 untitled03
```

removes the three objects UNTITLED01, UNTITLED02, UNTITLED03 from SPOOL01. Note that the order at which objects are specified is not important.

Extracting Objects

Objects within a spool are not confined to spools forever; they may be extracted to other spools using `Spool::extract`. An independent copy of the specified object will be made. Note that only one object may be extracted at a time. For instance, referring to our example above, where we have a STATE spool containing a COUNTY spool,

```
spool01.extract state/county
```

creates an untitled spool containing the objects in the COUNTY spool.

Similarly:

```
spool01.extract(mycounty) state/county
```

Customizing the Spool

Titles and Comments

Each object in a spool has both an object name and a display name. By default, the object name is shown. The object name is not case sensitive, while the display name can be multiple words and is case sensitive.

Setting a comment for an object is similar to setting the display name. Comments can be multiline; you may use "\n" to indicate the start of a new line in a comment.

```
Spool01.comment untitled01 "The state population of Alabama as
    found\nfrom http://www.census.gov/popest/states/NST-ann-
    est.html."
```

assigns the following comment to object UNTITLED01:

"The state population of Alabama as found
from http://www.census.gov/popest/states/NST-ann-est.html."

Customizing the Appearance

General properties of a spool may be modified using the `Spool::options` proc. These properties include the display of the object tree, borders, titles, comments, and the use of the object name or display name. To change these settings, use the `options` keyword followed by the characteristic you wish to change.

To turn off the tree and display titles, displaynames and comments for SPOOL01:

```
spool01.options -tree titles displaynames comments
```

creates a spool with the same objects and names it MYCOUNTY.

Printing the Spool

Printing a entire spool object is the same as printing any other object in EViews, simply use the `print` (p. 414) command followed by the name of the spool:

```
print spool01
```

prints all of SPOOL01.

To print an object stored in SPOOL01, us the `Spool::print` proc and specify the name of the object within the spool that you wish to print. For example,

```
spool01.print state/county
```

prints the COUNTY object, which is located in the STATE spool in SPOOL01. The `Spool::print` proc also allows you to print multiple objects in the spool.

```
spool01.print state county
```

prints both the STATE and COUNTY objects individually.

When printing from the command window, the **Print Options** dialog will be displayed for each object specified, allowing you to modify printer settings. When printing from a program, the current printer settings will be used. To modify the current printer settings, you may use **File/Print Setup** to set the global print defaults ("Print Setup," on page 779 of *User's Guide I*).

Spool Summary

See "Spool," on page 596 of the *Object Reference* for a full listing of procedures that may be used with spool objects.

Chapter 5. Strings and Dates

Strings

An alphanumeric *string* is a set of characters containing alphabetic ("alpha") and numeric characters, and in some cases symbols, found on a standard keyboard. Strings in EViews may include spaces and dashes, as well as single or double quote characters. Note also that EViews does not support unicode characters.

Strings are used in EViews in a variety of places. "Using Strings in EViews" on page 79 offers a brief overview.

When entering alphanumeric values into EViews, you generally should enclose your characters in double quotes. The following are all examples of valid string input:

```
"John Q. Public"
"Ax$23!*jFg5"
"000-00-0000"
"(949)555-5555"
"11/27/2002"
"3.14159"
```

You should use the double quote character as an escape character for double quotes in a string. Simply enter two double quote characters to include the single double quote in the string:

```
"A double quote is given by entering two "" characters."
```

Bear in mind that strings are simply sequences of characters with no special interpretation. The string values "3.14159" and "11/27/2002" might, for example, be used to represent a number and a date, but as strings they have no such intrinsic interpretation. To provide such an interpretation, you must use the EViews tools for translating string values into numeric or date values (see "String Information Functions" on page 70 and "Translating between Date Strings and Date Numbers" on page 88).

Lastly, we note that the *empty,* or *null, string* ("") has multiple interpretations in EViews. In settings where we employ strings as a building block for other strings, the null string is interpreted as a blank string with no additional meaning. If, for example, we concatenate two strings, one of which is empty, the resulting string will simply be the non-empty string.

In other settings, the null string is interpreted as a missing value. In settings where we use string values as a category, for example when performing categorizations, the null string is interpreted as both a blank string and a missing value. You may then choose

to exclude or not exclude the missing value as a category when computing a tabulation using the string values. This designation of the null string as a missing value is recognized by a variety of views and procedures in EViews and may prove useful.

Likewise, when performing string comparisons using blank strings, EViews generally treats the blank string as a missing value. As with numeric comparisons involving missing values, comparisons involving missing values will often generate a missing value. We discuss this behavior in greater detail in our discussion of "String Comparison (with empty strings)" on page 68.

String Operators

The following operators are supported for strings: (1) concatenation—plus (" + "), and (2) relational—equal to (" = "), not equal to (" < > "), greater than (" > "), greater than or equal to (" > = "), less than (" < "), less than or equal to (" < = ").

String Concatenation Operator

Given two strings, *concatenation* creates a new string which contains the first string followed immediately by the second string. You may concatenate strings in EViews using the concatenation operator, " + ". For example,

```
"John " + "Q." + " Public"
"3.14" + "159"
```

returns the strings

```
"John Q. Public"
"3.14159"
```

Bear in mind that string concatenation is a simple operation that does not involve interpretation of strings as numbers or dates. Note in particular that the latter entry yields the concatenated string, "3.14159", not the sum of the two numeric values, "162.14". To obtain numeric results, you will first have to convert your strings into a number (see "String Information Functions" on page 70).

Lastly, we note that when concatenating strings, the *empty* string is interpreted as a blank string, not as a missing value. Thus, the expression

```
"Mary " + "" + "Smith"
```

yields

```
"Mary Smith"
```

since the middle string is interpreted as a blank.

String Relational Operators

The relational operators return a 1 if the comparison is true, and 0 if the comparison is false. In some cases, relational comparisons involving null strings will return a NA.

String Ordering

To determine the ordering of strings, EViews employs the region-specific collation order as supplied by the Windows operating system using the user's regional settings. Central to the tasks of *sorting* or *alphabetizing*, the collation order is the culturally influenced order of characters in a particular language.

While we cannot possibly describe all of the region-specific collation order rules, we note a few basic concepts. First, all punctuation marks and other non alphanumeric characters, except for the hyphen and the apostrophe precede the alphanumeric symbols. The apostrophe and hyphen characters are treated distinctly, so that "were" and "we're" remain close in a sorted list. Second, the collation order is case specific, so that the character "a" precedes "A". In addition, similar characters are kept close so that strings beginning with "a" are followed by strings beginning with "A", ahead of strings beginning with "b" and "B".

Typically, we determine the order of two strings by evaluating strings character-by-character, comparing pairs of corresponding characters in turn, until we find the first pair for which the strings differ. If, using the collation order, we determine the first character precedes the second character, we say that the first string is *less than* the second string and the second string is *greater than* the first. Two strings are said to be *equal* if they have the same number of identical characters.

If the two strings are identical in every character, but one of them is shorter than the other, then a comparison will indicate that the longer string is greater. A corollary of this statement is that the null string is less than or equal to all other strings.

The multi-character elements that arise in many languages are treated as single characters for purposes of comparison, and ordered using region-specific rules. For example, the "CH" and "LL" in Traditional Spanish are treated as unique characters that come between "C" and "L" and "M", respectively.

String Comparison (with non-empty strings)

Having defined the notion of string ordering, we may readily describe the behavior of the relational operators for non-empty (non-missing) strings. The " = " (equal), " > = " (greater than or equal), and " < = " (less than or equal), " < > " (not equal), " > " (greater than), and " < " (less than) comparison operators return a 1 or a 0, depending on the result of the string comparison. To illustrate, the following (non region-specific) comparisons return the value 1,

```
"abc" = "abc"
"abc" <> "def"
```

```
"abc" <= "def"
"abc" < "abcdefg"
"ABc" > "ABC"
"abc def" > "abc 1ef"
```

while the following return a 0,

```
"AbC" = "abc"
"abc" <> "abc"
"aBc" >= "aB1"
"aBC" <= "a123"
"abc" >= "abcdefg"
```

To compare portions of strings, you may use the functions `@left`, `@right`, and `@mid` to extract the relevant part of the string (see "String Manipulation Functions" on page 72). The relational comparisons,

```
@left("abcdef", 3) = "abc"
@right("abcdef", 3) = "def"
@mid("abcdef", 2, 2) = "bc"
```

all return 1.

In normal settings, EViews will employ case-sensitive comparisons (see "Case-Sensitive String Comparison" on page 157 for settings that enable caseless element comparisons in programs). To perform a caseless comparison, you should convert the expressions to all uppercase, or all lowercase using the `@upper`, or `@lower` functions. The comparisons,

```
@upper("abc") = @upper("aBC")
@lower("ABC" = @lower("aBc")
```

both return 1.

To ignore leading and trailing spaces, you should use the `@ltrim`, `@rtrim`, and `@trim` functions remove the spaces prior to using the operator. The relational comparisons,

```
@ltrim(" abc") = "abc"
@ltrim(" abc") = @rtrim("abc ")
@trim(" abc ") = "abc"
```

all return 1.

String Comparison (with empty strings)

Generally speaking, the relational operators treat the empty string as a missing value and return the numeric missing value NA when applied to such a string. Suppose, for example that an observation in the alpha series X contains the string "Apple", and the corresponding observation in the alpha series Y contains a blank string. All comparisons ("X = Y", "X > Y",

"X > = Y", "X < Y", "X < = Y", and "X < > Y") will generate an NA for that observation since the Y value is treated as a missing value.

Note that this behavior differs from EViews 4 and earlier in which empty strings were treated as ordinary blank strings and not as a missing value. In these versions of EViews, the comparison operators always returned a 0 or a 1. The change in behavior, while regrettable, was necessary to support the use of string missing values.

It is still possible to perform comparisons using the previous behavior. One approach is to use the special functions `@eqna` and `@neqna` for equality and strict inequality comparisons without propagating NAs (see "String Information Functions" on page 70). For example, you may use the expressions

```
@eqna(x, y)
@neqna(x, y)
```

so that blanks in string X or Y are treated as ordinary string values. Using these two functions, the observation where X contains "Apple" and Y contains the "" will evaluate to 0 and 1, respectively instead of NA.

Similarly, if you specify a relational expression involving a literal blank string, EViews will perform the test treating empty strings as ordinary string values. If, for example, you test

```
x = ""
```

or

```
x < ""
```

all of the string values in X will be tested against the string literal "". You should contrast this behavior with the behavior for the non-literal tests "X = Y" and "X < Y" where blank values of X or Y result in an NA comparison.

Lastly, EViews provides a function for the strict purpose of testing whether a string value is an empty string. The `@isempty` function tests whether a string is empty. The relational equality test against the blank string literal "" is equivalent to this function.

String Lists

A *string list* is an ordinary string that is interpreted as a space delimited list of string elements. For example, the string

```
"Here I stand"
```

may be interpreted as containing three elements, the words "Here", "I" and "stand". Double quotes may be used to include multiword elements in a list. Bearing in mind that the quote is used as an escape character for including quotes in strings, the list

```
"""Chicken Marsala"" ""Beef Stew"" Hamburger"
```

contains three elements, the expressions “Chicken Marsala”, “Beef Stew”, and “Hamburger”. Notice how the escaped double quotes are used to group words into single list elements.

Interpreting a string as a list of elements allows us to make use of functions which operate on each element in the string, rather than on each character. These methods can be useful for string manipulation and pattern searching. For example, we may find the intersection, union, or cross of two string lists. Additionally, we may manipulate the elements of a string list and find the elements that match or do not match a given pattern. For example, the string list function

```
@wkeep("ABC ABCC AABC", "?B*")
```

uses the pattern “?B*” to filter the string list “ABC ABCC AABC”. Elements with a single character, followed by the character “B”, then followed by any number of other characters are kept, returning: “ABC ABCC”.

String Functions

EViews provides a number of functions that may either be used with strings, or return string values.

Functions that treat a string as a string list begin with a “w”. Some string functions have corresponding list functions with the same name, preceded by a “w”. For instance, `@left` returns the leftmost *characters* of a string, while `@wleft` returns the leftmost *elements* of a string list.

String Information Functions

The following is a brief summary of the basic functions that take strings as an argument and return a number. (Chapter 17. “String and Date Function Reference,” on page 567 offers a more detailed description.)

- `@length`(*str*): returns an integer value for the length of the string *str*.

  ```
  @length("I did not do it")
  ```

  returns the value 15.

  A shortened keyword form of this function, `@len`, is also supported.

- `@wcount`(*str_list*): returns an integer value for the number of elements in the string list *str_list*.

  ```
  @wcount("I did not do it")
  ```

  returns the value 5.

- `@instr`(*str1*, *str2[, int]*): finds the starting position of the target string *str2* in the base string *str1*. By default, the function returns the location of the first occurrence of *str2* in *str1*. You may provide an optional integer *int* to specify the occurrence. If the requested occurrence of the target string is not found, `@instr` will return a 0.

The returned integer is often used in conjunction with `@mid` to extract a portion of the original string.

```
@instr("1.23415", "34")
```

returns the value 4, since the substring "34" appears beginning in the fourth character of the base string, so

```
@mid("1.23415", @instr("1.23415", "34"))
```

returns "3415".

- `@wfind`(*str_list, str_cmp*): looks for the string *str_cmp* in the string list *str_list,* and returns the element position in the list or 0 if the string is not in the list.

  ```
  @wfind("I did it", "did")
  ```

  returns the value 2.

  The `@wfindnc` function performs the same operation, but the comparison is not case-sensitive.

- `@isempty`(*str*): tests for whether *str* is a blank string, returning a 1 if *str* is a null string, and 0 otherwise.

  ```
  @isempty("1.23415")
  ```

  returns a 0, while

  ```
  @isempty("")
  ```

  returns the value 1.

- `@eqna`(*str1*, *str2*): tests for equality of *str1* and *str2*, treating null strings as ordinary blank strings, and not as missing values. Strings which test as equal return a 1, and 0 otherwise. For example,

  ```
  @eqna("abc", "abc")
  ```

  returns a 1, while

  ```
  @eqna("", "def")
  ```

  returns a 0.

- *@*`neqna`(*str1*, *str2*): tests for inequality of *str1* and *str2*, treating null strings as ordinary blank strings, and not as missing values. Strings which test as not equal return a 1, and 0 otherwise.

  ```
  @neqna("abc", "abc")
  ```

  returns a 0,

  ```
  @neqna("", "def")
  ```

  returns a 1.

- `@val`(*str[, fmt]*): converts the string representation of a number, *str,* into a numeric value. If the string has any non-digit characters, the returned value is an NA. You may

provide an optional numeric format string *fmt*. See "String Conversion Functions" on page 76 and @val (p. 590) for details.

- @dateval(*str[, fmt]*): converts the string representation of a date string, *str,* into a date number using the optional format string *fmt*. See "String Conversion Functions" on page 76 and @dateval (p. 574) for details.
- @dtoo(*str*): (*D*ate *TO O*bs) converts the string representation of a date, *str*, into an observation value for the active workfile. 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.

  Note that @dtoo will generate an error if used in a panel structured workfile.

String Manipulation Functions

The following is a brief summary of the basic functions that take strings as an argument and return a string.

- @left(*str, int*): returns a string containing the *int* characters at the left end of the string *str*. If there are fewer than *int* characters, @left will return the entire string.

  ```
  @left("I did not do it", 5)
  ```

  returns the string "I did".
- @wleft(*str_list*, *int*): returns a string containing the *int* elements at the left end of the string list *str_list*. If there are fewer than *int* elements, @wleft will return the entire string list.

  ```
  @wleft("I did not do it", 3)
  ```

  returns the string "I did not".
- @right(*str, int*): returns a string containing the *int* characters at the right end of a string. If there are fewer than *int* characters, @right will return the entire string.

  ```
  @right("I doubt that I did it", 8)
  ```

  returns the string "I did it".
- @wright(*str_list*, *int*): returns a string containing the *int* elements at the right end of a string list. If there are fewer than *int* elements, @wright will return the entire string.

  ```
  @wright("I doubt that I did it", 3)
  ```

  returns the string "I did it".

- `@mid`(*str*, *int1[, int2]*): returns the string consisting of the characters starting from position *int1* in the string. By default, `@mid` returns the remainder of the string, but you may specify the optional integer *int2*, indicating the number of characters to be returned.

  ```
  @mid("I doubt that I did it", 9, 10)
  ```

  returns “that I did”.

  ```
  @mid("I doubt that I did it", 9)
  ```

  returns the string “that I did it”.

- `@wmid`(*str_list*, *int1[, int2]*): returns the string consisting of the elements starting from position *int1* in the string. By default, `@wmid` returns all remaining elements of the string, but you may specify the optional integer *int2*, indicating the number of elements to be returned.

  ```
  @wmid("I doubt that I did it", 2, 3)
  ```

  returns “doubt you did”.

  ```
  @mid("I doubt that I did it", 4)
  ```

  returns the string “I did it”.

- `@word`(*str_list*, *int*): returns the *int* element of the string list.

  ```
  @word("I doubt that I did it", 2)
  ```

  returns the second element of the string, “doubt”.

- `@wordq`(*str_list*, *int*): returns the *int* element of the string list, while preserving quotes.

  ```
  @wordq("""Chicken Marsala"" ""Beef Stew""", 2)
  ```

  returns the second element of the string, “Beef Stew”. The `@word` function would return the same elements, but would not include quotation marks in the string.

- `@insert`(*str1*, *str2*, *int*): inserts the string *str2* into the base string *str1* at the position given by the integer *int*.

  ```
  @insert("I believe it can be done", "not ", 16)
  ```

  returns “I believe it cannot be done”.

- `@wkeep`(*str_list*, "*pattern_list*"): returns the list of elements in *str_list* that match the string pattern *pattern_list*. The *pattern_list* is space delimited, and may be made up of any number of “?” (indicates any single character) or “*” (indicates any number of characters).

  ```
  @wkeep("ABC DEF GHI JKL", "?B? D?? *I")
  ```

  keeps the first three elements of the string list, returning the string “ABC DEF GHI”.

- `@wdrop`(*str_list*, "*pattern_list*"): returns a string list, dropping elements in *str_list* that match the string pattern *pattern_list*. The *pattern_list* is space delimited, and may be made up of any number of "?" (indicates any single character) or "*" (indicates any number of characters).

  ```
  @wdrop("ABC DEF GHI JKL", "?B? D?? *I")
  ```

  drops the first three elements of the string list, returning the string "JKL".

- `@replace`(*str1*, *str2*, *str3[, int]*): returns the base string *str1*, with the replacement *str3* substituted for the target string *str2*. By default, all occurrences of *str2* will be replaced, but you may provide an optional integer *int* to specify the number of occurrences to be replaced.

  ```
  @replace("Do you think that you can do it?", "you", "I")
  ```

  returns the string "Do I think that I can do it?", while

  ```
  @replace("Do you think that you can do it?", "you", "I", 1)
  ```

  returns "Do I think that you can do it?".

- `@wreplace`(*str_list*, "*src_pattern*", "*replace_pattern*"): returns the base string list *str_list*, with the replacement pattern *replace_pattern* substituted for the target pattern *src_pattern*. The pattern lists may be made up of any number of "?" (indicates any single character) or "*" (indicates any number of characters).

  ```
  @wreplace("ABC AB", "*B*", "*X*")
  ```

  replaces all instances of "B" with "X", returning the string "AXC AX".

  ```
  @wreplace("ABC DDBC", "??B?", "??X?")
  ```

  replaces all instances of "B" which have two leading characters and one following character, returning the string "ABC DDXC".

- `@ltrim`(*str*): returns the string *str* with spaces trimmed from the left.

  ```
  @ltrim(" I doubt that I did it. ")
  ```

  returns "I doubt that I did it. ". Note that the spaces on the right remain.

- `@rtrim`(*str*): returns the string *str* with spaces trimmed from the right.

  ```
  @rtrim(" I doubt that I did it. ")
  ```

  returns the string " I doubt that I did it.". Note that the spaces on the left remain.

- `@trim`(*str*): returns the string *str* with spaces trimmed from the both the left and the right.

  ```
  @trim(" I doubt that I did it. ")
  ```

  returns the string "I doubt that I did it.".

- `@upper`(*str*): returns the upper case representation of the string *str*.

  ```
  @upper("I did not do it")
  ```

returns the string “I DID NOT DO IT”.

- `@lower`(*str*): returns the lower case representation of the string *str*.

  ```
  @lower("I did not do it")
  ```

  returns the string “i did not do it”.

- `@addquotes`(*str*): returns the string *str* with quotation marks added to the left and right. Given a string S1 that contains the unquoted text: I did not do it,

  ```
  @addquotes(S1)
  ```

  returns the quoted string “I did not do it”.

- `@stripquotes`(*str*): returns the string *str* with quotation marks removed from the left and right. Given a string S1 that contains the text: “I did not do it”,

  ```
  @stripquotes(S1)
  ```

  returns the unquoted string: “I did not do it”.

- `@stripparens`(*str*): returns the string *str* with parentheses removed from the left and right. Given a string S1 that contains the text: “(I did not do it)”,

  ```
  @stripparens(S1)
  ```

  returns the string: “I did not do it”.

- `@wintersect`(*str_list1, str_list2*): returns the intersection of *str_list1* and *str_list2*.

  ```
  @wintersect("John and Greg won", "Mark won but Greg lost")
  ```

  returns the string “won Greg”.

- `@wunion`(*str_list1*, *str_list2*): returns the union of *str_list1* and *str_list2*.

  ```
  @wunion("ABC DEF", "ABC G H def")
  ```

  returns the string “ABC DEF G H def”. Each new element is added to the string list, skipping elements that have already been added to the list.

- `@wunique`(*str_list*): returns *str_list* with duplicate elements removed from the list.

  ```
  @wunique("fr1 fr2 fr1")
  ```

  returns the string “fr1 fr2”.

- `@wnotin`(*str_list1*, *str_list2*): returns elements of *str_list1* that are not in *str_list2*.

  ```
  @wnotin("John and Greg won", "and Greg")
  ```

  returns the string “John won”.

- `@wcross`(*str_list1*, *str_list2[, “pattern”]*): returns *str_list1* crossed with *str_list2,* according to the string *pattern.* The default pattern is “??”, which indicates that each element of *str_list1* should be crossed individually with each element of *str_list2.*

  ```
  @wcross("ABC DEF", "1 2 3", "?-?")
  ```

returns the string list “ABC-1 ABC-2 ABC-3 DEF-1 DEF-2 DEF-3”, inserting a dash (“-”) between each crossed element as the “?-?” pattern indicates.

- `@winterleave`(*str_list1*, *str_list2[, count1, count2]*): Interleaves *str_list1* with *str_list2,* according to the pattern specified by *count1* and *count2.* The default uses counts of one.

  ```
  @winterleave("A B C", "1 2 3")
  ```

  interleaves “A B C” with “1 2 3” to produce the string list “A 1 B 2 C 3”.

- `@wsort`(*str_list*[,”D”]): Returns sorted elements of *str_list*. Use the “D” flag to sort in descending order.

  ```
  @wsort("fq8 Fq8 xpr1", "D")
  ```

  sorts the string in descending order: “xpr1 Fq8 fq8”.

- `@wdelim`(*str_list,* "*src_delim*", "*dest_delim*"): returns a string list, replacing every appearance of the *src_delim* delimiter in *str_list* with a *dest_delim* delimiter. Delimiters must be single characters.

  ```
  @wdelim("Arizona, California, Washington", ",", "-")
  ```

  identifies the comma as the source delimiter and replaces each comma with a dash, returning the string “Arizona-California-Washington”.

String Conversion Functions

The following functions convert between numbers or date numbers and strings:

- `@datestr`(*date1[, fmt]*): converts the date number *date1* to a string representation using the optional date format string, *fmt*.

  ```
  @datestr(730088, "mm/dd/yy")
  ```

  will return “12/1/99”,

  ```
  @datestr(730088, "DD/mm/yyyy")
  ```

  will return “01/12/1999”, and

  ```
  @datestr(730088, "Month dd, yyyy")
  ```

  will return “December 1, 1999”, and

  ```
  @datestr(730088, "w")
  ```

  will produce the string “3”, representing the weekday number for December 1, 1999.

  See the function reference entry `@datestr` (p. 573) and “Dates” on page 82 for additional details on date numbers and date format strings.

- `@dateval`(*str[, fmt]*): converts the string representation of a date string, *str,* into a date number using the optional format string *fmt*.

  ```
  @dateval("12/1/1999", "mm/dd/yyyy")
  ```

will return the date number for December 1, 1999 (730088) while

```
@dateval("12/1/1999", "dd/mm/yyyy")
```

will return the date number for January 12, 1999 (729765). See the function reference entry @dateval (p. 574) and "Dates," beginning on page 82 for discussion of date numbers and format strings.

- `@str`(*num[, fmt]*): returns a string representation of the number *num*. You may provide an optional numeric format string *fmt*.

  ```
  @str(153.4)
  ```

  returns the string "153.4".

  To create a string containing 4 significant digits and leading "$" character, use

  ```
  @str(-15.4435, "g$.4")
  ```

  The resulting string is "-$15.44".

  The expression

  ```
  @str(-15.4435, "f7..2")
  ```

  converts the numerical value, -15.4435, into a fixed 7 character wide decimal string with 2 digits after the decimal and comma as decimal point. The resulting string is " -15,44". Note that there is a leading space in front of the "-" character making the string 7 characters long.

  The expression

  ```
  @str(-15.4435, "e(..2)")
  ```

  converts the numerical value, -15.4435, into a string written in scientific notation with two digits to the right of the decimal point. The decimal point in the value will be represented using a comma and negative numbers will be enclosed in parenthesis. The resulting string is "(1,54e + 01)". A positive value will not have the parenthesis.

  ```
  @str(15.4435, "p+.1")
  ```

  converts the numeric value, 15.4435, into a percentage where the value is multiplied by 100. Only 1 digit will be included after the decimal and an explicit " + " will always be included for positive numbers. The resulting value after rounding is " + 1544.4".

  See the function reference entry @str (p. 582) for a detailed description of the conversion rules and syntax, along with additional examples.

- `@val`(*str[, fmt]*): converts the string representation of a number, *str,* into a numeric value. If the string has any non-digit characters, the returned value is an NA. You may provide an optional numeric format string *fmt*.

  ```
  @val("1.23415")
  ```

  See the function reference entry @val (p. 590) for a detailed description of the conversion rules.

String Vector Functions

The following functions either take a string vector as an argument, or return a string vector:

`@rows`(*str_vector*): returns the number of rows in *str_vector*.

For a string vector SV01 with 10 rows,

```
@rows(sv01)
```

returns the integer 10.

- `@wsplit`(*str_list*): returns a string vector containing the elements of *str_list*.

  If the string list SS01 contains "A B C D E F", then

  ```
  @wsplit(ss01)
  ```

  returns an untitled svector, placing an element of SS01 in each row. Row one of the svector contains "A", row two contains "B", etc.

Special Functions that Return Strings

EViews provides a special, workfile-based function that uses the structure of the active workfile page and returns a *set* of string values representing the date identifiers associated with the observations.

- `@strdate`(*fmt*): returns the set of workfile row dates as strings in an Alpha series, formatted using the date format string *fmt*. See "Special Date Functions" on page 100 for details.

In addition, EViews provides two special functions that return a string representations of the date associated with a specific observation in the workfile, or with the current time.

- `@otod`(*int*): (*O*bs *TO D*ate) : returns a string representation of the date associated with a single observation (counting from the start of the workfile). Suppose, for example, that we have a quarterly workfile ranging from 1950Q1 to 1990Q4. Then

  ```
  @otod(16)
  ```

  returns the date associated with the 16th observation in the workfile in string form, "1953Q4".

- `@otods`(*int*): (*O*bs *TO D*ate in *S*ample) : returns a string representation of the date associated with a single observation (counting from the start of the *sample*). Thus

  ```
  @otods(2)
  ```

  will return the date associated with the second observation in the current sample. Note that if *int* is negative, or is greater than the number of observations in the current sample, an empty string will be returned.

- `@strnow`(*fmt*): returns a string representation of the current date number (at the moment the function is evaluated) using the date format string, *fmt*.

```
@strnow("DD/mm/yyyy")
```

returns the date associated with the current time in string form with 2-digit days, months, and 4-digit years separated by a slash, "24/12/2003".

You may also ask EViews to report information about objects in the current workfile or database, or a directory on your computer, in a form suitable for list processing:

- `@wlookup`("*pattern_list*"[, "*object_type_list*"]): Returns a string list of all objects in the workfile or database that satisfy the *pattern_list* and, optionally, the *object_type_list*. The *pattern_list* may be made up of any number of "?" (indicates any single character) or "*" (indicates any number of characters).

  If a workfile contains a graph object named "GR01" and two series objects named "SR1" and "SER2", then

  ```
  @wlookup("?R?","series")
  ```

  returns the string "SR1".

- `@wdir`(*directory_str*): returns a string list of all files in the directory *directory_str*. Note that this does not include other directories nested within *directory_str*.

  ```
  @wdir("C:\Documents and Settings")
  ```

  returns a string list containing the names of all files in the "C:\Documents and Settings" directory.

Lastly, all EViews objects have data members which return information about themselves in the form of a string. For example:

```
ser1.@updatetime
```

returns the last update time for the series SER1

```
eq1.@command
```

returns the full command line form of the estimation command.

For lists of the relevant data members see the individual object descriptions in Chapter 1. "Object View and Procedure Reference," on page 2.

Using Strings in EViews

Strings in EViews are primarily used in four distinct contexts: string variables, string objects, string vectors, or Alpha series.

String Variables

A *string variable* is a temporary variable used in a program whose value is a string. String variables, which only exist during the time that your EViews program is executing, have names that begin with a "%" symbol. For example,

```
%value = "value in millions of u.s. dollars"
%armas = "ar(1) ar(2) ma(1) ma(2)"
```

are string variables declarations that may be used in program files.

See "String Variables," on page 116 for extensive discussion of the role that these variables play in programming.

String Objects

A *string object* is an EViews workfile object that holds a string of text:

```
string lunch = "Apple Tuna Cookie"
string dinner = """Chicken Marsala"" ""Beef Stew"" Hamburger"
```

creates the string objects LUNCH and DINNER, each containing the corresponding string literal. Note that we have used the double quote character as an escape character for double quotes.

Since a string object is an EViews workfile object, we may open and display its views. A string object's view may be switched between **String** and **Word list** views. The **String** view for DINNER displays the text as a single string,

```
"Chicken Marsala" "Beef Stew" Hamburger
```

while the **Word list** view breaks up the text by element,

```
"Chicken Marsala"
"Beef Stew"
Hamburger
```

with each element on a separate line.

We emphasize the important distinction that string objects are named objects in the workfile that may be saved with the workfile, while string variables are temporary variables that only exist while an EViews program is running. Thus, string objects have the advantage that they may be used interactively, while string variables may not be used outside of programs. String objects can, however, go out of scope when the active workfile page changes, while string variables are always in scope.

In all other respects, strings objects and string variables may be used interchangeably in programs. Either string object or string variables can be passed into subroutines for arguments declared as type string.

String Vectors

An svector, or *string vector*, is an EViews object that holds a string in each row of the vector. A string vector can be created by specifying the number of rows in the vector and providing a name:

```
svector(3) svec
```

If a length is not specified, a one row svector will be created.

An svector can be populated by assigning a string or string literal to each row:

```
svec(1) = "gdp cost total"
```

fills the first row of SVEC with the string “gdp cost total”. To assign the same string to all rows, omit the row number. The command

```
svec = "invalid"
```

will assign the string “invalid” to all rows of SVEC.

A multiple row svector may be populated using the `@wsplit` command, which creates a string vector from a string list. For example,

```
svector svec
string st = "gdp cost total"
svec = @wsplit(st)
```

creates the string vector SVEC of default length one and a string object ST containing “gdp cost total”. The `@wsplit` command creates a three element svector from the elements of ST, placing the string “gdp” in the first row of the string vector, the string “cost” in the second row, and the string “total” in the third row, and assigns it to SVEC, which is resized accordingly.

Similarly, an svector will shrink if assigned to a smaller svector. For example,

```
svector svec3 = @wsplit("First Middle Last")
svector(10) svec10
svec10 = svec3
```

first creates the three row svector SVEC3, then assigns the strings “First”, “Middle”, and “Last” to the first, second, and third rows, respectively. The third line creates a second ten row svector, SVEC10. When SVEC3 is assigned to SVEC10, its values are copied over and rows four through ten are removed from SVEC10.

An svector may also be filled by concatenating two strings or svectors. For instance,

```
svector s1 = @wsplit("A B C")
svector s2 = @wsplit("1 2 3")
svector ssvec = s1 + s2
```

creates two svectors S1 and S2, each with three rows. S1 contains the characters “A”, “B”, and “C”, while S2 contains “1”, “2”, and “3”. The third command creates the svector SSVEC and fills it with the concatenation of the other two svectors, producing “A1” on the first row, “B2” on the second row, and “C3” on the third row.

More generally, any operation that can be performed on a string may be performed on element of an svector. For example, given an svector whose first element contains the string "Hello World" and whose second element contains "Hi there world", the element assignment statement

```
svector sv(3) = @left(sv(1),5) + " " + @mid(sv(2),4,5)
```

takes the left five characters of the first row ("Hello"), adds a space, concatenates five characters from the second row, starting at the fourth character ("there"), and assigns it to the third element of SV. Element three of SV now contains the string "Hello there".

The row count of a string vector may be retrieved using the `@rows` command:

```
scalar sc = @rows(sv)
```

This is especially useful since svectors are dynamically resized when necessary.

Alpha Series

EViews has a special series type for holding string data. An *alpha series* object contains a set of observations on string values. Alpha series should be used when you wish to work with variables that contain alphanumeric data, such as names, addresses, and other text.

Alpha series are distinguished from string vectors primarily in that their length is tied to the length of the workfile.

See "Alpha Series," on page 194 for discussion.

Dates

There are a variety of places in EViews where you may work with calendar dates. For most purposes, users need not concern themselves with the intricacies of working with dates. Simply enter your dates in familiar text notation and EViews will automatically interpret the string for you.

Those of you who wish to perform more sophisticated operations with dates will, however, need to understand some basic concepts.

In most settings, you may simply use text representations of dates, or *date strings*. For example, an EViews sample can be set to include only observations falling between two dates specified using date strings such as "May 11, 1997", "1/10/1990" or "2001q1". In these settings, EViews understands that you are describing a date and will interpret the string accordingly.

Date information may also be provided in the form of a *date number*. A date number is a numeric value with special interpretation in EViews as a calendar date. EViews allows you to convert date strings into date numbers which may be manipulated using a variety of tools. These tools allow you to perform standard calendar operations such as finding the

number of days or weeks between two dates, the day of the week associated with a given day, or the day and time 36 hours from now.

The remainder of this section summarizes the use of dates in EViews. (See Chapter 5. "Strings and Dates," on page 65 for reference material.) There are several tasks that are central to working with dates:

- Translating between date strings and date numbers.
- Translating ordinary numbers into date numbers.
- Manipulating date numbers using operators and functions.
- Extracting information from date numbers.

Before turning to these tasks, we must first provide a bit of background on the characteristics of date strings, date numbers, and a special class of strings called *date formats*, which are sometimes employed when translating between the former.

Date Strings

Date strings are simply text representations of dates and/or times. Most of the conventional ways of representing days, weeks, months, years, hours, minutes, *etc.*, as text are valid date strings.

To be a bit more concrete, the following are valid date strings in EViews:

```
"December 1, 2001"
"12/1/2001"
"Dec/01/01 12am"
"2001-12-01 00:00"
"2001qIV"
```

As you can see, EViews is able to handle a wide variety of representations of your dates and times. You may use everything from years represented in 1, 2, and 4-digit Arabic form ("1", "01", "99", "1999"), to month names and abbreviations ("January", "jan", "Jan"), to quarter designations in Roman numerals ("I" to "IV"), to weekday names and abbreviations ("Monday", "Mon"), to 12 or 24-hour representations of time ("11:12 pm", "23:12"). A full list of the recognized date string components is provided in "Date Formats" on page 85.

It is worth noting that date string representations may be divided up into those that are *unambiguous* and those that are *ambiguous*. Unambiguous date strings have but a single interpretation as a date, while ambiguous date strings may be interpreted in multiple ways.

For example, the following dates may reasonably be deemed unambiguous:

```
"March 3rd, 1950"
"1980Q3"
"9:52PM"
```

while the following dates are clearly ambiguous:

```
"2/3/4"
"1980:2"
"02:04"
```

The first date string in the latter set is ambiguous because we cannot tell which of the three fields is the year, which is the month, and which is the day, since different countries of the world employ different orderings. The second string is ambiguous since we cannot determine the period frequency within the year. The "2" in the string could, for example, refer to the second quarter, month, or even semi-annual in the year. The final string is ambiguous since it could be an example of a time of day in "hour:minute" format (2:04 am), or a date in "year:period" notation (*i.e.*, the fourth month of the year 2002) or "period:year" notation (*i.e.*, the second month of 2004).

In settings where date input is required, EViews will generally accept date string values without requiring you to provide formatting information. It is here that the importance of the distinction between ambiguous and unambiguous date strings is seen. If the date string is unambiguous, the free-format interpretation of the string as a date will produce identical results in all settings. On the other hand, if the date string is ambiguous, EViews will use the context in which the date is being used to determine the most likely interpretation of the string. You may find that ambiguous date strings are neither interpreted consistently nor as desired.

These issues, and methods of getting around the problem of ambiguity, are explored in greater detail in "Translating between Date Strings and Date Numbers" on page 88.

Date Numbers

Date information is often held in EViews in the form of a *date number*. A date number is a double precision number corresponding to an instance in time, with the integer portion representing a specific day, and the decimal fraction representing time during the day.

The integer portion of a date number represents the number of days in the Gregorian proleptic calendar since Monday, January 1, A.D. 0001 (a "proleptic" calendar is a calendar that is applied to dates both before and after the calendar was historically adopted). The first representable day, January 1, A.D. 1 has an integer value of 0, while the last representable day, December 31, A.D. 9999, has an integer value of 3652058.

The fractional portion of the date number represents a fraction of the day, with resolution to the millisecond. The fractional values range from 0 (12 midnight) up to (but not including) 1 (12 midnight). A value of 0.25, for example, corresponds to one-quarter of the day, or 6:00 a.m.

It is worth noting that the time of day in an EViews date number is accurate up to a particular millisecond within the day, although it can always be displayed at a lower "precision"

(larger unit of time). When date numbers are formatted to lower precisions, they are always rounded down to the requested precision and never rounded up. Thus, when displaying the week or month associated with a date number, EViews always rounds down to the beginning of the week or month.

Date Formats

A *date format string* (or *date format*, for short) is a string made up of text expressions that describe how components of a date and time may be encoded in a date string. Date formats are used to provide an explicit description of a date string representation, and may be employed when converting between strings or numbers and date numbers.

Before describing date formats in some detail, we consider a simple example. Suppose that we wish to use the date string "5/11/1997" to represent the date May 11, 1997. The date format corresponding to this text representation is

```
"mm/dd/yyyy"
```

which indicates that we have, in order, the following components: a one or two-digit month identifier, a "/" separator, a one or two-digit day identifier, a "/" separator, and a 4-digit year identifier.

Alternatively, we might wish to use the string "1997-May-11" to represent the same date. The date format for this string is

```
"yyyy-Month-dd"
```

since we have a four-digit year, followed by the full name of the month (with first letter capitalized), and the one or two-digit day identifier, all separated by dashes.

Similarly, the ISO 8601 representation for 10 seconds past 1:57 p.m. on this date is "1997-05-11 13:57:10". The corresponding format is

```
"yyyy-MM-DD HH:mi:ss"
```

Here, we have used the capitalized forms of "MM", "DD", and "HH" to ensure that we have the required leading zeros.

A full description of the components of a date format is provided below. Some of the more commonly used examples of date formats are listed in the options for the `setformat` object commands (see, for example, `Table::setformat` (p. 703) in the *Object Reference*).

Date Format Components

A date format may contain one or more of the following string fragments corresponding to various date components. In most cases, there are various upper and lowercase forms of the format component, corresponding either to the presence or absence of leading zeros, or to the case of the string identifiers.

The following format strings are the basic components of a date format:

Years

Year formats use either two or four digit years, with or without leading zeros. The corresponding date format strings are:

- "yyyy" or "YYYY": four digit year without/with leading zeros.
- "yy" or "YY": two digit year without/with leading zeros.
- "year" or "YEAR": synonym for "yyyy" and "YYYY", respectively.

Semi-Annual

The semi-annual format corresponds to a single digit representing the period in the year:

- "s" or "S": one digit half-year (1 or 2).

Quarters

Quarter formats allow for values entered in either standard (Arabic) or Roman numbers:

- "q" or "Q": quarter number, always without leading zeros (1 to 4).
- "qr" or "QR": quarter in Roman numerals following the case of the format string ("i" to "iv" or "I" to "IV".)

Months

Month formats may represent two-digit month values with or without leading zeros, three-letter abbreviations for the month, or the full month name. The text identifiers may be all lowercase, all uppercase or "namecase" in which we capitalize the first letter of the month identifier. The corresponding format strings are given by:

- "mm" or "MM": two-digit month without/with leading zeros.
- "mon", "Mon", or "MON": three-letter form of month, following the case of the format string("jan", "Feb", "MAR").
- "month", "Month", or "MONTH": full month name, following the case of the format string ("january", "February", "MARCH").

Weeks

Week of the year formats may be specified with or without leading zeros:

- "ww" or "WW": week of year (with first week starting from Jan 1st) without/with leading zeros.

Days

Day formats correspond to day of the year, business day of the year, day of the month, or day of the week, in various numeric and text representations.

- "ddd" or "DDD": day of year without/with leading zeros.
- "bbb" or "BBB": business day of year without/with leading zeros (only counting Monday-Friday).
- "dd" or "DD": day of month without/with leading zeros.
- "day" or "DAY": day of month with suffix, following the case of the format string ("1st", "2nd", "3RD").
- "w" or "W": weekday number (1-7) where 1 is Monday.
- "wdy", "Wdy", or "WDY": three-letter weekday abbreviation, following the case of the format string ("Mon", "Tue", "WED").
- "weekday", "Weekday", or "WEEKDAY": full weekday name, following the case of the format string ("monday", "Tuesday", "WEDNESDAY").

Time (Hours/Minutes/Seconds)

The time formats correspond to hours (in 12 or 24 hour format), minutes, seconds, and fractional sections, with or without leading zeros and with or without the AM/PM indicator where appropriate.

- "hh" or "HH": hour in 24-hour format without/with leading zeros.
- "hm" or "HM": hour in 12-hour format without/with leading zeros.
- "am" or "AM": two letter AM/PM indicator for 12-hour format, following the case of the format string.
- "a" or "A": single letter AM/PM indicator for 12-hour format, following the case of the format string.
- "mi" or "MI": minute, always with leading zeros.
- "ss.s", "ss.s", "ss.ss", or "ss.sss": seconds and tenths, hundreths, and thousandths-of-a-second, with leading zeros. The capitalized forms of these formats ("SS", "SS.S", ...) yield identical results.

Delimiters

You may use text to delimit the date format components:

- "f" or "F": use frequency delimiter taken from the active, regular frequency workfile page. The delimiter corresponds to the letter associated with the current workfile frequency ("a", "m", "q", ..., "A", "M", "Q", ...), following the case of the format string, or the colon (":"), as determined by the Global Options setting (**Options/General Options.../Date representation**).
- "?" : used as "wildcard" single character for skipping a character formats used on date number input. Passed through to the output string.

- Other alphabetical characters are errors unless they are enclosed in square brackets *e.g.* "[Q]", in which case they are passed through to the output (for example, the "standard-EViews" quarterly format is "YYYY[Q]Q", where we use a four digit year identifier, followed by a "Q" delimiter/identifier, followed by a single digit for the quarter "1990Q2").
- All other characters (*e.g.*, punctuation) are passed through to the input or output without special interpretation.

Translating between Date Strings and Date Numbers

There are times when it is convenient to work with date strings, and times when it is easier to work with date numbers.

For example, when we are describing or viewing a specific date, it is easier to use a "human readable" date string such as "2002-Mar-20", "3/20/2002", or "March 20, 2002 12:23 pm" than the date number 730928.515972.

Alternatively, since date strings are merely text representations of dates, working with date numbers is essential when manipulating calendar dates to find elapsed days, months or years, or to find a specific date and time 31 days and 36 hours from now.

Accordingly, translating between string representations of dates and date numbers is one of the more important tasks when performing advanced operations with dates in EViews. These translations occur in many places in EViews, ranging from the interpretation of date strings in sample processing, to the spreadsheet display of series containing date numbers, to the import and export of data from foreign sources.

In most settings, the translations take place automatically, without user involvement. For example, when you enter a sample command of the form

```
smpl 1990q1 2000q4
```

EViews automatically converts the date strings into a range of date numbers. Similarly, when you edit a series that contains date numbers, you typically will enter your data in the form of a date string such as

```
"2002-Mar-20"
```

which EViews will automatically translate into a date number.

In other cases, you will specifically request a translation by using the built-in EViews functions `@datestr` (to convert a date number to a string) and `@dateval` (to convert a date string to a date number).

For example, the easiest way to identify the date 1,000 days after May 1, 2000 is first to convert the string value "May 1, 2000" into a date number using `@dateval`, to manipulate the date number to find the value 1000 days after the original date, and finally to convert the

resulting date number back into a string using `@datestr`. See "Formatted Conversion" on page 92 and "Manipulating Date Numbers" on page 97 for additional details.

All translations between dates strings and date numbers involve one of two methods:

- First, EViews may perform a *free-format conversion* in which the date format is inferred from the string values, in some cases other contextual information.
- Second, EViews may perform a *formatted conversion* in which the string representation of the dates is provided explicitly via a date format.

For the most part, you should find that free-format conversion is sufficient for most needs. Nevertheless, in some cases the automatic handling of dates by EViews may not produce the desired results. If this occurs, you should either modify any ambiguous date formats, or specify an explicit formatted conversion to generate date numbers as necessary.

Free-format Conversion

EViews will perform free-format conversions between date strings and numbers whenever: (1) there is an automatic translation between strings and numbers, or (2) when you use one of the translation functions without an explicit date format.

When converting from strings to numbers, EViews will produce a date number using the "most likely" interpretation of the date string. For the most part, you need not concern yourself with the details of the conversion, but if you require additional detail on specific topics (*e.g.*, handling of date intervals, the implicit century cutoff for 2-digit years) see "Free-format Conversion Details" on page 102.

When converting from date numbers to strings, EViews will use the global default settings to determine the default date format, and will display all significant information in the date number.

Converting Unambiguous Date Strings to Numbers

The free-format conversion of unambiguous date strings (see "Date Strings" on page 83), to numbers will produce identical results in all settings. The date string:

```
"March 3rd, 1950"
```

will be interpreted as the third day of the third month of the year A.D. 1950, and will yield the date value 711918.0. Note that the date value is the smallest associated with the given date, corresponding to 12 midnight.

Similarly, the date string:

```
"1980Q3"
```

is interpreted as the first instance in the third quarter of 1980. EViews will convert this string into the date number representing the smallest date value in that quarter, 722996.0 (12 midnight on July 1, 1980).

If we specify a time string without a corresponding day,

```
"9:52PM"
```

the day portion of the date is set to 0 (effectively, January 1, A.D. 1), yielding a value of 0.91111111 (see "Incomplete Date Numbers" on page 102) for details.

Consider also the following *ambiguous* date string:

```
"1 May 03"
```

While this entry may appear to be ambiguous since the "03" may reasonably refer to either 1903 or 2003, EViews resolves the ambiguity by assuming that if the two-digit year is greater than or equal to 30, the year is assumed to be from the twentieth century, otherwise the year is assumed to be from the twenty first century (see "Two-digit Years" on page 103 for discussion). Consequently free-format conversion of two-digit years will produce consistent results in all settings.

Converting Ambiguous Date Strings to Numbers

Converting from ambiguous date strings will yield context sensitive results. In cases involving ambiguity, EViews will determine the most likely translation format by examining surrounding data or applicable settings for clues as to how the date strings should be interpreted.

The following contextual information is used in interpreting ambiguous free-form dates:

- For implicit period notation (*e.g.*, "1990:3") the current workfile frequency is used to determine the period.
- Choosing between ambiguous "mm/dd" or "dd/mm" formats is determined by examining the values of related date strings (*i.e.*, those in the same series), user-specified date/time display formats for a series or column of a spreadsheet, or by examining the EViews global setting for date display, (**Options/General Options.../Date representation**).

To fix ideas, we consider a few simple examples of the use of contextual information.

If you specify an ambiguous sample string, EViews will use the context in which the sample is used, the frequency of the workfile, to determine the relevant period. For example, given the sample statement

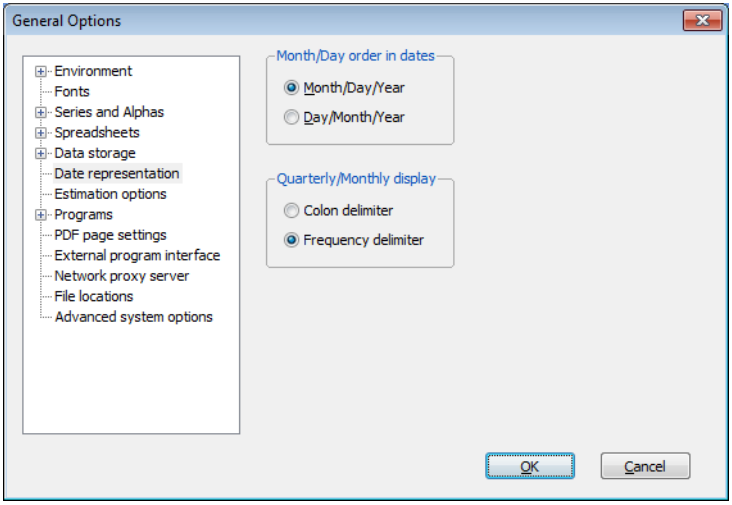
```
smpl 90:1 03:3
```

and a quarterly workfile, the sample will be set from 1990q1 to 2003q3. If the workfile is monthly, the sample will be set from January 1990 to March 2003.

Suppose instead that you are editing a series spreadsheet where your date numbers are *displayed* as dates strings using a specified format. In this setting, EViews allows you to enter your values as date strings, instead of having to enter the underlying date numbers. In this

context, it is natural for EViews to use the current display format as a hint in interpreting ambiguous data. For example, if the current display format is set to "Month dd, YYYY" then an input of "2/3/4" or "@dateval("2/3/4")" will be interpreted as February the 3rd, 2004. On the other hand, if the current display format is set to "YYYY-MM-DD" then the same input will be interpreted as the March the 4th, 2002.

In settings where an entire series is provided to an EViews procedure, EViews is able to use all of the values in the series to aid in determining the underlying data format. For example, when an alpha series is provided as a date identifier for restructuring a workfile, EViews will first scan all the values of the series in order to decide on the most likely format of all of the data before converting the string in each element into a date number. If the first observation of the series is an ambiguous "2/3/4" but a later observation is "3/20/95" then the "2/3/4" will be interpreted as the 3rd of February 2004 since that is the only order of year, month and day that is consistent with the "3/20/95" observation.



Conversely, when generating new series values with a `genr` or series assignment statement, EViews processes observation individually and is therefore unable to obtain contextual information to aid in interpreting ambiguous date strings. In this case, EViews will use the global workfile setting for the **Month/Day order in dates** to determine the ordering of the days and months in the string.

For example, when the expression

```
series dnums = @dateval("2/3/4")
```

is used to generate a series containing date values, EViews will interpret the value as February 3, 2004, if the global setting is **Month/Day/Year**, and March 2, 2004, if the global setting is **Day/Month/Year**.

Converting Date Numbers to Strings

EViews provides the `@datestr` function to translate a date number to a date string. We describe the function in detail in "Formatted Conversion" on page 92, but for now, simply note that `@datestr` takes an optional argument describing the date format to be used when exporting the string. If the optional argument is not provided, EViews will perform a free-format conversion.

In performing the free-format conversion, EViews examines two pieces of information. First, the global default settings for the **Month/Day order in dates** will be used to determine the ordering of days and months in the string. Next, EViews examines the date values to be translated and looks for relevant time-of-day information.

If there is no relevant time-of-day information in the date numbers (*e.g.*, the non-integer portions are all zero), EViews writes the string corresponding to the date using either the date format

```
"dd/mm/yyyy"
```

or

```
"mm/dd/yyyy"
```

with preference given to the order favored in the global settings.

If there is relevant time-of-day information, EViews will extend the date format accordingly. Thus, if days are favored in the ordering, and relevant hours (but not minutes and seconds) information is present, EViews will use

```
"dd/mm/yyyy hh"
```

while if hours-minutes are present, the format will be

```
"dd/mm/yyyy hh:mi"
```

and so forth.

Formatted Conversion

While the free-format conversions will generally produce the desired results, there may be times when you want to exercise precise control over the conversion. EViews will perform a formatted conversion between date strings and date numbers whenever you use the `@dateval` or `@datestr` functions *with* the optional second argument specifying an explicit date format.

To convert a date string into a date number using a date format, you should use the `@dateval` function with two arguments. The first argument must be a valid date string, and the second must be the corresponding date format string. If you omit the optional second argument, EViews will perform a free-format conversion.

- `@dateval`(*str[, fmt]*): takes the string *str* and evaluates it to a date number using the optional date format string, *fmt*.

A few simple examples will illustrate the wide range of string to date number conversions that are possible using `@dateval` and a date format. The simplest format strings involve the standard month/day/year date descriptions:

```
@dateval("12/1/1999", "mm/dd/yyyy")
```

will return the date number for December 1, 1999 (730088),

```
@dateval("12/1/1999", "dd/mm/yyyy")
```

returns the date number for January 12, 1999 (729765). Here we have changed the interpretation of the date string from "American" to "European" by reversing the order of the parts of the format string.

Likewise, we may find the first date value associated with a given period

```
@dateval("1999", "yyyy")
```

returns the value 729754.0 corresponding to 12 midnight on January 1, 1999, the first date value for the year 1999.

Conversion of an broad range of date strings is possible by putting together various date format string components. For example,

```
@dateval("January 12, 1999", "Month dd, yyyy")
```

returns the date number for 12 midnight on January 12, 1999 (729765), while

```
@dateval("99 January 12, 9:37 pm", "yy Month dd, hm:mi am")
```

yields the value 729765.900694 corresponding to the same date, but at 9:37 in the evening. In this example, the "hm:mi" corresponds to hours (in a 12 hour format, with no leading 0's) and minutes, and the "am" indicates that there is an indicator for "am" and "pm". See "Date Strings" on page 83 and "Date Formats" on page 85 for additional details.

To translate a date number to a date string using a date format, you should use the `@datestr` function with two arguments. The first argument must be a valid date number, and the second must be a date format string describing a string representation of the date.

- `@datestr`(*date_val[, fmt]*): converts the date number into a string, using the optional date format *fmt*. If a format is not provided, EViews will use a default method (see "Converting Date Numbers to Strings" on page 92).

For example,

```
@datestr(730088,"mm/dd/yy")
```

will return "12/1/99",

```
@datestr(730088,"DD/mm/yyyy")
```

will return "01/12/1999", and

```
@datestr(730088,"Month dd, yyyy")
```

will return "December 1, 1999", and

```
@datestr(730088,"w")
```

will produce the string "3", representing the weekday number for December 1, 1999. See "Date Numbers" on page 84 and "Date Formats" on page 85 for additional details.

Translating Ordinary Numbers into Date Numbers

While date information is most commonly held in the form of date strings or date numbers, one will occasionally encounter data in which a date is encoded as a (non-EViews format) numeric value or values. For example, the first quarter of 1991 may be given the numeric representation of 1991.1, or the date "August 15, 2001" may be held in the single number 8152001, or in three numeric values 8, 15, and 2001.

The `@makedate` function is used to translate ordinary numbers into date numbers. It is similar to `@dateval` but is designed for cases in which your dates are encoded in one or more numeric values instead of date strings:

- `@makedate`(*arg1[, arg2[,arg3]], fmt*): takes the numeric values given by the arguments *arg1*, and optionally, *arg2*, etc. and returns a date number using the required format string, *fmt*. Only a subset of all date formats are supported by `@makedate`.

  If more than one argument is provided, the arguments must be listed from the lowest frequency to the highest, with the first field representing either the year or the hour.

The simplest form of `@makedate` involves converting a single number into a date or a time. The following are the supported formats for converting a single number into a date value:

- "yy" or "yyyy": two or four-digit years.
- "yys" or "yyyys": year*10 + half-year.
- "yy.s" or "yyyy.s": year + half-year/10.
- "yyq" or "yyyyq": year*10 + quarter.
- "yy.q" or "yyyy.q": year + quarter/10.
- "yymm" or "yyyymm": year*10 + month.
- "yy.mm" or "yyyy.mm": year + month/10.
- "yyddd" or "yyyyddd": year*1000 + day in year.
- "yy.ddd" or "yyyy.ddd": year + day in year/1000.
- "yymmdd" or "yyyymmdd": year*10000 + month*100 + day in month.
- "mmddyy": month*10000 + day in month*100 + two-digit year.

- "mmddyyyy": month*100000 + day in month*10000 + four-digit year.
- "ddmmyy": day in month*10000 + month*100 + two-digit year.
- "ddmmyyyy": day in month*1000000 + month*10000 + four-digit year.

The following formats are supported for converting a single number into intraday values:

- "hh": hour in day (in 24 hour units)
- "hhmi": hour*100 + minute.
- "hhmiss": hour*10000 + minute*100 + seconds.

Note that the `@makedate` format strings are not case sensitive, since the function requires that *all* non-leading fields must have leading zeros where appropriate. For example, when using the format "YYYYMMDD", the date March 1, 1992 must be encoded as 19920301, and not 199231, 1992031, or 1992301.

Let us consider some specific examples of `@makedate` conversion of a single number. You may convert a numeric value for the year into a date number using a format string to describe the year. The expressions:

```
@makedate(1999, "yyyy")
@makedate(99, "yy")
```

both return the date number 729754.0 corresponding to 12 midnight on January 1, 1999. Similarly, you may convert a numeric value into the number of hours in a day using expressions of the form,

```
@makedate(12, "hh")
```

Here, EViews will return the date value 0.5 corresponding to 12 noon on January 1, A.D. 1. While this particular date value is not intrinsically of interest, it may be combined with other date values to obtain the value for a specific hour in a particular day. For example using date arithmetic, we may add the 0.5 to the 729754.0 value (12 midnight, January 1, 1999) obtained above, yielding the date value for 12 noon on January 1, 1999. We consider these sorts of operations in greater detail in "Manipulating Date Numbers" on page 97.

If your number contains "packed" date information, you may interpret the various components using `@makedate` with an appropriate format string. For example,

```
@makedate(199003, "yyyymm")
@makedate(1990.3, "yyyy.mm")
@makedate(1031990, "ddmmyyyy")
@makedate(30190, "mmddyy")
```

all return the value 726526.0, representing March 1, 1990.

Cases where `@makedate` is used to convert more than one argument into a date or time are more limited and slightly more complex. The arguments must be listed from the lowest frequency to the highest, with the first field representing either the year or the hour, and the remaining fields representing sub-periods. The valid date format strings for the multiple argument `@makedate` are a subset of the date format strings, with components applied sequentially to the numeric arguments:

- "yy s" or "yyyy s": two or four-digit year and half-year.
- "yy q" or "yyyy q": year and quarter.
- "yy mm" or "yyyy mm": year and month.
- "yy ddd" or "yyyy ddd": year and day in year.
- "yy mm dd" or "yyyy mm dd": year, month, and day in month.

Similarly, the valid formats for converting multiple numeric values into a time are:

- "hh mi": hour*100 + minute.
- "hh mi ss": hour*10000 + minutes*100 + seconds.

For convenience, the non-space-delimited forms of these format strings are also supported (*e.g.*, "yymm", and "hhmi").

For example, the expressions,

```
@makedate(97, 12, 3, "yy mm dd")
@makedate(1997, 12, 3, "yyyymmdd")
```

will return the value 729360.0 corresponding to midnight on December 3, 1997. You may provide a subset of this information so that

```
@makedate(97, 12, "yymm")
```

returns the value 729358.0 representing the earliest date and time in December of 1997 (12 midnight, December 1, 1997). Likewise,

```
@makedate(1997, 37, "yyyy ddd")
```

yields the value 729060.0 (February 6, 1997, the 37th day of the year) and

```
@makedate(14, 25, 10, "hh mi ss")
```

returns the value 0.600810185 corresponding to 2:25:10 pm on January 1, A.D. 1.

It is worth pointing out that in the examples above, the numeric arguments are entered from lowest frequency to high, as required. The following example, in which days appear before months and years, is *not* a legal specification

```
@makedate(7, 10, 98, "dd mm yy")
```

and will generate an error reporting a "Bad date format".

Lastly, we note that limitations on the date formats supported by `@makedate` imply that in some cases, you are better off working with strings and the `@dateval` function. In cases, where `@makedate` does not support a desired conversion, you should consider converting your numbers into strings, performing string concatenation, and then using the richer set of `@dateval` conversions to obtain the desired date values.

Manipulating Date Numbers

One of the most important reasons for holding your date information in the form of date numbers is so that you may perform sophisticated calendar operations.

Date Operators

Since date values are simply double precision numbers, you may perform standard mathematical operations using these values. While many operations such as division and multiplication do not preserve the notion of a date number, you may find addition and subtraction and relational comparison of date values to be useful tools.

If, for example, you add 7 to a valid date number, you get a value corresponding to the same time exactly seven days later. Adding 0.25 adds one-quarter of a day (6 hours) to the current time. Likewise, subtracting 1 gives the previous day, while subtracting 0.5 gives the date value 12 hours earlier. Taking the difference between two date values yields the number of days between the two values.

While using the addition and subtraction operators is valid, we strongly encourage you to use the EViews specialized date functions since they allow you to perform arithmetic at various frequencies (other than days), while taking account of irregularities in the calendar (see "Functions for Manipulating Dates" on page 97).

Similarly, while you may round a date number down to the nearest integer to obtain the first instance in the day, or you may round down to a given precision to find the first instance in a month, quarter or year, the built-in functions provide a set of simple, yet powerful tools for working with dates.

Note further that all of the relational operators are valid for comparing date numbers. Thus, if two date numbers are equal, the " = ", " > = ", and " < = " relational operators all return a 1, while the " < > ", " > ", and " < " comparison operators return a 0. If two date numbers are not equal, " < > " returns a 1 and " = " returns a 0. If the first date number is less than a second date number, the corresponding first date precedes the second in calendar time.

Functions for Manipulating Dates

EViews provides several functions for manipulating dates that take date numbers as input and return numeric values that are also date numbers. These functions may be used when you wish to find a new date value associated with a given date number, for example, a date number 3 months before or 37 weeks after a given date and time.

The functions described below all take a *time unit string* as an argument. As the name suggests, a time unit string is a character representation for a unit of time, such as a month or a year. The valid time unit string values are: "A" or "Y" (annual), "S" (semi-annual), "Q" (quarters), "MM" (months), "WW" (weeks), "DD" (days), "B" (business days), "HH" (hours), "MI" (minutes), "SS" (seconds).

There are three primary functions for manipulating a date number:

- `@dateadd`(*date1*, *offset[, u]*): returns the date number given by *date1* offset by *offset* time units as specified by the time unit string *u*. If no time unit is specified, EViews will use the workfile regular frequency, if available.

  Suppose that the value of *date1* is 730088.0 (midnight, December 1, 1999). Then we can add and subtract 10 days from the date by using the functions

  ```
  @dateadd(730088.0, 10, "dd")
  ```

  ```
  @dateadd(730088.0, -10, "dd")
  ```

  which return 730098.0 (December 11, 1999) and (730078.0) (November 21, 1999). Note that these results could have been obtained by taking the original numeric value plus or minus 10.

  The `@dateadd` function allows for date offsets specified in various units of time. For example, to add 5 weeks to the existing date, simply specify "W" or "WW" as the time unit string, as in

  ```
  @dateadd(730088.0, 5, "ww")
  ```

  which returns 730123.0 (January 5, 2000).

- `@datediff`(*date1*, *date2[, u]*): returns the number of time units between *date1* and *date2*, as specified by the time unit string *u*. If no time unit is specified, EViews will use the workfile regular frequency, if available.

  Suppose that *date1* is 730088.0 (December 1, 1999) and *date2* is 729754.0 (January 1, 1999), then,

  ```
  @datediff(730088.0, 729754.0, "dd")
  ```

  returns 334 for the number of days between the two dates. Note that this is result is simply the difference between the two numbers.

  The `@datediff` function is more powerful in that it allows us to calculate differences in various units of time. For example, the expressions

  ```
  @datediff(730088.0, 729754.0, "mm")
  ```

  ```
  @datediff(730088.0, 729754.0, "ww")
  ```

  return 11 and 47 for the number of months and weeks between the dates.

- `@datefloor`(*date1*, *u[, step]*): finds the first possible date number in the given time unit, as in the first possible date value in the current quarter, with an optional step offset.

  If *step* is omitted, the frequency will use a step of 1 so that by default, `@datefloor` will find the beginning of the period defined by the time unit.

  Suppose that *date1* is 730110.5 (12 noon, December 23, 1999). Then the `@datefloor` values

  ```
  @datefloor(730110.5, "dd")
  @datefloor(730110.5, "mm")
  ```

  yield 730110.0 (midnight, December 23, 1999) and 730088.0 (midnight, December 1, 1999), since those are the first possible date values in the current day and month. Note that the first value is simply the integer portion of the original date number, but that the latter required more complex analysis of the calendar.

  Likewise, we can find the start of any corresponding period by using different time units:

  ```
  @datefloor(730098.5, "q")
  @datefloor(730110.5, "y", 1)
  ```

  returns 730027.0 (midnight, October 1, 1999), and 729754.0 (midnight, January 1, 1999. Notice that since the latter example used an offset value of 1, the result corresponds to the first date value for the year 1999, which is the start of the year following the actual value.

Extracting Information from Date Numbers

Given a date number you may wish to extract numeric values associated with a portion of the value. For example, you might wish to know the value of the month, the year, or the day in the year associated with a given date value. EViews provides the `@datepart` function to allow you to extract the desired information.

- `@datepart`(*date1*, *u*): returns a numeric part of a date value given by *u*, where *u* is a time unit string.

Consider the *date1* date value 730110.5 (noon, December 23, 1999). The `@datepart` values for

```
@datepart(730110.5, "dd")
@datepart(730110.5, "w")
@datepart(730110.5, "ww")
@datepart(730110.5, "mm")
@datepart(730110.5, "yy")
```

are 23 (day of the month), 1 (day in the week), 52 (week in the year), 12 (month in the year), and 99 (year), respectively.

Note that the numeric values returned from `@datepart` are not themselves date values, but may be used with `@makedate` to create date values.

Special Date Functions

In addition to the functions that convert strings or numbers into date values, EViews provides the following special ways to obtain one or more date values of interest.

- `@now`: returns the date number associated with the current time.

The remaining functions return information for each observation in the current workfile.

- `@date`: returns the date number corresponding to every observation in the current workfile.
- `@year`: returns the four digit year in which the current observation begins. It is equivalent to `@datepart(@date, "YYYY")`
- `@quarter`: returns the quarter of the year in which the current observation begins. It is equivalent to `@datepart(@date, "Q")`.
- `@month`: returns the month of the year in which the current observation begins. It is equivalent to `@datepart(@date, "MM")`.
- `@day`: returns the day of the month in which the current observation begins. It is equivalent to `@datepart(@date, "DD")`.
- `@weekday`: returns the day of the week in which the current observation begins, where Monday is given the number 1 and Sunday is given the number 7. It is equivalent to `@datepart(@date, "W")`.
- `@hour`: returns the current observation hour as an integer. For example, 9:30AM returns 9, and 5:15PM returns 17.
- `@minute`: returns the current observation minute as an integer. For example, 9:30PM returns 30.
- `@second`: returns the current observation second as an integer.
- `@hourf`: returns the current observation time as a floating point hour. For example, 9:30AM returns 9.5, and 5:15PM returns 17.25.
- `@strdate`(*fmt*): returns the set of workfile row dates as strings, using the date format string *fmt*. See "Date Formats" on page 85 for a discussion of date format strings.

The `@date` function will generally be used to create a series containing the date value associated with every observation, or as part of a series expression involving date manipulation. For example:

```
series y = @date
series x = @dateadd(@date, 12, "ww")
```

which generates a series containing the date values for every observation, and the date values for every observation 12 weeks from the current values.

`@strdate` should be used when you wish to obtain the date string associated with every observation in the workfile—for example, to be used as input to an alpha series. It is equivalent to using the `@datestr` function on the date number associated with every observation in the workfile.

Free-format Conversion Formats

EViews supports the free-format conversion of a wide variety of date strings in which the string is analyzed for the most likely corresponding date.

Any of the following date string types are allowed:

Day, month, year

- “"YYYY-MM-DD"” (IEEE, with the date enclosed in double quotes)
- “dd/mm/yy” (if American, “mm/dd/yy” instead)
- “dd/mm/yyyy” (if American, “mm/dd/yyyy” instead)
- “yyyy/mm/dd”
- “dd/mon/yy”
- “dd/mon/yyyy”
- “yyyy/mon/dd”
- “ddmmyy” (if American, “mmddyy”)
- “ddmmyyyy” (if American, “mmddyyyy”)

The resulting date values correspond to the first instance in the day (12 midnight).

Month in year

- “mon/yy”
- “mon/yyyy”
- “yy/mon”
- “yyyy/mon”

The results are rounded to the first instance in the month (12 midnight of the first day of the month).

Period in year

- "yyyy[S|Q|M|W|B|D|T|F|:]period"
- "yy[S|Q|M|W|B|D|T|F|:]period"

The date value is rounded to the first instance in the period in the year

Whole year

- "yyyy[A]". The "A" is generally optional, but required if current WF is undated.
- "yy[A]". The "A" is generally optional, but required if current WF is undated.

The date value is rounded to the first instance in the year (12 midnight on January 1).

Free-format Conversion Details

Note that the following conventions may be used in interpreting ambiguous free-form dates.

Dates and Date Intervals

A date in EViews is generally taken to represent a single point in calendar time. In some contexts, however, a date specification is used to refer to a range of values contained in a time, which can be referred to as an interval.

When a date specification is treated as an interval, the precision with which the date is specified is used to determine the duration of the interval. For example, if a full day specification is provided, such as "Oct 11 1980", then the interval is taken to run from midnight at the beginning of the day to just before midnight at the end of the day. If only a year is specified, such as "1963", then the interval is taken to run from midnight on the 1st of January of the year to just before midnight on the 31st of December at the end of the year.

An example where this is used is in setting the sample for a workfile. In this context, pairs of dates are provided to specify which observations in the workfile should be included in the sample. The pairs of dates are provided are processed as intervals, and the sample is defined to run from the start of the first interval to the end of the second interval. As an example, if the sample "1980q2 1980q2" is specified for a daily file, the sample will include all observations from April 1st 1980 to June 30th 1980 inclusive.

Incomplete Date Numbers

An EViews date number can be used to represent both a particular calendar day, and a particular time of day within that day. If no time of day is specified, the time of day is set to midnight at the beginning of the day.

When no date is specified, the day portion of a date is effectively set to 1st Jan A.D. 1. For example, the date string "12 p.m." will be translated to the date value 0.5 representing 12 noon on January 1, A.D. 1. While this particular date value is probably not of intrinsic inter-

est, it may be combined with other information to obtain meaningful values. See "Manipulating Date Numbers" on page 97

Two-digit Years

In general, EViews interprets years containing only two digits as belonging to either the twentieth or twenty-first centuries, depending on the value of the year. If the two digit year is greater than or equal to 30, the year is assumed to be from the twentieth century and a century prefix of "19" is added to form a four digit year. If the number is less than 30, the year is assumed to be from the twenty first century and a century prefix of "20" is added to form a four digit year.

Note that if you wish to refer to a year after 2029 or a year before 1930, you must use the full four-digit year identifier.

Because this conversion to four digit years is generally performed automatically, it is not possible to specify years less than A.D. 100 using two digit notation. Should the need ever arise to represent these dates, such two digit years can be input directly by specifying the year as a four digit year with leading zeros. For example, the 3rd of April in the year A.D. 23 can be input as "April 3rd 0023".

Implicit Period Notation

In implicit period notation (*e.g.*, "1990:3"), the current workfile frequency is used to determine the period.

American vs. European dates

When performing a free-format conversion in the absence of contextual information sufficient to identify whether data are provided in "mm/dd" or "dd/mm" format, the global workfile setting for the **Options/Dates & Frequency Conversion.../Month/Day order in dates** ("Date Representation" on page 770 of the *User's Guide I*) will be used to determine the ordering of the days and months in the string.

For example, the order of the months and years is ambiguous in the date pair:

```
1/3/91 7/5/95
```

so EViews will use the default date settings to determine the desired ordering. We caution you, however, that using default settings to define the interpretation of date strings is not a good idea since a given date string may be interpreted in different ways at different times if your settings change. You may instead use the IEEE standard format, "YYYY-MM-DD" to ensure consistent interpretation of your daily date strings. The presence of a dash in the format means that you must enclose the date in quotes for EViews to accept this format. For example:

```
smpl "1991-01-03" "1995-07-05"
```

will always set the sample to run from January 3, 1991 and July 5, 1995.

Time of Day

Free-format dates can also contain optional trailing time of day information which must follow the pattern:

```
hh[[[[:mi:]ss].s]s]s][am|AM|pm|PM]
```

where "[]" encloses optional portions or the format and "|" indicates one of a number of possibilities. In addition, either the "am" or "pm" field or an explicit minute field must be provided for the input to be recognized as a time. An hour by itself is generally not sufficient.

The time of day in an EViews date is accurate up to a particular millisecond within the day, although any date can always be displayed at a lower precision. When displaying times at a lower precision, the displayed times are always rounded down to the requested precision, and never rounded up.

When both a day and a time of day are specified as part of a date, the two can generally be provided one after the other with the two fields separated by one or more spaces. If, however, the date is being used in a context where EViews does not permit spaces between input fields, a single letter "t" can also be used to separate the day and time so that the entire date can be contained in a single word, *e.g.* "1990-Jan-03T09:53".

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.

You may, for example, write a program containing 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.

The remainder of this chapter outlines the basics of EViews programming. If you have experience with computer programming and batch or macro processing, you will find most of the features of the EViews language to be quite familiar. At the same time, non-programmers should feel welcome to examine the material as you need not have any experience with programming to take advantage of these powerful features.

Program Basics

What is a Program?

A program simply a text file containing EViews commands. It is not an EViews object in a workfile. It exists as a file on your computer hard disk, generally with a ".PRG" extension.

Creating a Program

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.

Program Formatting

As noted earlier, an EViews program is simply a text file, consisting of one or more lines of text. Generally, each line of a program corresponds to a single EViews command, so you may 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, by default, autowrap the text of the line. Autowrapping alters the appearance of the program line by display-

ing 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. You may turn off autowrapping in programs via **Options/General Options/Programs** and deselecting the **Enable word wrap** check box, or by clicking the **Wrap** +/- button on the program window.

When autowrapping is turned off via the option menu, you may elect to show program line numbers in your program window by selecting **Display line numbers**. You may then right-click anywhere in your program and select **Go To Line...** to jump directly to a specific line number.

If you desire 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 of text

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

are equivalent to the single line

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

formed by joining the lines at the continuation character. We emphasize that the "_" must be the very last character in the line.

The apostrophe "'" is the comment character in programs. You may place the comment character anywhere in a line to treat all remaining characters in the line as a comment which will be ignored when executing the program command.

```
equation eq1.ls y x c ar(1) ar(2) ' this is a comment
```

A block of lines may be commented or uncommented in the EViews program file editor by highlighting the lines, right-mouse clicking, and selecting **Comment Selection** or **Uncomment Selection**.

You can instruct EViews can automatically format your program by selecting the lines to which you wish to apply formatting, right-mouse clicking and selecting **Format Selection**. Automatic formatting will clean up the text in the selection, and will highlight the structure of the program by indenting lines of code inside loops, if conditions and subroutines, *etc.* You should find that automatic formatting makes your programs easier to read and edit.

You may elect to have EViews automatically indent your program as you type by changing the **Indent** option in the main EViews options menu (**Options/General Options/Programs**).

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".

If saving your program will overwrite an existing program file and you have instructed EViews to make backup copies ("Programs," on page 772 of *User's Guide I*), EViews will automatically create the backup. The backup copy will have the same name as the file, but with the first character in the extension changed to " ~ ".

Saving the Command Window

One convenient method of creating a program is to execute several commands using the EViews command window and then save the history of those commands to a program file. Click in the command window then select **File/Save As...** from the main EViews menu. EViews will prompt you to save the command log as a text file. Simply save the file with the ".PRG" extension.

You may then edit the program file and save the edited version in the usual fashion.

Encrypting a Program

EViews offers you the option of encrypting a program file so that you may distribute it to others in a form where they may not view the original text. Encrypted files may be opened and the program lines may be executed, but the source lines may not be viewed. To encrypt a program file simply click on the **Encrypt** button on the program window.

EViews will create an untitled program containing the contents of the original program, but with only the visible text "Encrypted program". You may save this encrypted program in the usual fashion using **Save** or **SaveAs**.

Note that *once a program is encrypted it may not be unencrypted; encryption should not be viewed as a method of password protecting a program*. You should always keep a separate copy of the original source program. To use an encrypted program, simply open it and run the program in the usual fashion.

Opening a Program

To load a program file previously saved on disk, click on **File/Open/Program...**, navigate to the appropriate directory, and click on the desired name. You may also drag an EViews program file onto the EViews window to open it. Alternatively, from the command line, you may type the keyword `open` followed by the full program name, including the file extension ".PRG". By default, EViews will look for the program in the default directory, but you may include the full path to the file and enclosed the entire name in quotations, if desired. For example, the commands:

```
open mysp500.prg
open "c:\mywork is here\eviews\myhouse.prg"
```

open the file "Mysp500.PRG" in the default EViews directory, and "Myhouse.PRG" located in the directory "c:\mywork is here\eviews".

Executing a Program

Executing a program is the process of running all of the commands in a program file.

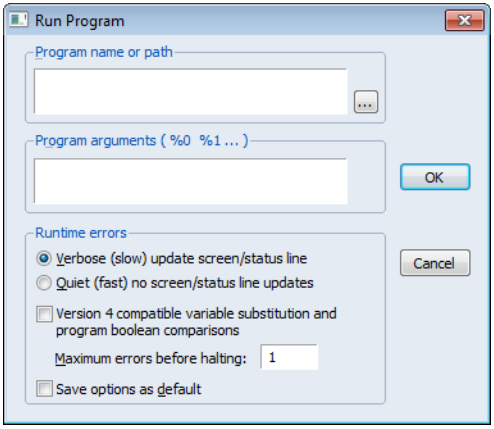
Note that EViews commands can be executed in two distinct ways. When you enter and run, line by line, a series of commands in the command window, we say that you are working in *interactive mode*. Alternatively, you can type all of the commands in a program and execute or run them collectively as a batch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

There are several ways to execute an EViews program:

- The easiest method is by pushing the **Run** button on an open program window and entering settings in the **Run Program** dialog.
- Alternately, you may use the `run` or `exec` command to execute a program.
- You may use external automation tools to run EViews and execute a program from outside of the EViews application environment.
- You may select a set of subset of lines to execute and run the selected lines.

The Run Program Dialog

To run a program from a dialog, click on the **Run** button on the program window. The **Run** dialog opens.



The default run options are taken from the global settings ("Programs" on page 772 of *User's Guide I*), but may be overridden on a one-time basis using the dialog settings or command options. For purposes of discussion, we will assume that the global options are set to their out-of-the-box values.

The **Program name or path edit field** will show the name and in some cases, path, of the program you are running. You may enter a file specification to instruct EViews to look for a particular program.

The **Program arguments** edit field is used to define special string variables that will be passed to your program when it is running. See "Program Arguments" on page 124 of *User's Guide I* for details.

The **Runtime errors** section allows you to modify the behavior of the program when it encounters an error. 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, if possible, continue to execute the program until the maximum number of errors is reached. If it encounters a serious error that makes it impossible to continue, EViews will halt the program even if the maximum number of errors is not reached. See "Execution Errors" on page 134.

If the **Compatibility** section checkbox labeled **Version 4 compatible variable substitution and program boolean comparisons** is selected, EViews will use the variable substitution behavior found in EViews 4 and earlier. To support the use of alpha series, EViews 5 and subsequent versions altered the way that % substitution variables are evaluated in expressions. To return to EViews 4 compatible rules for substitution, you may either use this checkbox or include a "MODE VER4" statement in your program. See "Version 4 Compatibility Notes" on page 155 and "Program Modes" on page 122 for additional discussion.

Lastly, you may select the **Save options as default** checkbox to update your global options with the specified settings. Alternately, you may change the global options from the **Options/General Options.../Programs/General** dialog.

The Run and Exec Commands

You may use the `run` command to execute a program from the EViews command window. Simply enter the keyword `run` along with any options, followed by a program file specification and any arguments (see `run` (p. 427)).

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

```
run mysp500
```

or, using an explicit path,

```
run c:\eviews\myprog arg1 arg2 arg3
```

Note that use of the ".PRG" extension is not required since EViews will automatically append one to your specification.

The default `run` options will be taken from the global settings ("Programs" on page 772 of *User's Guide I*), but may be overridden using the command options. For example, you may use the "v" or "verbose"options to run the program in verbose mode, and the "q" or "quiet" options 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:

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

or

```
run(q, ver4) progarg
```

Alternatively, you may modify your program to include statements for quiet or verbose mode, and statements to specify version compatibility. For example, to return to EViews 4 compatible rules for substitution, you may use the "ver4" option or include a "MODE VER4" statement in your program. See "Version 4 Compatibility Notes" on page 155 and "Program Modes" on page 122 for additional discussion.

You may provide a list of program arguments after the name of the program. These arguments will be passed on to the program as %0, %1 etc. See "Program Arguments" on page 124 for more details.

Program options may be passed on to the program by entering them, surrounded by parenthesis immediately after the name of the program. See "Program Options" on page 125 for details.

For example:

```
run myprog(opt1, opt2, opt3=k) arg0 arg1 arg1
```

will run the program MYPROG passing on the options OPT1, OPT2, OPT3 = k as options, and ARG0, ARG1 and ARG1 as arguments (%0, %1 and %2 respectively).

You may have launch EViews and run a program automatically on 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. If the program has as its last line the command `exit`, EViews will close following the execution of the program.

The `exec` command is similar to the `run` command. It can also be used to execute a program file. The main differences between `run` and `exec` commands are the default directory they will run from, and the behavior when returning from executing a program. For more details see "Multiple Program Files" on page 136.

External Automation Tools

Lastly, you may use external automation tools to run EViews and execute a program from outside of the EViews application environment. In particular, EViews may be used as a COM Automation server so that an external program or script may launch and control EViews programmatically. See "EViews COM Automation Server" on page 162 and the *EViews COM Automation Server* whitepaper, which is available in your EViews documentation directory or from our website www.eviews.com, for additional discussion.

Stopping a Program

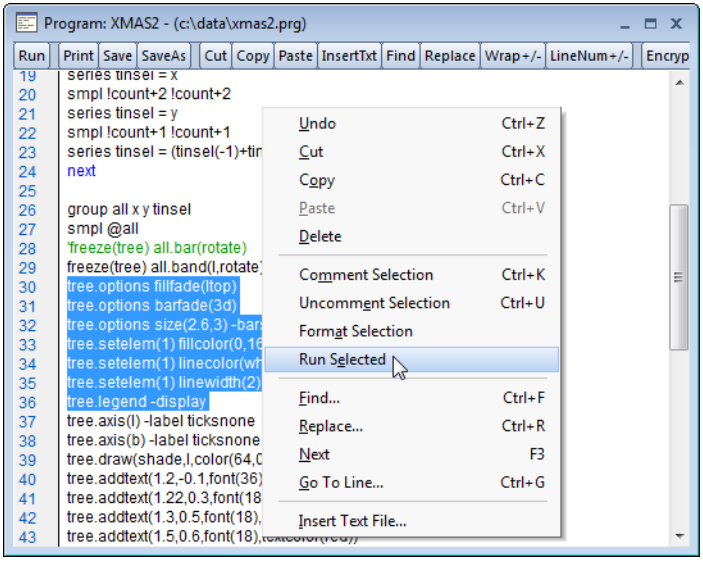
Pressing the ESC or F1 keys halts execution of a program. It may take a few seconds for EViews to respond to the halt command.

Programs will also stop when they encounter a `stop` command, when they reach the maximum number of errors, when then encounter a non-recoverable error in a program statement, 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.

Running Part of a Program

You may choose to only run part of your program by highlight the lines you wish to run, then right-clicking and selecting **Run Selected**. EViews will then execute only the selected line of code as a new program.



You should note that there are potential pitfalls when using **Run Selected**. The first is that no arguments or options can be passed into the selected lines of code. If the selected lines rely on program arguments or options, they will fail to execute properly. Similarly, any program variables ("Program Variables" on page 114) declared outside of the selected lines may not be initialized properly when the selected lines are run. Finally, any subroutines declared outside of the selected lines of code cannot be called, since they do not exist in the selected code.

Alternately, you may use add the `stop` command to your EViews program to halt execution at a particular place in the program file.

Simple Programs

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

While you may 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. (Note that you may combine the two methods by entering the commands in the command window and then save them to a file as described in "Saving the Command Window" on page 107.)

Since an EViews program is a collection of commands, familiarity with the EViews command language is an essential part of programming. The command language may be split into four different groups in EViews.

- General commands are commands that create, structure and manipulate workfiles (including importing and exporting data) and create workfile objects. These commands are listed in Chapter 12. "Command Reference," beginning on page 263.
- Object commands are commands that execute views and procedures on objects in the workfile (such as series, groups and equations).

  For example, unit root tests are done via a series command (`series.uroot`), co-integration tests are a group command (`group.coint`), and regression and forecasting are performed via equation commands (`equation.ls` and `equation.forecast`). The full set of object commands can be found in Chapter 1. "Object View and Procedure Reference," beginning on page 2, where they are listed by object type.
- Functions may be used to create new data, either in the form of workfile objects or as program variables.

  Creating a new series that is equal to the moving average of an existing series is done with the `@movav` function. The `@rbinom` function is used to generate a random scalar or series from the binomial distribution. `@pagefreq` may be used to generate a string object or program variable containing the frequency of the current workfile page. In general, functions all start with the @ symbol.

  Listings of the available @-functions available can be found in Chapters 13–19 ("Operator and Function Reference", "Operator and Function Listing", "Workfile Functions", "Special Expression Reference", "String and Date Function Reference", "Matrix Language Reference", and "Programming Language Reference").
- Object data members allow you to retrieve a fundamental piece of data from an object. Unlike object commands that execute a view or procedure on an object, data members merely provide access to existing results. Data members almost all start with

an @ symbol and can be accessed by typing the name of the object followed by a "." and the name of the data member.

For example, `eq1.@coefs` returns a vector containing the estimated coefficients from the equation object EQ1. `ser01.@last` returns a string containing the date (or observation number) of the last non-NA value in the series SER01. The list of data members available for each object is listed in the object's section of Chapter 1. "Object View and Procedure Reference," on page 2.

Let's examine 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
wfcreate(wf=myhouse) m 1968m3 1997m6
fetch progdemo::hsf
smpl 1968m5 1992m12
equation reg1.ls hsf c hsf(-1)
reg1.makeresid hsfres
smpl 1993m1 1997m6
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.

The line containing the `wfcreate` command creates a new workfile, called MYHOUSE, with a monthly frequency spanning the dates March 1968 to June 1997. A series called HSF (total housing units started) is fetched from the database PROGDEMO. The rest of the program results in a saved workfile named MYHOUSE containing the HSF series, an equation object REG1, residual and forecast series HSFRES and HSFFIT, and a graph HSFPLOT 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
```

```
wfcreate(wf=mysp500) m 1968m3 1997m6
fetch progdemo::fspcom
smpl 1968m5 1992m12
equation reg1.ls fspcom c fspcom(-1)
reg1.makeresid fspcomres
smpl 1993m1 1997m6
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 these two programs are almost identical, it seems inefficient to have two separate programs for performing the analysis. In "Program Arguments" on page 124 we describe a method for performing these two forecasting problems using a single program. First however, we must define the notion of program variables that exist solely when running programs.

Program Variables

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

Program variables are variables that you may use in place of numeric or string values in your EViews programs. Accordingly, there are two basic types of program variables: *control* (numeric) *variables* and *string variables*.

In the remainder of this section we describe the use of these two types of program variables.

Control Variables

Control variables are program variables that you can use in place of numeric 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.

It is important to note that control variables do not exist outside of your program and are automatically deleted after a program finishes. In this regard, control variables should not be confused with scalar objects as the former 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.

The name of a control variable starts with an "!" character. After the "!", the name should be a legal EViews name of 23 characters or fewer (leading numbers are allowed). Examples of control variable names are:

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

You need not declare control variables, but you must assign them values before use. Control variable values 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 1950q1+!i 1960q4+!i
```

For example, the following commands:

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

use the numeric values assigned to the control variables !VARX, !LENGTH, and !Z.

It is important to note that control variables are used in programs via text substitution. Whenever EViews encounters a control variable in a program it substitutes the text value of that variable into the program. *One of the implications of the text substitution is that you may lose some numeric accuracy when using a program variable due to the conversion in to and out of a text representation of the number.*

A second unintended consequence of text substitution can arise when raising a negative control variable to a power:

```
!a = -3
!b = !a^2
```

When evaluating these lines, EViews will substitute the value of !A in the second line, leaving the line:

```
!b = -3^2
```

Which it then evaluates as –9 (since the square operator takes precedence over the negation operator), rather than the expected 9.

You should also take care to avoid inadvertent replacement of program variables, as outlined in "Replacement Variables" on page 118.

String Variables

A *string expression* or *string* is text enclosed in double quotes:

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

A *string program variable* is a program variable whose value is a string of text. String variables, which exist only during the time that your program is executing, have names that begin with a "%" symbol. String variables should not be confused with "String Objects" on page 80 which are objects that exist in a workfile.

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, the following lines assign values to string variables:

```
%value = "value in millions of u.s. dollars"
%armas = "ar(1) ar(2) ma(1) ma(2)"
%mysample = " 83m1 96m12"
%dep = " hs"
%pi = " 3.14159"
```

You may use strings variables to help you build up command text, variable names, or other string values. EViews provides a number of operators and functions for manipulating strings; a complete list is provided in "Strings" on page 65.

Once assigned a value, a string variable may appear in any expression in place of the underlying string. When substituted for, the string variable will be replaced by the contents of the string variable, enclosed in double quotes.

Here is a simple example where we use string operations to concatenate the contents of three string variables.

```
!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 string variable assignments. After string variable substitution, the latter assignment is equivalent to entering

```
%st3 = "500" + " draws from the " + "Cauchy " + "distribution"
```

Similarly, the table assignment statement

```
table1(1,1) = %st3
```

is equivalent to entering the command

```
table(1,1) = "500 draws from the Cauchy distribution"
```

One important use for string variables is assigning string values to string objects, string vectors, or Alpha series. For example, we may have the assignment statement

```
%z = "Ralph"
alpha full_name = %z + last_name
```

which is equivalent to the expression

```
alpha full_name = "Ralph" + last_name
```

We again emphasize that string variable substitution involves replacing the string variable by its string value contents, enclosed in double quotes.

As with any string value, you may convert a string variable containing the text representation of 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 string cannot be evaluated to a number, `@val` returns the value "NA".

String variables are closely related to the concept of a string object ("String Objects," on page 80). A *string object* is an EViews workfile object that holds a string:

```
string a = "Hello World"
string b = """First Name"" Middle ""Last Name"""
```

Unlike string variables, string objects are named objects in the workfile that may exist apart from a running program.

In programming applications, string variables and string objects are sometimes defined using string literal expressions as illustrated above. However, in most settings, string variables and objects are defined using results from string functions, or by manipulation of string lists and string vectors. For example, you could use the `@wlookup` and `@wsplit` functions to obtain a set of strings corresponding to the names of all of the series in a workfile. See "Strings," on page 65 for a full discussion.

Replacement Variables

When working with EViews commands, you may wish to use a string variable, not simply to refer to a string value, but as *an indirect way of referring to something else*, perhaps a command, or an object name, or portion of names for one or more items.

Suppose, for example, that we assign the string variable %X the value "GDP":

```
%x = "gdp"
```

We may be interested, however, not in the actual string variable *value* "gdp", but rather in an EViews *object named* "GDP". Accordingly, we require a method of distinguishing between a string value and the item corresponding to the string value.

If you enclose a string variable in curly braces ("{ " and "}") EViews will replace the expression with the name, names, or name fragment given by the string value. In this context we refer to the expression "{%X}" as a *replacement variable* since the string variable %X is replaced in the command line by the name or names of objects to which the string refers.

Suppose we want to create the series Y and set it equal to the values in the series GDP. Given the string variable %X defined above, the series declaration,

```
series y = %x
```

creates a numeric series Y and sets it equal to the string value "gdp",

```
series y = "gdp"
```

which generates an error since the `series` declaration expects the *name* of a series or a numeric value, not a string value. In this circumstance, we would like to replace the string value with the name of an object. With the replacement variable "{%X}", the command

```
series y = {%x}
```

is properly interpreted as

```
series y = gdp
```

Similarly, the program line using replacement variables,

```
equation eq1.ls {%x} c {%x}(-1)
```

would be interpreted by EViews as

```
equation eq1.ls gdp c gdp(-1)
```

Changing the contents of %X to "M1" changes the interpretation of the original program line to

```
equation eq1.ls m1 c m1(-1)
```

since the replacement variable uses the name obtained from the new value of %X.

To take another example, when trying to find the number of valid (non-missing) observations in a series named INCOME, you may use the `@obs` function along with the name of the series:

```
@obs(income)
```

If you wish to use a string variable %VAR to refer to the INCOME series, you must use the replacement variable in the `@obs` function, as in

```
%var = "income"
@obs({%var})
```

since you wish to refer indirectly to the object named in %VAR. Note that the expression

```
@obs(%var)
```

will return an error since `@obs` requires a series or matrix object name as an argument.

Any string variable may be used as the basis of a replacement variable. Simply form your string using one or more string operations

```
%object = "group"
%space = " "
%reg1 = "gender"
%reg2 = "income"
%reg3 = "age"
%regs = %reg1 + %space + %reg2 + %space + %reg3
```

then enclose the string variable in braces. In the expression,

```
{%object} g1 {%regs}
```

EViews will substitute the names found in %OBJECT and %REGS so that the resulting command is

```
group g1 gender income age
```

It is worth noting that replacement variables may be used as building blocks to form object names. For example, the commands

```
%b = "2"
%c = "temp"
series z{%b}
matrix(2, 2) x{%b}
vector(3) x_{%c}_y
```

declare a series named Z2, a 2×2 matrix named X2, and a vector named X_TEMP_Y.

Up to this point, we have focused on replacement variables formed from string variables. However, control variables may also be used to form replacement variables. For example, the commands

```
!i = 1
series y{!i} = nrnd
!j = 0
series y{!j}{!i} = nrnd
```

are equivalent to

```
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. Conveniently, in cases where there is no possibility of ambiguity, EViews will treat a control variable as a replacement variable, even if the braces are not provided. For example:

```
!x = 3
series y!x = 4
```

will generate the series Y3 containing the value 4.

While convenient, this loose interpretation of control variables can, however, lead to unexpected results if one is not careful. Take the following program:

```
!x1 = 10
series y = !x1
```

This program will create a series equal to the value of !X1 (i.e. 10). However if you were to mis-type the program slightly:

```
!x = 10
series y = !x1
```

where the first line has `!X` rather than `!X1`, EViews will not generate an error due to the missing `!X1` variable, but will instead evaluate the second line by substituting `!X` into the expression, and evaluating the result as 101.

Replacement variables may be constructed using nested evaluation of replacement variables. Suppose we have the strings:

```
%str1 = "x"
%name = "%str1"
```

Then we can declare the series X and fill it with random normals using the command

```
series {{%name}} = nrnd
```

After evaluation of the innermost replacement variable, the expression reduces to

```
series {%str1} = nrnd
```

and then to

```
series x = nrnd
```

when we evaluate the remaining brace. The double braces allow us perform a double replacement in which the string variable used to form a replacement variable is itself obtained as a replacement variable.

The double nested braces need not be adjacent. For example:

```
%x1 = "x"
%x2 = "y"
scalar x = 10
scalar y = 20
!y = 1
scalar r1 = {%x{!y}}
!y = 2
scalar r2 = {%x{!y}}
```

First we create two string variables, %X1 and %X2, and three scalar objects, X, Y, and R. First, the control variable is !Y is set to 1 and the replacement variable {!Y} is used to construct the name "%X1" of a string variable. The resulting replacement variable {%X1} refers to the scalar object X. We assign the scalar X value of 10 to the scalar R1. Next, we set !Y to 2, the replacement variable {%X{!Y}} evaluates to Y, and we assign the Y value of 20 to the scalar R2.

Multiple sets of braces may be used to create various string names:

```
string uslabel = "USA"
string nzlabel = "New Zealand"
%a1 = "US"
%a2 = "NZ"
%b = "LABEL"
!y = 1
string label1 = {%a{!y}}{%b}
!y = 2
string label2 = {%a{!y}}{%b}
```

First we create two string objects, USLABEL and NZLABEL, which hold the label we wish to assign to each country. Then we create the string variables %A1 and %A2 to hold the country abbreviations. When the control variable !Y = 1, {%A{!Y}}{%B} evaluates to the object USLABEL and when !Y = 2, {%A{!Y}}{%B} evaluates to the object NZLABEL. Then the string LABEL1 contains "USA" and LABEL2 contains "New Zealand".

Replacement variables may also be formed using string objects in place of string variables. To use a string object as a replacement variable, simply surround it with curly braces ("{" and "}"). For example,

```
string LAGGDP = "GDP(-1) GDP(-2) GDP(-4)"
equation eq1.ls GDP C {LAGGDP}
```

executes the command

```
equation eq1.ls GDP C GDP(-1) GDP(-2) GDP(-4)
```

In most respects, there is no difference between using a string object or a string variable in a program. String objects have the advantage that they may be used interactively, outside of programs, while string variables may not. This can be useful when debugging a program; string variables disappear once the program has finished, making it difficult to identify problems in their construction. Note that string objects do, however, go out-of-scope when the active workfile page changes, while string variables are always in scope.

Lastly, while replacement variables provide you with great flexibility in referencing objects in your programs, used carelessly, they can lead to confusion. We suggest, for example, that you avoid using similar base names to refer to different objects. Consider the following program:

```
' possibly confusing commands (avoid)
!a = 1
series x{!a}
!a = 2
matrix x{!a}
```

In this 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 control variable assignment !A = 1 may be separated from the declaration by many program lines, it may be difficult to tell at a glance what kind of object X1 represents. A better approach might be to use different names for different kinds of variables:

```
!a = 1
series ser{!a}
!a = 2
matrix mat{!a}
```

so that the meaning of replacement variable names are more apparent from casual examination of the program.

Program Modes

Program modes describe different settings associated with the execution of your program. You may, for example, choose to provide verbose messaging where your program will echo each command to the status line, or you may choose quiet mode. Alternately, you may wish to run a legacy program file in Version 4 compatibility mode.

EViews provides you with the opportunity to set program execution modes at the time that the program is first run. In addition, you may use the "MODE" statement to change the execution mode of a program from within the program itself. One important benefit to using "MODE" statements is that the program can begin executing in one mode, and switch to a second mode as the program executes.

To change the mode for quiet or verbose mode, simply add a line to your program reading "MODE" followed by either the "QUIET" or the "VERBOSE" keyword, as in

```
mode quiet
```

For version 4 compatibility, you should use the keyword "VER4":

```
mode ver4
```

as a line in the body of the program.

Multiple settings may be set in a single "MODE" line:

```
mode quiet ver4
```

and multiple mode statements may be specified in a program to change the mode as the program runs:

```
mode quiet
[some program lines]
mode verbose
[additional program lines]
```

Note that setting the execution mode explicitly in a program overrides any settings specified at the time the program is executed.

Program Message Logging

When executing a program in EViews, it may be useful to keep track of what is happening during execution. Log windows allow you to determine more accurately the state of various objects in your workfile or follow program progression.

Log windows may be switched on using the `logmode` (p. 370) command. One log window is created for each program. If a program is executed more than once and a log window has already been created, the log window will be cleared and all subsequent messages will be sent to the existing log window. If you wish to preserve a log, you may either save the log to a text file or freeze it, creating a text file object.

There are several types of messages which can be logged: program lines, status line messages, user log messages, and program errors. When displayed in a log message, each type will appear in a different color, making it easier to differentiate one type from another. Program lines are echoes of the line of code in the program currently being executed, and are displayed in black. Status line messages are the messages displayed in the status line (see

"The Status Line" on page 9 of *User's Guide I*) and appear in blue. User log messages are custom messages created by the program via the `logmsg` (p. 372) command and appear in green. Program errors are messages generated when a logical or syntax error has occurred and appear in red.

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. The individual arguments will be available in the string variables "%0", "%1", "%2", and so on; the "%ARGS" variable will contain a space delimited list of all the arguments passed into the program.

When you run a program that takes arguments, you may 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` or `exec` command, you may list the arguments consecutively after the name of the program.

For example, suppose we have a program named "REGPROG":

```
equation eq1
smpl 1980q3 1994q1
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 Program** dialog or `run` command are treated as empty string variables. For example, suppose you have a one-line program named "REGRESS":

```
equation eq1.ls y c time {%0} {%1} {%2} {%3} {%4} {%5} {%6} {%7}
    {%8}
```

The command,

```
run regress x x(-1) x(-2)
```

executes

```
equation eq1.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":

```
wfcreate(wf={%0}) m 1968m3 1997m6
fetch progdemo::{%1}
smpl 1968m5 1992m12
equation reg1.ls {%1} c {%1}(-1)
reg1.makeresid {%1}res
smpl 1993m1 1997m6
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
```

Program Options

Much like program arguments, *program options* are special string variables that may be passed to your program when you run the program. You may specify options by providing a comma separated list of options in parentheses in the `run` or `exec` command statement, immediately after the name of the program as in:

```
run myprogram(myoption1, myoption2)
```

Note that options are only supported via the command line method using `run` or `exec`. You cannot pass an option to a program when running a program via the menus. Options are included via the command line by entering them in parenthesis immediately following the name of the program to be run.

In contrast with arguments, options may not be accessed directly from within your program. Rather you can only test for the existence of an option, or retrieve part of an option. The `@hasoption` command lets you test for the existence of an option. For example, `@hasoption("k")` will return a value of 1 if the "k" option was passed into the program at run time, or 0 if it was not.

`@equaloption` may be used to return the value to the right of an equality sign in an option. For example if "cov = hac" is entered as an option, `@equaloption("cov")` would return "hac". If the option was not entered at all, `@equaloption` will return an empty string.

For example, suppose you have the following program:

```
%filename = @equaloption("file")
wfcreate(wf={%filename}) m 1968m3 1997m6
fetch progdemo::{%0}
if (@hasoption("LS")=1) then
    smpl 1968m5 1992m12
    equation reg1.ls {%0} c {%0}(-1)
endif
```

If you were to run this program with:

```
run myprog(file=myhouse, ls) hsf
```

the program would create a workfile called MYHOUSE, would fetch a series called HSF, and would then create an equation called REG1 by performing least squares using the series HSF (for discussion of the "if" condition used in this example, see "IF Statements" on page 127).

If we had run the program with just:

```
run myprog hsf
```

the workfile would not have been named (and would be given the default name of UNTITLED), and the regression would not have been run.

Note that if your program name has spaces or illegal characters, it must be enclosed within quotes in `run` or `exec` commands. In this case, program options should be included after the closing quote without a space. For example, if we were to name our above program as MY PROG, then the correct method to issue options is:

```
run "my prog"(file=myhouse, ls) hsf
```

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 repeatedly under changing conditions. The tools for controlling execution will be familiar from other computer languages.

IF Statements

There are situations where you may want to execute commands only if some condition is satisfied. EViews uses IF and ENDIF, or IF, ELSE, and 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.

All comparisons in the IF statement follow the rules outlined in "String Relational Operators" on page 67 and in "Numeric Relational Operators" on page 168 of *User's Guide I*. Note in particular that string comparisons are case-sensitive. You may perform caseless comparison using the `@upper` or `@lower` string functions as in

```
if (@lower(%x) = "abc") then
```

or

```
if (@upper(%x) = "ABC") then
```

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. 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 1950q1 1970q1+!a
endif
```

only generates the series GNPSTD and CONSTD and sets the sample if the corresponding IF statements are true. Note that we have indented the lines between the if and the endif statements. The indentation is added for program clarity and has no effect on the execution of the program lines.

The expression to be tested may also take a numerical value. In this case, 0 and NA are equivalent to false and any other non-zero value evaluates to true. For example,

```
if !scale then
    series newage = age/!scale
endif
```

executes the series statement if !SCALE is a non-missing, non-zero value.

An IF statement may include a matching 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
```

(It is worth noting that this example performs a conditional recode in which the series NEWAGE is assigned using one expression if a condition is true, and a different expression otherwise. EViews provides a built-in `@recode` function for performing this type of evaluation; see `@recode(s,x,y)` (p. 506).)

IF statements may also be applied to string variables:

```
if %0="CA" or %0="IN" then
    series stateid = 1
else
    if %0="MA" then
       series stateid=2
    else
       if %0="IN" then
          series stateid=3
       endif
    endif
endif
```

Note that the nesting of our comparisons does not cause any difficulties.

You should note when using the IF statement with series or matrix objects that the comparison 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 does not equal the corresponding value of Y in the default sample. If X and Y are identically sized vectors or matrices, the comparison is over each of the elements X and Y. This element-wise comparison is described in greater detail in "Relational Operators (=, >, >=, <, <=, <>)" on page 253.

If you wish to operate on individual elements of a series on the basis of element-wise conditions, you should use the `@recode` function or use `smpl` statements to set the observations

you wish to change. Element-wise operations on a vector or matrix should use comparisons of individual element comparisons

```
if x(3)=7 then
    x(3) = 2
endif
```

or the element-wise matrix functions ("Matrix Element Functions" on page 607).

The FOR Loop

The FOR loop allows you to repeat a set of commands for different values of numerical or string variables. 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 numerical variables or string variables.

FOR Loops with Numerical Control Variables or Scalars

To repeat statements for different values of a control variable, you should follow the for keyword by a control variable initialization, the word `to`, and then an ending value. After the ending value you may include the word `step` followed by a number indicating an amount to change the control variable each time the loop is executed. If you don't include `step`, the step is assumed to be 1. Consider, for example the loop:

```
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.

We may add the `step` keyword and value to the FOR loop to modify the step:

```
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 in naming the ten constructed series RESCALE10 through RESCALE1 and as a control variable scalar divisor in the series ORIGINAL.

The commands inside the FOR loop are first executed for the initial control value (unless that value is already beyond the terminal value). After the initial execution, the control variable is incremented by `step` and EViews compares the new control variable value to the limit. If the limit is exceeded, execution stops.

One important use of FOR loops with control variables is to change the workfile sample using a `smpl` command. If you add a control variable to a date in a `smpl` command statement, 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 recursive regression:

```
for !horizon=10 to 72
    smpl 1970m1 1970m1+!horizon
    equation eq{!horizon}.ls sales c orders
next
```

One other important case uses loops and control variables to access elements of a matrix object. 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 extracts the series element associated with a given date.

You may 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. Consider, for example, the commands:

```
' 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 specific need to change a control variable inside the loop, you should consider using a WHILE loop ("The WHILE Loop" on page 133) as an alternative to the FOR loop.

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 scalar objects I and SUM remain in the workfile after the program has finished running, unless they are explicitly deleted. When using scalar objects as the looping variable you should be careful that the scalar is always available white the FOR loop is active. You should not, for example, delete the scalar or change the workfile page within the FOR loop.

FOR Loops with String Variables and String Objects

To repeat statements for different values of a string variable, you may use the FOR loop to let a string variable range over a list of string values. You should list the FOR keyword, followed by the name of the string program 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 may include multiple string variables in the same FOR statement—EViews will process the string values in sets. For example, we may define a loop with list three string variables:

```
for %1 %2 %3 1955q1 1960q4 early 1970q2 1980q3 mid 1975q4 1995q1
   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 1955q1 1960q4
equation earlyeq.ls sales c orders
smpl 1970q2 1980q3
equation mideq.ls sales c orders
smpl 1975q4 1995q1
equation lateeq.ls sales c orders
```

Both string objects and replacement variables may be used to specify a list for use in loops, by surrounding the object name or replacement variable with curly braces ("**{ }**"). For example,

```
string dates = "1960m1 1960m12"
%label = "year1"
for %1 %2 %3 {dates} {%label}
   smpl {%1} {%2}
   equation {%3}eq.ls sales c orders
next
```

finds the three strings for the loop by taking two elements of the string list and one element of the string variable:

```
smpl 1960m1 1960m12
equation year1eq.ls sales c orders
```

Note the difference between using a FOR loop with multiple string variables and using nested FOR loops. In the multiple string variable case, all string variables are advanced at the same time, while with nested loops, the inner variable is advanced over all choices, for each value of the outer variable. For example:

```
!eqno = 1
for %1 1955q1 1960q4
   for %2 1970q2 1980q3 1975q4
      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 1955Q1–1970Q2 for EQ1, 1955Q1–1980Q3 for EQ2, 1955Q1–1975Q4 for EQ3, 1960Q4–1970Q2 for EQ4, 1960Q4–1980Q3 for EQ5, and 1960Q4–1975Q4 for EQ6.

Note that you may use the `exitloop` command to exit a FOR loop early. See "Exiting Loops" on page 135.

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 or scalar object. 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 1950q1 1970q1+!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 include an explicit statement inside the loop to change 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.

Earlier, we cautioned against this behavior creating FOR loops that explicitly change the control variable inside the loop and offered an example to show the resulting lack of clarity (p. 130). Note that the equivalent WHILE loop provides a much clearer program:

```
!i = 1
while !i<=25
    vector a{!i}
    !i = !i + 11
wend
```

Note that you may use the `exitloop` command to exit a WHILE loop early. See "Exiting Loops" on page 135.

Execution Errors

By default, EViews will stop executing after encountering any errors. You can instruct the program to continue running even if errors are encountered by changing the maximum error count from the **Run** dialog (see "Executing a Program" on page 108), or by using the `setmaxerrs` (p. 666) command inside a program.

Handling Errors

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, or you may wish to overwrite the default EViews error with one of your own, using the `seterr` (p. 665) command.

EViews offers a number of different ways to test for and handle execution errors. For example, the `@lasterrstr` (p. 662) command will return a string containing the previous line's error. If the previous line of your program did not error, this string will be empty. Alternatively you could use the `@errorcount` (p. 657) function to check the number of errors currently encountered before and after a program line has been executed.

For example, to test whether the estimation of an equation generated an error, you can 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.

For additional error handling functions, see "Support Commands" on page 649 and "Support Functions" on page 650.

Stopping Programs

Occasionally, you may want to stop a program 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 elements of TEST will also be missing. But 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.

Exiting Loops

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.

As a simple 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.

If the `exitloop` is issued inside nested loops it will stop execution of the innermost loop. Execution of the remaining loops is unaffected.

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 a run-referenced file. Thus, a program containing the lines

```
run powers.prg
series x = nrnd
```

will only execute the commands in the file "Powers.PRG", and will stop before generating the series X. This behavior is probably not what you intended.

The `exec` command may be used execute commands in a file in place of the `run` command. Though `exec` is quite similar to the `run` command, there are important differences between the two commands:

- First, `exec` allows you to write general programs that execute other programs, something that is difficult to do using `run`, since the `run` command ends all program execution when processing of the named file is completed. In contrast, once `exec` processing completes, the calling program will continue to run.
- Second, the default directory for `exec` is the Add-ins directory (in contrast with both `run` and `include` which defaults to using the EViews default directory). Thus, the command

  ```
  exec myprog1.prg
  ```

  will run the program file "Myprog1.prg" located in the default Add-ins directory. You may specify files using relative paths in the standard fashion. The command:

  ```
  exec MyAddIn\myprog2.prg
  ```

  runs the program "Myprog2.prg" located in the "MyAddIn" subdirectory of the Add-ins directory.

If you wish to run a program that is located in the same directory as the calling program, simply issue a ".\" at the start of the program name:

```
exec .\myprog2.prg
```

Alternatively you may 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", will execute the commands in "Durbin_h.PRG", and then will execute the remaining lines in the program file.

If you do not provide an absolute path for your include file, EViews will use the location of the executing program file as the base location. In the example above, EViews will search for the "Durbin_h.PRG" file in the directory containing the executing program.

Note that in contrast to using `exec` to execute another program, `include` simply inserts the child program into the parent. This insertion is done prior to executing any lines in either program. One important consequence of this behavior is that any program variables that are declared in the parent program will not be available in the child/included program, even if they are declared prior to the `include` statement.

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 may also use subroutines from one program to perform the same task in other programs.

Defining Subroutines

A subroutine begins 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 may 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 (Greene, 2008, p. 646, or Davidson and MacKinnon, 1993, p. 360) for testing serial correlation with a lagged dependent variable cannot be computed:

```
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 we have seen thus far have been written to work with a specific set of variables. More generally, subroutines can take arguments. Arguments allow you to change the behavior of the group of commands each time the subroutine is used. You may be familiar with the concept from other programming languages, but if now, you are probably familiar with similar concepts in mathematics. You can define a function, say

$$f(x) = x^2 \tag{6.1}$$

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 the full function expression every time you wish to evaluate it for a different value.

To define a subroutine with arguments, you start with the `subroutine` keyword, followed by the subroutine name and (with no space) the arguments separated by commas, enclosed in parentheses. Each argument is specified by listing a type of EViews object, followed by the name of the argument. For example:

```
subroutine power(series v, series y, scalar p)
    v = y^p
endsub
```

This subroutine generalizes the example subroutine Z_SQUARE. Calling the subroutine 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 equivalent of the subroutine Z_SQUARE above. See the discussion below on how to call subroutines.

When creating subroutines with scalar or string arguments, you will define your arguments using the `scalar` or the `string` types. Beyond that, you have a choice of whether you can to make the corresponding argument a (temporary) program variable or a (permanent) workfile object:

- To make the argument a program variable, you should use a program variable name (beginning with a "!" for a control variable and a "%" for a string variable). If you choose to use program variables, they should be referred to using the "!" or "%" name inside the subroutine.
- To make the argument a workfile object, you should use a standard EViews object name. The object should be referred to by the argument name inside the subroutine.

Obviously, you can mix the two approaches in the definition of any subroutine.

For example, the declaration

```
subroutine mysub(scalar !a, string %b)
```

uses program variable names, while

```
subroutine mysub(scalar a, string b)
```

uses object names. In the first case you should refer to "!A" and "%B" inside the subroutine; in the latter case, you should refer to the objects named "A" and "B".

If you define your subroutine using program variables, the subroutine will operate on them as though they were any other program variable. The variables, which cannot go out-of-scope, should be referred to using the "!" or "%" argument name inside the subroutine.

If you define your subroutine using object names, the subroutine will operate on those variables as though they were scalar or string objects. The variables, which may be deleted and may go out-of-scope (if, for example, you change the workfile page), should be referred to using the argument names as though they were scalar or string objects.

(We discuss in detail related issues in "Calling Subroutines," beginning on page 139.)

You should note that EViews makes no distinction between input or output arguments in a subroutine. Each argument may be an input to the subroutine, an output from the subroutine, or both (or neither!). There is no way to declare that some arguments are inputs and some are outputs. There is no need to order the arguments in any particular order. However we find it much easier to read subroutine code when we stick to a convention, such as listing all output arguments prior to all input arguments (or vice versa).

Calling Subroutines

Once a subroutine is defined, 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 (with

no space after the subroutine name). If the subroutine takes arguments, the arguments must 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 call of the Z_SQUARE subroutine fills the series X with the value of Z squared. Next, the call to the POWER subroutine creates the series GDP2 which is GDP squared. The last call to POWER creates the series GDP3 as the cube of GDP.

When calling your subroutine, bear in mind that:

- 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 with a string argument may be called with a string object, a string program variable, simple text (such as "hello") or an element of an svector object.
- 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 an object name referring to a single object of the correct type.

In "Subroutine with Arguments" on page 138 we described how you can *define* subroutines that use either program variables or objects for scalar or string arguments. However you define your subroutine, you may call the subroutine using either program variables or objects—you are not required to match the calling arguments with the subroutine definition. Suppose, for example, that you define your subroutine as

```
subroutine mysub(scalar a, string b)
```

Then for scalar and string objects F and G, and program variables !X and %Y,

```
scalar f = 3
string g = "hello"
!x = 2
%y = "goodbye"
```

you may call the subroutine using any of the following commands:

```
call mysub(!x, %y)
call mysub(!x, g)
call mysub(f, %y)
call mysub(f, g)
```

Note that changes to the scalars A and B inside the subroutine will change the corresponding program variable or object that you pass into the routine.

Similarly, you may define

```
subroutine mysub(scalar !a, string !b)
```

and use the same four `call` statements to execute the subroutine commands.

However the subroutine is called, bear in mind that behavior inside the subroutine is dependent on whether the subroutine declaration is in terms of program variables or objects, not on the variable type that is passed into the subroutine.

Subroutine Placement

Subroutine definitions may be placed anywhere throughout your program. However, for clarity, we recommend grouping subroutine definitions together either at the start or at the end 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."

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 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.

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, and to 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 is defined within the subroutine. Local variables are deleted from the workfile once a subroutine finishes. The program that calls the subroutine will not be able to use a local variable since the local variable disappears once the subroutine finishes and the original program continues.

Global Subroutines

By default, subroutines in EViews are *global*. 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:

- All objects created by a global subroutine are global and will remain 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 so that 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 sername)
    series {sername} = 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 series passed-in via the argument Z instead of the global Z series. On completion of the routine, the new series Y2 will contain the square of the series Y, not the square of the series Z. Since keeping track of variables can become confusing when subroutine arguments take the same name as existing workfile objects, we encourage you to name subroutine arguments as clearly and distinctly as possible.

Global subroutines may call global subroutines. You should make certain to pass along 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 must explicitly pass along arguments to the subroutine OLS so that it uses the correct series and equation objects.

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 matrix X generates an error since X exists and is a series, but the declaration of the matrix Y works (assuming there is no Y in the workfile MYWORK, or that Y exists and is already a matrix):

```
subroutine newxy
   matrix(2,2) x = 0
   matrix(2,2) y = 0
endsub
load mywork
series x
call newxy
```

If you call this subroutine, EViews will return an error indicating that the global series X already exists and is of a different type than a matrix.

Local Subroutines

If you include the word `local` in the definition of the subroutine, you create a local 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:

- All objects created by a local subroutine will be local and will be removed from the workfile upon exit from the subroutine.
- The global objects corresponding to subroutine arguments may be used and updated in the subroutine by referring to the arguments.
- You may not use or update global objects that do not correspond to subroutine arguments.

There is one exception to the general inaccessibility of non-argument global variables in local subroutines. When a global group is passed as an argument to a local subroutine, all of the series in the group are accessible to the local routine.

The last two rules deserve a bit of elaboration. In general, local subroutines do not have access to any global variables unless those variables are associated with arguments passed into the subroutine. Thus, if there is a series X in the workfile, a local subroutine will not be

allowed to use or modify X unless it is passed into the subroutine using a series argument. Conversely, if X is passed into the subroutine, it may be modified.

Since locally created objects will vanish upon completion of the subroutine, to save results from a local subroutine, you have to include them as arguments. For example, consider the subroutine:

```
subroutine local ols_local(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
    scalar se = ssr/temp_eq.@df
endsub
```

This local subroutine takes the series Y as input and modifies the argument series RES and argument scalar SSR as output. Note that since Y, RES, and SSR are the only arguments of this local subroutine, the only global variables that may be used or modified are those associated with these arguments.The equation object TEMP_EQ and the scalar SE are 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_local(hsf, ures, ussr)
```

Note that 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. Since we call the local subroutine by passing these global objects as arguments, the subroutine can use and update these global variables.

Object commands that require access to global variables should be used with care in local subroutines since that the lack of access to global variables can create problems for views or procs of objects passed into the routine. For example, a subroutine of the form:

```
subroutine local bg(equation eq)
    eq.hettest z c
endsub
```

will fail because the `hettest` equation proc requires access to the original variables in the equation and the global variable Z, and these series are not available since they are not passed in as arguments to the subroutine.

Care should also be taken when using samples and local subroutines. If the workfile sample is based upon a series in the workfile (for example "smpl @all if x > 0"), most procedures inside the local subroutine will fail unless all of the series used in the sample are passed into the subroutine.

Local Samples

Local samples in subroutines allow you to indicate that changes to the workfile sample are temporary, with the original sample restored when you exit the routine. This feature is useful when designing subroutines which require working on a subset of observations in the original sample.

You may, in a subroutine, use the `local smpl` statement to indicate that subsequent changes to the sample are temporary, and should be undone when exiting the subroutine. The command

```
local smpl
```

makes a copy of the existing sample specification. You may then change the sample as many times as desired using the `smpl` statement, and the original sample specification will be reapplied when existing from the subroutine.

You may use the `global smpl` statement to indicate that subsequent `smpl` statements will result in permanent changes to the workfile sample. Thus, the commands:

```
global smpl
smpl 5 100
```

in a subroutine permanently change the sample.

For example, consider the following program snippet which illustrates the behavior of local and global samples:

```
workfile temp u 100
call foo
subroutine foo
    smpl 2 100
    local smpl
    smpl 10 100
endsub
```

Here, we create a workfile with 100 observations and an initial workfile sample of "1 100", then call the subroutine FOO. Inside FOO, the first `smpl` statement changes the workfile

sample to “2 100”. We then issue the `local smpl` statement which backs up the existing sample and identifies subsequent sample changes as local. The subsequent change to the “10 100” sample is local so that when the subroutine exits, the sample is reset to “2 100”.

If instead we define FOO to be

```
subroutine foo
    smpl 2 100
    local smpl
    smpl 10 100
    global smpl
    smpl 5 100
    local smpl
    smpl 50 100
endsub
```

As before, first `smpl` statement changes the workfile sample to “2 100” and the `local smpl` statement and following `smpl` statement set the local sample to “10 100”. The `global smpl` indicates that subsequent sample statements will once again be global so the next line permanently changes the workfile sample to “5 100”. Note that the last `local smpl` and subsequent `smpl` statement change the local sample only. When we exit the subroutine the sample will be set to the last global sample of “5 100”.

User-Defined Dialogs

EViews offers the ability to construct several types of user-interface controls, or dialogs, within your program. These dialogs permit users to input variables or set options during the running of the program, and allow you to pass information back to users.

There are five different functions that create dialogs in EViews:

- `@uiprompt` (p. 676) – creates a prompt control, which displays a message to the user.
- `@uiedit` (p. 674) – creates an edit control, which lets users input text.
- `@uilist` (p. 675) – creates a list control, which lets users select from a list of choices.
- `@uiradio` (p. 677) – creates a set of radio controls, which lets users select from a set of choices.
- `@uidialog` (p. 671) – creates a dialog which contains a mixture of other controls.

Each dialog function returns an integer indicating how the user exited the dialog:

User Selection	Return Value
Cancel	-1

OK	0
Yes	1
No	2

Note that the only dialog types that can return exit conditions other than "Cancel" or "OK" are `@uiprompt` and `@uidialog`. If "Cancel" is pressed, the variables passed into the dialog will not be updated with whatever settings the user has chosen. If "OK" is pressed, then the dialog changes are accepted.

Each of the dialog functions accept arguments that are used to define what will be displayed by the dialog, and that will be used to store the user's inputs to the dialog. You may use string or a scalar arguments, where both the string and scalar can be a literal, a program variable, or a workfile object.

@uiprompt

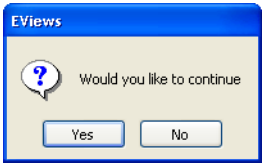
The `@uiprompt`(**string** *prompt*, **string** *type*) function creates a simple message/prompt box that displays a single piece of text, specified using the *prompt* argument, and lets the user click a button to indicate an action. The choices of action offered the user will depend upon the string specified in *type*.

- if *type* is equal to "O", or is omitted completely, then the dialog will only have an "OK" button. This type of prompt dialog would typically be used to provide a message to the user.
- if *type* is equal to "OC", then the dialog will have an "OK" button and a "Cancel" button. This type of prompt dialog would be used to pass a message onto the user and let them continue or cancel from the procedure.
- if *type* is equal to "YN", then the dialog will contain a "Yes" button and a "No" button which can be used to ask the user a question requiring an affirmative or negative response.
- if *type* is equal to "YNC" the dialog will have a "Yes" button, a "No" button and a "Cancel" button.

For example, the command:

```
@uiprompt("Welcome to EViews")
```

will display a simple dialog box with a simple welcome message and an "OK" button, while the command




```
@uiprompt("Would you like to continue",
    "YN")
```

displays a dialog asking the user if they would like to continue the program, with a "Yes" or "No" answer.

Note that the arguments to the function may be a program variable or string object rather than string literals. The following sets of commands give identical results to that above:

```
%type = "YN"
%msg = "Would you like to continue"
scalar ret = @uiprompt(%msg, %type)
```

The return value of the control is determined by the user response: Cancel (-1), OK (0), Yes (1), No (2).

See `@uiprompt` (p. 676) for additional detail.

@uiedit

The `@uiedit`(**string** *IOString*, **string** *prompt*, **scalar** *maxEditLen*) function provides an edit control that lets the user enter text which is then stored in the string *IOString*. If *IOString* contains text before the `@uiedit` function is called, the edit box will be initialized with that text.

The string *prompt* is used to specify text to be displayed above the edit box, which may be used to provide a message or instructions to the user.

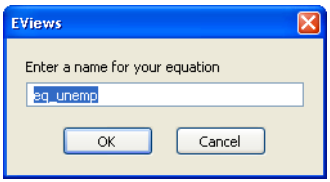
maxEditLen is an option integer scalar that specifies the maximum length that *IOString* can take. If the user tries to enter text longer than *maxEditLen* characters, the extra characters will be truncated. By default *maxEditLen* is set to 32.

Both *prompt* and *maxEditLen* may be written in-line using string literals, but *IOString* must be either a program variable or a string object in your workfile.

As an example,

```
%eqname = "eq_unemp"
scalar ret = @uiedit(%eqname, "Enter a name for your equation")
equation {%eqname} unemp c gdp gdp(-1)
```

will display the following dialog box and then create an equation object with a name equal to whatever was entered for %EQNAME.



The return value of the control is determined by the user response: Cancel (-1) or OK (0).

See `@uiedit` (p. 674) for additional detail.

@uilist

This function creates a list box dialog, which lets the user select one item from a list. There are two forms of the `@uilist` function, one that returns the user's selection as a string *IOString*,

`@uilist`(**string** *IOString*, **string** *prompt*, **string** *list*)

and one that stores it as an integer *IOScalar* representing the position of the selection in the list,

`@uilist`(**scalar** *IOScalar*, **string** *prompt*, **string** *list*)

The string *prompt* is used to specify text to be displayed above the list box, providing a message or instructions to the user.

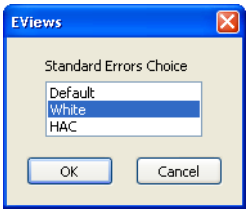
The string *list* is a space delimited list of items that will be displayed in the list box. To specify entries with spaces, you should enclose the entry in double-quotes using double-quote escape characters.

Both *prompt* and *list* may be provided using in-line text, but *IOString* or *IOScalar* must be either a program variable or an object in your workfile.

If the IO variable (*IOString* or *IOScalar*) is defined before the function is called, then the list box control will have the item defined by the variable pre-selected. If the IO variable does not match an item in the list box, or if it is not pre-defined, the first item in the list will be selected.

The following program lines provide the user with a choice of robust standard errors, and then displays that choice on the statusline:

```
%choice = "White"
%list = "Default White HAC"
scalar ret = @uilist(%choice, "Standard Errors Choice", %list)
statusline %list
```



Note that the above program could also be run with the following lines:

```
!choice = 2
%list = "Default White HAC"
scalar ret = @uilist(!choice, "Standard Errors Choice", %list)
```

```
%choice = @word(%list,!choice)
statusline %choice
```

The return value of the control is determined by the user response: Cancel (-1) or OK (0).

See `@uilist` (p. 675) for detail.

@uiradio

`@uiradio`(**scalar** *IOScalar*, **string** *prompt*, **string** *list*) is similar to `@uilist` in that it provides a dialog that lets the user select from a list of choices. However rather than selecting an item from a list box, the user must select the desired item using radio buttons. The `@uiradio` function will return the user's choice in *IOScalar*.

The string *prompt* should be used to specify a message or instructions to the user to be displayed above the set of radio buttons.

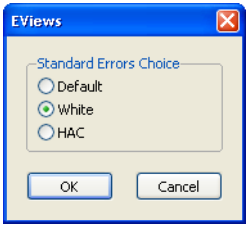
The string *list* is a space delimited list of items that contains the items for the radio buttons. To specify entries with spaces, you should enclose the entry in double-quotes using double-quote escape characters.

Both *prompt* and *list* may be specified using in-line text, but *IOScalar* must be either a program variable or an object in your workfile.

If *IOScalar* is defined and set equal to a valid integer before the function is called, the radio button associated with that integer will be the default selection when the dialog is shown. If *IOScalar* is not a valid choice, the first radio will be selected.

As an example, we replicate the standard error choice example from the `@uilist` function, but this time use radio buttons:

```
!choice = 2
%list = "Default White HAC"
scalar ret = @uiradio(!choice, "Standard Errors Choice", %list)
%choice = @word(%list,!choice)
statusline %choice
```



The return value of the control is determined by the user response: Cancel (-1) or OK (0).

See @uiradio (p. 677) for additional detail.

@uidialog

The @uidialog(*control_spec1[*, *control_spec2*, ….*]*) function displays a dialog which may be composed of different controls, including simple text, edit boxes, list boxes, radio buttons and check boxes. The dialog is specified using a list of control specifications passed into the function as arguments. Each control specification is a type keyword specifying the type of control, followed by a list of parameters for that control.

The type keywords should be from the following list:

Keyword	Control
“caption”	Dialog title
“text”	Plain text
“edit”	Edit box
“list”	List box
“radio”	Radio buttons
“check”	Check box
“button”	OK-type button
“buttonc”	Cancel-type button
“colbreak”	Column break
“setoc”	Set OK/Cancel text

The “edit”, “list” and “radio” controls are similar to their individual dialog functions, and the specifications for those controls follow the same pattern. Thus the specification for an edit box would be (“edit”, string *IOString*, string *prompt*, scalar *maxEditLen*).

The “caption” control changes the title of the dialog, shown in the title bar. The caption keyword should be followed by a string containing the text to be used as the caption, yielding a specification of (“caption”, string *caption*).

The “text” control adds basic text to the dialog. Like the caption control, the text control keyword, “text”, should be followed by a string containing the text to be used, yielding a specification of (“text”, string *text*).

The “check box” control adds a check box to the dialog. The check keyword should be followed by a scalar, *IOScalar*, which stores the selection of the check box - 1 for checked, and 0 for unchecked, and then by a string prompt which contains the text to be used as a prompt/instructions for the check box. The specification for the check box control is then: (“check”, scalar *IOScalar*, string *prompt*).

The “button” and “buttonc” controls add a custom button to the dialog. The dialog will close after a button has been pressed. The behavior of the button will depend on the type of button —buttons of type “button” will behave in the same way as the “OK” button (*i.e.*, all variables passed into the dialog will be updated to reflect changes made to their corresponding controls). Buttons of type “buttonc” will behave in the same way as the “Cancel” button (*i.e.,* all variables will be reset to the values that were passed into the dialog).

The return value of the dialog will correspond to the order in which buttons are placed in the dialog. If only one button (apart from the standard “OK” and “Cancel”) is included in the dialog, the return value for that button will be “1”. If there is more than one button, then the first button will return a value of “1”, the second will return a value of “2” and so on. Note that the return value is independent of whether the button was of type “button” or “buttonc”. The specification for the button controls is (“button[c]”, “*text*”) where *text* specifies the text that will be on the button.

The column break control inserts a column break. By default, EViews will automatically choose the number of columns in the constructed dialog. There is still a maximum of only two columns allowed, but by adding a “colbreak” control, you can force the position of a break in the dialog.

“setoc” allows you to change the text of the “OK” and “Cancel” buttons on the dialog. You should supply two words, separated by a space as the text for “OK” and “Cancel”.

As an example, a dialog that offers a choice of covariance matrix options, plus a check box for degree of freedom adjustment, could be made with:

```
!choice = 2
!doDF = 1
%list = "Default White HAC"
scalar ret = @uidialog("caption", "Covariance Options", "list",
    !choice, "Covariance method", %list, "check", !doDF, "DF-
    adjust")
```

See `@uidialog` (p. 671) for details.

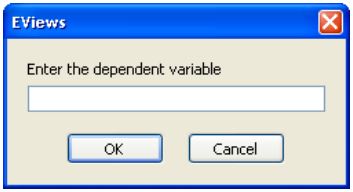
Example Program

This program creates a workfile and some series, then puts up dialogs asking the user to specify the dependent variable and the regressors to be used in a least-squares equation. Note that the part of the example program that generates the data could be replaced with a command to open up an existing workfile.

```
'Create a workfile and some series
wfcreate temp u 100
series y=nrnd
series x=nrnd
series w=nrnd
'-------------------------------------
'variable to store name of dependent variable
%dep = ""
'variable to store list of regressors. Default is "c" for a con-
   stant.
%regs = "c "
'variable that will track the result of dialogs. -1 indicates user
   hit Cancel. 0 Indicates user hit OK.
!result  = 0
'put up an Edit dialog asking for dependent variable.
!result = @uiedit(%dep,"Enter the dependent variable")
if !result=-1 then    'if user cancelled, then stop program
stop
endif
'put up an Edit dialog asking for regressors list.
!result = @uiedit(%regs,"Enter the list of regressors")
if !result=-1 then   'if user cancelled, then stop program
stop
endif

equation eq1.ls {%dep} {%regs}   'estimate equation.
```

Three program variables are used in this program: %DEP, %REGS and !RESULT. %DEP is the string variable that will contain the user's entry for the dependent variable. To begin, we set this equal to an empty string. %REGS is used to store the user's entry for the list of regressors. We set this equal to "C" to begin with. This means that the default setting for the regressor list will be a constant. !RESULT will be used to track the completion of the dialogs. Every dialog returns an integer value depending on whether the user clicked **OK** or **Cancel** (or in some cases **Yes** or **No**). We initialize this value to 0.



Following the declaration of the variables, the first dialog is brought up using the `@uiedit` command. This will create an Edit dialog asking the user to "Enter the dependent variable." The user's response will be stored in %DEP. Following the dialog command, we check whether the value of !RESULT is equal to -1. A -1 indicates that the user pressed **Cancel** in the dialog. If this is the case, the program quits, using the `stop` command.

The second dialog command is similar to the first, but rather than asking for the dependent variable, it asks the user to "Enter the list of regressors," and stores that list in %REGS. Note that since %REGS was set equal to "C" prior to the dialog being put up, the dialog will be pre-filled with the constant term. Users can still delete the constant or add extra regressors.

Finally, having obtained entries for both %DEP and %REGS, equation EQ1 is estimated via least squares with the specified variables.

Version 4 Compatibility Notes

While the underlying concepts behind strings, string variables, and replacement variables have not changed since the first version of EViews, there were three important changes in the implementation of these concepts introduced in EViews 5. In addition, there has been an important change in the handling of boolean comparisons involving numeric NA values, and blank string values.

String vs. Replacement Variables

First, the use of contextual information to distinguish between the use of string and replacement variables has been eliminated.

Prior to EViews 5, the underlying notion that the expression "%X" refers exclusively to the string variable %X while the expression "{%X}" refers to the corresponding replacement variable was modified slightly to account for the context in which the expression was used. In particular, the string variable expression "%X" was treated as a string variable in cases where a string was expected, *but was treated as a replacement variable* in other settings.

For example, suppose that we have the string variables:

```
%y = "cons"
%x = "income"
```

When used in settings where a string is expected, all versions of EViews treat %X and %Y as string variables. Thus, in table assignment, the command,

```
table1(2, 3) = %x + " " + %y
```

is equivalent to the expression,

```
table1(2, 3) = "cons" + " " + "income"
```

However, when string variables were used in other settings, early versions of EViews used the context to determine that the string variable should be treated as a replacement variable; for example, the three commands

```
equation eq1.ls %y c %x
equation eq1.ls {%y} c {%x}
equation eq1.ls cons c income
```

were all viewed as equivalent. Strictly speaking, the first command should have generated an error since string variable substitution would replace %Y with the double-quote delimited string "cons" and %X with the string "income", as in

```
equation eq1.ls "cons" c "income"
```

Instead, EViews determined that the only valid interpretation of %Y and %X in the first command was as replacement variables so EViews simply substituted the names for %Y and %X.

Similarly, the commands,

```
series %y = %x
series {%y} = {%x}
series cons = income
```

all yielded the same result, since %Y and %X were treated as replacement variables in the first line, not as string variables.

The contextual interpretation of string variables was convenient since, as seen from the examples above, it meant that users rarely needed to use braces around string variables. The EViews 5 introduction of alphanumeric series meant, however, that the existing interpretation of string variables was no longer tenable. The following example clearly shows the problem:

```
alpha parent = "mother"
%x = "parent"
alpha temp = %x
```

Note that in the final assignment statement, the command context alone is not sufficient to determine whether %X should refer to the string variable value "parent" or to the replacement variable PARENT, which is an Alpha series containing the string "mother".

Consequently, in EViews 5 and later, users must now always use the expression "{%X}" to refer to the replacement variable corresponding to the value of %X. Thus, under the new interpretation, the final line in the example above resolves to

```
alpha temp = "parent"
```

Under the EViews 4 interpretation of the final line, "%X" would have been treated as a replacement variable so that TEMP would contain the value "mother".

To interpret the last line as a replacement variable in EViews 5 and later, you must now explicitly provide braces around the string variable

```
alpha temp = {%x}
```

to resolve to the command

```
alpha temp = parent
```

String Variables in String Expressions

The second major change in EViews 5 is that text in a string expression is now treated as a literal string. The important implication of this rule is that string variable text is no longer substituted for inside a string expression.

Consider the assignment statements

```
%b = "mom!"
%a = "hi %b"
table(1, 2) = %a
```

In EViews 4 and earlier, the "%B" text in the string expression was treated as a string variable, not as literal text. Accordingly, the EViews 4 string variable %A contains the text "hi mom!". One consequence of this approach is that there was no way to get the literal text of the form "%B" into a string using a program in EViews 4.

Beginning in EViews 5, the "%B" in the second string variable assignment is treated as literal text. The string variable %A will contain the text "hi %b". Obtaining a %A that contains the EViews 4 result is straightforward. Simply move the first string variable %B outside of the string expression, and use the string concatenation operator:

```
%a = "hi " + %b
```

assigns the text "hi mom!" to the string variable %A. This expression yields identical results in all versions of EViews.

Case-Sensitive String Comparison

In early versions of EViews, program statements could involve string comparisons. For example, you might use an if-statement to compare a string variable to a specific value, or you could use a string comparison to assign a boolean value to a cell in a matrix or to a numeric series. In all of these settings, the string comparisons were performed caselessly, so that the string "Abc" was viewed as equal to "ABC" and "abc".

The introduction of mixed case alpha series in EViews 5 meant that caseless string comparisons could no longer be supported. Accordingly, the behavior has been changed so that all EViews 5 and later string comparisons are case-sensitive.

If you wish to perform caseless comparison in newer versions of EViews, you can use the `@upper` or `@lower` string functions, as in

```
if (@lower(%x) = "abc") then
```

or

```
if (@upper(%x) = "ABC") then
```

Alternately, programs may be run in version 4 compatibility mode to enable caseless comparisons for element operations (see "Version 4 Compatibility Mode" on page 159). For example, the if-statement comparison:

```
if (%x = "abc") then
```

will be performed caselessly in compatibility mode.

Note that compatibility mode does not apply to string comparisons that assign values into an entire EViews series. Thus, even in compatibility mode, the statement:

```
series y = (alphaser = "abc")
```

will be evaluated using case-sensitive comparisons for each value of ALPHASER.

Comparisons Involving NAs/Missing Values

Prior to EViews 5, NA values were always treated as ordinary values for purposes of numeric equality (" = ") and inequality (" < > ") testing. In addition, when performing string comparisons in earlier versions of EViews, empty strings were treated as ordinary blank strings and not as a missing value. In these versions of EViews, the comparison operators (" = " and " < > ") always returned a 0 or a 1.

In EViews 5 and later, the behavior of numeric and string inequality comparisons involving NA values or blank strings has been changed so that comparisons involving two variables propagate missing values. To support the earlier behavior, the `@eqna` and `@neqna` functions are provided so that users may perform comparisons without propagating missing values. Complete details on these rules are provided in "String Relational Operators" on page 67 of the *Command and Programming Reference* and in "Numeric Relational Operators" on page 168 of *User's Guide I*.

Programs may be run in version 4 compatibility mode to enable the earlier behavior of comparisons for element operations. For example, the if-statement comparison:

```
if (!x = !z) then
```

will not propagate NA values in compatibility mode.

Note that compatibility mode does not apply to comparisons that assign values into an EViews numeric or alpha series. Thus, even in compatibility mode, the statement:

```
series y = (x = z)
```

will propagate NA values from either X or Z into Y.

Version 4 Compatibility Mode

While the changes to the handling of string variables and element boolean comparisons are required for extending the programming language to handle the new features of the EViews 5 and later, we recognize that users may have a large library of existing programs which make use of the previous behavior.

Accordingly, EViews provides a version 4 compatibility mode in which you may run EViews programs using the previous context sensitive handling of string and substitution variables, the earlier rules for resolving string variables in string expressions, and the rules for caseless string comparison and propagation of missing values in element comparisons.

There are two ways to ensure that your program is run in version 4 compatibility mode. First, you may specify version 4 compatibility mode at the time the program is run. Compatibility may be set interactively from the **Run Program** dialog ("Executing a Program" on page 108) by selecting the **Version 4 compatible variable substitution and program boolean comparisons** checkbox, or in a program using the "ver4" option (see `run` (p. 427)).

Alternatively, you may include "MODE VER4" statement in your program. See "Program Modes" on page 122 for details.

References

Davidson, Russell and James G. MacKinnon (1993). *Estimation and Inference in Econometrics*, Oxford: Oxford University Press.

Greene, William H. (2008). *Econometric Analysis*, 6th Edition, Upper Saddle River, NJ: Prentice-Hall.

Chapter 7. External Interfaces

EViews offers several methods for interacting with external applications:

- The EViews OLEDB driver provides an easy-to-use interface for external programs to read data stored in EViews workfiles (WF1) and EViews databases (EDB).
- The EViews Excel Add-in offers a simple interface for reading data stored in EViews workfiles and databases from within Microsoft Excel.
- EViews offers COM Automation server support so that external programs or scripts can launch or control EViews, transfer data, and execute EViews commands.
- EViews offers COM Automation client support for MATLAB and R application servers so that EViews may be used to launch and control the application, transfer data, and execute commands.

Reading EViews Data

The EViews OLEDB driver provides an easy way for OLEDB-aware clients or custom programs to read data stored in EViews workfiles (WF1) and EViews databases (EDB).

We also provide an EViews Microsoft Excel Add-in that allows users to fetch and link to EViews data located in workfiles and databases. The Add-in offers an easy-to-use interface to OLEDB for reading EViews data from within Excel.

The OLEDB Driver

The EViews OLEDB driver is automatically installed and registered on your computer when you install EViews. Once installed, you may use OLEDB-aware clients or custom programs to read series, vector, and matrix objects directly from EViews workfiles and databases.

See "The OLEDB Driver" on page 149 of *User's Guide I* for discussion. For additional details, see the *Using the EViews OLEDB Driver* whitepaper.

The Excel Add-in

The EViews Excel Add-in offers a simple interface for fetching and linking to data stored in EViews workfiles and databases from within Microsoft Excel (2000 and later).

See "The Excel Add-in" on page 145 of *User's Guide I* for discussion. For additional details, see the *Using the EViews OLEDB Driver* whitepaper.

EViews COM Automation Server

EViews may be used as a COM Automation server so that an external program or script may launch and control EViews programmatically. EViews COM is comprised of two class objects: Manager and Application.

The Manager class is used to manage and create instances of the main EViews Application class. The Application class provides access to EViews functionality and data. Most notably, the Application class **Run** and a variety of **Get** and **Put** methods provide you with access to EViews commands and allow you to obtain read or write access to series, vectors, matrix, and scalar objects.

For a complete description of these methods, please refer to the *EViews COM Automation Server* whitepaper, which is available in your EViews documentation directory or from our website www.eviews.com.

Note that web server access to EViews via COM is not allowed. Furthermore, EViews will limit COM access to a single instance when run by other Windows services or run remotely via Distributed COM.

EViews COM Automation Client Support (MATLAB and R)

EViews offers COM Automation client support for select external application servers. Support is currently provided for two applications: MATLAB and R.

The client support includes a set of EViews functions for exporting an EViews data object into the external application, running commands and programs in the application, and importing data back into EViews. These functions provide easy access to the powerful programming languages of MATLAB and R to create programs and routines that perform tasks not currently implemented in EViews. The interface also offers access to the large library of statistical routines already written in the MATLAB and R languages.

There are six EViews commands that control the use of external applications: `xclose` (p. 493), `xget` (p. 493), `xlog` (p. 496), `xopen` (p. 496), `xput` (p. 498), and `xrun` (p. 500).

`xopen` and `xclose` are used to open and close a connection to the external application (MATLAB or R). `xput` and `xget` are used to send data to and from the external application. `xrun` is used to send a command to the external application, and, finally, `xlog` lets you show or hide an external application log window within EViews.

Using MATLAB®

To open a connection to MATLAB, simply use the `xopen(type=m)` command. EViews will then attempt to launch MATLAB on your computer. Note, your computer must have access

to MATLAB, either through a local installation, or through a network. EViews has been tested with MATLAB release R2008a, although other versions may work as well.

Once a connection to MATLAB has been made, `xput` (p. 498) may be used to pass data from EViews over to MATLAB. All numerical data is passed to MATLAB as a matrix. String data (in the form of an alpha series or svector) will be passed to a MATLAB char if each string contains the same number of characters. Otherwise the string data will be passed as a cell array. Series and group objects are always filtered by the current sample unless you specify an explicit sample in the `xput` command.

Note that object names in EViews are not case sensitive. Unless otherwise specified, EViews objects that are passed into MATLAB using `xput` will have the same name as the EViews objects that are being pushed, with the case determined by the case established for the COM connection (see `xopen` (p. 496)).

`xrun` can be used to issue a single line command to MATLAB. You may, for example, use `xrun` to invert a matrix or run a program in MATLAB. If you wish to run multiple commands, each command must be entered with a separate `xrun` command. Commands should be surrounded in quotes.

`xget` can be used to fetch data from MATLAB into EViews. The "type = " option lets you specify the type of object that will be created in EViews. If no option is specified, MATLAB matrices will be brought in as EViews matrix objects, while chars and cell arrays will be brought in as svectors. If you use "type = series" or "type = alpha" to specify that the data is brought in as a series or an alpha series, EViews will create a series of workfile length, and either truncate the data, or pad with NAs, if the incoming matrix does not have the same number of rows as there are workfile observations.

The follow program offers a simple example using MATLAB to perform a least-squares regression (more complicated operations may be performed by using `xrun` to run a MATLAB program):

```
'create a workfile
wfcreate u 100
'create some data
series y=nrnd
series x1=nrnd
series x2=nrnd
'group regressor data into a group
group xs c x1 x2
'open a connection to Matlab with lower-case default output names
xopen(type=m, case=lower)
'put regressors and dependent variable into Matlab
```

```
xput xs
xput y
'run a command to perform least squares as a matrix operation
xrun "beta = inv(xs'*xs)*xs'*y"
'retrieve betas back into EViews
xget beta
'perform same least squares estimation in EViews
equation e1.ls y xs
show e1
show beta
'close Matlab connection
xclose
```

The program first creates a new workfile, and then creates some series objects. The series called Y is the dependent variable, and the series X1 and X2 are the regressors (along with a constant). `xopen` is used to open a connection to MATLAB, and then `xput` is used to pass the dependent variable and the regressors over to MATLAB. Note that the names of the matrices and vectors will all be lower-cased in MATLAB since the connection was opened with the "case = lower" option.

`xrun` is used to create a vector in MATLAB, called "beta", equal to the least squares coefficient, using the matrix algebra formula for LSQ. beta is brought back into EViews with the `xget` command, and then we estimate the same equation inside EViews and show the two results for comparison. Finally the connection to MATLAB is closed.

For additional detail and examples see the whitepaper: *Using the EViews COM Automation Client for* MATLAB.

Using R

R is a GNU-project software environment for statistical computing and graphics. R is free software (under the terms of the GNU General Public License) that is readily available for download from the Official R Project website: http://www.r-project.org/ and other mirror sites.

To use the EViews external interface to R, you must have R installed on your Windows computer, along with the rscproxy and statconnDCOM packages. Once installed, you may use the commands listed above to communicate with R from within EVIews.

For additional detail and examples see the whitepaper: *Using EViews COM Automation Client for* R.

Installing R Components

EViews was developed and tested with the following R components. You must have the following installed on the machine running EViews (or have R server components registered on the same machine via DCOM):

- R, version 2.8.1 or later.
- rscproxy, version 1.0-12.
- statconnDCOM, version 3.1-1B2.

Newer versions of rscproxy or statconnDCOM may work, but have not been tested with EViews. Note that rscproxy version 1.0-13 has known issues with the console output feature, such that you may not see any output in the R Output Log window in EViews.

New R installation

If you do not currently have R installed, the easiest procedure is to run the latest RAnd-Friends setup file located at

http://rcom.univie.ac.at/download.html

As of this documentation, RAndFriendsSetup2090V3.0-14-2.exe is the latest file, which includes the proper rscproxy version 1.0-12. Run the basic installation of RAndFriendsSetup2090V3.0-14-2.exe, leaving the **Direct internet connection** box checked, to install R version 2.9.0, rscproxy version 1.0-12, and statconnDCOM version 3.1-1B2. After installing statconnDCOM, you may cancel out of the installation of RExcel, as it is not needed by EViews.

Modifying an Existing R installation

If you already have R installed, verify that it is at least version 2.8.1. Also, you should check your R installation write path and version information in your windows registry, as this information is needed by statconnDCOM to find your R installation. You should have an entry in the registry that looks like the following:

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SOFTWARE\R-core]

[HKEY_LOCAL_MACHINE\SOFTWARE\R-core\R]
"InstallPath" = "C:\\Program Files\\R\\R-2.8.1"
"Current Version" = "2.8.1"

[HKEY_LOCAL_MACHINE\SOFTWARE\R-core\R\2.8.1]
```

"InstallPath" = "C:\\Program Files\\R\\R-2.8.1"

If you don't have these entries in your registry, you can re-install R and check the checkbox **Save version number in registry** (this option is checked by default) or you can manually add them to the registry (with your proper values for "Install Path" and "Current Version"). If you choose to edit your registry, we highly recommend that you first make a backup copy.

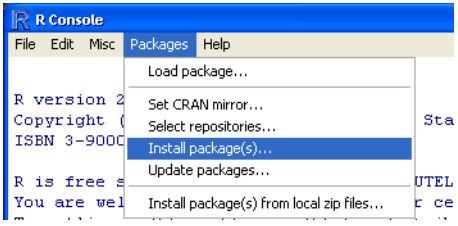
The current version of R is available at one of the Comprehensive R Archive Network sites near you (go to "http://cran.r-project.org/mirrors.html" for a current list). You should navigate to the "bin/windows/base" directory and download the installer for the current version (it should have a name that looks like "R-X.X.X-win32.exe". For more info, see the R Windows FAQ, which currently may be found at

http://cran.r-project.org/bin/windows/base/rw-FAQ.html

Installing rscproxy and statconnDCOM

When you have verified that the R installation path and version in your registry are correct, you can install rscproxy and statconnDCOM.

To install the rscproxy package, open R and select the **Packages\Install Package(s)...** menu item to see a list of available packages. First select your country and region, and then find **rscproxy** and click **OK**.



To install statconnDCOM, download the latest version from

http://rcom.univie.ac.at/download.html

and run the setup program.

Verifying installation

Once installation is complete, run the statconn "Server 01 - Basic Test" program, usually found under **Start- > All Programs- > statconn- > DCOM- > Server01 – Basic Test**, to verify that everything is working. Click on the **Start R** button to begin the test.

Once you have run the test, you should be able to run EViews and use the `xopen` command to open a connection to R.

```
xopen(type=r)
```

This command will open a connection to R from within EViews and will also open an R output log window in EViews.

You may then use the `xput`, `xrun`, and `xget` commands to transfer data and run commands in R from within EViews.

By default, EViews series, alpha, vector, and matrix objects transferred to R using `xput` will be written as R vectors. EViews group objects are sent to an R data.frame that contains both column and row identifiers.

Series objects may be grouped into a data.frame by including the "rtype = data.frame" and "name = " options. Certain series objects may be written as R time-series objects using the "rtype = ts" option. Objects may be grouped into an R list object by including the "rtype = list" option.

Series and group objects are always filtered by the current sample unless you specify an explicit sample in the `xput` command.

Example

The following commands show you how to move data from EViews into R, how to use EViews to run an R command, and how to retrieve results from R into EViews.

```
'create a workfile
wfcreate u 100
'create some data
series y=rnd
series x1=rnd
series x2=rnd
'open a connection to R with upper-case default output names
xopen(type=r, case=upper)
'put regressors and dependent variable into R
xput(rtype=data.frame, name=vars) y x1 x2
'run a command to perform GLM
xrun "z<-glm(Y~X1+X2, family=Gamma(link=log), data=vars)"
```

```
xrun "summary(z)"
'retrieve coefs
xget(name=beta, type=vector) z$coef
'create EViews equation
equation e1.glm(family=gamma,link=log) y c x1 x2
show e1
show beta
'close R connection
xclose
```

The program first creates a workfile, then creates some series objects. The series Y is the dependent variable, and the series X1 and X2 and a constant are regressors. `xopen` is used to open a connection to R, and then `xput` is used to pass the series into an R data.frame container, which we name "VARS". Note that the names of the data.frame, and its contents are all uppercased in R since the connection was opened with the "case = upper" option.

`xrun` is used to estimate a GLM equation in R, with the results being stored in a GLM output object, "z". `xget` is used to retrieve the coefficient member of z back into EViews, where it is stored as a vector called BETA.

Finally, the same GLM specification is estimated inside EViews, and the coefficient estimates are shown on the screen for comparison.

Output Display

Note that the statconnDCOM package does not always automatically capture all of your R output. Consequently, you may find that using `xrun` to issue a command that displays output in R may return only a subset of the usual output to your log window. In the most extreme case, you may see a simple "OK" message displayed in your log window. To instruct statconnDCOM to show all of the output, you should use enclose your command inside an explicit `print` statement in R. Thus, to display the contents of a matrix X, you must issue the command

```
xrun print(X)
```

instead of the more natural

```
xrun X
```

R Related Links

The Official R Project website: http://www.r-project.org/

StatConn Project website: http://rcom.univie.ac.at/

Chapter 8. Add-ins

In Chapter 6. "EViews Programming," beginning on page 105, we explain how you can put commands in program files to repeat tasks, produce a record of your research project, or augment the built-in features of EViews.

The chapter describes Add-ins, which extend the utility of the programming tools by providing seamless access to programs from the standard EViews menus and command line. Creating an Add-in is a simple procedure and involves little more than defining a command and menu items for your existing EViews program.

Keep in mind that Add-ins aren't just for EViews programmers. Even if you have never written an EViews program, you may take advantage of these tools by installing prepackaged Add-ins from the IHS EViews website or from third-parties. Once installed, Add-ins can provide you with user-defined features that are virtually indistinguishable from built-in EViews features.

What is an Add-in?

Fundamentally, an Add-in is simply an EViews program that is integrated into the EViews menus and command line, allowing you to execute the program using the menus or user-defined command. In this regard, any EViews program can be used as the basis of an Add-in.

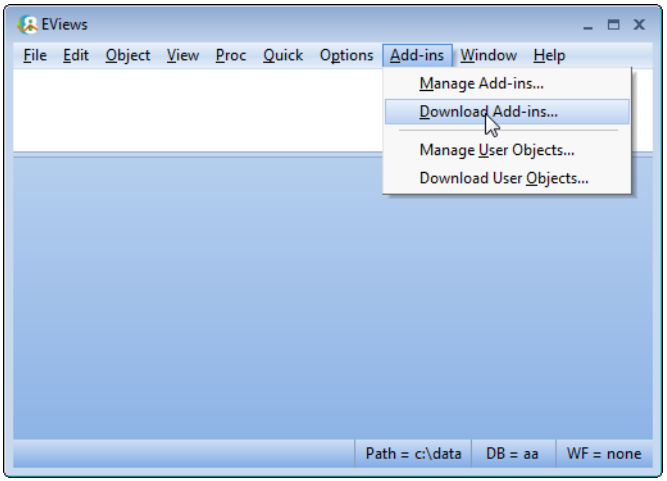
More specifically, the Add-ins infrastructure lets you:

- add entries to EViews global or object-specific menus to offer point-and-click execution of the Add-in program.
- specify a user-defined single-word global or object-specific command which may be used to run the Add-in program.
- display Add-in output in standard EViews object windows.

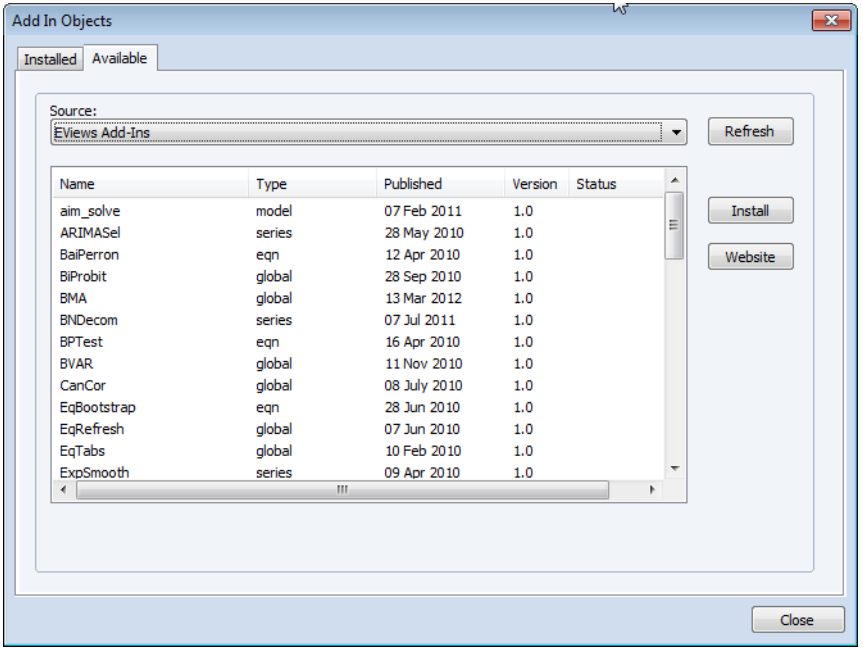
For example, suppose you have created a program to implement an econometric procedure that prompts the user for needed input. You may turn this program into an EViews Add-in that may be run by selecting a menu item or typing a command. Lastly, the Add-in might display the output in the window of an existing EViews object.

Getting Started with Add-ins

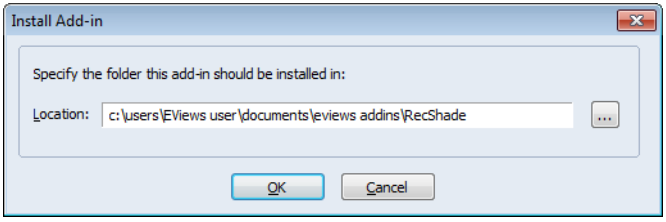
The easiest way to get started with Add-ins is to download and install one of the previously written Add-ins packages from the EViews website. Simply go to the main menu in EViews and select **Add-ins/Download Add-ins...**



EViews will open the corresponding **Add-ins** dialog opened to the **Available** tab showing the list of Add-ins that are available for download from the EViews.com website. The list shows the name of the Add-in, the publication date, version, and status. The status field indicates whether the entry has not been installed (blank), has previously been installed, or has previously been installed and is out-of-date.



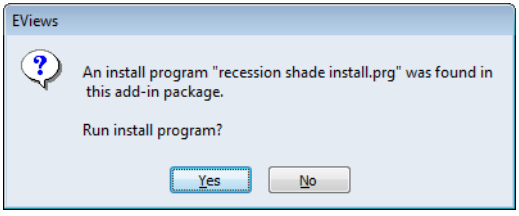
Select the desired entry to display additional information in the bottom of the dialog and to enable the **Install** button. Clicking on **Install** instructs EViews to download the Add-in and begin the installation procedure. (Alternately, you may click on the **Website** button and follow the navigation links to the Add-ins page. Download the appropriate file to your computer then open it using EViews by double-clicking on the file or dropping it onto the EViews application window or the application icon.)



The first step in the installation procedure is to unpack the Add-in files onto your computer. By default, EViews will put the files in a subfolder of your default directory (see "Managing Add-ins" on page 180), but you may choose an alternate location if desired (you may use the "..." button on the right-hand side to navigate to a specific directory). Click on **OK** to proceed.

Note that if you manually downloaded the Add-in from the EViews website, you need not use EViews to unpack the files. You may instead change the download file extension from "AIPZ" to "ZIP" and use your favorite ZIP file tools to extract the contents into the desired directory.

Next, EViews will prompt you to run the installation program that is included in the Add-in package.



The installation program is a file containing EViews commands that automatically registers the Add-in by defining menu entries and commands for the program.

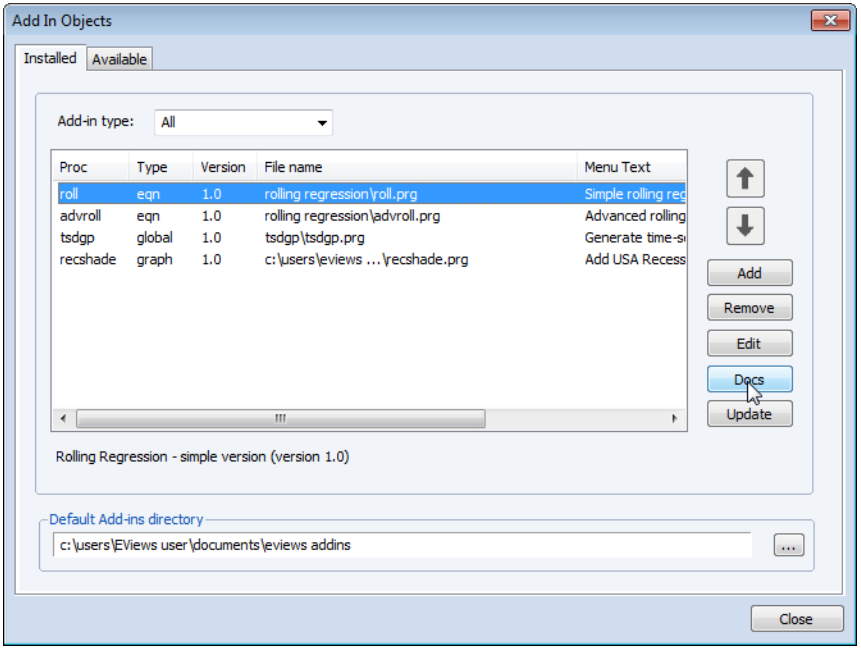
If you click on **No** in response to the prompt, EViews will finish the automatic installation procedure without running the installation program and registering the Add-in. The Add-in program files will be on your computer but will not be integrated into the standard command or menu interface. You may, at a later time, examine and run the installation program as you would any other EViews program, or you may manually register your programs as described in "Registering an Add-in" on page 184.

Click on **Yes** to finish registering the Add-in. If there are conflicts with existing directory names or existing Add-ins, EViews will warn you before proceeding.

After completion of the automatic installation procedure, EViews will report that the Add-in was installed successfully:

We note that installing and running an EViews program file provided by an unknown individual has risks. Accordingly, we recommend that care be taken when installing packages from non-trusted sites. All of the packages provided on the EViews website have been examined to ensure that they do not include potentially harmful commands.

Once your Add-in is installed, you may wish to consult the Add-in documentation to obtain additional information on features, options, and commands. Open the Add-ins management dialog by selecting **Add-ins/Manage Add-ins...** from the main EViews menu, select the Add-in of interest and click on the **Docs** button to display any documentation provided by the author:



Using Add-ins

Add-ins are integrated into the EViews menus and command line so that they work much like built-in routines. To run an Add-in program, simply select the corresponding menu entry or issue the appropriate command.

Beyond that, working with Add-ins menu and command entries does require some understanding of the difference between the two types of Add-ins: *object-specific* and *global*. As the names suggest, object-specific Add-ins are designed to work with a single object type, while global Add-ins are designed to work more generally with more than one object or object type.

For example, an Add-in that computes a spline using data in a series is likely to be object-specific, since it operates on a single series, while an Add-in that copies tables, graphs, and spools into an RTF file would naturally be defined as global.

The menu entries and form of commands differs between the two Add-in types.

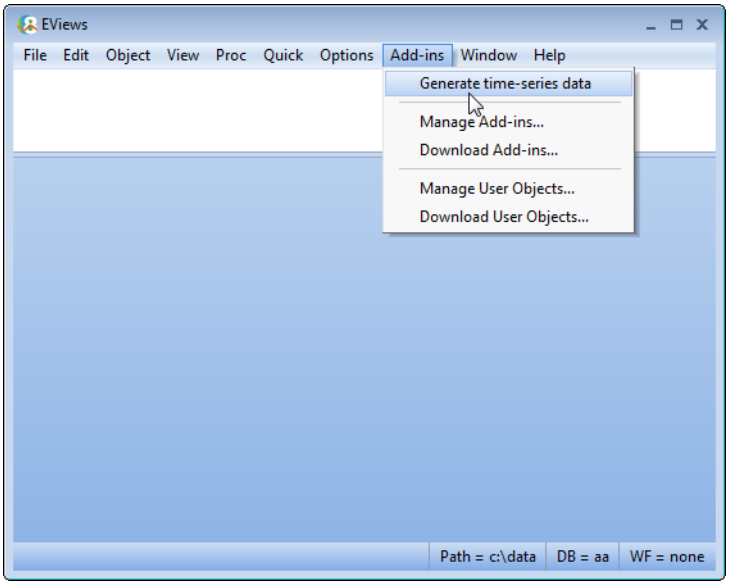
- Global Add-ins have menu entries that appear only in the main Add-ins menu. Global add-in commands follow the EViews command syntax:

  command(options) [args]

- Object-specific Add-ins have menu entries that appear in *both* the main Add-ins menu and in the menu of objects of the specified object type. Object-specific Add-in commands follow the standard EViews object command syntax:

  object_name.command(options) [args]

Suppose, for example, we have a global **Generate time-series data** Add-in with associated command `tsdgp`. Since the Add-in is global, it will have a menu item in the main Add-ins menu,

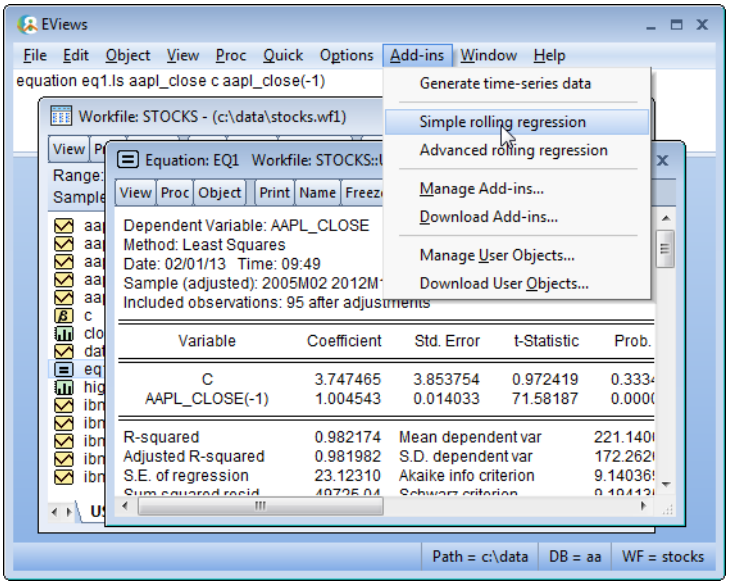


Moreover, the global command

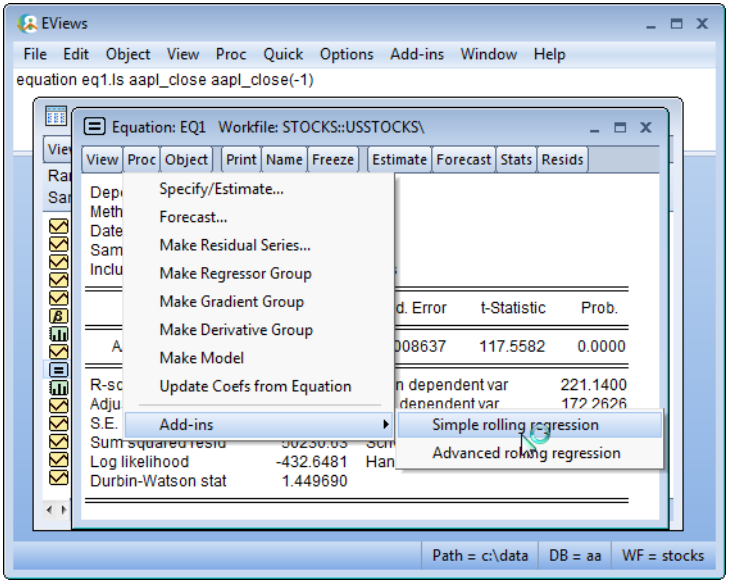
```
tsdgp(diff="2", seed=100, meanconst="2", ar="0.1", ma="0.15",
   varconst="0.8", arch = "0.15", garch="0.2 0.2") y
```

will run the Add-in program with the specified options.

Suppose, in addition, that we have two equation-specific Add-ins, **Simple rolling regression** and **Advanced rolling regression**, with associated object-specific commands, `roll` and `advroll`. If equation EQ1 is the active object, the main **Add-ins** menu will contain *both* the global (**Generate time series data**) and the two equation-specific entries (**Simple rolling regression** and **Advanced rolling regression**):



In contrast, the EQ1 equation object will have an object **Add-ins** menu contains only the two object-specific entries:

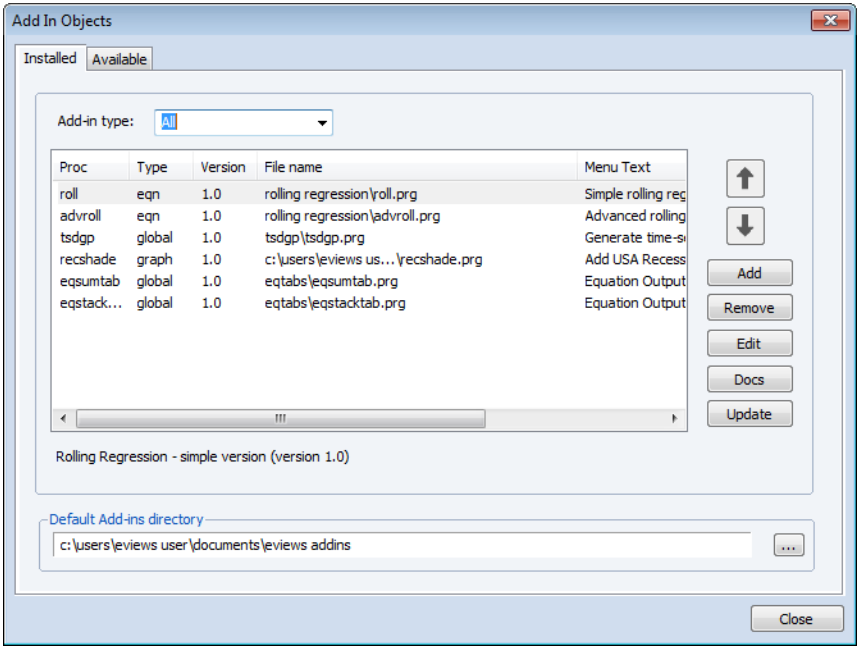


To run the simple rolling regression Add-in you may select either the main or the equation menu Add-ins entries, or you may enter the equation object command:

```
eq1.roll
```

in the EViews command window.

If you wish to see the available Add-ins and their types, you may click on the **Add-ins/Manage Add-ins...** entry in the main menu to display the Add-ins management dialog. EViews will display the list of installed Add-ins with a **Type** column showing the type associated with each entry:



Note that you may use the **Add-in type** drop-down to filter the display.

In this example, the Recshade (Add USA Recession Shading, ROLL (Simple Rolling Regression), and Advroll (Advanced Rolling Regression), Add-ins are all object-specific, while the Tsdgp (Generate time series data) Add-in is global.

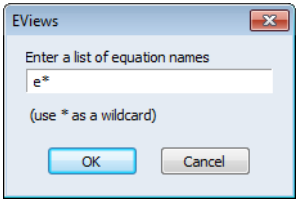
Add-ins Examples

To further illustrate the use of Add-ins, we examine two of the Add-ins currently available for download from the EViews website. *(To follow along with these examples, we recommend that you first download and install the corresponding Add-in using the steps outlined in* "Getting Started with Add-ins" on page 169*.)*

Summarize Equation Results

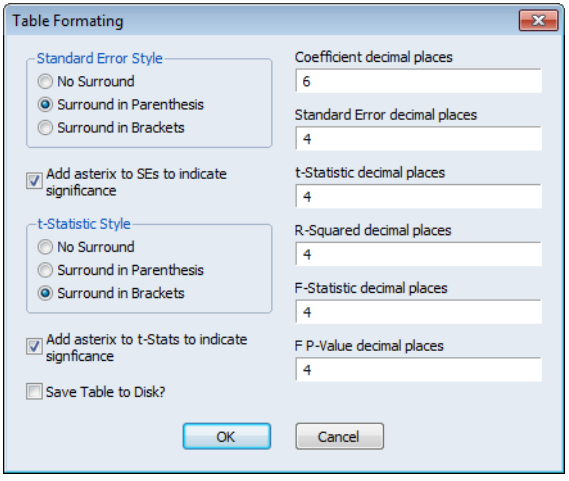
Our first example is a global Add-in (EqTabs) that creates a table summarizing the results from multiple equations. If you have not already downloaded and installed this Add-in, we recommend that you do so now. We employ the workfile "Demo.WF1" (which may be found in the examples subdirectory of your EViews installation directory.

To run the Add-in, go to the main EViews menu, and select **Add-ins/Equation Output Table (Summary form)**. EViews will display a dialog prompting you for the names of the equations you wish to summarize:



The default entry of "e*" is sufficient for this example since we want to summarize results for the previously estimated equations EQ01 and EQ02. Click on **OK** to continue.

EViews will display a series of dialogs prompting you to specify the headers you wish to include in the summary table, the information you wish to display, and the display format. For example, the last dialog lets you choose the standard errors and *t*-statistics display formats, along with the number of significant digits for coefficients and other statistics:



EViews will use the specified settings in constructing a table that summarizes the results from all of the specified equations.

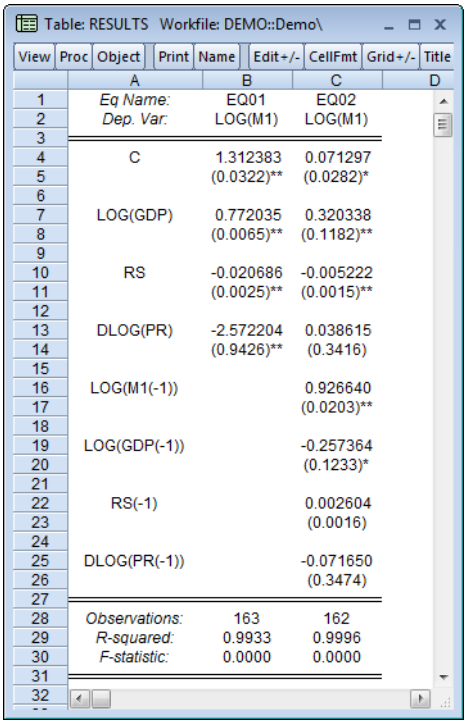


Table: RESULTS Workfile: DEMO::Demo\

View | Proc | Object | Print | Name | Edit+/- | CellFmt | Grid+/- | Title

	A	B	C	D
1	*Eq Name:*	EQ01	EQ02	
2	*Dep. Var:*	LOG(M1)	LOG(M1)	
3				
4	C	1.312383	0.071297	
5		(0.0322)**	(0.0282)*	
6				
7	LOG(GDP)	0.772035	0.320338	
8		(0.0065)**	(0.1182)**	
9				
10	RS	-0.020686	-0.005222	
11		(0.0025)**	(0.0015)**	
12				
13	DLOG(PR)	-2.572204	0.038615	
14		(0.9426)**	(0.3416)	
15				
16	LOG(M1(-1))		0.926640	
17			(0.0203)**	
18				
19	LOG(GDP(-1))		-0.257364	
20			(0.1233)*	
21				
22	RS(-1)		0.002604	
23			(0.0016)	
24				
25	DLOG(PR(-1))		-0.071650	
26			(0.3474)	
27				
28	*Observations:*	163	162	
29	*R-squared:*	0.9933	0.9996	
30	*F-statistic:*	0.0000	0.0000	
31				
32				

You may also launch the Add-in from the command line or via batch execution of a program file. Simply enter the user-defined command:

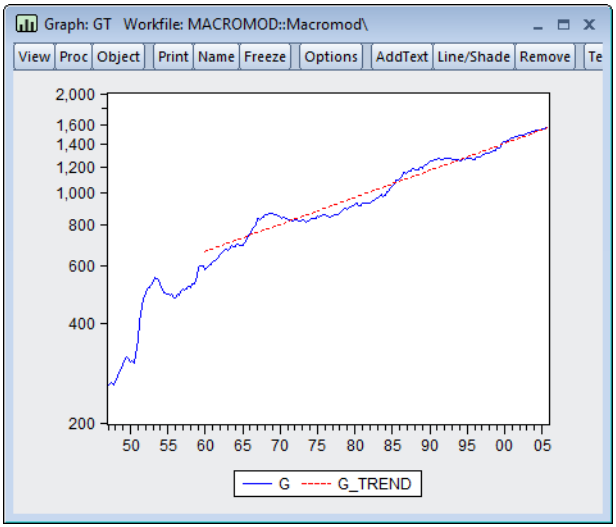
```
eqsumtab eq*
```

in the command line or include it in your batch program. The command instructs EViews to run the program, displaying the set of dialogs prompting you for additional input, and constructing the table accordingly.

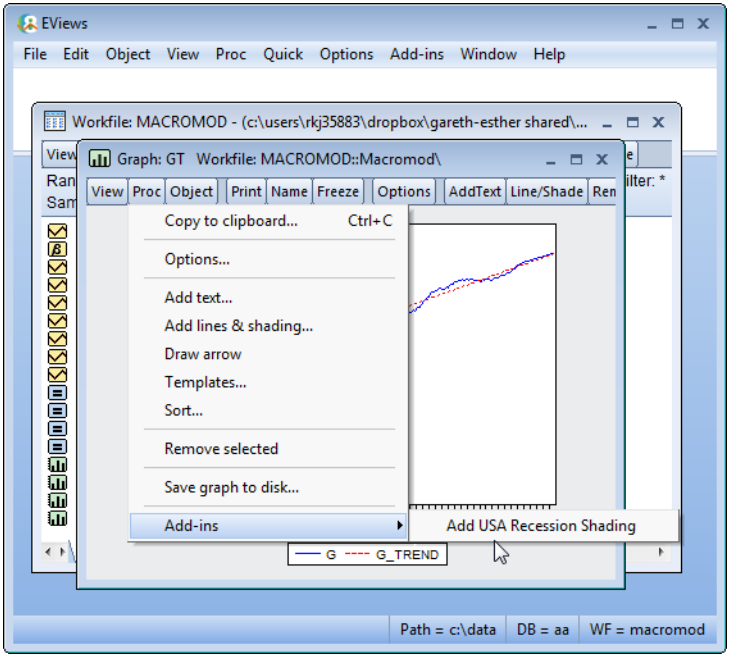
U.S. Recession Graph Shading

Our second example uses the graph-specific Add-in (RecShade) to add shading for periods of U. S. recession (as determined by the National Bureau of Economic Research).

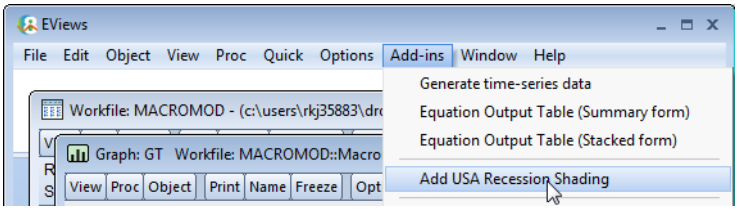
We start by opening the graph object GT in the "Macromod.WF1" workfile (which may be found in the example files subdirectory of your EViews installation directory).



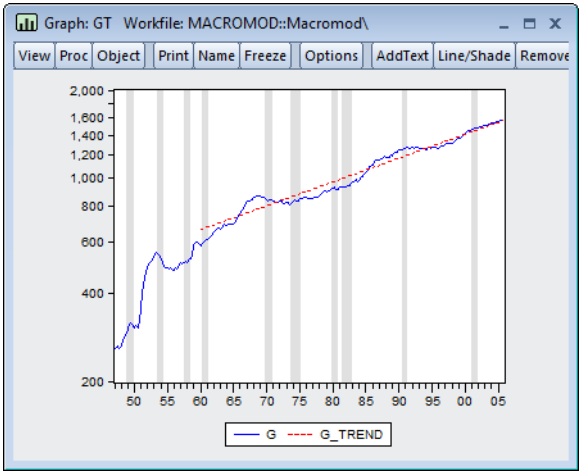
Next, click on the **Proc/Add-ins** menu item in the graph toolbar to display the Add-ins we have installed that work with graph objects. In this case, there is only a single menu item **Add US Recession Shading**:



You may also access the active object menu items by clicking on the **Add-ins** entry in the main EViews menu. When the graph object GT is the active object, the main **Add-ins** menu shows both the global and graph specific menu items:



Selecting the **Add US Recession Shading** entry in either the main or graph object Add-ins menu runs the Add-in program which adds shading to the existing graph:



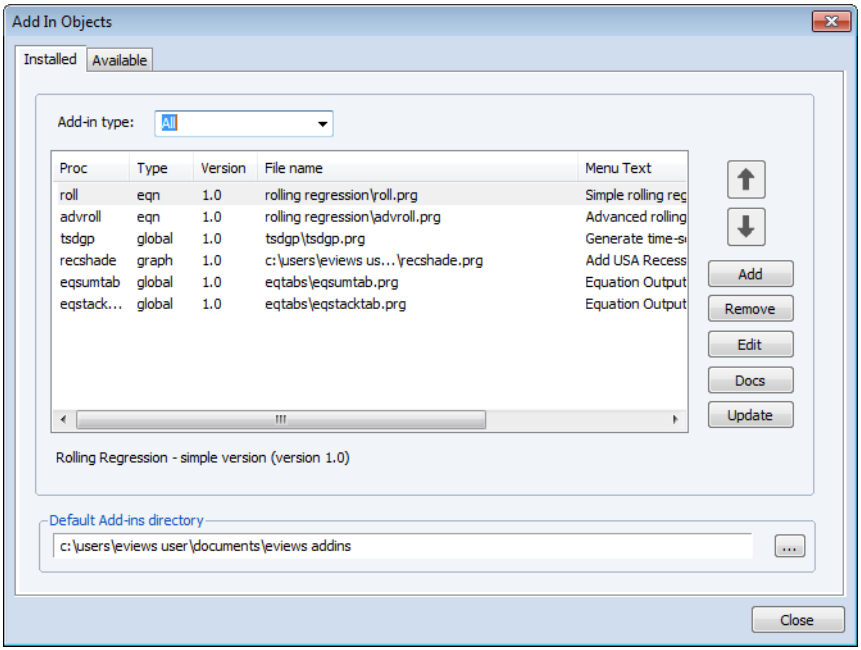
For those who prefer to use a command, you may go to the command line and enter the user-defined object command

```
gt.recshade
```

to apply the recession shading. This command may also be included in a program file for batch execution.

Managing Add-ins

EViews offers a complete system for managing your Add-ins. To bring up the management dialog, you should select **Add-ins/Manage Add-ins...** from the main EViews menu:

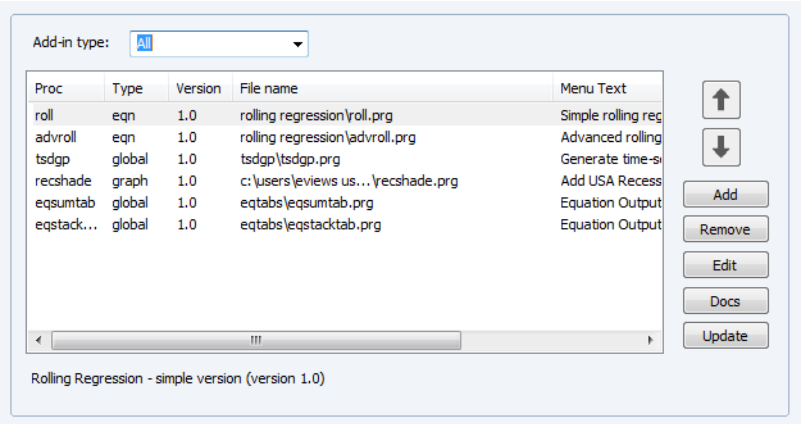


The management dialog is divided into two sections:

- The **Registered Add-ins** section is where you will perform most of the tasks of managing your Add-ins (add, delete, edit, update, and reorder Add-in definitions, and view the documentation file).
- The **Default Add-ins directory** section allows you to set the default directory for your Add-ins.

Registered Add-ins

The top portion of the dialog shows settings for currently installed Add-ins, with each line devoted to an Add-in.



By default, all of the installed Add-ins will be displayed. You may use the **Add-in-type** drop-down menu to filter the list, showing for example only global or only equation-specific Add-ins.

The **File name** column shows the name (and possibly location) of the Add-in program, and the **Type** column indicates whether the Add-in is global or object-specific. The **Proc** column shows the command keyword associated with the Add-in (if any), while the **Menu Text** column shows the text used in the user-defined menu entries. The **Version** column show the version number of the Add-in.

You may use the buttons and arrows on the right-hand side of the dialog to manage your Add-ins:

- To add a new Add-in to the list, simply click on the **Add** button to display the **Add/Edit Program** dialog. The **Add/Edit Program** dialog is described in detail in "Registering an Add-in" on page 184.
- To delete an Add-in, simply click on the name in the Add-ins management dialog and press the **Remove** button.
- The order in which your Add-ins appear in the menus may be controlled using the up and down arrows. If you have many Add-ins, putting the most frequently used Add-ins first in the list may simplify menu access (see "Menu Congestion" on page 187). In addition, the order in which Add-ins appear can have important consequences in the event that Add-ins duplicate command names (see "Command Masking" on page 186).
- To edit the settings of an existing Add-in, select it from the list and chick on the **Edit** button to display the **Add/Edit Program** dialog. The **Add/Edit Program** dialog is described in detail in "Registering an Add-in" on page 184.

- To examine the documentation file associated with an Add-in, click on the name and press the **Docs** button.
- To check whether the Add-in has an updated version available, and to install the update if available, click on the **Update** button.

Note you may select multiple Add-ins at the same time and click on the **Remove** or **Update** buttons to perform the specified operation. You may also right click anywhere on the list of Add-ins and select **Update All** to check for updates on all of your Add-ins.

After modifying the desired settings, click on **OK** to accept any changes.

Default Add-ins Directory

The bottom portion of the Add-ins dialog shows the current default Add-ins directory. The default directory is where an Add-in will search for supplementary files if explicit directory locations are not provided. To change the default directory, click on the button on the right and navigate to the desired directory, or simply enter the desired folder name. Click on **OK** to accept the settings.

Creating an Add-in

You can use Add-ins developed by others without ever having to create one yourself. Indeed, many users will never need to go beyond running Add-ins downloaded from the EViews website or other repositories.

You may find, however, that creating your own Add-ins is both useful, if only to add a menu item or assign a one-word command for running your favorite EViews program, and easy-to-do.

Assuming that you already have an EViews program, creating an Add-in requires at most two steps:

- Register the program as an Add-in. Registering an Add-in is the process of defining the Add-in type and assigning menu entry and command keywords to an EViews program.
- (*optional*) Create an Add-in package for distribution. Bundling your program files into a self-installing package file and providing a program for automatically registering the Add-in means that others can more easily incorporate your Add-in into their EViews environment.

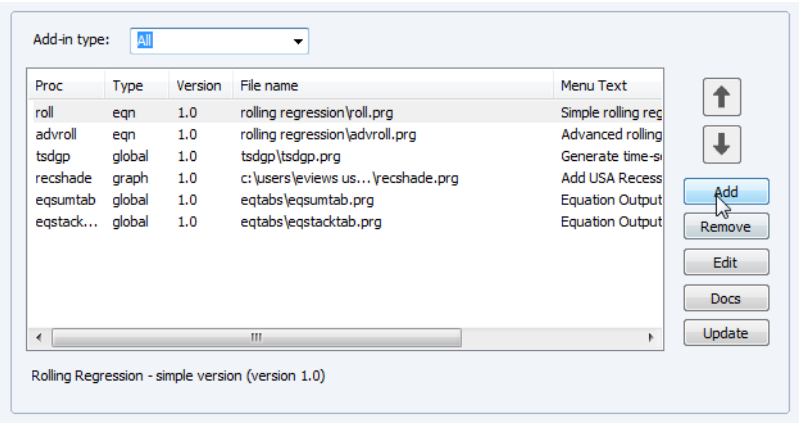
While only the first step is required, you should consider the optional step of creating a self-installing Add-in package if you wish to distribute your Add-ins more widely.

In the remainder of this section we describe the steps required to register an Add-in. In addition, we provide design tips and describe additional programming tools that will aid you in writing sophisticated Add-in programs.

Registering an Add-in

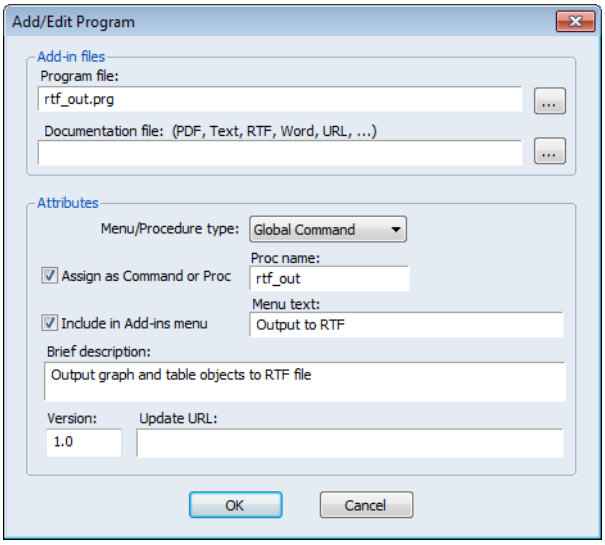
The process of defining the Add-in type and assigning menu entry and command keywords to an EViews program is termed *registration*. The Add-in registration process may also be used to associate documentation with the Add-in.

To register an EViews program file as an new Add-in, click on the **Add-ins/Manage Add-ins...** menu entry in the main EViews menu. The top portion of the management dialog shows the list of currently registered Add-ins:



Click on the **Add** button to display a standard file **Open** dialog. Navigate to the program file you wish to register and click on **OK** to continue.

EViews will display the **Add/Edit Program** dialog with various default settings.



The dialog allows you to specify a command keyword for the Add-in, to add menu items for point-and-click access to the Add-in, and to attach documentation and descriptions.

- The **Program file** edit field in the dialog should be used to specify the program to be used as an Add-in. You may enter the name of a file (with an absolute or Add-ins directory relative path, if necessary) or you may click on the button to the right-hand side of the edit field to navigate to the file.
- The **Documentation file** edit field allows you specify a PDF, Text, RTF, Microsoft Word file, or URL containing documentation for the Add-in. Note that relative paths are evaluated with respect to the directory of the Add-in program, not the default Add-in directory.
- The **Menu/Procedure type** dropdown setting determines whether the program is a global or an object specific Add-in. (Recall that global Add-ins are those designed to work with multiple objects or object types, while object-specific Add-ins work with a single object and object type.)
- If you check **Assign as Command or Proc**, EViews will add the specified single-word command or proc keyword to the EViews language, so you may run the Add-in using a global or object command. Names for the keyword must follow standard EViews command naming convention. Global Add-ins should not be given the same name as built-in EViews commands. (See "Command Masking" below, for additional discussion.)
- If you check **Include in Add-ins menu**, EViews will add the specified **Menu text** to the appropriate menus.

- You may use the **Brief description** edit field to describe the purpose of the Add-in.
- You may enter a **Version** number for your Add-in and an **Update URL** indicating where to find the update XML file (see "XML File Specification" on page 189).

In the example above, we instruct EViews to use the file "Rtf_out.PRG", which is located in the default Add-ins directory, as an Add-in program. We indicate that the program is a global Add-in that may be run from the main EViews menu by selecting the Add-ins menu item **Output to RTF** or by issuing the command `rtf_out`. There is no documentation file.

Add-ins Design Issues

For the most part, defining commands and menu items for an Add-ins is straightforward. There are, however, a few issues that you should bear in mind when designing your Add-in.

Command Masking

Allowing you to define user-specified Add-in command names opens up the possibility that an Add-in will be assigned the same command as an existing EViews command, or that multiple Add-in programs will be assigned the same name. Duplicate command names will generate an error or will lead to command masking where some of the instances will be ignored.

- If you specify a *global* Add-in command name that is identical to an EViews command or a previously defined global Add-in command, EViews will issue an error message, and will display the **Add/Edit Program** dialog so you can provide a different name. EViews will not, for example, permit you to register an Add-in with the global command `copy` since this conflicts with the built-in command.
- EViews will not generate an error message if you provide an *object-specific* command name that is identical to a *built-in* command, but the EViews command will mask the user-defined command. EViews will, for example, permit you to register an equation command with the name `resid`, but if you enter the equation command "eq01.resid", EViews will display the built-in `resid` view of equation EQ01, instead of running the user-defined Add-in.
- If you specify an *object-specific* Add-in command name that is identical to another *object-specific* command, EViews will not error. When multiple Add-ins are assigned the same object-specific command, the first Add-in listed in the list of registered Add-ins will be used and the remainder will be masked. To eliminate masking, you must edit the conflicting Add-in command definitions. If you wish to retain the same command name for multiple Add-ins, you should use the Add-ins management dialog to reorder the Add-ins list so that the desired Add-in has priority ("Managing Add-ins" on page 180).

We emphasize that masking only occurs within like Add-ins. You may have a global, series, and group Add-in that each use the same command without experience masking, but two global Add-ins or two series Add-ins with the same name will lead to masking.

Menu Congestion

While you may define as many Add-in menu items as you like, EViews does place limits on the number of menu items that may be displayed by default. If you have more than 10 global menu entries or more than 10 object-specific menus of a given type, the corresponding menus will display the first 10 entries, along with an 11th menu entry, **More...** Clicking on **More...** displays a listbox showing the full set of entries. In such circumstances, we recommend that you reorder your Add-ins so that the most used Add-ins are given priority.

Multiple Object Add-ins

You may wish to use a single program file to define Add-ins for several object types. Since it is not possible to use a single Add-in entry to define multiple object-specific types, you must create separate entries for each object-type which point to the single program file. *For each relevant object type,* select **Add-ins/Manage Add-ins...** from the main EViews menu, then navigate to and select the program file. Use the **Add/Edit Program** dialog to define the object-specific entry.

Creating an Add-in Package

If you are developing Add-ins for use by others, we recommend that you create a self-installing package so that users may easily incorporate your Add-ins in their EViews environment. You can then host your package on a company intranet or perhaps submit your package to be hosted on the EViews.com website for wide distribution.

The process of creating an Add-in package is straightforward, requiring at most two steps:

- (*optional*) Create a table of contents (TOC) information file and installer program file.
- Create a self-extracting Add-in package file containing the program and support files (including the TOC and installer program file, if available).

The second step, creating the self-extracting package, is trivial. Simply create a standard ZIP archive file containing all of the files for your Add-in, then rename the ZIP file so that it has the extension "AIPZ". You now have an self-extracting Add-in package file.

Opening a self-extracting package file, either automatically by a browser after completing the download, by double clicking on the file, or by dropping it onto EViews, will prompt you to unpack and copy the files to the location of your choice.

The Add-in will not, however, automatically be installed and registered unless you include a table of contents (TOC) information file and installer program along with your program files. Creating the TOC and installer program files takes only a few minutes and allows you to

automate the Add-in installation and allow for automatic updating of your Add-in as you provide newer versions. We strongly recommend that package distributors take the time to create these files as described below.

Table of Contents

First, you should create a table-of-contents file named "Toc.INI" for inclusion in your package. The TOC file should contain setup information which describes the directory in which the Add-in files should be installed, and if appropriate, the name of an EViews installer program for registering the Add-in. The format of the TOC file is:

```
[package]
installer = <name of installer file>
folder = <name of folder to create>
```

A TOC file should always begin with the line "[package]", followed by lines which give the directory and installer information.

The `installer` keyword is used to indicate the name of the EViews program file, if one exists, that should be run to register the Add-in (see "Installer Program" on page 189). If, for example, a registration program file named "Recession shade install.PRG" is included in your package, you should include the line

```
installer = recession shade install.prg
```

If you include this line in your TOC, EViews will automatically run the installer program when it opens the AIPZ package. If you do not wish to provide an installer program, you should include the line "installer = none" in the TOC file.

The `folder` keyword may be used to indicate a subfolder of the default Add-ins directory into which you will extract the packaged files. Thus,

```
folder = RecShade
```

tells EViews to extract the contents of the AIPZ file into the "RecShade" folder of the Add-ins directory. If no folder is specified, the name of the AIPZ file will be used as the target folder name. Additionally, you may use the special folder name "<addins>" to indicate that the contents of the AIPZ archive should be placed in the main Add-ins directory. (Note, however, that only folders in an AIPZ archive may be written to the main Add-ins directory in this fashion; individual files in AIPZ files must be written into subdirectories.

We emphasize that creating a TOC file and providing an installer program are not required. In the absence of a TOC file or an `installer=` specification, EViews will, after unpacking the AIPZ files, simply display a message reminding the user that the installed programs may be registered manually using the Add-ins management dialog.

Nevertheless, we strongly recommend that package distributors provide both a TOC file and installation program to facilitate use of their Add-ins. Packages hosted on the EViews website must include both a TOC and an installer.

Installer Program

Next, you should create a simple EViews program that uses the `addin` (p. 270) command to register the Add-in with appropriate type, menu, and command settings. Note that the TOC file `installer=` specification should point to this installer program.

For example, the graph-specific Add-in described in "U.S. Recession Graph Shading" on page 178 may be registered by including the following command in a program file:

```
addin(type="graph", menu="Add USA Recession Shading",
   proc="recshade", docs=".\recession shade.txt", desc="Applies US
   recession shading to a graph object.") ./recshade.prg
```

The options in this example should be self-explanatory. The command registers the program "./recshade.PRG" as a graph-specific Add-in with menu item "Add USA Recession Shading", command name "recshade", and description text "Applied US Recession shading to a graph object".

See `addin` (p. 270) for details and a complete list of options. Use of the following `addin` options is highly recommended:

Documentation

We recommend that you provide documentation for your Add-in, and use the "docs = " option to point to the documentation file.

Documentation could be anything from a simple text file with some syntax hints, to a lengthy PDF document that describes the Add-in features in detail.

Version

EViews allows Add-ins to have a version number which allows the users of your Add-in to use automatic updating to ensure they have the latest version of the Add-in. When a user uses the **Update** button on the **Manage Add-ins** dialog to check for Add-in updates, EViews will compare the hosted version number with the currently registered version number and download the latest version if necessary.

You may use the "version = " option to specify the version number. If omitted, EViews will assume that the Add-in version number is 1.0.

XML File Specification

One of the most useful Add-ins management features is the ability of users to automatically update their installed Add-ins as newer versions become available. To support this feature,

EViews must know where to look to determine the most recent version of the Add-in and where to download any updates.

This information is communicated in an XML file, typically located on the Add-ins package hosting site. If you will be hosting this file, you should use the `addin` option "url = " to specify the URL for the XML file. If this option is not supplied, EViews will look for the XML file on EViews.com.

The XML file should contain one or more item definitions, where each item contains information on a specific Add-in. An item definition is contained in the lines between an `<item>` and `</item>` tag. An example of the full specification of an item is as follows:

```
<item>
<title>BMA</title>
<path>bma\bma.prg</path>
<path>bma\bmamlogit.prg</path>
<version>1.0</version>
<description>Computes different Bayesian Model Averaging methods
    including LM, GLM and Multinomial Logit models.</description>
<link>http://eviews.com/Addins/BMA.aipz</link>
<pubDate>13 Mar 2012</pubDate>
</item>
```

Note that the only required specifications are the `<title>` and `<link>`.

The `<path>` specification is used to identify the paths and file names of the main program files used by the Add-in, with the path location specified relative to the Add-ins directory. Automatic updating will update these files when a newer Add-in version is available. If `<path>` is not specified, EViews will use the `<title>` specification to determine the relevant Add-in proc name, and use registration information for the proc name to determine the files to update.

When an add-in package has multiple main program files, a `<path>` statement is required. You should list each file using a separate `<path>` entry. In the example above, the BMA Add-in has two program files, called "bma.PRG", and "bmalogit.PRG" that are associated with procs. EViews will update all of the files associated with these procs when updating the Add-in.

The `<version>` is used to specify the current Add-in version number. When the user checks for updates, EViews will download the updated version if the version number they have currently installed is lower than the one given in the `<version>` tag.

Finally, the `<link>` specification contains the URL (or network file location) of the AIPZ file containing the Add-in. This is the location from which EViews will download the updated Add-in package should the user request an update.

The <description> and <pubDate> specifications should be self-explanatory.

Add-ins Design Support

EViews offers several programming language features that will aid you in developing and working with Add-ins.

Add-ins Registration Command

The addin command may be used to register an EViews program file as an Add-in. You may use the command to specify the Add-in file, Add-in type, menu text, user-defined command name, description, version number, documentation file, XML file, *etc.*

See addin (p. 270) for details.

The Active Object Keyword

"_this" Keyword

Central to the construction of an object-specific Add-in program is the ability to reference the object on which the program should act. If, for example, you wish to write an Add-in that computes a statistic based on the data in a series object, you must have a convenient method of referring to the series.

Accordingly, EViews provides an object placeholder keyword, _this, which refers to the currently active object upon which a program may operate. Typically the use of _this in an EViews program indicates that it has been designed to work with a single object.

There are three ways in which the identity of the _this object may be set:

- _this refers to the active object whose window was last active in the workfile; when used in a program, _this refers to the active object at the time the program was run.
- executing an Add-in using the object-command syntax, *obj_name.proc,* sets _this to *obj_name.*
- _this can be set to a specific object using the "this = " option in an exec or run command.

While the above description is a bit abstract, a simple example should illustrate the concepts that lay behind the three methods. Suppose we have the trivial (silly) program "Myline.PRG" which consists of the command:

```
_this.line
```

First, if we register this program as a global Add-in with menu item text "Show line", we can display a line graph of a series or group object by opening the series or group and selecting **Show line** from the Add-in menu. From the program's point of view, the _this object is

simply the opened series or group whose menu we are using (the last one active in the workfile).

Alternately, if we had registered the program as a series-specific Add-in with proc name "myl", the command:

```
ser01.myl
```

identifies SER01 as the `_this` object, so that the object used by the Add-in will be the series SER01, regardless of which object is active in the workfile.

Lastly, you may specify `_this` explicitly when using the `exec` or `run` command to run the program by including the "this = " option to identify an object by name. The command:

```
exec(this=ser01) myline.prg
```

explicitly specifies SER01 as the `_this` object.

Custom Object Output

EViews allows you to display the contents of a table, graph, or spool object in the window of another object. This feature will undoubtedly most often be used to mimic the behavior of EViews views and procedures in Add-ins programs.

Suppose, for example, that your Add-in program performs some calculations and constructs an output table TABLE_OUT. You may instruct EViews to display the contents of TABLE_OUT in the object OBJECT_01 using the `display` command:

```
object_01.display table_out
```

Thus, a useful approach to constructing an object-specific Add-in involves creating a program of the form:

```
[use _this to perform various calculations]
[create an output table or graph, say the table TABLE01]
' put the output in the _this window
_this.display table01
delete table01
```

Note that we delete the TABLE01 after we put it in the window of `_this`. (You may instead wish to employ local subroutine to enable automatic cleanup of the temporary table.)

If the above program is registered as a series-specific Add-in with the command "FOO", then you may run it by issuing the command

```
series01.foo
```

which will display the output in the SERIES01 window.

The `display` object command is a documented view for each supported object. See for example, `Series::display` (p. 491) in *Object Reference*.

Chapter 9. User Objects

As the name suggests, the EViews user object allows you to create your own object types inside of EViews. A user object may be as simple as a storage container for other EViews objects, or it may be a sophisticated new estimation object defined by multiple EViews programs, with views containing post-estimation tests and results, and procedures producing output from the estimation results. Once defined, a user object is almost indistinguishable from a built-in EViews object.

Defining a user object is quite easy—simply specify the types of data and objects stored inside your object, and if desired, define a set of views and procedures that be accessed via commands, menus and dialogs.

Even if you do not go to the trouble of creating your own objects, you may take advantage of this powerful tool by using user objects downloaded from the IHS EViews website or obtained from third-parties.

What is a User Object?

An EViews user object is a custom object that can contain data and objects and may offer views and procedures. In its simplest form, a user object is a storage container for EViews objects. More sophisticated user objects also provide views and procs that allow you to run EViews programs to perform various tasks and display results. These latter objects work almost identically to built-in EViews objects such as a series or equation.

In the discussion to follow it will be important to distinguish between user objects that are *unregistered* or *registered*:

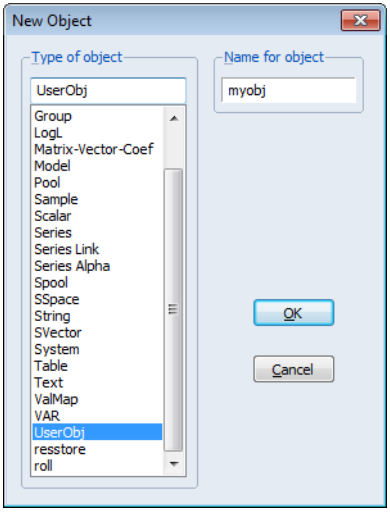
- Unregistered user objects are simple container objects which require virtually no effort to create.
- Registered user objects are more powerful than unregistered user objects. Registering a user object class is the process of describing what happens each time a new instance of the user object is created, and defining data and a set of views and procs available to the user object class.

A relatively complex registered user object might be a complete econometric estimator. Each time a new instance of the estimator object is created, it could specify and perform estimation, saving results inside the user object in the form of data objects such as coefficient vectors and covariance matrices. The object could also offer views such as coefficient tests and procs to perform forecasting. And like an EViews equation object, you may have multiple instances of this estimator in the workfile, each corresponding to a different set of estimates.

We note that a registered user object need not be particularly complex. For example, you could have a simple user object called "RESULTS" that contains a collection of graphs, tables, and estimation objects obtained from a particular form of analysis. You could define simple views for your user object that display the stored tables or graphs, and define procs that let you extract those tables or graphs into new EViews objects. Registering the object allows you to have multiple results objects in your workfile.

Unregistered User Objects

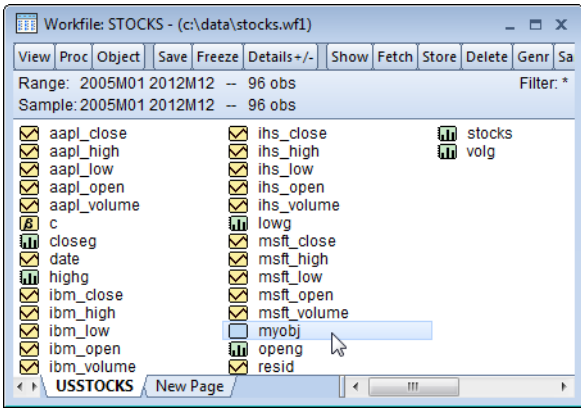
To create a new, *unregistered* user object, select **Object/New Object** in the main EViews menu.



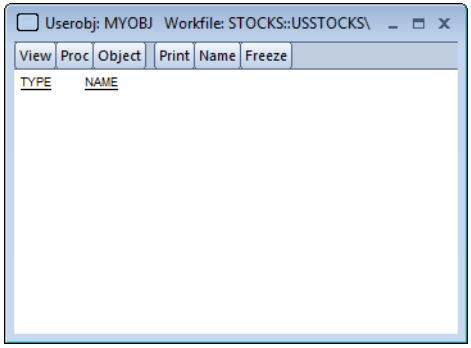
Scroll down and select **UserObj**, enter a name for the object in the workfile (in this example, MYOBJ), and click on **OK**. Alternately, you may enter the generic user object declaration command

```
userobj myobj
```

to create a user object named MYOBJ in the workfile.



Notice that MYOBJ has a black icon. A user object created in this fashion is empty, with no user-defined views or procedures. Double-clicking on MYOBJ opens the object and displays its contents:



As you can see, the object is empty. Clicking on the **View** menu of the object shows only a single **Label** entry. The **Proc** menu is completely empty.

An empty, unregistered userobj is not particularly interesting. You may, however, use the object as a container for EViews matrix objects (including scalars), string objects, estimation objects, and view objects (graphs, tables, spools):

- The `add` and `drop` procs may be used to populate the user object and the `extract` proc may be employed to extract objects into the workfile.
- You may use user object data members to provide information about the contents of the object: the `@hasmember(`*obname*`)` member function can be used to determine whether *obname* exists inside the user object, while `@members` returns a space delimited string listing all objects in the user object.

See "User Object Programming Support" on page 218 for details.

The following program offers a simple example showing the use of these commands:

```
userobj myobj
myobj.add mygraph
myobj.add mytable
%list = myobj.@members
myobj.drop mygraph
myobj.extract mytable mynewtable
```

The first line creates a new, empty, user object called "MYOBJ". The second and third lines copy the workfile objects "MYGRAPH" and "MYTABLE" into MYOBJ. The fourth line creates a string variable whose contents are "mygraph mytable". The fifth line removes MYGRAPH from MYOBJ, and the final line copies MYTABLE back into the workfile under the name MYNEWTABLE.

Registered User Objects

While simple, unregistered user objects may only be employed as storage containers, registering a user object class creates a more powerful working environment. Note that we used the term *user object class*, reflecting the fact that when you register a user object, you are not simply declaring a single object but rather are defining the general characteristics of a *type* of object.

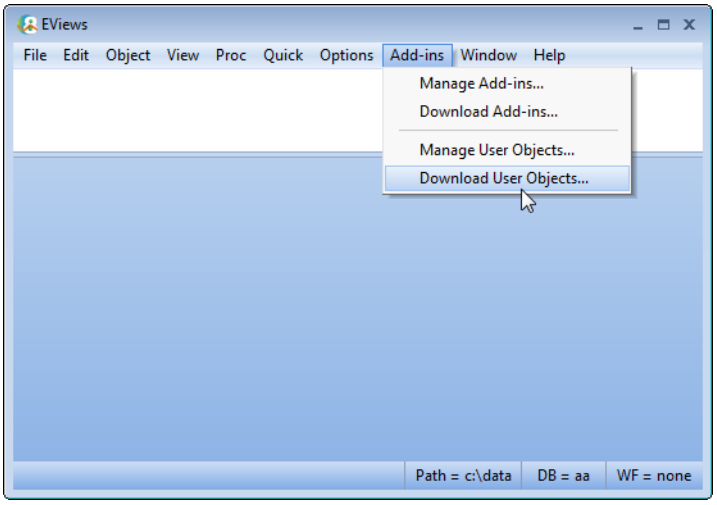
Registering a user object class allows you to have multiple objects of a given type in your workfile, each of which has its own data. Additionally, registering allows you, if desired, to define views and procs that can be used by any object of that class. These views and procs will execute a set of EViews programs that you specify as part of the registration procedure.

While not difficult, creating a registered user object class is a bit more involved than creating an unregistered user object. Details are provided in "Defining a Registered User Object Class" on page 212.

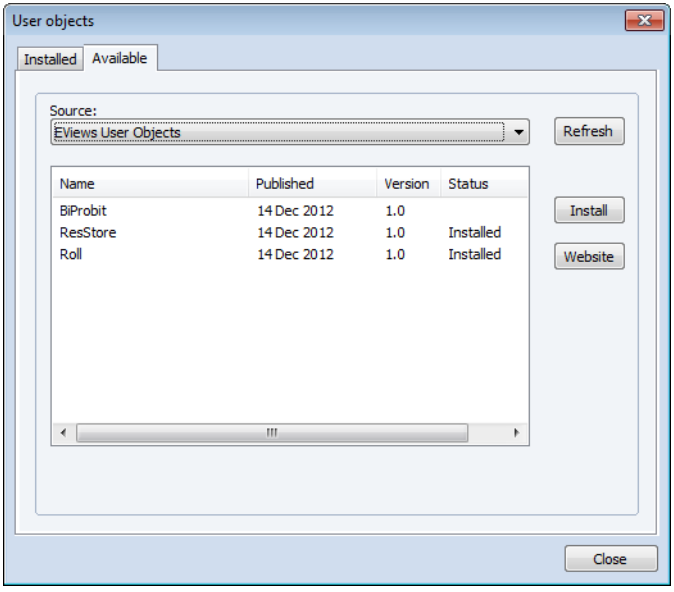
You may, of course, download and register user object classes created by others. Since working with example user objects provides good introduction to this powerful tool, we begin by discussing the steps required to download an object from the EViews website.

Downloading a Registered User Object

To download and install object class definitions from the EViews website, simply select **Add-ins/Download User Objects**...from the main EViews menu.

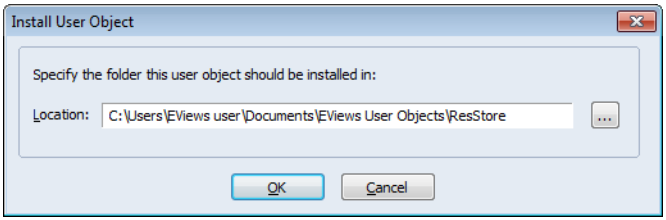


EViews opens the **User objects** management dialog opened to the **Available** tab, which shows a list of the user objects classes (in this case ResStore, Roll, and BiProbit), that are available for download along with the date they were published, their version number, and their status (blank for un-installed, installed, or installed but out of date):



Selecting an entry displays a description of what the user object does below the listbox. Clicking on the **Install** button downloads the selected user object and prompts you to install

the package on your the local computer. (Alternately, you may click on the **Website** button and follow the navigation links to the user objects page. Download the appropriate file to your computer then open it using EViews by double-clicking on the file or dropping it onto the EViews application window or the application icon.)



The first step in installation is to unpack and copy the files to your computer. By default, EViews will put the files in a sub-folder of your default directory (see "Default User Objects Directory" on page 212) but you may choose an alternate location if desired (you may use the "..." button on the right-hand side to navigate to a specific directory). Click on **OK** to proceed.

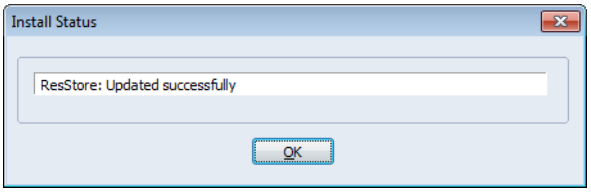
To unpack the files without using EViews, simply change the download file extension from "AIPZ" to "ZIP" and use your favorite ZIP file tools to extract the contents into the desired directory.

Next, EViews will prompt you to run and installation program that is included in the user object package.

If you click on **No** in response to the installation prompt, EViews will finish the automatic install procedure without running the installation program and registering the userobj. You may later examine the installation program prior to running it as you would any other EViews program, or you may manually register your object as described in "Registering a User Object Class" on page 214.

Click on **Yes** to finish the installation and registration. If there are conflicts with existing directory names or existing user objects, EViews will warn you before proceeding.

After completion of the automatic installation procedure, EViews will report the status of the installation:



We note that installing and running an EViews program file provided by an unknown individual has risks. Accordingly, we recommend that care be taken when installing packages from non-trusted sites.

All of the packages provided on the EViews website have been examined to ensure that they do not include potentially harmful commands.

Working with Registered User Objects

Once you have registered your user object you may work with it much as you would any built-in EViews object.

You can create a new instance of the object using the **Object/New Object...** main menu item, or by declaring it on the command line using the name of the object and any relevant options or arguments:

userobj_class_name(options) my_objname [args]

You may use the defined views and procs of the object using the object **View** or **Proc** menu, or via the command line using the standard syntax:

userobj_name.view_name(options) [args]
userobj_name.proc_name(*options*) *[args]*

The user object data member @-functions may be accessed using the syntax:

[result_type] result = userobj_name.@datamember_name[(arg)]

Examples

To illustrate the use of registered user objects, we examine two of the EViews user objects that are currently available for download from our website. *(To follow along with these examples, we recommend that you first download and install the corresponding user object using the steps outlined in* "Downloading a Registered User Object" on page 198*.)*

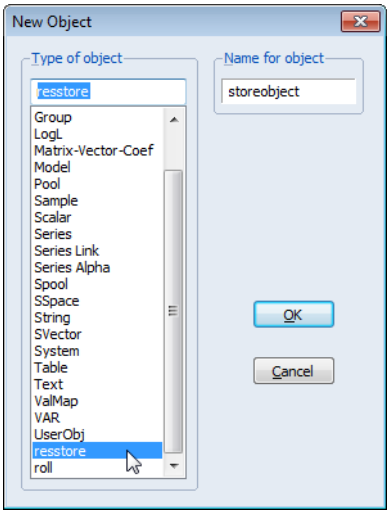
The first example uses a simple container user object (of type ResStore) that allows you to add, extract, and display graphs, tables and estimation objects. The second example performs rolling regression estimation using the Roll user object.

Simple Container Object (ResStore)

This first example uses the ResStore user object to create a storage container for objects from our workfile. This is a bare bones registered object that nonetheless shows the basic features of registered user objects.

We use the workfile "Demo.WF1" (which may be found in the example files subdirectory of your EViews installation directory) and assume that you have already installed the ResStore object class.

You may create a new ResStore object by clicking on **Object/New Object...** and then selecting **resstore** in the list of object types.

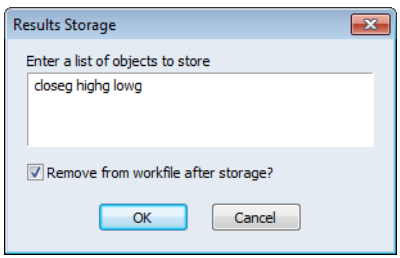


Notice that while the built-in EViews objects are listed alphabetically, the two user objects (ResStore and Roll) are simply placed at the bottom of the listbox. Select **resstore** and specify the name STOREDOBJECT for our new object. Click on **OK** to create the object. Alternately, enter the command

```
resstore storedobject
```

in the EViews command line and hit ENTER.

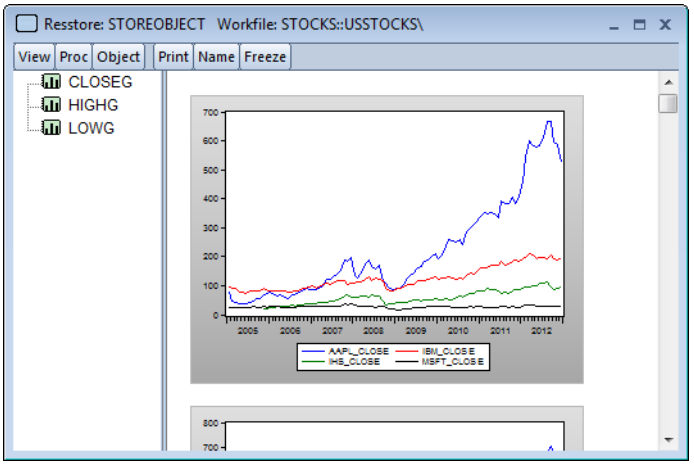
As part of its construction, the ResStore object will display a dialog asking you to enter the names of the workfile objects you would like to store:



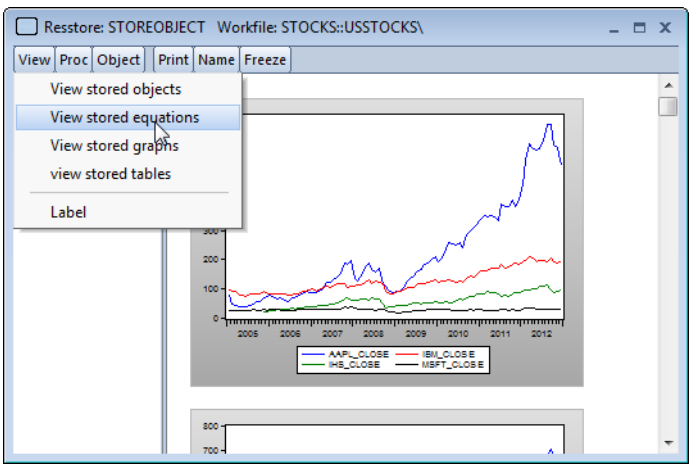
You may enter the names of any matrix objects (including scalars), strings objects, estimation objects, or view objects (graphs, tables, spools).

Note that there is a check-box that lets you specify whether to remove the objects from the workfile after storing them. You should selection this option if you wish to move, rather than copy, the specified objects into the ResStore.

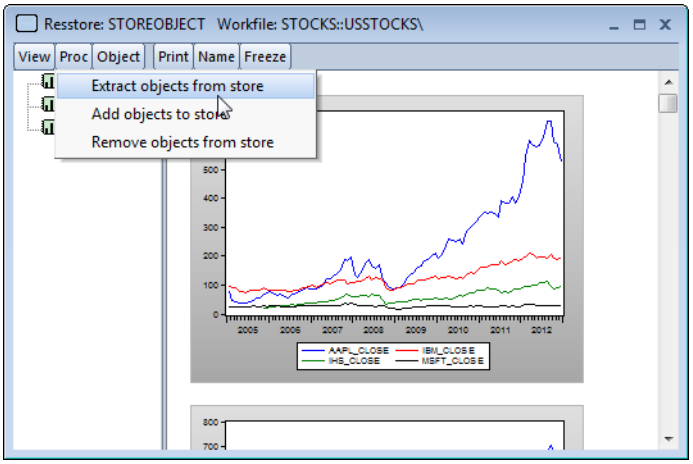
Once you hit **OK**, a new ResStore object named STOREDOBJECT will be added to your workfile. If you open up the ResStore object, a spool view display of all objects currently stored is shown:



You may use the **View** menu to access the defined views for this type of object which allow you to show various subset types of the objects in the container:



Similarly, the **Proc** menu lists procs which allow you to add, remove, and extract objects from the storage container:



As with other EViews objects, you may use the command language to work with the ResStore object. For example the defined view command,

```
storedobject.graphs
```

displays all of the graph objects in the object

```
storedobject.extractobjects
```

extracts all of the objects from STOREDOBJECT into the workfile.

The command

```
string storednames = storedobject.@members
```

saves a list of the stored object names in the string object STOREDNAMES.

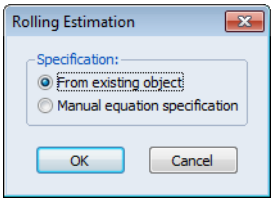
Rolling Regression Estimation Object (Roll)

Our second example uses the Roll user object to estimate rolling regressions. We again use the workfile "Demo.WF1", and we assume that you have already installed the Roll object class.

You can create a new Roll object by clicking on **Object/New Object...** and then selecting **roll** in the list of object types, or by entering the `roll` command followed by the name of a new object in the command line:

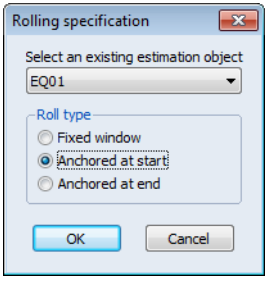
```
roll myroll
```

As part of its creation, the Roll object will display a series of dialogs prompting you to provide information on how the object should be constructed. First, you will be asked whether to create the new object using the specification from an existing equation or by specifying an equation manually:



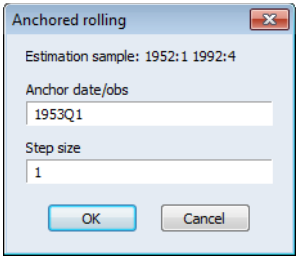
Since we will use one of the previously estimated equations in our workfile as the basis of our rolling regression, click on **OK** to accept the default.

Next, you will be asked to select your base equation and to specify the basic type of rolling regressions you wish to perform:



To obtain recursive estimates based on equation EQ01 choose **EQ01** in the drop-down menu and select **Anchored at start**. Click on **OK** to continue.

Lastly, you will be prompted to provide sample information and a step size:



Click on **OK** to create the Roll object using the specified settings.

(Note that if you had chosen **Manual equation specification** in the first dialog or **Fixed window** estimation in the second dialog, the subsequent dialogs would provide a different set of options).

EViews estimates the rolling regression and, like built-in estimation objects, displays basic estimation information in the object window:

Roll: MYROLL
Roll type: Anchored at start
Specification: EQ01
Estimation command: ROLL(AS,STEP=1,ANCHOR=1953Q1) MYROLL @ EQ01

Anchor point:	1953Q1
Number of subsamples:	160
Number of coefficients:	4
Step size:	1

Full sample estimation results:

Dependent Variable: LOG(M1)
Method: Least Squares
Date: 01/31/13 Time: 10:19
Sample (adjusted): 1952Q2 1992Q4
Included observations: 163 after adjustments

Variable	Coefficient	Std. Error	t-Statistic	Prob.
C	1.312383	0.032199	40.75850	0.0000
LOG(GDP)	0.772035	0.006537	118.1092	0.0000
RS	-0.020686	0.002516	-8.221196	0.0000
DLOG(PR)	-2.572204	0.942556	-2.728967	0.0071

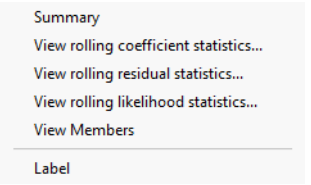
R-squared	0.993274	Mean dependent var	5.692279
Adjusted R-squared	0.993147	S.D. dependent var	0.670253
S.E. of regression	0.055485	Akaike info criterion	-2.921176
Sum squared resid	0.489494	Schwarz criterion	-2.845256
Log likelihood	242.0759	Hannan-Quinn criter.	-2.890354
F-statistic	7826.904	Durbin-Watson stat	0.140967
Prob(F-statistic)	0.000000		

These basic results may be viewed at any time by selecting **View/Summary** in the object view menu or by entering the object command

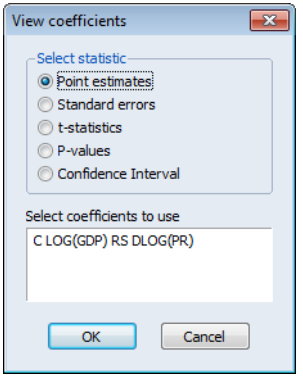
```
myroll.summary
```

in the command line.

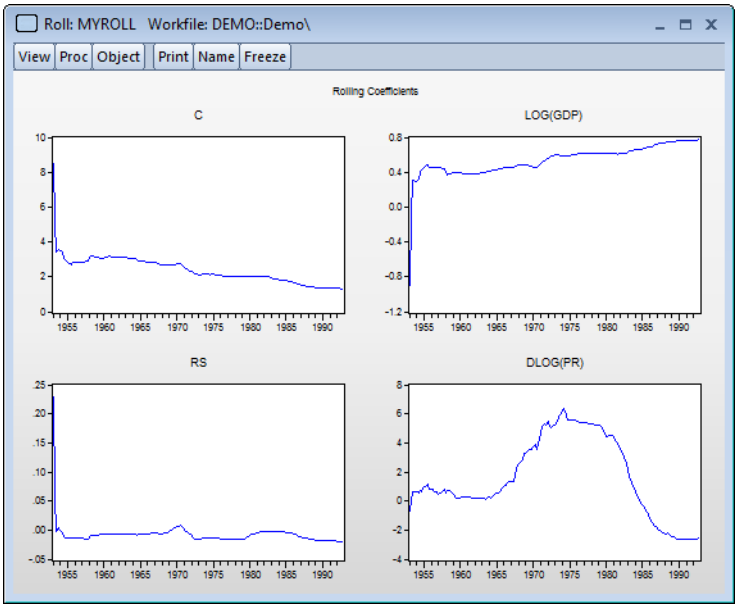
Next, consider the custom views that have been defined for this object. In addition to the summary view, you may display information on the coefficient statistics, residual statistics, likelihood statistics, members of the object, and the standard label information view for the Roll object:



Clicking on **View/View rolling coefficient statistics...** display a dialog prompting you to select the coefficients and statistics you wish to display. By default, the object will display the coefficient estimates for all of the coefficients in the regression:



Click on **OK** to accept the default values and to display a graph of the results in the object window:



Equivalently, you could have issued the command:

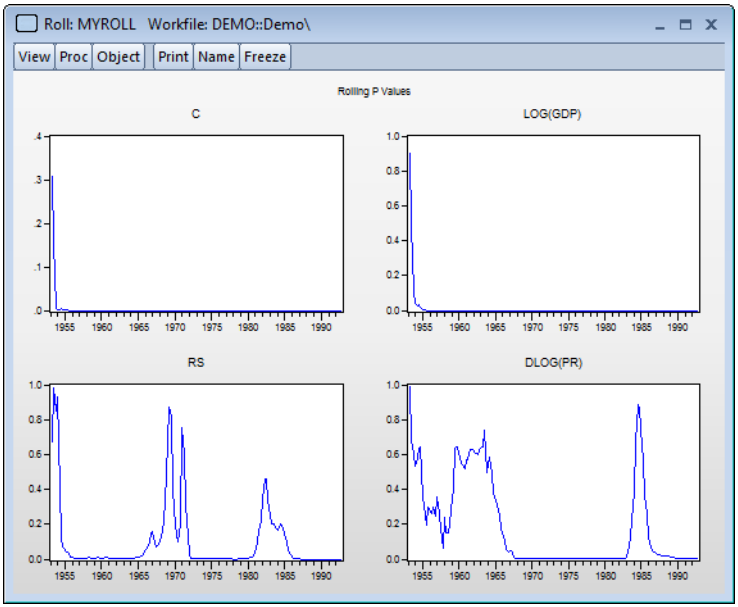
```
myroll.rollcoefs c log(gdp) rs dlog(pr)
```

in the command line.

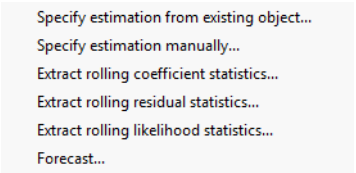
Note that you could have specified a different statistic in the dialog or add the "stat = " option to the command to display a different coefficient statistic. For example, selecting **P-values** in the dialog or entering,

```
myroll.rollcoefs(stat=pvals) c log(gdp) rs dlog(pr)
```

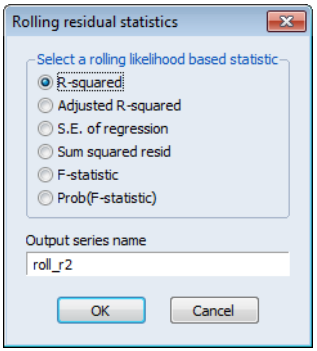
displays the coefficient t-statistic p-values:



Similarly, you may click on the Proc menu to display a list of the user defined procs:



The first two entries re-initialize the Roll object using the dialogs we first encountered when creating MYROLL. The next three menu entries extract results into the workfile. For example, clicking on **Extract rolling residual statistics...** opens a dialog prompting you to identify the results you wish to extract along with the destination:



Clicking on **OK** t saves the R^2 statistics in the workfile in the series ROLL_R2. The command

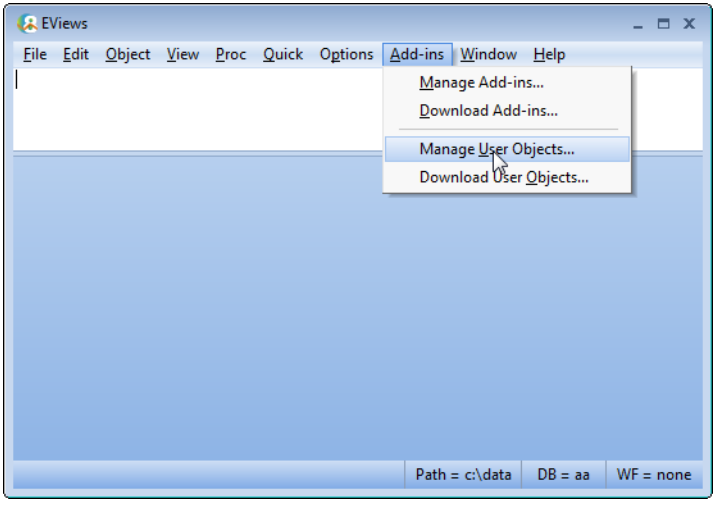
```
myroll.extractresidstat(stat=r2s) roll_r2
```

performs an equivalent operation.

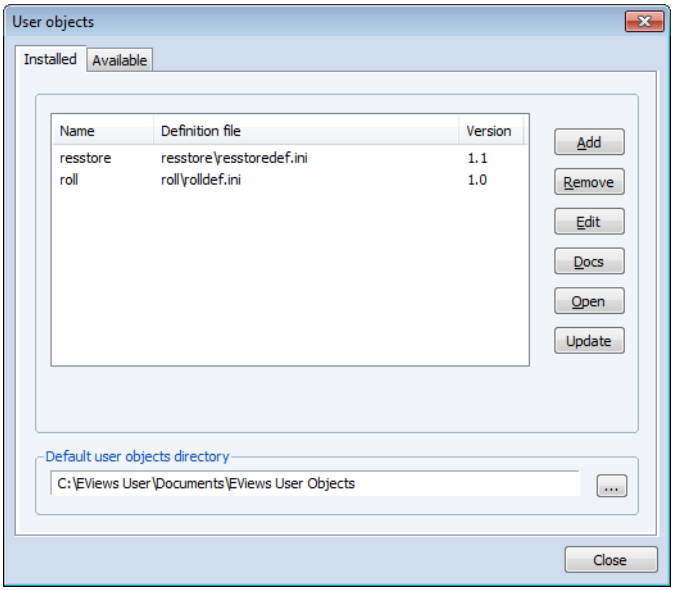
It is worth noting that behind-the-scenes in this object is a set of user programs that use standard EViews programming tools to display dialogs, perform computations, and display and extract results.

Managing User Object Classes

To manage your user object definitions, select **Add-ins/Manage User Objects...** from the main EViews menu.



EViews will display the **User objects** management dialog opened to the **Installed** tab:



The top portion of the dialog shows settings for currently registered user object classes. The **Name** column shows the name of the class, the **Definition file** column displays the location and file name of the definition ".INI" file for the class (see "Creating an Object Definition File" on page 212), and the **Version** column shows the version number of the user object class.

Note that you may click on the column headers in the list to sort the list by the contents of column.

You may use the buttons on the right-hand side of the dialog to manage your classes:

- To add a new user object to the list, simply click on the **Add** button to display the **Add/Edit User Object** dialog. The dialog settings are described in "Registering a User Object Class" on page 214.
- To delete a class, simply click on the name in the Add-ins management dialog and press the **Remove** button.
- To edit the settings of an existing user object class, select it from the list and click on the **Edit** button to display the **Add/Edit User Object** dialog. The dialog settings are described in "Registering a User Object Class" on page 214.
- To open and edit the INI definition file for the user object class, select it from the list and click on the **Open** button.

- To examine the documentation file associated with a class, click on the name and press the **Docs** button.
- To check whether the userobj class has an updated version available, and to install the update if available, click on the **Update** button. Note you may select multiple classes at the same time and click the **Update** button to update the set. You may also right click anywhere on the list of user object classes and select **Update All** to update all of your classes.

Default User Objects Directory

The bottom portion of the **Installed** tab shows the default user objects directory. The default directory is where user objects will be installed and where the user object programs will search for supplementary files if explicit directory locations are not provided. To change the default directory, click on the button on the right and navigate to the desired directory, or simply enter the desired folder name. Click on **OK** to accept the settings.

Defining a Registered User Object Class

To define a registered user object class, you must provide information on how to *construct* (create) an instance of the object. While the constructor information is all that is required, you may optionally specify menu items and custom command names for views and procs that will execute EViews programs.

We may divide the registration procedure into two distinct steps:

- Create an object definition file which includes constructor, view, and proc definitions.
- Register the object definition file with EViews.

This discussion assumes that you have already written programs to initialize your object and possibly to display views and execute procs. These programs will be standard EViews programs that use ordinary EViews commands. There are, however, several programming features which are designed specifically for user object (and Add-in) program development that you should find useful. See "Add-ins Design Support" on page 191 in the *Command and Programming Reference*.

Creating an Object Definition File

The object definition file is a simple text file with a "INI" extension. This file describes how to construct the custom object using an EViews program and optionally provides menu items and custom command names for views and procs that will execute EViews programs.

There are three sections in the file, corresponding to the constructor, the views, and the procs of the object.

The first section of the definition file should start with a line consisting of the text "[constructor]" (without the quotes). The line immediately following should contain the path and name of an EViews program file that will be used as the constructor. The constructor program file describes how the object should be initialized, or constructed, when you create a new instance using the **Object/New Object...** menu item or the command line.

The second section contains the view definition specifications. This section should start with a line consisting of the keyword "[views]". Each line following this keyword will define a view for the user object. Each view definition line should consist of the menu text, followed by a comma, a custom command name for the view, a comma, the path and name of the program file to be run when the view is selected from the menu or run from the command line.

The third section contains the proc definition specifications. It follows the same format as the views section, but begins with "[procs]" rather than "[views]".

Note that when providing the path of the programs in your definitions, the ".\" shortcut can be used to denote the folder containing the definition INI file.

For example, the following is the definition file for the ResStore user object:

```
[constructor]
".\resstore construct.prg"
[views]
"View stored objects", objects, ".\viewall.prg"
"View stored equations", equations, ".\viewequations.prg"
"View stored graphs", graphs, ".\viewgraphs.prg"
"view stored tables", tables, ".\viewtables.prg"
[procs]
"Extract objects from store", extractobjects, ".\extract
   objects.prg"
"Add objects to store", addobjects, ".\resstore construct.prg"
"Remove objects from store", dropobjects, ".\remove objects.prg"
```

The ResStore user object use a constructor program called "resstore construct.PRG" which is called when you create a new ResStore object from the dialogs or command line.

There are four view menu items, each associated with a different EViews program. The first view definition tells EViews that it should create a view menu item **View stored objects** and object command `objects`, and associate both with the EViews program "viewall.PRG" (which is located in the ResStore directory). This definition means that selecting **View/View stored objects** from the object menu, or entering the object command

```
my_object.objects
```

will run the "viewall.PRG" program which displays all of the stored objects in MY_OBJECT.

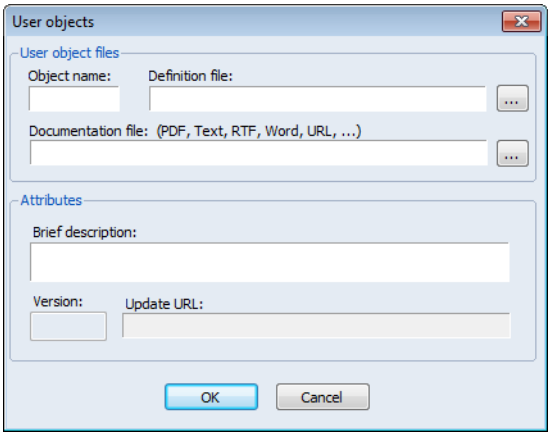
Similarly, there are three proc menu items. Selecting **Proc/Remove objects from store** or issuing the command

```
my_object.dropobjects
```

runs the "remove objects.PRG" program which displays a dialog prompting you for the name of the objects to remove from MY_OBJECT.

Registering a User Object Class

To register a new user object class, select **Add-ins/Manage User Objects...** from the main EViews menu to display the **User objects** management dialog, then click on the **Add** button to display the **Add/Edit Program** dialog.



The dialog allows you to provide a name for the user object class, to specify the INI file, attach a documentation file, and provide various object attributes:

- The **Object name** edit field should be used to specify the user object class name. Note that if you provide the name of a built-in EViews object, the EViews built-in object will take precedence.
- The **Definition file** edit field must be used to enter the name and path of the INI definition file. Note you may press the "**...**" button to navigate to and select the file.
- The **Documentation file** edit field allows you specify a PDF, Text, RTF, Microsoft Word file, or URL containing documentation for the user object class. (Note that relative paths are evaluated with respect to the directory of the INI file, not the default user object directory.)
- You may use the **Brief description** edit field to describe the purpose of the User Object.

- You may enter a **Version** number for your user object and an **Update URL** indicating where to find the update XML file (see "XML File Specification" on page 217).

You must provide an object name and a definition file to register your object. The remaining information is recommended, but not required.

Creating a User Object Package

In "Defining a Registered User Object Class" on page 212 we showed how you can take use EViews program files and an object definition file to register a new user object class for personal use.

If you wish to distribute your custom object to others, we highly recommend that you create a user object package. Packaging the user object allows you to bundle all of the files for distribution and, if you provide an installer program, offers automatic installation of the object by dragging-and-dropping the package onto EViews or double clicking on the file.

The process of creating a user object package is virtually identical to packaging of an EViews Add-in, with a few minor exceptions. We summarize briefly the main steps. Related discussion may be found in.

Creating the Self-installing Package

The process of creating a user object package is straightforward, requiring only two steps:

- (*optional*) Create a table of contents (TOC) information file and installer program file to allow for automatic registration and updating of the Add-in.
- Create a self-extracting user object package file containing the object definition file and any support files (including the TOC and installer program file, if available).

To create the self-extractive file simply create a standard ZIP archive file containing all of the files for your user object, then rename the ZIP file so that it has the extension "UOPZ". Opening such an file, automatically after completing the download, by double clicking on the file, or by dropping it onto EViews, will begin the automatic installation procedure.

The user object will not, however be automatically installed and registered unless you include a table of contents (TOC) information file and installer program along with your program files. Creating the TOC and installer program files takes only a few minutes and allows you to support automatic registration and updating of the user object. We strongly recommend that package distributors take the time to create these files as described below.

Table of Contents

Next, you should create a table-of-contents file named "Toc.INI". The TOC file should contain setup information for the user object which describes the directory in which it should

be installed, and the name of the EViews program, if any, that should be run to register the user object files. The format of the TOC file is:

```
[package]
installer = <name of installer file>
folder = <name of folder to create>
```

A TOC file should always begin with the line "[package]". The "installer" keyword is used to indicate the name of the EViews program file that should be run to register the user object. If, for example, a registration file named "Roll install.PRG" is included in your package, you should include the line

```
installer = roll install.prg
```

The "folder" keyword may be used to indicate the subfolder of the default user object directory into which you wish to extract the package files. Thus,

```
folder = Roll
```

tells EViews to extract the contents of the UOPZ file into the "Roll" folder of the User Object directory. If no folder is specified, the basename of the UOPZ file will be used as the target folder name.

Installer Program

If you wish to facilitate automatic registration of the user object class, you should create a simple EViews ".PRG" program that uses the `adduo` command to register the User Object class. For example, the rolling regression user object described in "Rolling Regression Estimation Object (Roll)" on page 204 may be registered by including the command

```
adduo(name="roll", version="1.0", desc="Rolling regression
    object") ./rolldef.ini
```

in a program file, and including a reference to this file in the package table-of-contents.

Documentation

We recommend that you provide documentation for your user object, and use the "docs = " option to point to the documentation file. Providing some documentation for the command line methods of initializing the object and accessing views and procs is especially important. Documentation could be anything from a simple text file with some syntax hints, to a lengthy PDF document that describes the user object features in detail.

Version

You may specify an version number for your user object. Version numbers allow users to use automatic updating to ensure they have the latest version of the object class definition files. When a user uses the **Update** button on the **Manage User Objects** dialog to check for

updates, EViews will compare the hosted version number with the currently registered version number and download the latest version if newer.

You may use the "version = " option to specify the version number. If omitted, EViews will assume that the user object version number is 1.0.

XML File Specification

One of the most useful user object management features is the ability of users to automatically update their installed user object as newer versions become available. To support this feature, EViews must know where to look to determine the most recent version of the user object and where to download any updates.

(Note that this specification is identical to the file specification for Add-ins and the material below is virtually identical to the discussion for Add-ins.)

This information is communicated in an XML file, typically located on the Add-ins package hosting site. If you will be hosting this file, you should use the `adduo` option "url = " to specify the URL for the XML file. If this option is not supplied, EViews will look for the XML file on EViews.com.

The XML file should contain one or more item definitions, where each item is a set of information on a specific user object. The item definition is contained in the lines between an `<item>` and `</item>` tag. The full specification of an item is as follows:

```
<item>
<title>Roll</title>
<version>1.0</version>
<description>User Object for performing rolling regression.</
    description>
<link>http://eviews.com/Addins/Roll.uopz</link>
<pubDate>14 Dec 2012</pubDate>
</item>
```

The only required specifications are the `<title>` and `<link>`.

The `<version>` is used to specify the current user object version number. When the user checks for updates, EViews will download the user object if the version number they have currently installed is lower than the one given in the `<version>` tag.

The `<link>` specification contains the URL (or network file location) of the UOPZ file containing the user object package. This is the location from which EViews will download the updated Add-in package should the user request an update.

The `<description>` and `<pubDate>` specifications should be self-explanatory.

User Object Programming Support

EViews offers several programming language features that will aid you in creating, registering and working with user objects.

Note that there is additional programming language support for creating the programs that will be used to define your user object. These features are described in "Add-ins Design Support" on page 191.

Declaration

The `userobj` command is used to create a new, unregistered user object as in

```
userobj myobject
```

where *myobject* is the name of the object to be created. A userobj created using this command will be empty, and will have no constructor or defined views and procs defined.

To declare a registered user object, you will use the name of the class followed by the name of the object:

userobj_class_name(options) myobject [args]

Depending on how the user object is designed, the declaration program may use *options* and additional arguments *args* when running the constructor program.

See `Userobj::userobj` (p. 733).

Registration

The `adduo` command may be used to register a user object, as in

```
adduo(name="roll", version="1.0", desc="Rolling regression
    object") ./rolldef.ini
```

You may use the command to specify the object definition file, path, description, version number, documentation file, XML file, etc.

See `adduo` (p. 272) for details.

View and Procs

Each type of user object will have its own views and procs:

- All user objects provide a small number of generic built-in views and procs. For example, the standard EViews `label` view for viewing and modifying the label contents and the `display` view for showing output in the object window are supported.
- All user objects support the `add` and `drop` procs which may be used to populate the user object, and the `extract` proc may be employed to extract objects into the workfile.

- In addition, registered user objects may provide custom views and procs. You should view the specific user object documentation file for details.

As with any EViews object, you may access the views and procs of the user object using the object **View** or **Proc** menu, or via the command line using the standard syntax:

myobject.view_name(options) [args]
myobject.proc_name(*options*) *[args]*

See "User Object Views," on page 726 and "User Object Procs," on page 726 for a listing of the built-in views and procs.

Data Members

The `@hasmember(obname)` function, available as a data member for all user objects, returns a boolean value depending on whether an object called *obname* currently exists inside the User Object.

The `@members` data member returns a space delimited string containing a list of all objects currently inside the user object.

See "User Object Data Members," on page 726.

Chapter 10. User-Defined Optimization

EViews offers a wide variety of built-in estimation methods that involve optimization, including (but not limited to) those supported by the Equation, System, Sspace, and VAR objects.

In addition, the EViews Logl object lets you maximize user-defined likelihood functions. While useful in a wide range of settings, the Logl object is nevertheless restricted in the types of functions that it can handle. In particular, the Logl requires that all computations be specified using series expressions, and that the log-likelihood objective can be expressed as a series containing log-likelihood contributions for each observation.

In contrast, the `optimize` (p. 379) command provides tools that allow you to find the optimal parameters or control values of a user-defined function. Notably, `optimize` supports quite general functions so that the computations and the user-defined objective need not be series-based.

Defining the Objective and Controls

To use `optimize`, you must first construct an EViews subroutine with arguments to define an output *objective* which depends on input *controls*.

Recall that a subroutine with arguments is simply a set of commands in a program that can be called one or more times within the program ("Subroutine with Arguments" on page 138). The arguments of the subroutine will correspond to the objective and to inputs that are required to calculate the objective. Each time the subroutine is called, the objective will be computed using the current values of the input controls.

The objective, which must be associated with an argument of the subroutine, may be a scalar, or may consist of many values stored in an EViews object such as a vector, matrix, or series.

The controls, which may be thought of as input parameters, are passed into the subroutine as an argument. As with the objective, the controls may be a scalar value, or a multi-valued object such as a vector, matrix, or series.

Note that when series objects are employed as either the objective or control, only the corresponding elements in the current workfile sample will be used.

`optimize` will determine the values of the controls that optimize the objective. If the objective is many-valued, EViews will optimize the sum or sum-of-squares of the values, with respect to the control elements.

Since the objective is defined using an EViews subroutine, you may optimize almost anything that may be computed using EViews commands. Notable, you may use `optimize` to optimize general functions as well as likelihoods involving matrix computations (neither of which may be optimized using the Logl object).

Consider, for example, the simple quadratic function defined as an EViews subroutine:

```
subroutine f(scalar !y, scalar !x)
    !y = 5*!x^2 - 3*!x - 2
endsub
```

This subroutine has one output and one input, the program variable scalars !X and !Y, respectively. For a given control value for !X, the subroutine computes the value of the scalar objective !Y.

In its simplest form, a subroutine designed to work with `optimize` requires only two arguments—an objective and control parameters. However you may include additional arguments, some of which may be used by optimize, while others are ignored. For example, the subroutine,

```
subroutine SqDev(series out, scalar in, series y)
    out = (y - in)^2
endsub
```

computes the squared deviations of the argument series Y from the control scalar, and places the element results in the output objective series OUT. The subroutine argument for the series Y will not be used by `optimize`, but allows optimization to be performed on arbitrary series without re-coding the subroutine.

By default, `optimize` will assume that the first subroutine argument corresponds to the objective and the second argument corresponds to the controls. As we will see, the default associations may be changed through the use of options in the `optimize` command ("The Optimize Command" on page 223).

Typically, multiple control values are passed into the subroutine in the form of a vector or matrix, as in

```
subroutine local loglike(series logl, vector beta, series dep,
    group regs)
    !pi = @acos(-1)
    series r = dep - beta(1) - beta(2)*regs(1) - beta(3)*regs(2) -
       beta(4)*regs(3)
    logl = @log((1/beta(5)*@dnorm(r/beta(5))
endsub
```

where the control vector BETA and the auxiliary arguments for the dependent variable series DEP and the regressors group REGS are used as inputs for the computation of the normal

log-likelihood contributions in the objective series LOGL. Note that the first four elements of the vector BETA correspond to the mean regression coefficients, and the last element is the parameter for the standard deviation of the error distribution.

Lastly, when designing your subroutine, you should always define the objective to return NA values for bad control values, since returning an arbitrary value may make numeric derivatives unreliable at points close to the invalid region.

The Optimize Command

The syntax for the `optimize` command is:

optimize*(options) subroutine_name(arguments)*

where *subroutine_name* is the name of the defined subroutine in your program (or included programs). The full set of options is provided in `optimize` (p. 379),

By default, EViews will assume that the first argument of the subroutine is the objective of the optimization, and that the second argument contains the controls. The default is to maximize the objective or sum of the objective values (with the sum taken over the current workfile sample, if a series).

Specifying the Method and Objective

You may control the type of optimization and which subroutine argument corresponds to the objective by providing one of the following options to the `optimize` command:

- max *[= integer]*
- min *[= integer]*
- ls *[= integer]*
- ml *[= integer]*

The four options correspond to different optimization types: maximization ("max"), minimization ("min"), least squares ("ls") and maximum likelihood ("ml"). If the objective is scalar valued only "max" and "min" are allowed.

As the names suggest, "min" and "max" correspond to minimizing and maximizing the objective. If the objective is multi-valued, `optimize` will minimize or maximize the *sum* of the elements of the objective.

"ls" and "ml" are special forms of minimization and maximization that may be specified only if the multi-valued objective argument has a value for each observation. "ls" tells `optimize` that you wish to perform least squares estimation so the optimizer should minimize the *sum-of-squares* of the elements of the objective. "ml" informs `optimize` you wish

to perform maximum likelihood estimation by maximizing the *sum* of the elements in the objective.

"ls" and "ml" differ from "min" and "max" in supporting an additional option for approximating the Hessian matrix (see "Calculating the Hessian" on page 225) that is used in the estimation algorithm. Indeed the only difference between the "max" and "ml" for a multivalued objective is that "ml" supports the use of this option ("hess = opg").

By default, the first argument of the subroutine is taken as the objective. However you may specify an alternate objective argument by providing an integer value identifier with one of the options above. For example, to identify the second argument of the subroutine as the objective in a minimization problem, you would use the option "min = 2".

Identifying the Control

By default, the second argument in the subroutine contains the controls for the optimization. You may modify this by including the "coef = *integer*" option in the `optimize` command, where *integer* is the argument identifier. For example, to identify the first argument of the subroutine as the control, you would use the option "coef = 1".

Starting Values

The values of the objects containing the control parameters at the onset of optimization are used as starting values for the optimization process. You should note that if any of the control parameters contain missing values at the onset of optimization, or if the objective function, or any analytic gradients cannot be evaluated at the initial parameter values, EViews will error and the optimization process will terminate.

Specifying Gradients

If included in the `optimize` command, the "grad = " option specifies which subroutine argument contains the analytic gradients for each of the coefficients. If you specify the "grad = " option, the subroutine should fill out the elements of the gradient argument with values of the analytical gradients at the current coefficient values.

- If the objective argument is a scalar, the gradient argument should be a vector of length equal to the number of elements in the coefficient argument.
- If the objective argument is a series, the gradient argument should be a group object containing one series per element of the coefficient argument. The series observations should contain the corresponding derivatives for each observation in the current workfile sample.
- For a vector objective, the gradient argument should be a matrix with number of rows equal to the length of the objective vector, and columns equal to the number of elements in the coefficient argument.

- "grad = " may not be specified if the objective is a matrix.

If "grad = " is not specified, `optimize` will use numeric gradients. In general, we have found that using numerical gradients performs as well as analytic gradients. Since programming the calculation of the analytic gradients into the subroutine can be complicated, omitting the "grad = " option should usually be one's initial approach.

Calculating the Hessian

The "hess = " option tells EViews which Hessian approximation should be used in the estimation algorithm. You may employ numeric Hessian ("hess = numeric"), Broyden-Fletcher-Goldfarb-Shanno ("hess = bfgs"), or outer-product of the gradients ("hess = opg") approximations to the Hessian (see "Hessian Approximation" on page 233).

You may not specify an analytic Hessian, though all three approximations use information from the gradients, so that there will be slight differences in the Hessian calculation depending on whether you use numeric versus analytical gradients.

The "finalh = " option allows you to save the Hessian matrix of the optimization problem at the final coefficient values as a matrix in the workfile. For least squares and maximum likelihood problems, the Hessian is commonly used in the calculation of coefficient covariances.

For OPG and numeric Hessian approximations, the final Hessian will be the same as the Hessian approximation used during optimization. For BFGS, the final Hessian will be based on the numeric Hessian, since the BFGS approximation need not converge to the true Hessian.

Numeric Derivatives

You can control the method of computing numeric derivatives for gradient or Hessian calculations using the "deriv = " option.

At the default setting of "deriv = auto", EViews will change the number of numeric derivative evaluation points as the optimization routine progresses, switching to a larger number of points as it approaches the optimum.

When you include the "deriv = high" option, EViews will always evaluate the objective function at a larger number of points.

Iteration and Convergence

The "m = " and "c = " options set the maximum number of iterations, and the convergence criterion respectively. Note that for optimization, the number of iterations is the number of successful steps that take place, and that each iteration may involve many function evaluations, both to evaluate any required numeric derivatives and for backtracking in cases where a trial step fails to improve the objective.

Reaching the maximum number of iterations will cause an error to occur (unless the "noerr" option is set).

Advanced Optimization Options

There are several advanced options which control different aspects of the optimization procedure. In general, you should not need to worry about these settings, but they may prove useful in cases where you are experiencing estimation difficulties.

Trust Region

You may use the "trust = " option to set the initial trust region size as a proportion of the initial control values. The default trust region size is 0.25.

Smaller values of this parameter may be used to provide a more cautious start to the optimization in cases where larger steps immediately lead into an undesirable region of the objective.

Larger values may be used to reduce the iteration count in cases where the objective is well behaved but the initial values may be far from the optimum values.

See "Technical Details," on page 232 for discussion.

Step Method

`optimize` offers several methods for determining the constrained step size which you may specify using the "step = " option. In additional to the default Marquardt method ("step = marquardt"), you may specify dogleg steps ("step = dogleg") or a line-search determined step ("step = linesearch").

Note that in most cases the choice of step method is less important than the selection of Hessian approximation. See "Step Method," on page 235 for additional detail.

Scale

By default, the optimization procedure automatically adjusts the scale of the objective and control variables using the square root of the maximum observed value of the second derivative (curvature) of each control parameter. Scaling may be switched off using the "scale = none" option. See "Scaling," on page 236 for discussion.

Objective Accuracy

The "feps = " option may be used to specify the expected relative accuracy of the objective function. The default value is 2.2e-16.

The value indicates what fraction of the observed objective value should be considered to be random noise. You may wish to increase the "feps = " value if the calculation of your objective may be relatively inaccurate.

Status Functions

To support the `optimize` command, EViews provides three functions that return information about the optimization process:

- `@optstatus` provides a status code for the optimizer, both during and post-optimization.
- `@optiter` returns the current number of iterations performed. If called post-optimization, it will return the number of iterations required for convergence.
- `@optmessage` returns a one line text message based on status and iteration information that summarizes the current state of an optimization.

All three of these functions may be used during optimization by including them inside the optimization subroutine, or post-optimization by calling them after the `optimize` command.

Error Handling

The "noerr" option may be used as an option to suppress any error messages created when the optimization fails. By default, the optimization procedure will generate an error whenever the results of the optimization appear to be unreliable, such as if convergence was not met, or the gradients are non-zero at the final solution.

If noerr is specified, these errors will be suppressed. In this case, your EViews program may still test whether the optimization succeeded using the `@optiter` function. Note that the noerr option is useful in cases where you are deliberately stopping optimization early using the m= maximum iterations option, since otherwise this will generate an error.

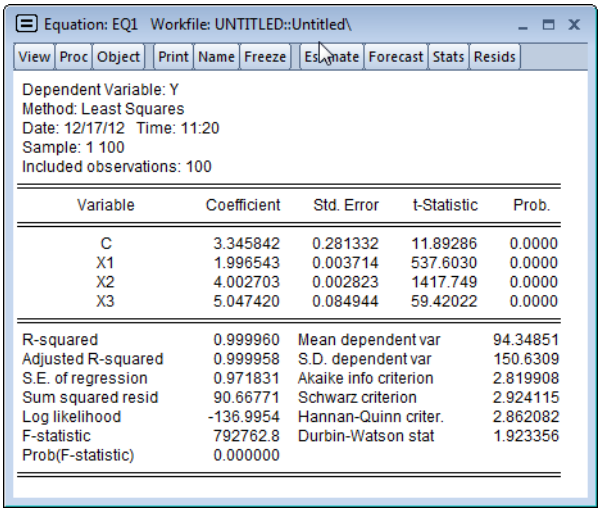
Examples

We demonstrate the use of the `optimize` command with several examples. To begin, we consider a regression problem using a workfile created with the following set of commands:

```
wfcreate u 100
rndseed 1
series e = nrnd
series x1 = 100*rnd
series x2 = 30*nrnd
series x3 = -4*rnd
group xs x1 x2 x3
series y = 3 + 2*x1 + 4*x2 + 5*x3 + e
equation eq1.ls y c x1 x2 x3
```

These commands create a workfile with 100 observations, and then generate some random data for series X1, X2 and X3, and E (where E is drawn from the standard normal distribution). The series Y is created as 3 + 2*X1 + 4*X2 + 5*X3 + E.

To establish a baseline set of results for comparison, we regress Y against a constant, X1, X2, and X3 using the built-in least squares method of the EViews equation object. The results view for the resulting equation EQ1 contains the regression output:



Equation: EQ1 Workfile: UNTITLED::Untitled\

View | Proc | Object | Print | Name | Freeze | Estimate | Forecast | Stats | Resids

Dependent Variable: Y
Method: Least Squares
Date: 12/17/12 Time: 11:20
Sample: 1 100
Included observations: 100

Variable	Coefficient	Std. Error	t-Statistic	Prob.
C	3.345842	0.281332	11.89286	0.0000
X1	1.996543	0.003714	537.6030	0.0000
X2	4.002703	0.002823	1417.749	0.0000
X3	5.047420	0.084944	59.42022	0.0000

R-squared	0.999960	Mean dependent var	94.34851
Adjusted R-squared	0.999958	S.D. dependent var	150.6309
S.E. of regression	0.971831	Akaike info criterion	2.819908
Sum squared resid	90.66771	Schwarz criterion	2.924115
Log likelihood	-136.9954	Hannan-Quinn criter.	2.862082
F-statistic	792762.8	Durbin-Watson stat	1.923356
Prob(F-statistic)	0.000000		

Next we use the `optimize` command with the least squares method to estimate the coefficients in the regression problem. Running a program with the following commands produces the same results as the built-in regression estimator:

```
subroutine leastsquares(series r, vector beta, series dep, group
    regs)
    r = dep - beta(1) - beta(2)*regs(1) - beta(3)*regs(2) -
       beta(4)*regs(3)
endsub

series LSresid
vector(4) LSCoefs
lscoefs = 1
optimize(ls=1, finalh=lshess) leastsquares(LSresid, lscoefs, y, xs)
scalar sig = @sqrt(@sumsq(LSresid)/(@obs(LSresid)-@rows(LSCoefs)))
vector LSSE = @sqrt(@getmaindiagonal(2*sig^2*@inverse(lshess)))
```

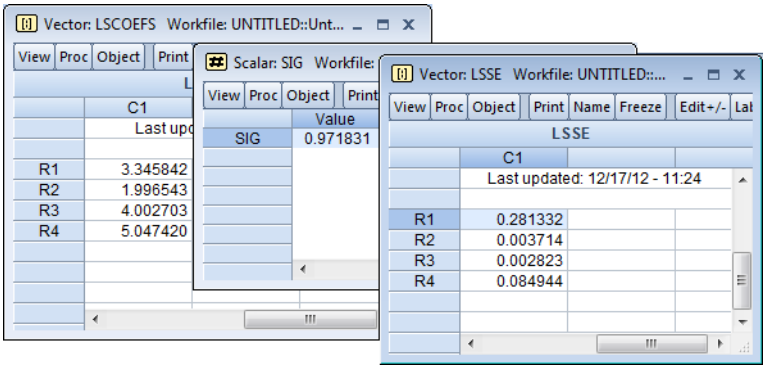
We begin by defining the LEASTSQUARES subroutine which computes the regression residual series R, using the parameters given by the vector BETA, the dependent variable given by the series DEP, and the regressors provided by the group REGS. All of these objects are arguments of the subroutine which are passed in when the subroutine is called.

Next, we declare the LSRESID series and a vector of coefficients, LSCOEFS, which we arbitrarily initialize at a value of 1 as starting values.

The `optimize` command is called with the "ls" option to indicate that we wish to perform a least squares optimization. The "finalh" option is included so that we save the estimated Hessian matrix in the workfile for use in computing standard errors of the estimates. `optimize` will find the values of LSCOEFS that minimize the sum of squared values of LSRESID as computed using the LEASTSQUARES subroutine.

Once optimization is complete, LSCOEFS contains the point estimates of the coefficients. For least squares regression, the standard error of the regression s is calculated as the square root of the sum of squares of the residuals, divided by $T - k$. We store s in the scalar SIG. Standard errors may be calculated from the Hessian as the square root of the diagonal of $2s^2 H^{-1}$. We store these values in the vector LSSE.

The coefficients in LSCOEFS, standard error of the regression s in SIG, and coefficient standard errors in LSSE, all match the results in EQ1.



Alternately, we may use `optimize` to estimate the maximum likelihood estimates of the regression model coefficients. Under standard assumptions, an observation-based contribution to the log-likelihood for a regression with normal error terms is of the form:

$$L_t = \log\left(\frac{1}{\sigma\sqrt{2\pi}}\exp\left(\frac{-e_t^2}{2\sigma^2}\right)\right) = \log\left(\frac{1}{\sigma}\phi\left(\frac{e_t}{\sigma}\right)\right) \tag{10.1}$$

The following code obtains the maximum likelihood estimates for this model:

```
subroutine loglike(series logl, vector beta, series dep, group regs)
   series r = dep - beta(1) - beta(2)*regs(1) - beta(3)*regs(2) -
      beta(4)*regs(3)
   logl = @log((1/beta(5))*@dnorm(r/beta(5)))
endsub

series LL
vector(5) MLCoefs
MLCoefs = 1
```

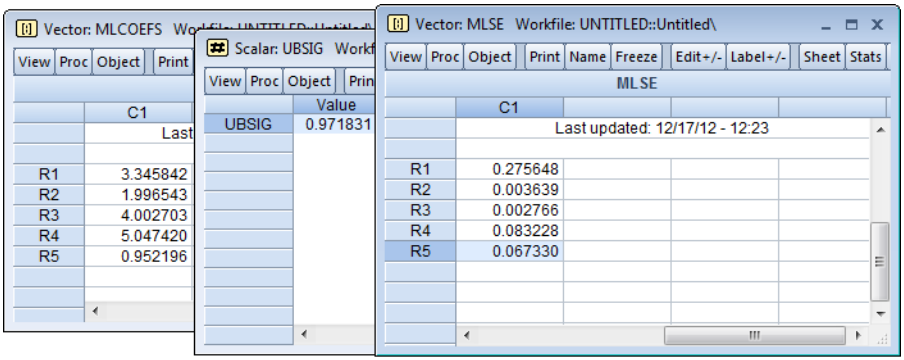
```
MLCoefs(5) = 100
optimize(ml=1, finalh=mlhess, hess=numeric) loglike(LL, MLCoefs, y,
    xs)
vector MLSE = @sqrt(@getmaindiagonal(-@inverse(mlhess)))
scalar ubsig = mlcoefs(5)*@sqrt(@obs(LL)/(@obs(LL) - @rows(MLCoefs)
    + 1))
%status = @optmessage
statusline {%status}
```

The subroutine LOGLIKE computes the regression residuals using the coefficients in the vector BETA, the dependent variable series given by DEP, and the regressors in the group REGS. Given R, the subroutine evaluates the individual log-likelihood contributions and puts the results in the argument series LOGL.

The next lines declare the series LL to hold the likelihood contributions and the coefficient vector BETA to hold the controls. Note that the coefficient vector, BETA, has five elements instead of the four used in least-squares optimization, since we are simultaneously estimating the four regression coefficients and the error standard deviation σ. We arbitrarily initialize the regression coefficients to 1 and the distribution standard deviation to 100.

We set the maximizer to perform a maximum likelihood based estimation using the "ml = " option and to store the OPG Hessian in the workfile in the sym objected MLHESS. The coefficient standard errors for the maximum likelihood estimates may be calculated as the square root of the main diagonal of the negative of the inverse of MLHESS. We store the estimated standard errors in the vector MLSE.

Although the regression coefficient estimates match those in the baseline, the ML estimate of σ in the fifth element of BETA differs. You may obtain the corresponding unbiased estimate of sigma by multiplying the ML estimate by multiplying BETA(5) by $\sqrt{T/(T-k)}$, which we calculate and store in the scalar UBSIG.



Note also that we use `@optmessage` to obtain the status of estimation, whether convergence was achieved and if so, how many iterations were required. The status is reported on the statusline after the `optimize` estimation is completed.

The next example we provide shows the use of the "grads = " option. This example re-calculates the least-squares example above, but provides analytic gradients inside the subroutine. Note that for a linear least squares problem, the derivatives of the objective with respect to the coefficients are the regressors themselves (and a series of ones for the constant):

```
subroutine leastsquareswithgrads(series r, vector beta, group grads,
   series dep, group regs)
   r = dep - beta(1) - beta(2)*regs(1) - beta(3)*regs(2) -
      beta(4)*regs(3)
   grads(1) = 1
   grads(2) = regs(1)
   grads(3) = regs(2)
   grads(4) = regs(3)
endsub

series LSresid
vector(4) LSCoefs
lscoefs = 1
series grads1
series grads2
series grads3
series grads4
group grads grads1 grads2 grads3 grads4
optimize(ls=1, grads=3) leastsquareswithgrads(LSresid, lscoefs,
   grads, y, xs)
```

Note that the series for the gradients, and the group containing those series, were declared prior to calling the `optimize` command, and that the subroutine fills in the values of the series inside the gradient group.

Up to this point, our examples have involved the evaluation of series expressions. The optimizer does, however, work with other EViews commands. We could, for example, compute the least squares estimates using the optimizer to "solve" the normal equation $(X'X)\beta = X'Y$ for β. While the optimizer is not a solver, we can trick it into solving that equation by creating a vector of residuals equal to $(X'X)\beta - X'Y$, and asking the optimizer to find the values of β that minimize the square of those residuals:

```
subroutine local matrixsolve(vector rvec, vector beta, series dep,
   group regs)
   stom(regs, xmat)
   xmat = @hcat(@ones(100), xmat)
   stom(dep, yvec)
   rvec = @transpose(xmat)*xmat*beta - @transpose(xmat)*yvec
```

```
    rvec = @epow(rvec,2)
endsub
vector(4) MSCoefs
MSCoefs = 1
vector(4) rvec
optimize(min=1) matrixsolve(rvec, mscoefs, y, xs)
```

Since we will be using matrix manipulation for the objective function, the first few lines of the subroutine convert the input dependent variable series and regressor group into matrices. Note that the regressor group does not contain a constant term upon input, so we append a column of ones to the regression matrix XMAT, using the `@hcat` command.

Lastly, we use the `optimize` command to find the minimum of a simply function of a single variable. We define a subroutine containing the quadratic form, and use the optimize command to find the value that minimizes the function:

```
subroutine f(scalar !y, scalar !x)
    !y = 5*!x^2 - 3*!x - 2
endsub
create u 1
scalar in = 0
scalar out = 0
optimize(min) f(out, in)
```

This example first creates an empty workfile and declares two scalar objects, IN and OUT, for use by the optimizer. IN will be used as the parameter for optimization, and is given an arbitrary starting value of 0. The subroutine F calculates the simple quadratic formula:

$$Y = 5X^2 - 3X - 2 \tag{10.2}$$

After running this program the value of IN will be 0.3, and the final value of OUT (evaluated at the optimal IN value) is -2.45. As a check we can manually calculate the minimal value of the function by taking derivatives with respect to X, setting equal to zero, and solving for X:

$$\begin{aligned} \frac{dY}{dX} &= 10X - 3 \\ X &= 0.3 \end{aligned} \tag{10.3}$$

Technical Details

The optimization procedure uses a Newton (or quasi-Newton) based approach to optimization. In this approach, the first and second derivatives of the objective are used to form a local quadratic approximation to the objective function around the current value of the control parameters. The procedure then calculates the change in the control values that would maximize (or minimize) the objective if the objective function were to exactly follow the local approximation.

Mathematically, if the local approximation of the objective f around the control values $x = x^*$ is:

$$\min f(p) = f(x^*) + g(x^*)'p + \frac{1}{2}p'H(x^*)p \tag{10.4}$$

where $f$is the objective function, g is the gradient, and H is the Hessian, then the first-order conditions for a maximum give the following expression for the Newton step:

$$p = -H(x^*)^{-1}g(x^*) \tag{10.5}$$

Note that this local approximation may become quite inaccurate as we move away from the current parameter values. At the full Newton step, the objective may improve by much less than the approximation suggests, or may even worsen. To deal with this possibility, the optimization procedure uses a trust region approach (More and Sorensen, 1983). In the trust region approach, the local quadratic approximation is only maximized within a limited neighborhood of the current control values, so that the change in control values at each step is not allowed to exceed a current maximum step size. We then evaluate the objective at the new proposed parameter values. If the local approximation appears to be accurate, the maximum allowed step size is increased. If the local approximation appears to be inaccurate, the maximum allowed step size is decreased. A step is only accepted when it results in a sufficiently large reduction in the objective relative to the reduction that was predicted by the local approximation.

Mathematically the constrained step can be written as:

$$\min f(p) \quad \text{s.t. } \|p\| \leq \delta \tag{10.6}$$

where δ is the trust region maximum step size. In the case where the maximum step constraint is binding, typically the step has a solution

$$p = -(H(x^*) + \lambda I)^{-1}g(x^*) \tag{10.7}$$

where λ is chosen so that $|\|p\| = \delta$.

Note that the Newton approach will work best when the objective can be fitted reasonably well by a local quadratic approximation. This will not be the case if the function is discontinuous or has discontinuous first or second derivatives. In these cases, the procedure may be slow to find an optimum, and the final parameter values may end up adjacent to a discontinuity so that the results will need to be interpreted with caution.

Hessian Approximation

In the discussion above we assumed that the Hessian matrix of second derivatives of the objective with respect to the control parameters are readily available. In practice these derivatives will need to be approximated. The `optimize` procedure provides three different methods: numeric Hessian, Broyden-Fletcher-Goldfarb-Shanno (BFGS), outer-product of the gradients (OPG).

Numeric Hessian

The numeric Hessian approach approximates the Hessian using numeric derivatives. If analytic gradients are provided, the Hessian is based on taking numeric first derivatives of the analytic gradients. If analytic gradients are not provided, the Hessian is based on numeric second derivatives of the objective function.

You may specify the use of numeric Hessians by including the option "hess = numeric" option in the `optimize` command.

Note that calculating numeric second derivatives may require many evaluations of the objective function. In the case of numeric second derivatives, each Hessian approximation will require additional evaluations proportional to the square of the number of control parameters in the problem. For a large number of control parameters, this method may be quite slow.

Broyden-Fletcher-Goldfarb-Shanno (BFGS)

The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method approximates the Hessian using an updating scheme where the previous iteration's approximation to the Hessian is adjusted after each step based on the observed change in the gradients.

The BFGS update makes as small a change as possible to the existing Hessian approximation so that it is compatible with the observed change in gradients, while ensuring that the approximation to the Hessian remains positive definite. (See Chapter 9 of Dennis and Schnabel (1983) for a detailed discussion.)

To specify the BFGS method, use the `optimize` command with the "hess = bfgs" option.

BFGS requires fewer objective function evaluations per step than computing a numeric Hessian, but may take more iterations to converge. Note that the BFGS approximation need not converge to the true Hessian at the optimized control parameter values, so it cannot be used for calculating the coefficient covariances in statistical problems. Note also that the iterations are started from a diagonal approximation to the Hessian.

Outer-product of Gradients (OPG)

For certain statistical problems, the Hessian can be approximated by a multiple of the sum of the outer products of the gradients (OPG) of individual contributions to the total objective with respect to the coefficients. In the case of least squares problems, this method is commonly referred to as the Gauss-Newton method. In maximum likelihood settings, this method is often referred to as the BHHH (Berndt, Hall, Hall, and Hausman, 1974) method.

In both settings, the approximations are based on the statistical idea that the expected value of the Hessian at the optimized parameter values is equal to a multiple of the expected value of the sum of the outer product of gradients and that the two will converge as the sample

size becomes large. The asymptotic equivalence implies that these OPG approximations will be closer to the true Hessian when working with medium to large sample sizes and when coefficients are close to the true coefficient values.

You may select the OPG approximation using the "hess = opg" option.

Note that the OPG method may only be used when the objective is a set of least squares residuals (specified using the "ls" option) or a set of maximum likelihood contributions (specified using the "ml" option), since there is no reason to believe the approximation is valid for an arbitrary maximization or minimization objective.

OPG uses the same number of objective evaluations per step as BFGS, which is less than the number required for evaluating the numeric Hessian.

Step Method

Different step methods are supported by `optimize`, with each following a trust region approach, where the full Newton step is taken whenever the step is less than the current maximum step size, and a constrained step is taken when the full Newton step exceeds the current maximum step size. The methods differ in how the constrained step is taken. Note that in most cases, the choice of step method is less important than the selection of Hessian approximation.

Marquardt

The default Marquardt option closely follows the method outlined above where the constrained step is calculated by an iterative procedure that searches for a diagonal adjustment to the Hessian that makes the step size equal to the maximum allowed step size. The Marquardt step has the highest computational cost, although since for most statistical estimation most computation time is spent evaluating the objective rather than calculating an optimal step, this is unlikely to matter unless the number of controls is fairly large and the objective can be evaluated cheaply.

Dogleg

The dogleg method is a cheaper approximation to the trust region problem where the constrained step is calculated by combining a Newton step with a Cauchy step (a step in the direction of the scaled gradients that minimizes the local quadratic approximation to the objective). For both the Marquardt and dogleg steps, the direction of the step shifts away from the direction of the Newton step towards the direction of steepest descent as the trust region contracts, but the dogleg step uses a simple linear combination of the two steps to achieve this. When the dogleg step is used with a BFGS Hessian (the hess = bfgs option) approximation, the calculations required per iteration are proportional to the square rather than the cube of the number of parameters. This makes the dogleg step attractive if the number of control variables is very large and the objective can be evaluated cheaply.

Line-search

The line-search method is the simplest approach in which the constrained step is formed by proportionally scaling down the Newton step until it satisfies the maximum step size constraint. With this method, only the length of the step is changed as the trust region contracts, but not its direction. The line-search method is the cheapest method in terms of calculational cost but may be less robust, particularly when used with poor initial values.

Note that for both the dogleg and line-search algorithms, an adjustment will be made to the diagonal of the Hessian to ensure positive definiteness before calculating the Newton step. There is also special handling for non-positive definite matrices in the Marquardt step following the method outlined in More and Sorensen (1983).

Scaling

The Newton step is theoretically invariant to both the scale of the objective and the scale of the control variables since any changes to the gradients and the Hessian cancel each other out in the expression for the Newton step. In practice, numerical issues may cause the equivalence to be inexact. Additionally, the constrained trust region steps do not have the invariance property unless scaling is applied to the control variables when calculating a constrained step.

By default, the optimization procedure scales automatically using the square root of the maximum observed value of the second derivative (curvature) of each control parameter. This makes the procedure theoretically invariant to the scaling of the variables.

In most cases you should leave the default scaling turned on, but in cases where the Hessian approximation may be unreliable, scaling may be switched off using the "scale = none" option. When scaling is switched off, you may wish to define your objective so that equal size changes to each control variable will have a similar order of magnitude of impact on the objective.

Optimization Termination

The optimization process will terminate immediately if the initial control parameters contain missing values, the objective function, or if provided, the analytical gradients cannot be evaluated at the starting parameter values.

Once the optimization procedure begins, it will proceed even if numerical errors (such as taking the log of a negative number) prevent the objective function from being evaluated at a trial step. An objective with missing values will be taken as indicating that the control values are invalid, and the optimization will step back from the problematic values.

Note that you should always define the objective to return NA values for bad control values since returning an arbitrary value may make numeric derivatives unreliable at points close to the invalid region.

The optimization procedure will terminate when:

- An unconstrained Newton step improved the objective and the length of the step was less than the specified convergence tolerance.
- A constrained step failed to improve the objective and the maximum allowed step size for the next iteration was decreased to become less than the specified convergence tolerance.
- The maximum number of iterations (successful steps) was reached without one of the above criteria being met.

When the procedure terminates for a condition other than the maximum iterations being reached, the procedure checks the gradients and curvature of the objective to see whether the first and second order conditions for an optimum appear to be satisfied. If the conditions are not met, the optimization will be considered to have failed. There are a variety of reasons that failure may occur:

- The objective may have no optimum value, but just gradually flatten out as a control variable becomes very large or small.
- The objective may not be defined for some values of the control parameters but may improve as we approach these values. This will cause the optimization to stall with control variables very close to the invalid region, but with non-zero gradients at the final control values.
- There may be values for some controls which make other controls included in the optimization have little or no impact on the objective, so that both the gradients and the elements of the Hessian corresponding to the variables gradually become zero as the optimization progresses.
- The control variables may 'collapse' so that two or more controls are serving the same role in the objective and their individual effect cannot be separated. This will result in a Hessian that is numerically singular since changes in one control can be exactly offset by changes in one or more of the other controls without changing the objective. (For statistical problems, this implies that the coefficients are unidentified).

In all these cases, a useful approach is to carefully consider starting values so that the initial values for the controls are as close as possible to what you believe the optimum values might be. You should also avoid starting values that are close to any regions in which the objective function cannot be evaluated. If the optimization continues to report problems from a wide range of starting values, this may indicate that your optimization problem is not well defined.

Successful convergence does not guarantee that the optimization procedure has found the global optimum of the function. The optimization procedure only tests whether the final point appears to satisfy the conditions necessary for a local optimum. In cases where more

than one local optimum may exist, the optimization procedure may converge to different final values depending on what starting values are used.

Note that when the optimization completes successfully (no error is reported) the last call to the subroutine that calculates the objective will always be with the control parameters set to the optimized values. (An additional final call to the subroutine will be made in situations where this is not already the case). This guarantees that any intermediate results saved inside the subroutine will also be left at their optimized results after the optimization is complete.

References

Berndt, E., Hall, B., Hall, R., and Hausman, J. (1974). *Estimation and Inference in Nonlinear Structural Models*, Annals of Economic and Social Measurement, Vol. 3, 653–665.

Dennis, J. E. and R. B. Schnabel (1983). "Secant Methods for Systems of Nonlinear Equations," *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, London.

More and Sorensen (1983). *Computing a Trust Region Step,* SIAM Journal of Scientific Statistical Computing, Vol. 4, 553–572.

Chapter 11. 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 may be created and manipulated using the matrix command language:

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

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

Declaring Matrix Objects

You must declare a matrix object for it to exist in the workfile. A listing of the declaration statements for the various matrix objects is provided in "Object Creation Commands" on page 263.

Briefly, a matrix object declaration consists of the object *keyword*, along with 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.

One common operation, creating and filling a vector in one-step, may be performed using the `@fill` (p. 623) function, as in

```
vector v = @fill(1,4,6,21.3)
```

which returns a 4 element vector, where the first element is set to 1, the second to 4, the third to 6 and the fourth to 21.3.

To change the size of a matrix object, you may 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

$$A = \begin{bmatrix} 1 & 0 \\ 4 & 0 \end{bmatrix}. \tag{11.1}$$

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 (see, for example, the matrix object version of the procedure, `Matrix::fill`).

Fill assignment

The second assignment method is to use the `fill` object 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 to fill by rows.

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:

$$V = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \end{bmatrix}, \qquad X = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \end{bmatrix} \tag{11.2}$$

If we replace the last line with

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

then X is given by:

$$X = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}. \tag{11.3}$$

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:

$$Y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \tag{11.4}$$

See `Matrix::fill` for a complete description of the fill procedure for a matrix. Equivalent procedures are available for the remaining matrix objects.

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 rules for how EViews performs the assignment which vary depending 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. In the second line, YDATA is automatically resized to be a 10×2 matrix and is filled with the contents of ZDATA.

The third line declares and initializes XDATA. 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 (see also "Copying Data From Matrix Objects" on page 244).

The matrix object on the right hand side of the declaration statement may also be the output from a matrix function or expression. For example,

```
sym eye4 = @identity(4)
```

declares the symmetric matrix EYE4 which is equal to the 4×4 identity matrix, while

```
vector b = @inverse(xx)*xy
```

inverts the matrix XX, multiplies it by XY, and assigns the value to the new vector B.

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 assignment statements. 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:

- The object type of the target Y cannot 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 260 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 ill 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 directly the 4 elements of the matrix X to the vector Y. Other examples of invalid assignment statements involve assigning matrix objects to scalars or sym objects to vector objects.

(In may be possible, however, to use the `@vec` (p. 647) or `@vech` (p. 647) functions to perform some of these operations.)

Copying Data From Parts Of Matrix Objects

In addition to the standard rules for conversion of data between objects, EViews provides matrix functions for extracting from and assigning to 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 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 "Matrix Command and Function Summary" on page 605. Here,

we consider a few examples that should provide you with a sense of the types 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 column of M1 while V3 will be a 10 element vector containing the fourth row of M1. The `@vec` (p. 647), `@vech` (p. 647), `@rowextract` (p. 639), and `@columnextract` (p. 614) functions also work with sym objects. V4 is a 10 element vector containing the fifth column of SYM1.

In some cases, it may be easier to take a subset of the elements of a matrix object using its data member functions. A subset of rows or columns of a matrix may be obtained using the `@row`, `@col`, `@droprow`, or `@dropcol` object data members. For example,

```
vector a = x.@row(3)
```

extracts the third column of X into the vector a. Similarly,

```
vector b = x.@col(2)
```

puts the second column of X into B.

You can also copy data from one matrix into a smaller matrix using `@subextract` (p. 643). 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.

You may use the data member functions to perform similar operations. For example,

```
matrix xsub = x.@row(@fill(1, 3, 4))
```

extracts the first, third, and fourth rows of the matrix X into the matrix XSIB.

Data from a matrix may be copied into another matrix object using the commands `colplace` (p. 614), `rowplace` (p. 639), and `matplace` (p. 632). 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.

You may combine `matplace` with `@fill` (p. 623) and the `@row`, `@col`, `@droprow`, or `@dropcol` data members to perform complex subsetting and filling

```
matplace(z, x.@row(@fill(1, 3, 4)), 1, 1)
```

puts rows 1, 3, and 4 of the matrix X into the upper-left hand corner of Z. Note that Z must be large enough to hold the X subset.

Copying Data Between Matrix 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 workfile-based 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-hand side and a vector on the left, EViews will only use observations from the current sample to make the vector. If you place a group object on the right and a matrix on the left, EViews will create a rectangular matrix out of the group using observations from the current sample.

While direct assignment is straightforward and convenient, there are two principal limitations to the approach. First, EViews uses only the 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 dropped. Thus, if the current sample contains 20 observations, but the series or group contains missing data, the dimension of the output vector or matrix will be less than 20. (Below, we describe methods which allow you to override the current sample and to retain missing values.)

Examples:

```
smpl 1963m03 1993m06
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 1963M03 to 1993M06. Note that if GMPYQ has a missing value in 1970M01, and HSF contains a missing value in 1980M01, 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,* not the current sample. 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` (p. 615) 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:

```
smpl 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, `@convert` 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 using Commands

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` (p. 642) (*S*eries *TO M*atrix) 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` (p. 642) (*S*eries *TO M*atrix with *NA*s) 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` (p. 633) (*M*atrix *TO S*eries) 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.

Example:

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

As with `stom` 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 with mathematical operators or relations, functions, and parentheses. While we discuss matrix functions in great detail below, some examples will demonstrate the relevant concepts.

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.

(Note that element multiplication, division, inverse, and powers are not available using operators, but are instead supported via functions).

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
```

To perform element multiplication where you multiply every element of a matrix by very element of another matrix, you should use the `@emult` (p. 622) function.

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.

To perform element division where you divide every element of a matrix by very element of another matrix, you should use the `@ediv` (p. 618) function.

Relational 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 0 (FALSE) is returned; otherwise, the value 1 (TRUE) is returned.

For 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) element functions, (3) matrix algebra functions, and (4) descriptive statistics functions.

Utility Commands and Functions

The utility commands and functions provide support for creating, manipulating, and assigning values to your matrix objects. We have already seen a number of these commands and functions, including the `@convert` (p. 615) function and the `stom` (p. 642) command, both of which convert data from series and groups into vectors and matrices, as well as `@vec` (p. 647), `@vech` (p. 647), `@rowextract` (p. 639), `@columnextract` (p. 614), and `matplace` (p. 632).

A random sampling of other useful commands and functions include:

```
matrix a = @ones(10, 5)
```

which creates a 10×5 matrix of ones,

```
vector f = @getmaindiagonal(x)
```

which extracts the main diagonal from the square matrix X,

```
matrix g = @explode(sym01)
```

which creates a square matrix from the symmetric matrix object SYM01, and

```
matrix h1 = @resample(y)
matrix h2 = @permute(y)
```

which create matrices by randomly drawing (with replacement) from, and by permuting, the rows of Y.

A full listing of the matrix commands and functions is included in the matrix summary on "Matrix Command and Function Summary" on page 605.

Element Functions

EViews offers two types of functions that work with individual elements of a matrix object. First, most of the element functions that may be used in series expressions can used with

matrix objects. When used with a matrix object, these functions will return a similar object whose elements are the values of a function evaluated at each element of the original.

For example, you may specify

```
matrix f = @log(y)
```

to compute the logarithm of each element of the matrix Y.

Similarly,

```
matrix tprob = @ctdist(x, df)
```

evaluates the cumulative distribution function value for the t-distribution for each element of the matrix X and places the results in the corresponding cells of the matrix TPROB. Note that DF may either be a scalar, or a matrix object of the same type and size as X.

(See "Basic Mathematical Functions" on page 505, "Special Functions" on page 522, "Trigonometric Functions" on page 524, and "Statistical Distribution Functions" on page 525 for summaries of the various element functions.)

Second, EViews provides a set of element functions for performing element-by-element matrix multiplication, division, inversion, and exponentiation. For example, to obtain a matrix Z containing the inverse of every element in X, you may use:

```
matrix z = @einv(x)
```

Likewise, to compute the elementwise (Hadamard) product of the matrices A and B, you may use

```
matrix ab = @eprod(a, b)
```

The (i,j)-th element of the matrix AB will contain the product of the corresponding elements of A and B: $a_{ij} \cdot b_{ij}$.

See "Matrix Element Functions" on page 607 for details.

Matrix Algebra Functions

The matrix algebra functions allow you to perform common matrix algebra operations. 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.

For example, to compute the inner product of two vectors A and B, you may use

```
scalar ip = @inner(a, b)
```

To compute the Cholesky factorization of a symmetric matrix G,

```
matrix cf = @cholesky(g)
```

The least squares coefficient vector for the data matrix X and vector Y may be computed as

```
vector b= @inverse(@inner(x))*@transpose(x)*y
```

A listing of the matrix algebra functions and commands is provided in "Matrix Algebra Functions" on page 606.

Descriptive Statistics Functions

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.

For example,

```
scalar xmean = @mean(xmat)
```

computes the mean taken over all of the non-missing elements of the matrix XMAT, and assigns the values to the scalar XMEAN. Similarly, the commands

```
scalar xquant95 = @quantile(xmat, .95)
```

computes the .95 quantile of the elements in XMAT.

Functions for computing descriptive statistics are discussed in "Descriptive Statistics" on page 508.

In addition, there are functions for computing statistics for each column in a matrix.

```
vector xmeans = @cmean(xmat)
```

computes the mean for each column of XMAT and assigns the values to the vector XMEANS.

```
vector xmin = @cmin(xmat)
```

saves the column minimums in the vector XMIN. If, instead you wish to find the index of the minimum element for the column, you may use `@cimin` instead:

```
vector ximin = @cimin(xmat)
```

The column statistics are outlined in "Matrix Column Functions" on page 608.

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 with 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, and may not be used 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. 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 (*with the exception of the descriptive statistics functions)*, 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` (p. 614) 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` or the `@nas` functions.

Matrix Views and Procs

The individual object descriptions in the *Object Reference* list the various views and procs for the various matrix objects. Listings are available for matrices ("Matrix," on page 340), vectors ("Vector," on page 781), symmetric matrices ("Sym," on page 627), rowvectors ("Rowvector," on page 451), and coefs ("Coef" on page 16).

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
```

By default, EViews performs listwise deletion by column when computing correlations and covariances, 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 and other views is provided in the reference for the specific object (*e.g.*, matrix, sym) 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 matrix data; Chapter 5. "Basic Data Handling," on page 109 of *User's Guide I* offers extensive discussion. See also the descriptions for the matrix procs `Matrix::read` and `Matrix::write` (similar descriptions are available for the other matrix objects.)

Matrix Operations versus Loop Operations

You may 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
```

This approach will, however, 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.

Similarly, suppose, for example, that 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 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 faster method for subtracting column means uses the built-in operators and functions:

```
matrix x = @convert(mygrp1)
vector xmean = @cmeans(x)
x = x - @scale(@ones(@rows(x), @columns(x)),@transpose(xmean))
```

The first line converts the data in MYGRP1 into the matrix X. The second line computes the column means of X and saves the results in XMEAN. The last line subtracts the matrix of column means from X. Note that we first create a temporary matrix of ones, then use the `@scale` function to scale each column using the element in the corresponding column of the transpose of XMEAN.

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 EViews does not permit 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 is 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)	*invalid*
matrix(n,k)	matrix(p,1)	matrix(p,q)
rowvector(k)	rowvector(p)	*invalid*
scalar	*invalid*	*invalid*
sym(k)	*invalid*	sym(p) if $p = q$
vector(n)	vector(p)	*invalid*

	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	*invalid*	scalar
sym(k)	*invalid*	*invalid*
vector(n)	rowvector(q)	vector(n)

	Object type and size for source X	
Object type for Y	sym(p)	vector(p)
coef(k)	*invalid*	coef(p)
matrix(n,k)	matrix(p,p)	matrix(p,1)
rowvector(k)	*invalid*	vector(p)
scalar	*invalid*	*invalid*

sym(k)	sym(p)	*invalid*
vector(n)	*invalid*	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.

Chapter 12. Command Reference

Commands

The following list summarizes the EViews basic commands.

Commands for working with matrix objects are listed in Chapter 18. "Matrix Language Reference," on page 605, and EViews programming expressions are described in Chapter 19. "Programming Language Reference," beginning on page 649.

A list of views and procedures available for each EViews object may be found in Chapter 1. "Object View and Procedure Reference," on page 2 of the *Object Reference*.

Command Actions

do .....execute action without opening window (p. 327).
freeze.....create view object (p. 339).
print .....print view (p. 414).
show.....show object window (p. 433).

Global Commands

cd .....change default directory (p. 285).
exit .....exit the EViews program (p. 329).
output.....redirect printer output (p. 387).
param .....set parameter values (p. 413).
rndseed.....set the seed of the random number generator (p. 423).
smpl .....set the sample range (p. 436).

Object Creation Commands

alpha .....alpha series.
coef.....coefficient vector.
data .....enter data from keyboard (p. 317).
equation.....equation object.
factor .....factor analysis object.
frml .....numeric or alpha series object with a formula for auto-updating (p. 340).
genr.....numeric or alpha series object (p. 342).
graph .....graph object—create using a graph command or by merging existing graphs.
group.....group object.
link.....series or alpha link object.

logl ..... likelihood object.
matrix ..... matrix object.
model..... model object.
pool ..... pool object.
rowvector ..... rowvector object.
sample ..... sample object.
scalar ..... scalar object.
series ..... numeric series.
spool..... spool object.
sspace ..... sspace object.
string ..... string object.
svector ..... svector object.
sym..... sym object.
system..... system object.
table..... table object.
text ..... text object.
userobj ..... user object.
valmap ..... valmap object.
var ..... var estimation object.
vector..... vector object.

Object Container, Data, and File Commands

ccopy ..... copy series from DRI database (p. 284).
cfetch ..... fetch series from DRI database (p. 287).
clabel ..... display DRI series description (p. 289).
close ..... close object, program, or workfile (p. 289).
copy..... copy objects within and between workfiles, workfile pages, and databases (p. 306).
db ..... open or create a database (p. 317).
dbcopy ..... make copy of a database (p. 318).
dbcreate ..... create a new database (p. 320).
dbdelete ..... delete a database (p. 322).
dbopen..... open a database (p. 322).
dbpack ..... pack a database (p. 324).
dbrebuild..... rebuild a database (p. 325).
dbrename ..... rename a database (p. 325).
delete ..... delete objects from a workfile (p. 326).
driconvert..... convert the entire DRI database to an EViews database (p. 327).
expand ..... expand workfile range (p. 330).

read ..... import data from a foreign disk file into series (p. 418).
save ..... save workfile to disk (p. 428).
sort ..... sort the workfile (p. 439).
store..... store objects in database and databank files (p. 444).
unlink..... break links and auto-updating series (formulae) in the specified series objects (p. 458).
wfclose..... close the active workfile (p. 464).
wfcompare ..... compare the contents of the current workfile or page with the contents of a different workfile, page, or database (p. 467).
wfcreate ..... create a new workfile (p. 467).
wfdetails ..... change the details displayed in the current workfile window (p. 471).
wfopen..... open workfile or foreign source data as a workfile (p. 472).
wfrefresh..... refresh all links and auto-series in the active workfile—primarily used to refresh links that use external database data (p. 485).
wfsave..... save workfile to disk as a workfile or a foreign data source (p. 485).
wfselect..... change active workfile page (p. 488).
wfstats ..... display the workfile statistics and summary view (p. 489).
wfunlink ..... break links in all link objects and auto-updating series (formulae) in the active workfile (p. 489).
wfuse ..... activate a workfile (p. 490).
workfile ..... create or change active workfile (p. 491).
write ..... write series to a disk file (p. 491).

Object Utility Commands

close ..... close window of an object, program, or workfile (p. 289).
copy..... copy objects (p. 306).
delete ..... delete objects (p. 326).
rename..... rename object (p. 421).

Object Assignment Commands

data..... enter data from keyboard (p. 317).
frml..... assign formula for auto-updating to a numeric or alpha series object (p. 340).
genr ..... create numeric or alpha series object (p. 342).
rndint..... assign random integer values to object (p. 422).
rndseed ..... set random number generator seed (p. 423).

Graph Creation Commands

Graph creation is discussed in detail in "Graph Creation Command Summary" on page 799 of the *Object Reference*. All of the page numbers in this section refer to the *Object Reference*.

area .....area graph (p. 801).
band .....area band graph (p. 804).
bar.....bar graph (p. 807).
boxplot .....boxplot graph (p. 811).
distplot .....distribution graph (p. 813).
dot.....dot plot graph (p. 820).
errbar .....error bar graph (p. 824).
hilo.....high-low(-open-close) graph (p. 826).
line.....line-symbol graph (p. 828).
pie.....pie chart (p. 831).
qqplot.....quantile-quantile graph (p. 834).
scat.....scatterplot (p. 838).
scatmat.....matrix of all pairwise scatterplots (p. 843).
scatpair.....scatterplot pairs graph (p. 845).
seasplot.....seasonal line graph (p. 849).
spike.....spike graph (p. 850).
xyarea.....XY area graph (p. 854).
xybar.....XY bar graph (p. 857).
xyline.....XY line graph (p. 859).
xypair.....XY pairs graph (p. 863).

Table Commands

setcell.....format and fill in a table cell (p. 429).
setcolwidth.....set width of a table column (p. 431).
setline.....place a horizontal line in table (p. 432).
tabplace.....insert a table into another table (p. 449).

Note that with the exception of `tabplace`, these commands are supported primarily for backward compatibility. There is a more extensive set of table procs for working with and customizing tables. See "Table Procs," on page 688 of the *Object Reference*.

Programming Commands

addin.....register an Add-in (p. 270).
adduo.....register a user object (p. 272).
exec.....execute a program (p. 328).
logclear.....clears the log window of a program (p. 369).
logmode.....sets logging of specified messages (p. 370).

logmsg ..... adds a line of text to the program log (p. 372).
logsave ..... saves the program log to a text file (p. 372).
open ..... open a program file (p. 378).
optimize ..... find the solution to a user-defined optimization problem (p. 379).
output ..... redirects print output to objects or files (p. 387).
poff ..... turns off automatic printing in programs (p. 664).
pon ..... turns on automatic printing in programs (p. 664).
program ..... create a new program (p. 415).
run ..... run a program (p. 427).
spawn ..... spawn a new process (p. 440).
statusline ..... send message to the status line (p. 442).
tic ..... reset the timer (p. 451).
toc ..... display elapsed time (since timer reset) in seconds (p. 452).

External Interface Commands

xclose ..... close an open connection to an external application (p. 493).
xget ..... retrieve data from an external application into an EViews object (p. 493).
xlog ..... switch on or off the external application log inside EViews (p. 496).
xopen ..... open a connection to an external application (p. 496).
xput ..... send an EViews object to an external application (p. 498).
xrun ..... run a command in an external application (p. 500).

Interactive Use Commands

The following commands have object command forms (*e.g.*, `Equation::arch`). These commands are particularly suited for interactive command line use. In general, we recommend that you use the object forms of the commands.

arch ..... estimate autoregressive conditional heteroskedasticity (ARCH and GARCH) models (p. 273).
archtest ..... LM test for the presence of ARCH in the residuals (p. 277).
auto ..... Breusch-Godfrey serial correlation Lagrange Multiplier (LM) test (p. 278).
binary ..... binary dependent variable models (includes probit, logit, gompit) models (p. 279).
breakls ..... least squares with breakpoints and breakpoint determination (p. 347).
cause ..... pairwise Granger causality tests (p. 283).
censored ..... estimate censored and truncated regression (includes tobit) models (p. 285).
chow ..... Chow breakpoint and forecast tests for structural change (p. 288).

Command Entries

The following section provides an alphabetical listing of commands. Each entry outlines the command syntax and associated options, and provides examples and cross references.

addin	Programming Commands

Register a program file as an EViews Add-in.

Syntax

addin*(options) [path\]prog_name*

registers the specified program file as an EViews Add-in. Note that the program file should have a ".PRG" extension, which you need not specify in the *prog_name*.

If you do not provide the optional path specification, EViews looks for the program file in the default EViews Add-ins directory.

Explicit path specifications containing ".\" and "..\" (to indicate the current level and one directory level up) are evaluated relative the directory of the installer program in which the `addin` command is specified, or the EViews default directory if `addin` is run from the command line.

You may use the special " < addins > "directory keyword in your path specification.

Options

type = *arg*	Specify the Add-ins type, where *arg* is the name of a EViews object type. The *default* is to create a global Add-in. Specifying an object-specific Add-in using a matrix object as in "type = matrix", "type = vector", etc. will register the Add-in for all matrix object types (including coef, rowvector, and sym objects). Sample objects do not support object-specific Add-ins so that "type = sample" is not allowed.
proc = *arg*	User--defined command/procedure name. If omitted, the Add-in will not have a command form.
menu = *arg*	Text for the Add-in menu entry. If omitted, the Add-in will not have an associated menu item. Note that you may use the "&" symbol in the entry text to indicate that the following character should be used as a menu shortcut.
desc = *arg*	Brief description of the Add-in that will be displayed in the Add-ins management dialog.
docs = *arg*	Path and filename for the Add-in documentation. Determination of the path follows the rules specified above for the `addin` filename.
version = *arg*	Version number of the Add-in. If no version number is supplied, EViews will assume version 1.0.
url = *arg*	Specify the location of an XML file containing information on the Add-in used for updating the Add-in to the latest version. If not supplied, EViews will default to an XML file hosted on the EViews website.

Examples

```
addin(proc="myaddin", desc="This is my add-in", version="1.0")
    .\myaddin.prg
```

registers the file "Myaddin.prg" as a global Add-in, with the user-defined global command `myaddin`, no menu support, and no assigned documentation file. The description "This is my add-in" will appear in the main Add-ins management dialog. The version number is "1.0". Note that the ".\" indicates the directory from which the program containing the `addin` command was run, or the EViews default directory if `addin` is run interactively.

```
addin(type="graph", menu="Add US Recession Shading",
    proc="recshade", docs=".\recession shade.txt", desc="Applies US
    recession shading to a graph object.") .\recshade.prg
```

registers the file "Recshade.prg" as a graph specific Add-in. The Add-in supports the object-command `recshade`, has an object-specific menu item "Add US Recession Shading", and has a documentation file "Recession shade.txt".

```
addin(type=equation, menu="Simple rolling regression", proc=roll,
    docs="<addins>\Roll\Roll.pdf", desc="Rolling Regression -
    simple version", url="www.mysite.com/myaddins.xml",
    version="1.2") "<addins>\Roll\roll.prg"
```

registers the Add-in file "Roll.prg" as an equation specific Add-in. Note that the documentation and program files are located in the "Roll" subdirectory of the default Add-ins directory. The XML file located at www.mysite.com/myaddins.xml is used when checking for available updates for the Add-in, and the current version number is set to "1.2".

Cross-references

See Chapter 8. "Add-ins," on page 169 for a detailed discussion of Add-ins.

adduo	Programming Commands

Register an EViews user object class.

Syntax

adduo*(options) [path\]definition_name*

registers the specified definition file as an EViews user object. Note that the definition file should have a ".INI" extension.

If you do not provide the optional path specification, EViews looks for the program file in the default EViews user objects directory.

Explicit path specifications containing ".\" and "..\" (to indicate the current level and one directory level up) are evaluated relative the directory of the installer program in which the `adduo` command is specified, or the EViews default directory if `adduo` is run from the command line.

Options

name = *arg*	Specify the name of the user object class.
desc = *arg*	Brief description of the user object that will be displayed in the user object management dialog.

docs = *arg*	Path and filename for the user object documentation. Determination of the path follows the rules specified above for the `adduo` filename.
version = *arg*	Version number of the Add-in. If no version number is supplied, EViews will assume version 1.0.
url = *arg*	Specify the location of an XML file containing information on the Add-in used for updating the Add-in to the latest version. If not supplied, EViews will default to an XML file hosted on the EViews website.

Examples

```
adduo(name="roll", desc="Rolling Regression Object") .\rolldef.ini
```

registers the roll class of user object, specifying a description of "Rolling Regression Object", and using the definition file rolldef.ini, located in the same folder as the installer program.

```
adduo(name="resstore", version="1.0",
    url="www.mysite.com/myuos.xml") .\resstoredef.ini
```

registers the resstore class of user object, specifying the version number as "1.0", and using the XML file located at "www.mysite.com/myuos.xml" to check for updates.

Cross-references

See Chapter 9. "User Objects," on page 195 for a discussion of user objects.

arch

Interactive Use Commands

Estimate generalized autoregressive conditional heteroskedasticity (GARCH) models.

Syntax

arch*(p,q,options) y [x1 x2 x3] [@ p1 p2 [@ t1 t2]]*

arch*(p,q,options) y = expression [@ p1 p2 [@ t1 t2]]*

The ARCH command estimates a model with p ARCH terms and q GARCH terms. *Note the order of the arguments in which the ARCH and GARCH terms are entered, which gives precedence to the ARCH term.*

The maximum value for p or q is 9; values above will be set to 9. The minimum value for p is 1. The minimum value for q is 0. If either p or q is not specified, EViews will assume a corresponding order of 1. Thus, a 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 wish to include variance regressors, list them after the mean equation using an "@"-sign to separate the mean from the variance equation.

When estimating component ARCH models, you may specify exogenous variance regressors for the permanent and transitory components. 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 as a permanent component regressor.

Options

General Options

egarch	Exponential GARCH.
parch*[= arg]*	Power ARCH. If the optional *arg* is provided, the power parameter will be set to that value, otherwise the power parameter will be estimated.
cgarch	Component (permanent and transitory) ARCH.
asy = *integer (default* = 1*)*	Number of asymmetric terms in the Power ARCH or EGARCH model. The maximum number of terms allowed is 9.
thrsh = *integer (default* = 0*)*	Number of threshold terms for GARCH and Component models. The maximum number of terms allowed is 9. For Component models, "thrsh" must take a value of 0 or 1.
archm = *arg*	ARCH-M (ARCH in mean) specification with the conditional standard deviation ("archm = sd"), the conditional variance ("archm = var"), or the log of the conditional variance ("archm = log") entered as a regressor in the mean equation.
tdist *[= number]*	Estimate the model assuming that the residuals follow a conditional Student's *t*-distribution (the default is the conditional normal distribution). Providing the optional number greater than two will fix the degrees of freedom to that value. If the argument is not provided, the degrees of freedom will be estimated.
ged *[= number]*	Estimate the model assuming that the residuals follow a conditional GED (the default is the conditional normal distribution). Providing a positive value for the optional argument will fix the GED parameter. If the argument is not provided, the parameter will be estimated.

h	Bollerslev-Wooldridge robust quasi-maximum likelihood (QML) covariance/standard errors. Not available when using the "tdist" or "ged" options.
z	Turn of backcasting for both initial MA innovations and initial variances.
b	Use Berndt-Hall-Hall-Hausman (BHHH) as maximization algorithm. The default is Marquardt.
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 (see also `param` (p. 413)).
s = *number*	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 = *key*	Set derivative method. 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.
backcast = *n*	Backcast weight to calculate value used as the presample conditional variance. Weight needs to be greater than 0 and less than or equal to 1; the default value is 0.7. Note that a weight of 1 is equivalent to no backcasting, i.e. using the unconditional residual variance as the presample conditional variance.
coef = *arg*	Specify the name of the coefficient vector (if specified by list); the default behavior is to use the "C" coefficient vector.
prompt	Force the dialog to appear from within a program.
p	Print estimation results.

GARCH options

vt	Variance target of the constant term. (Can't be used with integrated specifications).
integrated	Restrict GARCH model to be integrated, *i.e.* IGARCH. (Can't be used with variance targeting).

Saved results

Most of the results saved for the `ls` command are also available after ARCH estimation; see `ls` (p. 373) for details.

Examples

```
arch(4, 0, m=1000, h) sp500 c
```

estimates an ARCH(4) model with a mean equation consisting of the series SP500 regressed on a constant. The procedure will perform up to 1000 iterations, and will report Bollerslev-Wooldridge robust QML standard errors upon completion.

The commands:

```
c = 0.1
arch(thrsh=1, s, mean=var) @pch(nys) c ar(1)
```

estimate a TARCH(1, 1)-in-mean specification with the mean equation relating the percent change of NYS to a constant, an AR term of order 1, and a conditional variance (GARCH) term. The first line sets the default coefficient vector to 0.1, and the "s" option uses these values as coefficient starting values.

The command:

```
arch(1, 2, asy=0, parch=1.5, ged=1.2) dlog(ibm)=c(1)+c(2)*
    dlog(sp500) @ r
```

estimates a symmetric Power ARCH(2, 1) (autoregressive GARCH of order 2, and moving average ARCH of order 1) model with GED errors. The power of model is fixed at 1.5 and the GED parameter is fixed at 1.2. The mean equation consists of the first log difference of IBM regressed on a constant and the first log difference of SP500. The conditional variance equation includes an exogenous regressor R.

Cross-references

See Chapter 7. "ARCH and GARCH Estimation," on page 207 of *User's Guide II* for a discussion of ARCH models.

See `Equation::garch` (p. 82) in the *Object Reference* for the equivalent object command. See also `Equation::garch` (p. 82) and `Equation::makegarch` (p. 110) in the *Object Reference*.

archtest	Interactive Use Commands

Test for autoregressive conditional heteroskedasticity (ARCH).

Carries out Lagrange Multiplier (LM) tests for ARCH in the residuals.

Note that a more general version of the ARCH test is available using `Equation::archtest` (p. 42) as described in the *Object Reference*.

Syntax

archtest(*options*)

Options

You must specify the order of ARCH for which you wish to test. The number of lags to be included in the test equation should be provided in parentheses after the `arch` keyword.

Other Options:

prompt	Force the dialog to appear from within a program.
p	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
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 when performing an `archtest` after an `arch` estimation, EViews uses the standardized residuals (the residual of the mean equation divided by the estimated conditional standard deviation) to form the test.

Cross-references

See "ARCH LM Test" on page 162 of *User's Guide II* for further discussion of testing ARCH and Chapter 7. "ARCH and GARCH Estimation," on page 207 of *User's Guide II* for a discussion of working with ARCH models in EViews.

See `Equation::archtest` (p. 42) in the *Object Reference* for the equivalent object command. See `Equation::hettest` (p. 96) in the *Object Reference* for a more general version of the ARCH test.

auto	Interactive Use Commands

Compute serial correlation LM (Lagrange multiplier) test.

Carries out Breusch-Godfrey Lagrange Multiplier (LM) tests for serial correlation in the estimation residuals from the default equation.

Syntax

auto(*order, options*)

You must specify the order of serial correlation for which you wish to test. You should specify the number of lags in parentheses after the `auto` keyword, followed by any additional options.

Options

prompt	Force the dialog to appear from within a program.
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 you may use the commands:

```
ls output c labor capital
auto(4)
```

The commands:

```
output(t) c:\result\artest.txt
equation eq1.ls cons c y y(-1)
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 87 of *User’s Guide II* for further discussion of the Breusch-Godfrey test.

See `Equation::auto` (p. 45) in the *Object Reference* for the corresponding equation view.

binary	Interactive Use Commands

Estimate binary dependent variable models.

Estimates models where the binary dependent variable Y is either zero or one (probit, logit, gompit).

Syntax

binary*(options) y x1 [x2 x3 ...]*

binary*(options) specification*

Options

d = *arg* *(default* = "n"*)*	Specify likelihood: normal likelihood function, probit ("n"), logistic likelihood function, logit ("l"), Type I extreme value likelihood function, Gompit ("x").
q *(default)*	Use quadratic hill climbing as the maximization algorithm.
r	Use Newton-Raphson as the 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. The criterion will be set to the nearest value between 1e-24 and 0.2.
s	Use the current coefficient values in "C" as starting values (see also `param` (p. 413)).
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.

coef = *arg*	Specify the name of the coefficient vector (if specified by list); the default behavior is to use the "C" coefficient vector.
prompt	Force the dialog to appear from within a program.
p	Print results.

Examples

To estimate a logit model of Y using a constant, WAGE, EDU, and KIDS, and computing QML standard errors, you may use the command:

```
binary(d=l,h) y c wage edu kids
```

Note that this estimation uses the default global optimization options. The commands:

```
param c(1) .1 c(2) .1 c(3) .1
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 = @coefs
matrix cov_probit = @coefcov
```

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 259 of *User's Guide II* for additional discussion.

See `Equation::binary` (p. 46) in the *Object Reference* for the corresponding equation method.

breakls	Interactive Use Commands

Estimation by linear least squares regression with breakpoints.

Syntax

breakls*(options) y z1 [z2 z3 ...] [@***nv** *x1 x2 x3 ...]*

List the dependent variable first, followed by a list of the independent variables that have coefficients which are allowed to vary across breaks, followed optionally by the keyword **@nv** and a list of non-varying coefficient variables.

Options

Breakpoint Options

method = *arg* (*default* = "seqplus1")	Breakpoint selection method: "seqplus1" (sequential tests of single $l+1$ versus l breaks), "seqall" (sequential test of all possible $l+1$ versus l breaks), "glob" (tests of global l vs. no breaks), "globplus1" (tests of $l+1$ versus l globally determined breaks), "globinfo" (information criteria evaluation).
select = *arg*	Sub-method setting (options depend on "method = "). (1) if "method = glob": Sequential ("seq") (default), Highest significant ("high"), $UDmax$ ("udmax"), $WDmax$ ("wdmax"). (2) if "method = globinfo": Schwarz criterion ("bic" or "sic") (default), Liu-Wu-Zidek criterion ("lwz").
trim = *arg* (*default* = 5)	Trimming percentage for determining minimum segment size (5, 10, 15, 20, 25).
maxbreaks = *integer* (*default* = 5)	Maximum number of breakpoints to allow (not applicable if "method = seqall").
maxlevels = *integer* (*default* = 5)	Maximum number of break levels to consider in sequential testing (applicable when "method = sequall").
size = *arg* (*default* = 5)	Test sizes for use in sequential determination and final test evaluation (10, 5, 2.5, 1) corresponding to 0.10, 0.05, 0.025, 0.01, respectively
heterr	Assume regimes specific error distributions in variance computation.
commondata	Assume a common distribution for the data across segments (only applicable if original equation is estimated with a robust covariance method, "heterr" is not specified).

General Options

w = *arg*	Weight series or expression.
wtype = *arg* (*default* = "istdev")	Weight specification type: inverse standard deviation ("istdev"), inverse variance ("ivar"), standard deviation ("stdev"), variance ("var").
wscale = *arg*	Weight scaling: EViews default ("eviews"), average ("avg"), none ("none"). The default setting depends upon the weight type: "eviews" if "wtype = istdev", "avg" for all others.

cov = *keyword*	Covariance type *(optional)*: "white" (White diagonal matrix), "hac" (Newey-West HAC).
nodf	Do not perform degree of freedom corrections in computing coefficient covariance matrix. The default is to use degree of freedom corrections.
covlag = *arg* *(default* = 1*)*	Whitening lag specification: *integer* (user-specified lag value), "a" (automatic selection).
covinfo = *arg* *(default* = "aic"*)*	Information criterion for automatic selection: "aic" (Akaike), "sic" (Schwarz), "hqc" (Hannan-Quinn) (if "lag = a").
covmaxlag = *integer*	Maximum lag-length for automatic selection *(optional)* (if "lag = a"). The default is an observation-based maximum of $T^{1/3}$.
covkern = *arg* *(default* = "bart"*)*	Kernel shape: "none" (no kernel), "bart" (Bartlett, *default*), "bohman" (Bohman), "daniell" (Daniel), "parzen" (Parzen), "parzriesz" (Parzen-Riesz), "parzgeo" (Parzen-Geometric), "parzcauchy" (Parzen-Cauchy), "quadspec" (Quadratic Spectral), "trunc" (Truncated), "thamm" (Tukey-Hamming), "thann" (Tukey-Hanning), "tparz" (Tukey-Parzen).
covbw = *arg* *(default* = "fixednw" *)*	Kernel Bandwidth: "fixednw" (Newey-West fixed), "andrews" (Andrews automatic), "neweywest" (Newey-West automatic), *number* (User-specified bandwidth).
covnwlag = *integer*	Newey-West lag-selection parameter for use in nonparametric kernel bandwidth selection (if "covbw = neweywest").
covbwoffset = *integer (default* = 0*)*	Apply integer offset to bandwidth chosen by automatic selection method ("bw = andrews" or "bw = neweywest").
covbwint	Use integer portion of bandwidth chosen by automatic selection method ("bw = andrews" or "bw = neweywest").
coef = *arg*	Specify the name of the coefficient vector; the default behavior is to use the "C" coefficient vector.
prompt	Force the dialog to appear from within a program.
p	Print basic estimation results.

Examples

```
breakls m1 c unemp
```

uses the Bai-Perron sequential $L + 1$ versus L tests to determine the optimal breaks in a model regressing M1 on the breaking variables C and UNEMP.

```
breakls(method=glob, select=high) m1 c unemp
```

uses the global Bai-Perron L versus none tests to determine the breaks. The selected break will be the highest significant number of breaks.

```
breakls(size=5, trim=10) m1 c unemp
```

lowers the sequential test size from 0.10 to 0.05, and raises the trimming to 10 percent.

```
breakls(method=user, break="1990q1 2010q4") m1 c @nv unemp
```

estimates the model with two user-specified break dates. In addition, the variable UNEMP is restricted to have common coefficients across the regimes.

Cross-reference

See Chapter 12. "Least Squares with Breakpoints," beginning on page 369 of *User's Guide II* for discussion. See also "Multiple Breakpoint Tests" on page 174 of *User's Guide II*.

See `Equation::multibreak` (p. 117) of *Object Reference* for estimation of regression equations with breaks.

cause — Interactive Use Commands

Granger causality test.

Performs pairwise Granger causality tests between (all possible) pairs of the listed series or group of series.

Syntax

cause*(n, options)* ser1 ser2 ser3

Following the keyword, list the series or group of series for which you wish to test for Granger causality.

Options

You must specify the number of lags *n* to use for the test by providing an integer in parentheses after the 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:

prompt	Force the dialog to appear from within a program.
p	Print output of the test.

Examples

To compute Granger causality tests of whether GDP Granger causes M1 and whether M1 Granger causes GDP, you may enter the command:

```
cause(4) gdp m1
```

The regressors of each test are a constant and four lags of GDP and M1.

```
cause(12,p) m1 gdp r
```

prints the result of six pairwise Granger causality tests for the three series. 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).

Cross-references

See "Granger Causality" on page 523 of *User's Guide I* for a discussion of Granger's approach to testing hypotheses about causality.

See also `Group::cause` (p. 258) in the *Object Reference* for the corresponding group view.

ccopy	Object Container, Data, and File Commands

Copy one or more series from the DRI Basic Economics database to EViews data bank (.DB) files.

You must have the DRI database installed on your computer to use this feature.

Syntax

ccopy series_name

Type the name of the series or wildcard expression for series you want to copy after the `ccopy` keyword. The data bank files will be stored in the default directory with the same name as the series names in the DRI database. You can supply path information to indicate the directory for the data bank files.

Examples

The command:

```
ccopy lhur
```

copies the DRI series LHUR to "Lhur.DB" file in the default path directory.

```
ccopy b:gdp c:\nipadata\gnet
```

copies the GDP series to the "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. 287), clabel (p. 289), store (p. 444), fetch (p. 332).

cd	Global Commands

Change default directory.

The cd command changes the current default working directory. The current working directory is displayed in the "Path = ..." message in the bottom right of the EViews window.

Note that the default directory is not used for processing of include files (see include (p. 661)).

Syntax

cd *path_name*

Examples

To change the default directory to "sample data" in the "a:" drive, you may issue the command:

```
cd "a:\sample data"
```

Notice that the directory name is surrounded by double quotes. If your name does not contain spaces, you may omit the quotes. The command:

```
cd a:\test
```

changes the default directory to "a:\test".

Subsequent save operations will save into the default directory, unless you specify a different directory at the time of the operation.

Cross-references

See Chapter 3. "Workfile Basics," on page 41 of *User's Guide I* for further discussion of basic operations in EViews.

See also "include" on page 661, wfsave (p. 485), pagesave (p. 402), and save (p. 428).

censored	Interactive Use Commands

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

censored*(options) y x1 [x2 x3]*

censored*(options) specification*

Options

l = *number* *(default* = 0*)*	Set value for the left censoring limit.
r = *number* *(default* = none*)*	Set value for the right censoring limit.
l = *series_name*, i	Set series name of the indicator variable for the left censoring limit.
r = *series_name*, i	Set series name of the indicator variable for the right censoring limit.
t	Estimate truncated model.
d = *arg* *(default* = "n"*)*	Specify error distribution: normal ("n"), logistic ("l"), Type I extreme value ("x").
q *(default)*	Use quadratic hill climbing as the maximization algorithm.
r	Use Newton-Raphson as the maximization algorithm.
b	Use Berndt-Hall-Hall-Hausman 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. The criterion will be set to the nearest value between 1e-24 and 0.2.
s	Use the current coefficient values in "C" as starting values (see also `param` (p. 413)).
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.

coef = *arg*	Specify the name of the coefficient vector (if specified by list); the default behavior is to use the "C" coefficient vector.
prompt	Force the dialog to appear from within a program.
p	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, no right censoring, and the quadratic hill climbing algorithm.

Cross-references

See Chapter 9. "Discrete and Limited Dependent Variable Models," on page 259 of *User's Guide II* for discussion of censored and truncated regression models.

See `Equation::censored` (p. 53) in the *Object Reference* for the corresponding equation method.

cfetch

Object Container, Data, and File Commands

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

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 151 of *User's Guide I*.

See also `ccopy` (p. 284), `clabel` (p. 289), `store` (p. 444), `fetch` (p. 332).

chdir	Global Commands

Change default directory.

See cd (p. 285).

chow	Interactive Use Commands

Chow test for stability.

Carries out Chow breakpoint or Chow forecast tests for parameter constancy.

Syntax

chow*(options) obs1 [obs2 obs3 ...] @ x1 x2 x3*

You must provide the breakpoint observations (using dates or observation numbers) to be tested. To specify more than one breakpoint, separate the breakpoints by a space. For the Chow breakpoint test, if the equation is specified by list and contains no linear terms, you may specify a subset of the regressors to be tested for a breakpoint after an "@" sign.

Options

f	Chow forecast test. For this option, you must specify a single breakpoint to test (default performs breakpoint test).
p	Print the result of test.

Examples

The commands:

```
ls m1 c gdp cpi ar(1)
chow 1970Q1 1980Q1
```

perform a regression of M1 on a constant, GDP, and CPI with first order autoregressive errors, and employ a Chow breakpoint test to determine whether the parameters before the 1970's, during the 1970's, and after the 1970's are "stable".

To regress the log of SPOT on a constant, the log of P_US, and the log of P_UK, and to carry out the Chow forecast test starting from 1973, enter the commands:

```
ls log(spot) c log(p_us) log(p_uk)
chow(f) 1973
```

To test whether only the constant term and the coefficient on the log of P_US prior to and after 1970 are "stable" enter the commands:

```
chow 1970 @ c log(p_us)
```

Cross-references

See "Chow's Breakpoint Test" on page 170 of *User's Guide II* for further discussion.

See also ubreak (p. 457). See Equation::facbreak (p. 76), Equation::breaktest (p. 51), Equation::ubreak (p. 152), and Equation::rls (p. 135) in the *Object Reference* for related equation object views.

clabel — Object Container, Data, and File Commands

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

clabel series_name

Examples

```
clabel lhur
```

displays the description of the DRI series LHUR on the status line.

Cross-references

See also ccopy (p. 284), cfetch (p. 287), read (p. 418), fetch (p. 332).

close — Object Container, Data, and File Commands

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

close *option_or_name*

Options

option_or_name may be either an object name or one of the following options:

@all	Close down everything. This is the same as clicking on **Close All** from the EViews main menu.
@objects	Close down all objects. This is the same as clicking on **Close All Objects** from the EViews main menu.
@wf	Close down all open workfiles.
@db	Close down all open databases.
@prg	Close down all open program files.

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.

```
close @all
```

closes all windows within EViews.

```
close @objects
```

closes all objects in EViews, leaving workfiles, programs, and database windows open.

Cross-references

See Chapter 1. "Introduction," on page 5 of *User's Guide I* for a discussion of basic control of EViews.

See also `exit` (p. 329) and `save` (p. 428).

coint	Interactive Use Commands

Perform either (1) Johansen's system cointegration test, (2) Engle-Granger or Phillips-Ouliaris single equation cointegration testing, or (3) Pedroni, Kao, or Fisher panel cointegration testing for the specified series.

Syntax

There are three forms for the `coint` command which depend on the form of the test you wish to perform:

Johansen Cointegration Test Syntax

coint*(test_option, n, option) ser1 ser2 [...ser3 ser4 ...] [@ x1 x2 x3 ...]*

uses the `coint` keyword followed by the *test_option* and the number of lags *n,* and if desired, an "*@*"-sign followed by a list of exogenous variables. The first option must be 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).

Options for the Johansen Test

restrict	Impose restrictions as specified by the `append` (coint) proc.
m = *integer*	Maximum number of iterations for restricted estimation (only valid if you choose the restrict option).
c = *scalar*	Convergence criterion for restricted estimation. (only valid if you choose the restrict option).

save = *mat_name*	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.
cvtype = ol	Display 0.05 and 0.01 critical values from Osterwald-Lenum (1992). This option reproduces the output from version 4. The default is to display critical values based on the response surface coefficients from MacKinnon-Haug-Michelis (1999). Note that the argument on the right side of the equals sign are letters, not numbers 0-1).
cvsize = *arg* *(default* = 0.05*)*	Specify the size of MacKinnon-Haug-Michelis (1999) critical values to be displayed. The size must be between 0.0001 and 0.9999; values outside this range will be reset to the default value of 0.05. This option is ignored if you set "cvtype = ol".
prompt	Force the dialog to appear from within a program.
p	Print results.

This type of cointegration testing may be used in a non-panel workfile. For Fisher combined testing using the Johansen framework, see below. The remaining options for the Johansen cointegration test are outlined below ("Options for the Johansen Test" on page 291).

Note that the output for cointegration tests displays p-values for the rank test statistics. These p-values are computed using the response surface coefficients as estimated in MacKinnon, Haug, and Michelis (1999). The 0.05 critical values are also based on the response surface coefficients from MacKinnon-Haug-Michelis. *Note: the reported critical values assume no exogenous variables other than an intercept and trend*.

Single Equation Test Syntax

coint(method = *arg, options) ser1 ser2 [...ser3 ser4 ...] [@***determ** *determ_spec] [@***regdeterm** *regdeterm_spec]*

where

method = *arg*	Test method: Engle-Granger residual test ("eg"), Phillips-Ouliaris residual test ("po").

Cointegrating equation specifications that include a constant, linear, or quadratic trends, should use the "trend = " option to specify those terms. If any of those terms are in the stochastic regressors equations but not in the cointegrating equation, they should be specified using the "regtrend = " option.

Deterministic trend regressors that are not covered by the list above may be specified using the keywords **@determ** and **@regdeterm**. To specify deterministic trend regressors that enter into the regressor and cointegrating equations, you should add the keyword **@determ** followed by the list of trend regressors. To specify deterministic trends that enter in the regressor equations but not the cointegrating equation, you should include the keyword **@regdeterm** followed by the list of trend regressors.

Note that the p-values for the test statistics are based on simulations, and do not account for any user-specified deterministic regressors.

This type of cointegration testing may be used in a non-panel workfile. The remaining options for the single equation cointegration tests are outlined below.

Options for Single Equation Tests

Options for the Engle-Granger Test

The following options determine the specification of the Engle-Granger test (Augmented Dickey-Fuller) equation and the calculation of the variances used in the test statistic.

trend = *arg* *(default* = "const"*)*	Specification for the powers of trend to include in the cointegrating equation: None ("none"), Constant ("const"), Linear trend ("linear"), Quadratic trend ("quadratic"). Note that the specification implies all trends up to the specified order so that choosing a quadratic trend instructs EViews to include a constant and a linear trend term along with the quadratic.
regtrend = *arg* *(default* = "none"*)*	Additional trends to include in the regressor equations (but not the cointegrating equation): None ("none"), Constant ("const"), Linear trend ("linear"), Quadratic trend ("quadratic"). Only trend orders higher than those specified by "trend = " will be considered. Note that the specification implies all trends up to the specified order so that choosing a quadratic trend instructs EViews to include a constant and a linear trend term along with the quadratic.
lag = *arg* *(default* = "a"*)*	Method of selecting the lag length (number of first difference terms) to be included in the regression: "a" (automatic information criterion based selection), or *integer* (user-specified lag length).

lagtype = *arg* *(default* = "sic"*)*	Information criterion or method 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), "tstat" (*t*-statistic).
maxlag = *integer*	Maximum lag length to consider when performing automatic lag-length selection $$\textit{default} = \text{int}(\min((T-k)/3, 12) \cdot (T/100)^{1/4})$$ where k is the number of coefficients in the cointegrating equation. Applicable when "lag = a".
lagpval = *number* *(default* = 0.10*)*	Probability threshold to use when performing automatic lag-length selection using a *t*-test criterion. Applicable when both "lag = a" and "lagtype = tstat".
nodf	Do not degree-of-freedom correct estimates of the variances.
prompt	Force the dialog to appear from within a program.
p	Print results.

Options for the Phillips-Ouliaris Test

The following options control the computation of the symmetric and one-sided long-run variances in the Phillips-Ouliaris test.

Basic Options:

trend = *arg* *(default* = "const"*)*	Specification for the powers of trend to include in the cointegrating equation: None ("none"), Constant ("const"), Linear trend ("linear"), Quadratic trend ("quadratic"). Note that the specification implies all trends up to the specified order so that choosing a quadratic trend instructs EViews to include a constant and a linear trend term along with the quadratic.
regtrend = *arg* *(default* = "none"*)*	Additional trends to include in the regressor equations (but not the cointegrating equation): None ("none"), Constant ("const"), Linear trend ("linear"), Quadratic trend ("quadratic"). Only trend orders higher than those specified by "trend = " will be considered. Note that the specification implies all trends up to the specified order so that choosing a quadratic trend instructs EViews to include a constant and a linear trend term along with the quadratic.

nodf	Do not degree-of-freedom correct the coefficient covariance estimate.
prompt	Force the dialog to appear from within a program.
p	Print results.

HAC Whitening Options:

lag = *arg (default* = 0*)*	Lag specification: *integer* (user-specified lag value), "a" (automatic selection).
info = *arg* *(default* = "aic"*)*	Information criterion for automatic selection: "aic" (Akaike), "sic" (Schwarz), "hqc" (Hannan-Quinn) (if "lag = a").
maxlag = *integer*	Maximum lag-length for automatic selection *(optional)* (if "lag = a"). The default is an observation-based maximum.

HAC Kernel Options:

kern = *arg* *(default* = "bart"*)*	Kernel shape: "none" (no kernel), "bart" (Bartlett, *default*), "bohman" (Bohman), "daniell" (Daniel), "parzen" (Parzen), "parzriesz" (Parzen-Riesz), "parzgeo" (Parzen-Geometric), "parzcauchy" (Parzen-Cauchy), "quadspec" (Quadratic Spectral), "trunc" (Truncated), "thamm" (Tukey-Hamming), "thann" (Tukey-Hanning), "tparz" (Tukey-Parzen).
bw = *arg* *(default* = "nwfixed"*)*	Bandwidth: "fixednw" (Newey-West fixed), "andrews" (Andrews automatic), "neweywest" (Newey-West automatic), *number* (User-specified bandwidth).
nwlag = *integer*	Newey-West lag-selection parameter for use in nonparametric bandwidth selection (if "bw = neweywest").
bwint	Use integer portion of bandwidth.

Panel Syntax

coint*(option) ser1 ser2 […ser3 ser4 …]*

The `coint` command tests for cointegration among the series in the group. The second form of the command should be used with panel structured workfiles.

Options for the Panel Tests

For panel cointegration tests, you may specify the type using one of the following keywords:

Pedroni (default)	Pedroni (1994 and 2004).
Kao	Kao (1999)
Fisher	Fisher - pooled Johansen

Depending on the type selected above, the following may be used to indicate deterministic trends:

const (default)	Include a constant in the test equation. Applicable to Pedroni and Kao tests.
trend	Include a constant and a linear time trend in the test equation. Applicable to Pedroni tests.
none	Do not include a constant or time trend. Applicable to Pedroni tests.
a, b, c, d, or e	Indicate deterministic trends using the "a", "b", "c", "d", and "e" keywords, as detailed above in "Options for the Johansen Test" on page 291. Applicable to Fisher tests.

Additional Options:

ac = *arg* *(default* = "bt"*)*	Method of estimating the frequency zero spectrum: "bt" (Bartlett kernel), "pr" (Parzen kernel), "qs" (Quadratic Spectral kernel). Applicable to Pedroni and Kao tests.
band = *arg* *(default* = "nw"*)*	Method of selecting the bandwidth, where *arg* may be "nw" (Newey-West automatic variable bandwidth selection), or a number indicating a user-specified common bandwidth. Applicable to Pedroni and Kao tests.
lag = *arg*	For Pedroni and Kao tests, the method of selecting lag length (number of first difference terms) to be included in the residual regression. For Fisher tests, a pair of numbers indicating lag.
info = *arg* *(default* = "sic"*)*	Information criterion to use when computing automatic lag length selection: "aic" (Akaike), "sic" (Schwarz), "hqc" (Hannan-Quinn). Applicable to Pedroni and Kao tests.

maxlag = *int*	Maximum lag length to consider when performing automatic lag length selection, where *int* is an integer. The default is $$\text{int}(\min(T_i/3, 12) \cdot (T_i/100)^{1/4})$$ where T_i is the length of the cross-section. Applicable to Pedroni and Kao tests.
disp = *arg* *(default* = 500*)*	Maximum number of individual results to be displayed.
prompt	Force the dialog to appear from within a program.
p	Print results.

Examples

Johansen test

```
coint(s,4) x y z
```

summarizes the results of the Johansen cointegration test for the series X, Y, and Z for all five specifications of trend. The test equation uses lags of up to order four.

Engle-Granger Test

```
coint(method=eg) x y z
```

performs the default Engle-Granger test on the residuals from a cointegrating equation which includes a constant. The number of lags is determined using the SIC criterion and an observation-based maximum number of lags.

```
coint(method=eg, trend=linear, lag=a, lagtype=tstat, lagpval=.15,
   maxlag=10) x y z
```

employs a cointegrating equation that includes a constant and linear trend, and uses a sequential *t*-test starting at lag 10 with threshold probability 0.15 to determine the number of lags.

```
coint(method=eg, lag=5) x y z
```

conducts an Engle-Granger cointegration test on the residuals from a cointegrating equation with a constant, using a fixed lag of 5.

Phillips-Ouliaris Test

```
coint(method=po) x y z
```

performs the default Phillips-Ouliaris test on the residuals from a cointegrating equation with a constant, using a Bartlett kernel and Newey-West fixed bandwidth.

```
coint(method=po, bw=andrews, kernel=quadspec, nodf) x y z
```

estimates the long-run covariances using a Quadratic Spectral kernel, Andrews automatic bandwidth, and no degrees-of-freedom correction.

```
coint(method=po, trend=linear, lag=1, bw=4) x y z
```

estimates a cointegrating equation with a constant and linear trend, and performs the Phillips-Ouliaris test on the residuals by computing the long-run covariances using AR(1) prewhitening, a fixed bandwidth of 4, and the Bartlett kernel.

Panel Tests

For a panel structured workfile,

```
coint(pedroni,maxlag=3,info=sic) x y z
```

performs Pedroni's residual-based panel cointegration test with automatic lag selection with a maximum lag limit of 3. Automatic selection based on Schwarz criterion.

Cross-references

See Chapter 26. "Cointegration Testing," on page 849 of *User's Guide II* for details on the various cointegration tests.

See `Equation::coint` (p. 57) and `Group::coint` (p. 259) in the *Object Reference* for the related object routines.

cointreg	Interactive Use Commands

Estimate a cointegrating equation using Fully Modified OLS (FMOLS), Canonical Cointegrating Regression (CCR), or Dynamic OLS (DOLS) in single time series settings, and Panel FMOLS and DOLS in panel workfiles.

Syntax

cointreg*(options) y x1 [x2 x3 ...] [***@determ** *determ_spec] [***@regdeterm** *regdeterm_spec]*

List the **coint** keyword, followed by the dependent variable and a list of the cointegrating variables.

Cointegrating equation specifications that include a constant, linear, or quadratic trends, should use the "trend = " option to specify those terms. If any of those terms are in the stochastic regressors equations but not in the cointegrating equation, they should be specified using the "regtrend = " option.

Deterministic trend regressors that are not covered by the list above may be specified using the keywords **@determ** and **@regdeterm**. To specify deterministic trend regressors that enter into the regressor and cointegrating equations, you should add the keyword **@determ** followed by the list of trend regressors. To specify deterministic trends that enter in the regres-

sor equations but not the cointegrating equation, you should include the keyword **@regdeterm** followed by the list of trend regressors.

Basic Options

method = *arg* *(default* = "fmols"*)*	Estimation method: Fully Modified OLS ("fmols"), Canonical Cointegrating Regression ("ccr"), Dynamic OLS ("dols") Note that CCR estimation is not available in panel settings.
trend = *arg* *(default* = "const"*)*	Specification for the powers of trend to include in the cointegrating and regressor equations: None ("none"), Constant ("const"), Linear trend ("linear"), Quadratic trend ("quadratic"). Note that the specification implies all trends up to the specified order so that choosing a quadratic trend instructs EViews to include a constant and a linear trend term along with the quadratic.
regtrend = *arg* *(default* = "none"*)*	Additional trends to include in the regressor equations (but not the cointegrating equation): None ("none"), Constant ("const"), Linear trend ("linear"), Quadratic trend ("quadratic"). Only trend orders higher than those specified by "trend = " will be considered. Note that the specification implies all trends up to the specified order so that choosing a quadratic trend instructs EViews to include a constant and a linear trend term along with the quadratic.
regdiff	Estimate the regressor equation innovations directly using the difference specifications.
coef = *arg*	Specify the name of the coefficient vector; the default behavior is to use the "C" coefficient vector.
prompt	Force the dialog to appear from within a program.
p	Print results.

In addition to these options, there are specialized options for each estimation method.

Panel Options

panmethod = *arg* *(default* = "pooled"*)*	Panel estimation method: pooled ("pooled"), pooled weighted ("weighted"), grouped ("grouped")
pancov = sandwich	Estimate the coefficient covariance using a sandwich method that allows for cross-section heterogeneity.

Options for FMOLS and CCR

To estimate FMOLS or CCR use the "method = fmols" or "method = ccr" options. The following options control the computation of the symmetric and one-sided long-run covariance matrices and the estimate of the coefficient covariance.

HAC Whitening Options

lag = *arg (default = 0)*	Lag specification: *integer* (user-specified lag value), "a" (automatic selection).
info = *arg (default* = "aic"*)*	Information criterion for automatic selection: "aic" (Akaike), "sic" (Schwarz), "hqc" (Hannan-Quinn) (if "lag = a").
maxlag = *integer*	Maximum lag-length for automatic selection *(optional)* (if "lag = a"). The default is an observation-based maximum.

HAC Kernel Options

kern = *arg (default* = "bart"*)*	Kernel shape: "none" (no kernel), "bart" (Bartlett, *default*), "bohman" (Bohman), "daniell" (Daniell), "parzen" (Parzen), "parzriesz" (Parzen-Riesz), "parzgeo" (Parzen-Geometric), "parzcauchy" (Parzen-Cauchy), "quadspec" (Quadratic Spectral), "trunc" (Truncated), "thamm" (Tukey-Hamming), "thann" (Tukey-Hanning), "tparz" (Tukey-Parzen).
bw = *arg (default* = "nwfixed"*)*	Bandwidth:: "fixednw" (Newey-West fixed), "andrews" (Andrews automatic), "neweywest" (Newey-West automatic), *number* (User-specified bandwidth).
nwlag = *integer*	Newey-West lag-selection parameter for use in nonparametric bandwidth selection (if "bw = neweywest").
bwoffset = *integer (default = 0)*	Apply integer offset to bandwidth chosen by automatic selection method ("bw = andrews" or "bw = neweywest").
bwint	Use integer portion of bandwidth chosen by automatic selection method ("bw = andrews" or "bw = neweywest").

Coefficient Covariance

nodf	Do not degree-of-freedom correct the coefficient covariance estimate.

Panel Options

hetfirst	Estimate the first-stage regression assuming heterogeneous coefficients. For FMOLS specifications estimated using pooled or pooled weighted methods ("panmethod = pooled", "panmethod = weighted")

Options for DOLS

To estimate using DOLS use the "method = dols" option. The following options control the specification of the lags and leads and the estimate of the coefficient covariance.

lltype = *arg* *(default* = "fixed"*)*	Lag-lead method: fixed values ("fixed"), automatic selection - Akaike ("aic"), automatic - Schwarz ("sic"), automatic - Hannan-Quinn ("hqc"), None ("none").
lag = *arg*	Lag specification: *integer* (*required* user-specified number of lags if "lltype = fixed").
lead = *arg*	Lead specification: *integer* (*required* user-specified number of lags if "lltype = fixed").
maxll = *integer*	Maximum lag and lead-length for automatic selection (*optional* user-specified integer if "lltype = " is used to specify automatic selection). The default is an observation-based maximum.
cov = *arg*	Coefficient covariance method: (default) long-run variance scaled OLS, unscaled OLS ("ols"), White ("white"), Newey-West ("hac").
nodf	Do not degree-of-freedom correct the coefficient covariance estimate.

For the default covariance calculation or "cov = hac", the following options control the computation of the long-run variance or robust covariance:

HAC Covariance Whitening Options (if default covariance or "cov=hac")

covlag = *arg* *(default* = 0*)*	Lag specification: *integer* (user-specified lag value), "a" (automatic selection).
covinfo = *arg* *(default* = "aic"*)*	Information criterion for automatic selection: "aic" (Akaike), "sic" (Schwarz), "hqc" (Hannan-Quinn) (if "lag = a").
covmaxlag = *integer*	Maximum lag-length for automatic selection *(optional)* (if "lag = a"). The default is an observation-based maximum.

HAC Covariance Kernel Options (if default covariance or "cov=hac")

covkern = *arg* (*default* = "bart")	Kernel shape: "none" (no kernel), "bart" (Bartlett, *default*), "bohman" (Bohman), "daniell" (Daniel), "parzen" (Parzen), "parzriesz" (Parzen-Riesz), "parzgeo" (Parzen-Geometric), "parzcauchy" (Parzen-Cauchy), "quadspec" (Quadratic Spectral), "trunc" (Truncated), "thamm" (Tukey-Hamming), "thann" (Tukey-Hanning), "tparz" (Tukey-Parzen).
covbw = *arg* *(default* = "nwfixed"*)*	Bandwidth: "fixednw" (Newey-West fixed), "andrews" (Andrews automatic), "neweywest" (Newey-West automatic), *number* (User-specified bandwidth).
covnwlag = *integer*	Newey-West lag-selection parameter for use in nonparametric bandwidth selection (if "covbw = neweywest").
covbwoffset = *integer* *(default* = 0*)*	Apply integer offset to bandwidth chosen by automatic selection method ("bw = andrews" or "bw = neweywest").
covbwint	Use integer portion of bandwidth chosen by automatic selection method ("bw = andrews" or "bw = neweywest").

Panel Options

Weighted coefficient or coefficient covariance estimation in panel DOLS requires individual estimates of the long-run variances of the residuals. You may compute these estimates using the standard default long-run variance options, or you may choose to estimate it using the ordinary variance.

For weighted estimation we have:

panwgtlag = *arg* *(default* = 0*)*	Lag specification: *integer* (user-specified lag value), "a" (automatic selection).
panwgtinfo = *arg* *(default* = "aic"*)*	Information criterion for automatic selection: "aic" (Akaike), "sic" (Schwarz), "hqc" (Hannan-Quinn) (if "lrvarlag = a").
panwgtmaxlag = *integer*	Maximum lag-length for automatic selection *(optional)* (if "lrvarlag = a"). The default is an observation-based maximum.
panwgtkern = *arg* *(default* = "bart"*)*	Kernel shape: "none" (no kernel), "bart" (Bartlett, *default*), "bohman" (Bohman), "daniell" (Daniell), "parzen" (Parzen), "parzriesz" (Parzen-Riesz), "parzgeo" (Parzen-Geometric), "parzcauchy" (Parzen-Cauchy), "quadspec" (Quadratic Spectral), "trunc" (Truncated), "thamm" (Tukey-Hamming), "thann" (Tukey-Hanning), "tparz" (Tukey-Parzen).

panwgtbw = *arg* *(default* = "nwfixed"*)*	Bandwidth:: "fixednw" (Newey-West fixed), "andrews" (Andrews automatic), "neweywest" (Newey-West automatic), *number* (User-specified bandwidth).
panwgtnwlag = *integer*	Newey-West lag-selection parameter for use in nonparametric bandwidth selection (if "bw = neweywest").
panwgtbwoffset = *integer* *(default* = 0*)*	Apply offset to automatically selected bandwidth. For settings where "cov = hac", "covkern = " is specified, and "covbw = " is not user-specified.
panwgtbwint	Use integer portion of bandwidth chosen by automatic selection method ("bw = andrews" or "bw = neweywest").

For the coefficient covariance estimation we have:

lrvar = ordinary	Compute DOLS estimates of the long-run residual variance used in covariance calculation using the ordinary variance.
lrvarlag = *arg* *(default* = 0*)*	For DOLS estimates of the long-run residual variance used in covariance calculation, lag specification: *integer* (user-specified lag value), "a" (automatic selection).
lrvarinfo = *arg* *(default* = "aic"*)*	For DOLS estimates of the long-run residual variance used in covariance calculation, information criterion for automatic selection: "aic" (Akaike), "sic" (Schwarz), "hqc" (Hannan-Quinn) (if "lrvarlag = a").
lrvarmaxlag = *integer*	For DOLS estimates of the long-run residual variance used in covariance calculation, maximum lag-length for automatic selection *(optional)* (if "lrvarlag = a"). The default is an observation-based maximum.
lrvarkern = *arg* *(default* = "bart"*)*	For DOLS estimates of the long-run residual variance used in covariance calculation, Kernel shape: "none" (no kernel), "bart" (Bartlett, *default*), "bohman" (Bohman), "daniell" (Daniell), "parzen" (Parzen), "parzriesz" (Parzen-Riesz), "parzgeo" (Parzen-Geometric), "parzcauchy" (Parzen-Cauchy), "quadspec" (Quadratic Spectral), "trunc" (Truncated), "thamm" (Tukey-Hamming), "thann" (Tukey-Hanning), "tparz" (Tukey-Parzen).
lrvarbw = *arg* *(default* = "nwfixed"*)*	For DOLS estimates of the long-run residual variance used in covariance calculation, bandwidth:: "fixednw" (Newey-West fixed), "andrews" (Andrews automatic), "neweywest" (Newey-West automatic), *number* (User-specified bandwidth).

lrvarnwlag = *integer*	For DOLS estimates of the long-run residual variance used in covariance calculation, Newey-West lag-selection parameter for use in nonparametric bandwidth selection (if "bw = neweywest").
lrvarbwoffset = *integer (default* = 0*)*	For DOLS estimates of the long-run residual variance used in covariance calculation, apply offset to automatically selected bandwidth. For settings where "cov = hac", "covkern = " is specified, and "covbw = " is not user-specified.
lrvarbwint	For DOLS estimates of the long-run residual variance used in covariance calculation, use integer portion of bandwidth.

Examples

FMOLS and CCR

To estimate, by FMOLS, the cointegrating equation for LC and LY including a constant, you may use:

```
cointreg(nodf, bw=andrews) lc ly
```

The long-run covariances are estimated nonparametrically using a Bartlett kernel and a bandwidth determined by the Andrews automatic selection method. The coefficient covariances are estimated with no degree-of-freedom correction.

To include a linear trend term in a model where the long-run covariances computed using the Quadratic Spectral kernel and a fixed bandwidth of 10, enter:

```
cointreg(trend=linear, nodf, bw=10, kern=quadspec) lc ly
```

A model with a cubic trend may be estimated using:

```
cointreg(trend=linear, lags=2, bw=neweywest, nwlag=10,
    kernel=parzen) lc ly @determ @trend^3
```

Here, the long-run covariances are estimated using a VAR(2) prewhitened Parzen kernel with Newey-West nonparametric bandwidth determined using 10 lags in the autocovariance calculations.

```
cointreg(trend=quadratic, bw=andrews, lags=a, info=aic,
    kernel=none, regdiff) lc ly @regdeterm @trend^3
```

estimates a restricted model with a cubic trend term that does not appear in the cointegrating equation using a parametric VARHAC with automatic lag length selection based on the AIC. The residuals for the regressors equations are obtained by estimating the difference specification.

To estimate by CCR, we provide the "method = ccr" option. The command

```
cointreg(method=ccr, lag=2, bw=andrews, kern=quadspec) lc ly
```

estimates, by CCR, the constant only model using a VAR(2) prewhitened Quadratic Spectral and Andrews automatic bandwidth selection.

```
cointreg(method=ccr, trend=linear, lag=a, maxlag=5, bw=andrews,
    kern=quadspec) lc ly
```

modifies the previous estimates by adding a linear trend term to the cointegrating and regressors equations, and changing the VAR prewhitening to automatic selection using the default SIC with a maximum lag length of 5.

```
cointreg(method=ccr, trend=linear, regtrend=quadratic, lag=a,
    maxlag=5, bw=andrews) lc ly
```

adds a quadratic trend term to the regressors equations only, and changes the kernel to the default Bartlett.

DOLS

```
cointreg(method=dols, trend=linear, nodf, lag=4, lead=4) lc ly
```

estimates the linear specification using DOLS with four lags and leads. The coefficient covariance is obtained by rescaling the no d.f.-correction OLS covariance using the long-run variance of the residuals computed using the default Bartlett kernel and default fixed Newey-West bandwidth.

```
cointreg(method=dols, trend=quadratic, nodf, lag=4, lead=2,
    covkern=bohman, covbw=10) lc ly @determ @trend^3
```

estimates a cubic specification using 4 lags and 2 leads, where the coefficient covariance uses a Bohman kernel and fixed bandwidth of 10.

```
cointreg(method=dols, trend=quadratic, nodf, lag=4, lead=2,
    cov=hac, covkern=bohman, covbw=10) lc ly @determ @trend^3
```

estimates the same specification using a HAC covariance in place of the scaled OLS covariance.

```
cointreg(method=dols, trend=quadratic, lltype=none, cov=ols) lc ly
    @determ @trend^3
```

computes the Static OLS estimates with the usual OLS d.f. corrected coefficient covariance.

Cross-references

See Chapter 8. "Cointegrating Regression," beginning on page 231 of *User's Guide II* for a discussion of single equation cointegrating regression. See Chapter 24. "Panel Cointegration Estimation," beginning on page 797 of *User's Guide II* for discussion of estimation in panel settings.

See "Vector Error Correction (VEC) Models" on page 572 of *User's Guide II* for a discussion of VEC estimation.

See also coint (p. 291).

copy	Object Container, Data, and File Commands

Copy an object, or a set of objects matching a name pattern, within and between workfiles, workfile pages, and databases. Data in series objects may be frequency converted or match merged.

Syntax

copy*(options) src_spec dest_spec [src_id dest_id]*

copy*(options) src_spec dest_spec [@***src** *src_ids @***dest** *dest_id]*

where *src_spec* and *dest_spec* are of the form:

*[ctype][container::][page\]*object_name

There are three parts to the copy command: (1) a specification of the location and names of the source objects; (2) a specification of the location and names of the destination objects; (3) optional source and destination IDs if the copy operation involves match merging.

The source and destination objects are specified in multiple (optional) parts: (1) the *container* specification is the name of a workfile or database; (2) the *page* specification is the name of a page within a workfile or a subdirectory within a database; and (3) the *object_name* specification is the name of an object or a wildcard pattern corresponding to multiple objects.

The *ctype* specification is rarely required, but permits you to specify precisely your source or destination in cases where a database and workfile share the same name. In this case, *ctype* may be used to indicate the container to which you are referring by prefixing the container name with ":" to indicate the workfile, or "::" to indicate the database with the common name.

When parts of the source or destination specification are not provided, EViews will fill in default values where possible. The default container is the active workfile, unless the "::" prefix is used in which case the default container is the default database. The default page within a workfile is always the active page. The default name for the destination object is the name of the object within the source container.

If ID series are not provided in the command, then EViews will perform frequency conversion when copying data whenever the source and destination containers have different frequencies. If ID series are provided, then EViews will perform a general match merge between the source and destination using the specified ID series. In the case where you wish to copy your data using match merging with special treatment for date matching, you must use the special keyword "@DATE" as an ID series for the source or destination. If "@DATE"

is not specified as an identifier in either the source or destination IDs, EViews will perform an exact match merge using the given identifiers.

If ID series are not specified, but a conversion option requiring a general match merge is used (*e.g.*, "c = med"), "@DATE @DATE" will be appended to the list of IDs and a general date match merge will be employed.

See `Link::linkto` (p. 319) in the *Object Reference* for additional discussion of the differences embodied in these choices.

The general syntax described above covers all possible uses of the `copy` command. The following paragraphs provide examples of the specific syntax used for some common cases of the command.

Copying Within a Workfile

Copy an object within the default workfile page as a new object with a different name:

- **copy***(options) src_name dest_name*

Copy an object from the *src_page* page into the default workfile page using the specified name:

- **copy***(options) src_page**src_name dest_name*

Copy an object from the *src_page* page into the *dest_page* page, keeping the same name:

- **copy***(options) src_page**src_names dest_page*\

Copy an object from the *src_page* page to the default workfile page, match merging any series data using a single *src_id* and a single *dest_id* identifier series:

- **copy***(options) src_page**src_name dest_name src_id dest_id*

Copy an object from the *src_page* page to the *dest_page* match merging any series data using multiple source and destination identifier series:

- **copy***(options) src_page**src_name dest_page**dest_name* **@src** *src_id1 src_id2 … src_id_n* **@dest** *dest_id1 dest_id2 … dest_id_n*

Copying Between Containers (Workfiles and Databases)

Copy one or more objects from the *src_page* of the workfile *src_workfile* to the *dest_page* of the workfile *dest_workfile*, using the name or name pattern given in *src_names*:

- **copy***(options) src_workfile*::*src_page**src_names dest_workfile*::*dest_page*\

Copy an object from database *src_database* to the default page in the container *dest_container*:

- **copy***(options) src_database*::*src_name dest_container*::*dest_name*

Note that if both a workfile and database exist matching the name provided in *dest_container*, EViews will favor the workfile unless the "::" prefix is used to specify explicitly that the database should be used.

Options

Basic Options

overwrite / o	Overwrite any existing object with the destination name in the destination container. Error only if a non-editable series is encountered in the destination location.
merge / m	If the source object is a series, merge the data from the source series into any existing destination series, preserving any values in the destination series that are not present in the source. For all other object types, overwrite any existing object with the source object. Error if a non-editable series is encountered in the destination location.
protect / p	Protect objects in the destination location from overwriting or merging. If there is an existing object in the destination container, cancel the copy operation for that object, but do not generate an error.
noerr	Suppress errors that are generated during the copy. For example, if the overwrite option is used, suppress any error caused by attempting to overwrite a non-editable series such as an index series used in the workfile structure.
link	When copying from a database, copy as a database link.

Group Copy Options

When copying a group object from workfile to database:

g = *arg*	Method for copying group objects from a workfile to database: "s" (copy group definition and series as separate objects), "t" (copy group definition and series as one object), "d" (copy series only as separate objects), "l" (copy group definition only).

When copying a group object from a database to a workfile:

g = *arg*	Method for copying group objects from a database or workfile to a workfile: "b" (copy both group definition and series), "d" (copy only the series), "l" (copy only the group definition).

Note that copying a group object containing expressions or auto-updating series between workfiles only copies the expressions, and not the underlying series.

Frequency Conversion Options

If the `copy` command does not specify identifier series, EViews will perform frequency conversion of the data contained in series objects whenever the source and destination containers are dated, but do not have the same frequency.

If either of the pages are undated, EViews will, unless match merge options are provided (as described below), perform a raw copy, in which the first observation in the source workfile page is copied into the first observation in the destination page, the second observation in the source into the second observation in the destination, and so forth.

The following options control the frequency conversion method when copying series and group objects into a workfile page and converting from *low* to *high* frequency:

c = *arg*	Low to high conversion methods: "r" (constant match average), "d" (constant match sum), "q" (quadratic match average), "t" (quadratic match sum), "i" (linear match last), "c" (cubic match last).

The following options control the frequency conversion method when copying series and group objects into a workfile page and converting from *high* to *low* frequency:

c = *arg*	*High to low conversion methods removing NAs*: "a" (average of the nonmissing observations), "s" (sum of the nonmissing observations), "f" (first nonmissing observation), "l" (last nonmissing observation), "x" (maximum nonmissing observation), "m" (minimum nonmissing observation). *High to low conversion methods propagating NAs*: "an" or "na" (average, propagating missings), "sn" or "ns" (sum, propagating missings), "fn" or "nf" (first, propagating missings), "ln" or "nl" (last, propagating missings), "xn" or "nx" (maximum, propagating missings), "mn" or "nm" (minimum, propagating missings).

Note that if no conversion method is given in the command, the conversion method specified within the series object will be used as the default. If the series does not contain an explicit conversion method, the global option settings will used to determine the method.

Frequency conversion involving panel structured pages involves special handling:

- If both pages are dated panel pages that are structured with a single identifier, EViews will perform frequency conversion cross-section by cross-section.
- Conversion from a dated panel page to a dated, non-panel page will first perform a mean contraction across cross-sections to obtain a single time series (by computing the means for each period), and then a frequency conversion of the resulting time series to the new frequency.

- Conversion from a dated, non-panel page to a dated panel page will first involve a frequency conversion of the single time series to the new frequency. The converted time series will be used for each cross-section in the panel page.

In all three of these cases, all of the high-to-low conversion methods are supported, but low-to-high frequency conversion only offers **Constant-match average** (repeating of the low frequency observations).

Lastly, conversion involving a panel page with more than one dimension or an undated page will default to raw data copy unless general match merge options are provided.

Match Merge Options

These options are available when ID series are specified in the `copy` commmand.

smpl = *smpl_spec*	Sample to be used when computing contractions during copying using match merge. Either provide the sample range in double quotes or specify a named sample object. By default, EViews will use the entire workfile sample "@ALL".
c = *arg*	Set the match merge contraction method. If you are copying a numeric source series by general match merge, the argument can be one of: "mean", "med" (median), "max", "min", "sum", "sumsq" (sum-of-squares), "var" (variance), "sd" (standard deviation), "skew" (skewness), "kurt" (kurtosis), "quant" (quantile, used with "quant = " option), "obs" (number of observations), "nas" (number of NA values), "first" (first observation in group), "last" (last observation in group), "unique" (single unique group value, if present), "none" (disallow contractions). If copying an alpha series, only the non-summary methods "max", "min", "obs", "nas", first", "last", "unique" and "none" are supported. For copying of numeric series, the default contraction method is "c = mean"; for copying of alpha series, the default is "c = unique".
quant = *number*	Quantile value to be used when contracting using the "c = quant" option (*e.g,* "quant = .3").
nacat	Treat "NA" values as a category when copying using general match merge operations.

Most of the conversion options should be self-explanatory. As for the others: "first" and "last" give the first and last non-missing observed for a given group ID; "obs" provides the number of non-missing values for a given group; "nas" reports the number of NAs in the group; "unique" will provide the value in the source series if it is the identical for all obser-

vations in the group, and will return NA otherwise; "none" will cause the copy to fail if there are multiple observations in any group—this setting may be used if you wish to prohibit all contractions.

On a match merge expansion, copying with match merging will repeat the value of the source for every observation with matching identifier values in the destination. If both the source and destination have multiple values for a given ID, EViews will first perform a contraction across IDs in the source (if not ruled out by "c = none"), and then perform the expansion by replicating the contracted value in the destination. For example, converting from a quarterly panel to an annual panel using match merge, EViews will first contract the data across IDs down to a single quarterly time series, will convert the series to an annual frequency, then will assign the annual data to each of the cross-sections in the destination page.

Examples

```
copy good_equation best_equation
```

makes an exact copy of GOOD_EQUATION and names it BEST_EQUATION.

```
copy graph_1 wf2::wkly\graph1
```

copies GRAPH_1 from the default page of the current workfile to GRAPH1 in the page WKLY of the workfile WF2.

```
copy gdp usdat::
```

copies GDP from the current workfile to GDP in the USDAT database or workfile.

```
copy ::gdp macro1::gdp_us
```

copies GDP from the default database to either the open workfile MACRO1, or the database named MACRO1 if there is no open workfile with that name. If there is an open workfile MACRO1 you may use

```
copy ::gdp ::macro1::gdp_us
```

to specify explicitly that you wish to write to the MACRO1 database.

```
copy(smpl="1990 2000") page1\pop page2\ @src county @date @dest
    county @date
```

copies POP data for 1990 through 2005 from PAGE1 to PAGE2, match merge using the ids COUNTY and the date structure of the two pages.

```
copy(smpl="1990 2000", c=mean) panelpage\inc countypage\ county
    county
```

copies the INC data from the PANELPAGE to the COUNTYPAGE, match merging using the values of the COUNTY series, and contracting the panel data by computing means for each county using the specified sample.

```
copy countypage\pop panelpage\ county county
```

match merges the POP data from the COUNTYPAGE to the PANELPAGE using the values of the COUNTY series.

```
copy(c=x, merge) quarterly::page1\ser* annual::page6\*
```

copies all objects with names beginning with "SER" on page PAGE1 of workfile QUARTERLY into page PAGE6 of workfile ANNUAL using the existing names. Series objects with data that can be (high-to-low) frequency converted will take the maximum value within a low-frequency period as the conversion method. If destination series already exist with the same name as the source series, the data will be merged. If destination objects (non-series) exist with the same name as source series, they will be overwritten.

Note that since databases are read from disk, you may provide a path for the database in the container specification, as in:

```
copy "c:\my data\dba.edb::ser01" ser02
```

which copies the object SER01 from the database DBA.EDB located in the path "C:\MY DATA\" to SER02 in the default workfile page.

```
copy gd* "c:\my data\findat::"
```

makes a duplicate of all objects in the default page of the current workfile with names starting with "GD" to the database FINDAT in the root of "C:\MY DATA\".

Cross-references

See "Copying Objects" on page 311 of *User's Guide I* for a discussion of copying and moving objects.

See also `fetch` (p. 332), `store` (p. 444), and `Link::linkto` (p. 319) in the *Object Reference*.

cor	Interactive Use Commands

Compute Pearson product-moment (ordinary) correlations for the specified series or groups.

Syntax

cor*(options) arg1 [arg2 arg3...]*

where *arg1*, *arg2*, *etc.* are the names of series or groups.

Note that this command is a limited feature version of the group view `Group::cor` (p. 267) in the *Object Reference*.

Options

wgt = *name (optional)*	Name of series containing weights.
wgtmethod = *arg (default* = "sstdev"*)*	Weighting method (when weights are specified using "weight = "): frequency ("freq"), inverse of variances ("var"), inverse of standard deviation ("stdev"), scaled inverse of variances ("svar"), scaled inverse of standard deviations ("sstdev").
pairwise	Compute using pairwise deletion of observations with missing cases (pairwise samples).
out = *name*	Basename for saving output. All results will be saved in Sym matrices named using the string "CORR", appended to the basename (*e.g.*, the correlation specified by "out = my" is saved in the Sym matrix "MYCORR").
prompt	Force the dialog to appear from within a program.
p	Print the result.

Examples

```
cor height weight age
```

displays a 3×3 Pearson correlation matrix for the three series HEIGHT, WEIGHT, and AGE.

Cross-references

See `Group::cor` (p. 267) in the *Object Reference* for more general routines for computing correlations.

See also `cov` (p. 315). For simple functions to perform the calculations, see `@cor` (p. 616), and `@cov` (p. 616).

count

Interactive Use Commands

Estimates models where the dependent variable is a nonnegative integer count.

Syntax

count*(options) y x1 [x2 x3...]*

count*(options) specification*

Follow the `count` keyword by the name of the dependent variable and a list of regressors.

Options

d = *arg* *(default* = "p"*)*	Likelihood specification: Poisson likelihood ("p"), normal quasi-likelihood ("n"), exponential likelihood ("e"), negative binomial likelihood or quasi-likelihood ("b").
v = *positive_num* *(default* = 1*)*	Specify fixed value for QML parameter in normal and negative binomial quasi-likelihoods.
q *(default)*	Use quadratic hill-climbing as the maximization algorithm.
r	Use Newton-Raphson as the maximization algorithm.
b	Use Berndt-Hall-Hall-Hausman as the 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. The criterion will be set to the nearest value between 1e-24 and 0.2.
s	Use the current coefficient values in "C" as starting values (see also `param` (p. 413)).
s = *number*	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").
prompt	Force the dialog to appear from within a program.
p	Print the result.

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.

```
count arrest c job police
makeresid(g) res_g
```

estimates a Poisson count model of ARREST on a constant, JOB, and POLICE, and stores the generalized residuals in the series RES_G.

```
count(d=p) y c x1
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.

```
count(d=p, h) y c x1
```

estimates the same model with QML standard errors and covariances.

Cross-references

See "Count Models" on page 305 of *User's Guide II* for additional discussion.

See `Equation::count` (p. 70) of the *Object Reference* for the equivalent equation object command.

cov	Interactive Use Commands

Compute Pearson product-moment (ordinary) covariances for the specified series or groups.

Syntax

cor *arg1 [arg2 arg3...]*

where *arg1*, *arg2*, *etc.* are the names of series or groups.

Note that this command is a limited feature version of the group view `Group::cov` (p. 271) in the *Object Reference*.

Options

wgt = *name (optional)*	Name of series containing weights.
wgtmethod = *arg (default* = "sstdev"*)*	Weighting method (when weights are specified using "weight = "): frequency ("freq"), inverse of variances ("var"), inverse of standard deviation ("stdev"), scaled inverse of variances ("svar"), scaled inverse of standard deviations ("sstdev").
pairwise	Compute using pairwise deletion of observations with missing cases (pairwise samples).
out = *name*	Basename for saving output. All results will be saved in Sym matrices named using the string "CORR", appended to the basename (*e.g.*, the correlation specified by "out = my" is saved in the sym matrix "MYCORR").
prompt	Force the dialog to appear from within a program.
p	Print the result.

Examples

```
cov height weight age
```

displays a 3×3 Pearson covariance matrix for the three series HEIGHT, WEIGHT, and AGE.

Cross-references

See `Group::cov` (p. 271) in the *Object Reference* for more general routines for computing covariances.

See also `cor` (p. 312) .For simple functions to perform the calculations, see `@cor` (p. 616), and `@cov` (p. 616).

create	Object Container, Data, and File Commands

Create workfile.

This command has been replaced by `wfcreate` (p. 467) and `pagecreate` (p. 394).

cross	Interactive Use Commands

Displays cross correlations (correlograms) for a pair of series.

Syntax

cross*(n,options) ser1 ser2 [ser3 ...]*

You must specify the number of lags *n* to use in computing the cross correlations as the first option. EViews will create an untitled group from the specified series and groups, and will display the cross correlation view for the group.

Options

The following options may be specified inside the parentheses after the number of lags:

prompt	Force the dialog to appear from within a program.
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, using 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 squared standardized residual and the standardized residual, up to 24 leads and lags. This correlogram provides a rough check of asymmetry in the ARCH effect.

Cross-references

See "Cross Correlations and Correlograms" on page 517 of *User's Guide I* for discussion.

See `Group::cross` (p. 274) of the *Object Reference* for the equivalent group view command.

data	Object Creation Commands || Object Assignment Commands || Interactive Use Commands

Enter data from keyboard.

Opens an unnamed group window to edit one or more series.

Syntax

data *arg1 [arg2 arg3 ...]*

Follow the `data` keyword by a list of series and 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 129 of *User's Guide I* for a discussion of the process of entering data from the keyboard.

db	Object Container, Data, and File Commands

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

db(*options*) *[path\]db_name [***as** *shorthand_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 may include a path name to work with a database not in the default path.

You may use the "as" keyword to provide an optional *shorthand_name* or short text label which may be used to refer to the database in commands and programs. If you leave this field blank, a default *shorthand_name* will be assigned automatically.

See "Database Shorthands" on page 307 of *User's Guide I* for additional discussion.

Options

See `dbopen` (p. 322) 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 10. "EViews Databases," on page 303 of *User's Guide I* for a discussion of EViews databases.

See also `dbcreate` (p. 320) and `dbopen` (p. 322).

dbcopy	Object Container, Data, and File Commands

Make a copy of an existing database.

Syntax

dbcopy *[path\]source_name [path\]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 from or to a database that is not in the default directory. All files associated with the database will be copied.

Options

type = *arg*	Specify the source database type: *(see table below)*. The default is to read an EViews 7 database.
desttype = *arg*, t = *arg*	Specify the destination database type: *(see table below)*. The default is to create an EViews 7 database.

The following table summaries the various database formats, along with the corresponding allowable "type = " and "desttype = " keywords:

	Option Keywords
Aremos-TSD x	"a", "aremos", "tsd"
DRIBase x	"b" "dribase"
DRIPro Link x	"dripro"
DRI DDS	"dds"
EViews	"e", "evdb"
EViews 6 compatible	"eviews6"
FAME	"f", "fame"
GiveWin/PcGive	"g", "give"
RATS 4.x	"r", "rats"
RATS Portable / TROLL	"l", "trl"
TSP Portable	"t", "tsp"

For the source specification, the following options may be required when connecting to a remote server:

s = *server_id*, server = *server_id*	Server name
u = *user*, username = *user*	Username
p = *pwd*, password = *pwd*	Password

For the destination specification, the following options may be required when connecting to a remote server:

dests = *server_id*, destserver = *server_id*	Server name
destu = *user*, destusername = *user*	Username
destp = *pwd*, destpassword = *pwd*	Password

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 basename "Usdat".

Cross-references

See Chapter 10. "EViews Databases," on page 303 of *User's Guide I* for a discussion of EViews databases.

See also `dbrename` (p. 325) and `dbdelete` (p. 322).

dbcreate

Object Container, Data, and File Commands

Create a new database.

Syntax

dbcreate*(options) [path\]db_name* [**as** *shorthand_name]*

Follow the `dbcreate` keyword by a name for the new database. You may include a path name to create a database not in the default directory. The new database will become the default database.

You may use the "as" keyword to provide an optional *shorthand_name* or a short text label which may be used to refer to the open database in commands and programs. If you leave this field blank, a default *shorthand_name* will be assigned automatically. See "Database Shorthands" on page 307 of the *User's Guide I* for additional discussion.

Options

type = *arg*, t = *arg*	Specify the database type: *(see table below)*. The default is to create an EViews 7 database.

The following table summaries the various database formats, along with the corresponding "type = " keywords:

	Option Keywords
Aremos-TSD x	"a", "aremos", "tsd"
DRIBase x	"b" "dribase"
DRIPro Link x	"dripro"
DRI DDS	"dds"
EViews	"e", "evdb"
EViews 6 compatible	"eviews6"
FAME	"f", "fame"
GiveWin/PcGive	"g", "give"
RATS 4.x	"r", "rats"
RATS Portable / TROLL	"l", "trl"
TSP Portable	"t", "tsp"

You must have EViews Enterprise Edition to create DRIBase and FAME databases.

The following options may be required when connecting to a remote server:

s = *server_id*, server = *server_id*	Server name
u = *user*, username = *user*	Username
p = *pwd*, password = *pwd*	Password

Examples

```
dbcreate 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 (p. 322) or db (p. 317).

Cross-references

See Chapter 10. "EViews Databases," on page 303 of *User's Guide I* for a discussion of EViews databases.

See also dbopen (p. 322) or db (p. 317).

dbdelete	Object Container, Data, and File Commands

Delete an existing database (all files associated with the specified database).

Syntax

dbdelete *[path\]db_name*

Follow the `dbdelete` keyword by the name of the database to be deleted. You may include a path name to delete a database not in the default path.

Examples

```
dbdelete c:\temp\testdat
```

deletes all files associated with the TESTDAT database in the specified directory.

Cross-references

See Chapter 10. "EViews Databases," on page 303 of *User's Guide I* for a discussion of EViews databases.

See also `dbcopy` (p. 318) and `dbdelete` (p. 322).

dbopen	Object Container, Data, and File Commands

Open an existing database.

Syntax

dbopen*(options) [path\]db_name [***as** *shorthand_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.

You do not need to specify a database name when opening a Datastream or FRED connection ("type = datastream" or "type = fred") as EViews will automatically connect to the proper location.

You may use the "as" keyword to provide an optional *shorthand_name* or a short text label which is used to refer to the open database in commands and programs. If you leave this field blank, a default *shorthand_name* will be assigned automatically.

See "Database Shorthands" on page 307 of *User's Guide I* for additional discussion.

By default, EViews will use the extension of the database file to determine type. For example, files with the extension ".EDB" will be opened as an EViews database, while files with

the extension ".IN7" will be opened as a GiveWin database. You may use the "type = " option to specify an explicit type.

Options

type = *arg*, t = *arg* Specify the database type: *(see table below)*.

The following table summaries the various database formats, along with the corresponding "type = " keywords:

	Option Keywords
AREMOS Bank	"aremos
AREMOS TSD	"a", "tsd"
CEIC	"ceic"
Datastream	"datastream"
DRIBase	"b" "dribase"
DRIPro Link	"dripro"
DRI DDS	"dds"
EcoWin	"ecowin"
EViews	"e", "eviews"
FactSet	"factset"
FAME	"f", "fame"
FRED	"fred"
GiveWin/PcGive	"g", "give"
Haver	"h", "haver"
IHS Magellan	"magellan"
Moody's Economy.com	"economy"
RATS 4.x	"r", "rats"
RATS Portable / TROLL	"l", "trl"
TSP Portable	"t", "tsp"

You must have EViews Enterprise Edition to access CEIC, Datastream, DRIBase, EcoWin, FactSet, FAME, Haver, IHS Magellan, and Moody's Economy.com databases.

The use of CEIC, Datastream, DRIPro Link, EcoWin, FactSet, FRED, IHS Magellan, Economy.com databases requires an active connection to the internet.

In addition, specific types may require installation of additional software. For details see, "Notes on Particular Formats" on page 333 in *User's Guide I*.

The following options may be required when connecting to a remote server:

s = *server_id*, server = *server_id*	Server name
u = *user*, username = *user*	Username
p = *pwd*, password = *pwd*	Password

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` (p. 317) or `dbcreate` (p. 320) to create a new database.

Cross-references

See Chapter 10. "EViews Databases," on page 303 of *User's Guide I* for a discussion of EViews databases.

See also `db` (p. 317) and `dbcreate` (p. 320).

dbpack

Object Container, Data, and File Commands

Pack an existing database.

Syntax

dbpack *[path\]db_name*

Follow the `dbpack` keyword by a database name. You may 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 330 of *User's Guide I* for additional discussion.

See also `dbrebuild` (p. 325).

dbrebuild	Object Container, Data, and File Commands

Rebuild an existing database.

Rebuild a seriously damaged database into a new database file.

Syntax

dbrebuild *[path\]source_name [path\]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

See "Maintaining the Database" on page 329 of *User's Guide I* for a discussion.

See also `dbpack` (p. 324).

dbrename	Object Container, Data, and File Commands

Rename an existing database.

`dbrename` renames all files associated with the specified database.

Syntax

dbrename *[path\]old_name [path\]new_name*

Follow the `dbrename` keyword with the current name of an existing database and the new name for the database.

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 10. "EViews Databases," on page 303 of *User's Guide I* for a discussion of EViews databases.

See db (p. 317) and dbcreate (p. 320). See also dbcopy (p. 318) and dbdelete (p. 322).

delete	Object Utility Commands

Deletes objects from a workfile or a database.

Syntax

delete*(options) arg1 [arg2 arg3 ...]*

Follow the keyword by a list of the names of any objects you wish to remove from the current workfile. Deleting does *not* remove objects that have been stored on disk in EViews database files.

Options

noerr	Do not error if the object doesn't exist.

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.

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", you may use the command:

```
delete temp*
```

To delete the objects CONS and INVEST from the database MACRO1, use:

```
delete macro1::cons macro1::invest
```

Cross-references

See "Object Commands" on page 9 for a discussion of working with objects.

See Chapter 10. "EViews Databases," on page 303 of *User's Guide I* for a discussion of EViews databases.

do	Command Actions

Execute without opening window.

Syntax

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 without opening a window, and prints the results to the disk file.

Cross-references

See "Object Commands" on page 9 for a discussion of working with objects.

Chapter 1. "Object View and Procedure Reference," on page 2 in the *Object Reference* provides a complete listing of the views of the various objects.

See also `show` (p. 433).

driconvert	Object Container, Data, and File Commands

Convert the entire DRI Basic Economics database into an 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

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 may include a path name to specify a database not in the default path.

Examples

```
dbcreate dribasic
driconvert dribasic
driconvert c:\mydata\dridbase
```

The first line creates a new (empty) database named DRIBASIC in the default directory. The second line copies all the data in the DRI Basic Economics database into in the DRIBASIC database. The last example copies the DRI data into the database DRIDBASE that is located in the C:\MYDATA directory.

Cross-references

See Chapter 10. "EViews Databases," on page 303 of the *User's Guide I* for a discussion of EViews databases.

See also `dbcreate` (p. 320) and `db` (p. 317).

exec	Programming Commands

Execute a program.

The `exec` command executes a program. The program may be located in memory or stored in a program file on disk.

Syntax

exec*(options) [path\]prog_name(prog_options) [%0 %1 …]*

If you wish to pass one or more options to the program, you should enclose them in parentheses immediately after the filename. If the program has arguments, you should list them after the filename.

EViews first checks to see if the specified program is in memory. If the program is not located, EViews then looks for the program on disk *in the EViews Add-ins directory*, or in the specified path. The program file should have a ".PRG" extension, which you need not specify in the *prog_name*.

Options

integer (default = 1)	Set maximum errors allowed before halting the program.
c	Run program file without opening a window for display of the program file.
verbose / quiet	Verbose mode in which messages will be sent to the status line at the bottom of the EViews window (slower execution), or quiet mode which suppresses workfile display updates (faster execution).

v / q	Same as [verbose / quiet].
ver4 / ver5	Execute program in [version 4 / version 5] compatibility mode.
this = *object_name*	Set the `_this` object for the executed program. If omitted, the executed program will inherit the `_this` object from the parent program, or from the current active workfile object when the `exec` command is issued from the command window.

Examples

```
exec rollreg
```

will run the program "Rollreg.prg" in the EViews add-in directory.

```
exec(this=graph01) recshade
```

will run the program "Recshade" in the EViews add-in directory, setting the `_this` object to GRAPH01.

```
exec(4) c:\myfiles\simul.prg(h=3) xhat
```

will run the program "Simul.prg" in the path "c:\myfiles\", with program option string "h = 3", the %0 argument set to "XHAT", and with the maximum error count set to 4.

Note that in contrast to the `run` command, `exec` will not stop executing a running program after returning from the executed program. For example if you have a program containing:

```
exec simul
print x
```

the `print` statement will be executed after running the "Simul.prg" program. If you replace exec with run, the program will stop after executing the commands in "Simul.prg".

Cross-references

See "Executing a Program" on page 108 of the *User's Guide I* and "The Active Object Keyword" on page 191 in the *Command and Programming Reference* for further details.

See also `run` (p. 427) and `include` (p. 661).

exit	Global Commands

Exit from EViews (close 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 workfiles, if desired, since all changes that you do not save to a disk file will be lost.

Syntax

exit

Cross-references

See also `close` (p. 289) and `save` (p. 428).

expand	Object Container, Data, and File Commands

Expand a workfile.

No longer supported. See the replacement command `pagestruct` (p. 408).

facbreak	Interactive Use Commands

Factor breakpoint test for stability.

Carries out a factor breakpoint test for parameter constancy.

Syntax

facbreak*(options) ser1 [ser2 ser3 ...] @ x1 x2 x3*

You must provide one or more series to be used as the factors with which to split the sample into categories. To specify more than one factor, separate the factors by a space. If the equation is specified by list and contains no linear terms, you may specify a subset of the regressors to be tested for a breakpoint after an "*@*" sign.

Options

p	Print the result of the test.

Examples

The commands

```
ls log(spot) c log(p_us) log(p_uk)
facbreak season
```

perform a regression of the log of SPOT on a constant, the log of P_US, and the log of P_UK, and employ a factor breakpoint test to determine whether the parameters are stable through the different values of SEASON.

To test whether only the constant term and the coefficient on the log of P_US are "stable" enter the commands:

```
facbreak season @ c log(p_us)
```

Cross-references

See "Factor Breakpoint Test" on page 155 of *User's Guide II* for further discussion.

See also `Equation::facbreak` (p. 76) and `Equation::rls` (p. 135) in the *Object Reference*.

factest	Interactive Use Commands

Specify and estimate a factor analysis model.

Syntax

factest*(method = arg, options) x1 [x2 x3...] [@***partial** *z1 z2 z3...]*

factest*(method = arg, options) matrix_name [[obs] [conditioning]] [@ name1 name2 name3...]*

where:

method = *arg* *(default =* "ml"*)*	Factor estimation method: "ml" (maximum likelihood), "gls" (generalized least squares), "ipf" (iterated principal factors), "pace" (non-iterative partitioned covariance matrix estimation), "pf" (principal factors), "uls" (unweighted least squares)

and the available options are specific to the factor estimation method (see "Factor Methods" on page 159 of the *Object Reference*).

The `factest` command allows you to estimated a factor analysis model without first declaring a factor object and then applying an estimation method. It provides a convenient method of interactively estimating transitory specifications that are not to be named and saved with the workfile.

Estimation of a factor analysis specification using `factest` only differs from estimation using a named factor and a factor estimation procedure (e.g., `Factor::ipf` (p. 172) in the *Object Reference*) in the use of the "method = " option and in the fact that the command results in an unnamed factor object.

Examples

The command:

```
factest(method=gls) g1
```

estimates a factor analysis model for the series in G1 using GLS. The result is an unnamed factor object. (Almost) equivalently, we may declaring and estimate the factor analysis object using the `Factor::gls` estimation method procedure

```
factor f1.gls g1
```

which differs only in the fact that the former yields an unnamed factor object and the latter saves the object F1 in the workfile.

The command:

```
factest(method=ml) group01 @partial ser1 ser2
```

estimates the factor model using the partial correlation for the series in GROUP01, conditional on the series SER1 and SER2. The command is equivalent to:

```
factor f2.ml group01 @partial ser1 ser2
```

except the latter names the factor object F2.

Cross-references

See Chapter 27. "Factor Analysis," on page 869 of *User's Guide II* for a general discussion of factor analysis. The various estimation methods are described in "Estimation Methods" on page 902 of *User's Guide II*.

See `Factor::gls` (p. 168), `Factor::ipf` (p. 172), `Factor::ml` (p. 181), `Factor::pf` (p. 190), and `Factor::uls` (p. 204) all in the *Object Reference*.

fetch	Object Container, Data, and File Commands

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 using the object in the database or using the databank file name.

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 an attached conversion method, EViews will use the method set by **Options/Date-Frequency** in the main menu. You can override the conversion method by specifying an explicit conversion method option.

Syntax

fetch*(options) object_list*

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 optionally fetch from individual databank files or search among registered databases.

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.

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 including identifying extensions.

Options

d = *db_name*	Fetch from specified database.
d	Fetch all registered databases in registry order.
i	Fetch from individual databank files.
link	Fetch as a database link.
notifyillegal	When in a program, report illegal EViews object names. By default, objects with illegal names are automatically renamed. (Has no effect in the command window.)

The following options are available for fetch of group objects:

g = *arg*	Group fetch options: "b" (fetch both group definition and series), "d" (fetch only the series in the group), "l" (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 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:

c = *arg*	Low to high conversion methods: "r" (constant match average), "d" (constant match sum), "q" (quadratic match average), "t" (quadratic match sum), "i" (linear match last), "c" (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 = *arg*	*High to low conversion methods removing NAs:* "a" (average of the nonmissing observations), "s" (sum of the nonmissing observations), "f" (first nonmissing observation), "l" (last nonmissing observation), "x" (maximum nonmissing observation), "m" (minimum nonmissing observation). *High to low conversion methods propagating NAs:* "an" or "na" (average, propagating missings), "sn" or "ns" (sum, propagating missings), "fn" or "nf" (first, propagating missings), "ln" or "nl" (last, propagating missings), "xn" or "nx" (maximum, propagating missings), "mn" or "nm" (minimum, propagating missings).

If no conversion method is specified, the series-specific or global default conversion method will be employed.

Examples

To fetch M1, GDP, and 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 destination 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, use the command:

```
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".

Use the "notifyillegal" option to display a dialog when fetching the series MYILLEG@LNAME that will suggest a valid name and give you to opportunity to name the object before it is inserted into a workfile:

```
pool2.fetch(notifyillegal) myilleg@lname
```

Cross-references

See Chapter 10. "EViews Databases," on page 303 of *User's Guide I* for a discussion of databases, databank files, and frequency conversion. Appendix A. "Wildcards," on page 683 describes the use of wildcard characters.

See also `store` (p. 444), and `copy` (p. 306).

See `Series::setconvert` (p. 522) in the *Object Reference* for information on default conversion settings.

fit	Interactive Use Commands

Computes static forecasts or 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.

(Note that we recommend that you instead use the equation proc `Equation::fit` since it explicitly specifies the equation of interest.)

Not available for equations estimated using ordered methods; use `Equation::makemodel` to create a model using the ordered equation results.

Syntax

fit*(options) yhat [y_se]*

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

d	In models with implicit dependent variables, forecast the entire expression rather than the normalized variable.
u	Substitute expressions for all auto-updating series in the equation.
g	Graph the fitted values together with the ±2 standard error bands.
e	Produce the forecast evaluation table.
i	Compute the fitted values 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 fitted values.
f = *arg (default* = "actual"*)*	Out-of-fit-sample fill behavior: "actual" (fill observations outside the fit sample with actual values for the fitted variable), "na" (fill observations outside the fit sample with missing values).
prompt	Force the dialog to appear from within a program.
p	Print view.

Examples

```
equation eq1.ls cons c cons(-1) inc inc(-1)
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 +/- 2 standard error bounds. The fifth line plots the actual series together with the error bounds.

```
equation eq2.binary(d=l) y c wage edu
fit yf
fit(i) xbeta
genr yhat = 1-@clogit(-xbeta)
```

The first line estimates a logit specification for Y with a conditional mean that depends 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.makemodel(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

To perform dynamic forecasting, use `forecast` (p. 337).

See Chapter 5. "Forecasting from an Equation," on page 111 of *User's Guide II* for a discussion of forecasting in EViews and Chapter 9. "Discrete and Limited Dependent Variable Models," on page 259 of *User's Guide II* for forecasting from binary, censored, truncated, and count models.

See "Forecasting" on page 606 of *User's Guide II* for a discussion of forecasting from sspace models.

See `Equation::forecast` (p. 80) and `Equation::fit` (p. 77) in the *Object Reference* for the equivalent object commands.

See `Equation::makemodel` (p. 112) and `Model::solve` (p. 398) in the *Object Reference* for forecasting from systems of equations or ordered equations.

forecast	Interactive Use Commands

Computes (*n*-period ahead) dynamic forecasts of the default equation.

`forecast` computes the forecast using the default equation 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.

(Note that we recommend that you instead use the equation proc `Equation::forecast` since it explicitly specifies the equation of interest.)

Syntax

forecast*(options) yhat [y_se]*

You should enter a name for the forecast series and, optionally, a name for the series containing the standard errors. Forecast standard errors are currently not available for binary or censored models. `forecast` is not available for models estimated using ordered methods.

Options

d	In models with implicit dependent variables, forecast the entire expression rather than the normalized variable.
u	Substitute expressions for all auto-updating series in the equation.
g	Graph the forecasts together with the ±2 standard error bands.
e	Produce the forecast evaluation table.
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.
b = *arg*	MA backcast method: "fa" (forecast available). Only for equations estimated with MA terms. This option is ignored if you specify the "s" (structural forecast) option. The default method uses the estimation sample.
f = *arg* (*default* = "actual")	Out-of-forecast-sample fill behavior: "actual" (fill observations outside the forecast sample with actual values for the fitted variable), "na" (fill observations outside the forecast sample with missing values).
prompt	Force the dialog to appear from within a program.
p	Print view.

Examples

The following lines:

```
smpl 1970q1 1990q4
equation eq1.ls con c con(-1) inc
smpl 1991q1 1995q4
forecast con_d
plot con_d
```

estimate a linear regression over the period 1970Q1–1990Q4, computes dynamic forecasts for the period 1991Q1–1995Q4, and plots the forecast as a line graph.

```
equation eq1.ls m1 gdp ar(1) ma(1)
forecast m1_bj bj_se
```

```
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.

Cross-references

To perform static forecasting, see `fit` (p. 335).

See Chapter 5. "Forecasting from an Equation," on page 111 of *User's Guide II* for a discussion of forecasting in EViews and Chapter 9. "Discrete and Limited Dependent Variable Models," on page 259 of *User's Guide II* for forecasting from binary, censored, truncated, and count models.

See "Forecasting" on page 606 of *User's Guide II* for a discussion of forecasting from sspace models.

See `Equation::forecast` (p. 80) and `Equation::fit` (p. 77) in the *Object Reference*, for the equivalent object commands.

See `Equation::makemodel` (p. 112) and `Model::solve` (p. 398) in the *Object Reference* for forecasting from systems of equations or ordered equations.

freeze | Command Actions

Creates graph, table, or text objects from a view.

Syntax

freeze*(options, name)* object_name.*view_command*

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 may provide a destination name for the object containing the frozen view in parentheses.

Options

mode = overwrite	Overwrites the object *name* if it already exists.

Examples

```
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(gra1) rates.line(m)
show gra1.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) gra1 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.

```
freeze(mode=overwrite, mygra) gra1 gra2 gra3
show mygra.align(2,1,1)
```

creates a graph object MYGRA that combines the three graph objects GRA1, GRA2 and GRA3, and displays MYGRA in two columns. If the object MYGRA already exists, it would be replaced by the new object.

Cross-references

See "Object Commands" on page 9 for discussion. See also Chapter 4. "Object Basics," on page 93 of *User's Guide I* for further discussion of objects and views of objects.

Freezing graph views is described in "Creating Graph Objects" on page 657 of *User's Guide I*.

frml	Object Creation Commands || Object Assignment Commands

Declare a series object with a formula for auto-updating, or specify a formula for an existing series.

Syntax

frml series_name = *series_expression*

frml series_name = **@clear**

Follow the `frml` keyword with a name for the object, and an assignment statement. The special keyword "@CLEAR" is used to return the auto-updating series to an ordinary numeric or alpha series.

Examples

To define an auto-updating numeric or alpha series, you must use the `frml` keyword prior to entering an assignment statement. The following example creates a series named LOW that uses a formula to compute its values.:

```
frml low = inc<=5000 or edu<13
```

The auto-updating series takes the value 1 if either INC is less than or equal to 5000 or EDU is less than 13, and 0 otherwise, and will be re-evaluated whenever INC or EDU change.

If FIRST_NAME and LAST_NAME are alpha series, then the formula declaration:

```
frml full_name = first_name + " " + last_name
```

creates an auto-updating alpha series FULL_NAME.

You may apply a `frml` to an existing series. The commands:

```
series z = 3
frml z =(x+y)/2
```

makes the previously created series Z an auto-updating series containing the average of series X and Y. Note that once a series is defined to be auto-updating, it may not be modified directly. Here, you may not edit Z, nor may you generate values into the series.

Note that the commands:

```
series z = 3
z = (x+y)/2
```

while similar, produce quite different results, since the absence of the `frml` keyword in the second example means that EViews will generate fixed values in the series instead of defining a formula to compute the series values. In this latter case, the values in the series Z are fixed, and may be modified.

One particularly useful feature of auto-updating series is the ability to reference series in databases. The command:

```
frml gdp = usdata::gdp
```

creates a series called GDP that obtains its values from the series GDP in the database USDATA. Similarly:

```
frml lgdp = log(usdata::gdp)
```

creates an auto-updating series that is the log of the values of GDP in the database USDATA.

To turn off auto-updating for a series or alpha, you should use the special expression "@CLEAR" in your `frml` assignment. The command:

```
frml z = @clear
```

sets the series to numeric or alpha value format, freezing the contents of the series at the current values.

Cross-references

See "Auto-Updating Series" on page 189 of *User's Guide I* for a discussion of updating series.

See also `Link::link` (p. 318) in the *Object Reference*.

genr	Object Creation Commands || Object Assignment Commands

Generate series.

Generate series or alphas.

Syntax

genr ser_name = *expression*

Examples

```
genr y = 3 + x
```

generates a numeric series that takes the values from the series X and adds 3.

```
genr full_name = first_name + last_name
```

creates an alpha series formed by concatenating the alpha series FIRST_NAME and LAST_NAME.

Cross-references

See Chapter 6. "Working with Data," on page 165 of *User's Guide I* for discussion of generating series data.

See `Series::series` (p. 520) and `Alpha::alpha` (p. 6) in the *Object Reference* for a discussion of the expressions allowed in `genr`.

glm	Interactive Use Commands

Estimate a Generalized Linear Model (GLM).

Syntax

glm*(options) spec*

List the `glm` keyword, followed by the dependent variable and a list of the explanatory variables, or an explicit linear expression.

If you enter an explicit *linear* specification such as "Y = C(1) + C(2)*X", the response variable will be taken to be the variable on the left-hand side of the equality ("Y") and the linear predictor will be taken from the right-hand side of the expression ("C(1) + C(2)*X").

Offsets may be entered directly in an explicit linear expression or they may be entered as using the "offset = " option.

Specification Options

family = *arg* *(default* = "normal"*)*	Distribution family: Normal ("normal"), Poisson ("poisson"), Binomial Count ("binomial"), Binomial Proportion ("binprop"), Negative Binomial ("negbin"), Gamma ("gamma"), Inverse Gaussian ("igauss"), Exponential Mean ("emean)", Power Mean ("pmean"), Binomial Squared ("binsq"). The Binomial Count, Binomial Proportion, Negative Binomial, and Power Mean families all require specification of a distribution parameter:
n = *arg (default* = 1*)*	Number of trials for Binomial Count ("family = binomial") or Binomial Proportions ("family = binprop") families.
fparam = *arg*	Family parameter value for Negative Binomial ("family = negbin") and Power Mean ("family = pmean") families.
link = *arg* *(default* = "identity"*)*	Link function: Identity ("identity"), Log ("log"), Log Compliment ("logc"), Logit ("logit"), Probit ("probit"), Log-log ("loglog"), Complementary Log-log ("cloglog"), Reciprocal ("recip"), Power ("power"), Box-Cox ("boxcox"), Power Odds Ratio ("opow"), Box-Cox Odds Ratio ("obox"). The Power, Box-Cox, Power Odds Ratio, and Box-Cox Odds Ratio links all require specification of a link parameter specified using "lparam = ".
lparam = *arg*	Link parameter for Power ("link = power"), Box-Cox ("link = boxcox"), Power Odds Ratio ("link = opow") and Box-Cox Odds Ratio ("link = obox") link functions.
offset = *arg*	Offset terms.
disp = *arg*	Dispersion estimator: Pearson χ^2 statistic ("pearson"), deviance statistic ("deviance"), unit ("unit"), user-specified ("user"). The default is family specific: "unit" for Binomial Count, Binomial Proportion, Negative Binomial, and Poison, and "pearson" for all others. The "deviance" option is only offered for families in the exponential family of distributions (Normal, Poisson, Binomial Count, Binomial Proportion, Negative Binomial, Gamma, Inverse Gaussian).
dispval = *arg*	User-dispersion value (if "disp = user").
fwgts = *arg*	Frequency weights.
w = *arg*	Weight series or expression.

wtype = *arg* *(default* = "istdev*")*	Weight specification type: inverse standard deviation ("istdev"), inverse variance ("ivar"), standard deviation ("stdev"), variance ("var").
wscale = *arg*	Weight scaling: EViews default ("eviews"), average ("avg"), none ("none"). The default setting depends upon the weight type: "eviews" if "wtype = istdev", "avg" for all others.

In addition to the specification options, there are options for estimation and covariance calculation.

Additional Options

estmeth = *arg* *(default* = "marquardt")	Estimation algorithm: Quadratic Hill Climbing ("marquardt"), Newton-Raphson ("newton"), IRLS - Fisher Scoring ("irls"), BHHH ("bhhh").
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. The criterion will be set to the nearest value between 1e-24 and 0.2.
s	Use the current coefficient values in estimator coefficient vector as starting values (see also `param` (p. 413) in the *Command and Programming Reference*).
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.
preiter = *arg* *(default* = 0*)*	Number of IRLS pre-iterations to refine starting values (only available for non-IRLS estimation).
cov = *arg* *(default* = "invinfo*")*	Coefficient covariance method: Inverse information matrix ("invinfo"), Huber-White sandwich ("white" or "sandwich"), Newey-West HAC ("hac").

covinfo = *arg* *(default* = "default"*)*	Information matrix method: Estimation and covariance method specific default ("default"), Hessian, outer-product of gradients ("opg"), "hac" (Newey-West HAC), The default method depends on the estimation and covariance method. For "cov = invinfo", the default information matrix is computed to match the setting specified in "estmeth = ". IRLS ("irls") will default to the expected hessian, while the remaining methods will default to the observed Hessian.
nodf	Do not degree-of-freedom correct the coefficient covariance estimate.
covlag = *arg* *(default* = 1*)*	Whitening lag specification: *integer* (user-specified lag value), "a" (automatic selection). Applicable where "cov = hac".
covinfosel = *arg* *(default* = "aic"*)*	Information criterion for automatic selection: "aic" (Akaike), "sic" (Schwarz), "hqc" (Hannan-Quinn) (if "lag = a"). For settings where "cov = hac, covlag = a".
covmaxlag = *integer*	Maximum lag-length for automatic selection *(optional)* (if "lag = a"). The default is an observation-based maximum of $T^{1/3}$. For settings where "cov = hac, covlag = a".
covkern = *arg* *(default* = "bart"*)*	Kernel shape: "none" (no kernel), "bart" (Bartlett, *default*), "bohman" (Bohman), "daniell" (Daniel), "parzen" (Parzen), "parzriesz" (Parzen-Riesz), "parzgeo" (Parzen-Geometric), "parzcauchy" (Parzen-Cauchy), "quadspec" (Quadratic Spectral), "trunc" (Truncated), "thamm" (Tukey-Hamming), "thann" (Tukey-Hanning), "tparz" (Tukey-Parzen). For settings where "cov = hac".
covbw = *arg* *(default* = "fixednw"*)*	Kernel Bandwidth: "fixednw" (Newey-West fixed), "andrews" (Andrews automatic), "neweywest" (Newey-West automatic), *number* (User-specified bandwidth). For settings where "cov = hac" and "covkern = " is specified.
covnwlag = *integer*	Newey-West lag-selection parameter for use in nonparametric kernel bandwidth selection (if "covbw = neweywest"). For settings where "cov = hac" and "covkern = " is specified.

covbwoffset = *number*	Apply offset to automatically selected bandwidth. For settings where "cov = hac", "covkern = " is specified, and "covbw = " is not user-specified.
covbwint	Use integer portion of kernel bandwidth. For settings where "cov = hac" and "covkern = " is specified.
coef = *arg*	Specify the name of the coefficient vector (if specified by list); the default behavior is to use the "C" coefficient vector.
prompt	Force the dialog to appear from within a program.
p	Print results.

Examples

```
glm(link=log) numb c ip feb
```

estimates a normal regression model with exponential mean.

```
glm(family=binomial, n=total) disease c snore
```

estimates a binomial count model with default logit link where TOTAL contains the number of binomial trials and DISEASE is the number of binomial successes. The specification

```
glm(family=binprop, n=total, cov=white, nodf) disease/total c
    snore
```

estimates the same specification in proportion form, and computes the coefficient covariance using the White-Huber sandwich with no df correction.

```
glm(family=binprop, disp=pearson) prate mprate log(totemp)
    log(totemp)^2 age age^2 sole
```

estimates a binomial proportions model with default logit link, but computes the coefficient covariance using the GLM scaled covariance with dispersion computed using the Pearson Chi-square statistic.

```
glm(family=binprop, link=probit, cov=white) prate mprate
    log(totemp) log(totemp)^2 age age^2 sole
```

estimates the same basic specification, but with a probit link and Huber-White standard errors.

```
glm(family=poisson, offset=log(pyears)) los hmo white type2 type3 c
```

estimates a Poisson specification with an offset term LOG(PYEARS).

Cross-references

See Chapter 10. "Generalized Linear Models," beginning on page 319 of *User's Guide II* for discussion.

See `Equation::glm` (p. 82) in the *Object Reference* for the equivalent equation object command.

gmm	Interactive Use Commands

Estimation by generalized method of moments (GMM).

Syntax

gmm*(options) y x1 [x2 x3...] @ z1 [z2 z3...]*

gmm*(options) specification @ z1 [z2 z3...]*

Follow the name of the dependent variable by a list of regressors, followed by the "@" symbol, and a list of instrumental variables which are orthogonal to the residuals. Alternatively, you can specify an expression using coefficients, an "@" symbol, and a list of instrumental variables. There must be at least as many instrumental variables as there are coefficients to be estimated.

In panel settings, you may specify dynamic instruments corresponding to predetermined variables. To specify a dynamic instrument, you should tag the instrument using "@DYN", as in "@DYN(X)". By default, EViews will use a set of period-specific instruments corresponding to lags from -2 to "-infinity". You may also specify a restricted lag range using arguments in the "@DYN" tag. For example, to use lags from-5 to "-infinity" you may enter "@DYN(X, -5)"; to specify lags from -2 to -6, use "@DYN(X, -2, -6)" or "@DYN(X, -6, -2)".

Note that dynamic instrument specifications may easily generate excessively large numbers of instruments.

Options

Non-Panel GMM Options

Basic GMM Options

nocinst	Do not include automatically a constant as an instrument.
method = *keyword*	Set the weight updating method. *keyword* should be one of the following: "nstep" (N-Step Iterative, or Sequential N-Step Iterative, default), "converge" (Iterate to Convergence or Sequential Iterate to Convergence), "simul" (Simultaneous Iterate to Convergence), "oneplusone" (One-Step Weights Plus One Iteration), or "cue" (Continuously Updating".
gmmiter = *integer*	Number of weight iterations. Only applicable if the "method = nstep" option is set.
w = *arg*	Weight series or expression.
wtype = *arg* *(default* = "istdev*")*	Weight specification type: inverse standard deviation ("istdev"), inverse variance ("ivar"), standard deviation ("stdev"), variance ("var").
wscale = *arg*	Weight scaling: EViews default ("eviews"), average ("avg"), none ("none"). The default setting depends upon the weight type: "eviews" if "wtype = istdev", "avg" for all others.
m = *integer*	Maximum number of iterations.
c = *number*	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients. The criterion will be set to the nearest value between 1e-24 and 0.2.
l = *number*	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.

coef = *arg*	Specify the name of the coefficient vector (if specified by list); the default behavior is to use the "C" coefficient vector.
prompt	Force the dialog to appear from within a program.
p	Print results.

Estimation Weighting Matrix Options

instwgt = *keyword*	Set the estimation weighting matrix type. *Keyword* should be one of the following: "tsls" (two-stage least squares), "white" (White diagonal matrix), "hac" (Newey-West HAC, default) or "user" (user defined).
instwgtmat = *name*	Set the name of the user-defined estimation weighting matrix. Only applicable if the "instwgt = user" option is set.
instlag = *arg* *(default* = 1*)*	Whitening Lag specification: *integer* (user-specified lag value), "a" (automatic selection).
instinfo = *arg* *(default* = "aic"*)*	Information criterion for automatic whitening lag selection: "aic" (Akaike), "sic" (Schwarz), "hqc" (Hannan-Quinn) (if "instlag = a").
instmaxlag = *integer*	Maximum lag-length for automatic selection *(optional)* (if "instlag = a"). The default is an observation-based maximum of $T^{1/3}$.
instkern = *arg* *(default* = "bart"*)*	Kernel shape: "none" (no kernel), "bart" (Bartlett, *default*), "bohman" (Bohman), "daniell" (Daniell), "parzen" (Parzen), "parzriesz" (Parzen-Riesz), "parzgeo" (Parzen-Geometric), "parzcauchy" (Parzen-Cauchy), "quadspec" (Quadratic Spectral), "trunc" (Truncated), "thamm" (Tukey-Hamming), "thann" (Tukey-Hanning), "tparz" (Tukey-Parzen).
instbw = *arg* *(default* = "fixednw"*)*	Kernel Bandwidth: "fixednw" (Newey-West fixed), "andrews" (Andrews automatic), "neweywest" (Newey-West automatic), *number* (User-specified bandwidth).
instnwlag = *integer*	Newey-West lag-selection parameter for use in nonparametric bandwidth selection (if "instbw = neweywest").
instbwint	Use integer portion of bandwidth.

Covariance Options

cov = *keyword*	Covariance weighting matrix type *(optional)*: “updated” (estimation updated), “tsls” (two-stage least squares), “white” (White diagonal matrix), “hac” (Newey-West HAC), “wind” (Windmeijer) or “user” (user defined). The default is to use the estimation weighting matrix.
nodf	Do not perform degree of freedom corrections in computing coefficient covariance matrix. The default is to use degree of freedom corrections.
covwgtmat = *name*	Set the name of the user-definied covariance weighting matrix. Only applicable if the “covwgt = user” option is set.
covlag = *arg* *(default* = 1*)*	Whitening lag specification: *integer* (user-specified lag value), “a” (automatic selection).
covinfo = *arg* *(default* = ”aic”*)*	Information criterion for automatic selection: “aic” (Akaike), “sic” (Schwarz), “hqc” (Hannan-Quinn) (if “lag = a”).
covmaxlag = *integer*	Maximum lag-length for automatic selection *(optional)* (if “lag = a”). The default is an observation-based maximum of $T^{1/3}$.
covkern = *arg* *(default* = “bart”*)*	Kernel shape: “none” (no kernel), “bart” (Bartlett, *default*), “bohman” (Bohman), “daniell” (Daniel), “parzen” (Parzen), “parzriesz” (Parzen-Riesz), “parzgeo” (Parzen-Geometric), “parzcauchy” (Parzen-Cauchy), “quadspec” (Quadratic Spectral), “trunc” (Truncated), “thamm” (Tukey-Hamming), “thann” (Tukey-Hanning), “tparz” (Tukey-Parzen).
covbw = *arg* *(default* = “fixednw”*)*	Kernel Bandwidth: “fixednw” (Newey-West fixed), “andrews” (Andrews automatic), “neweywest” (Newey-West automatic), *number* (User-specified bandwidth).
covnwlag = *integer*	Newey-West lag-selection parameter for use in nonparametric kernel bandwidth selection (if “covbw = neweywest”).
covbwint	Use integer portion of bandwidth.

Panel GMM Options

cx = *arg*	Cross-section effects method: (default) none, fixed effects estimation ("cx = f"), first-difference estimation ("cx = fd"), orthogonal deviation estimation ("cx = od")
per = *arg*	Period effects method: (default) none, fixed effects estimation ("per = f").
levelper	Period dummies always specified in levels (even if one of the transformation methods is used, "cx = fd" or "cx = od").
wgt = *arg*	GLS weighting: (default) none, cross-section system weights ("wgt = cxsur"), period system weights ("wgt = persur"), cross-section diagonal weighs ("wgt = cxdiag"), period diagonal weights ("wgt = perdiag").
gmm = *arg*	GMM weighting: 2SLS ("gmm = 2sls"), White period system covariances (Arellano-Bond 2-step/n-step) ("gmm = perwhite"), White cross-section system ("gmm = cxwhite"), White diagonal ("gmm = stackedwhite"), Period system ("gmm = persur"), Cross-section system ("gmm = cxsur"), Period heteroskedastic ("cov = perdiag"), Cross-section heteroskedastic ("gmm = cxdiag"). By default, uses the identity matrix unless estimated with first difference transformation ("cx = fd"), in which case, uses (Arellano-Bond 1-step) difference weighting matrix. In this latter case, you should specify 2SLS weights ("gmm = 2sls") for Anderson-Hsiao estimation.
cov = *arg*	Coefficient covariance method: *(default)* ordinary, White cross-section system robust ("cov = cxwhite"), White period system robust ("cov = perwhite"), White heteroskedasticity robust ("cov = stackedwhite"), Cross-section system robust/PCSE ("cov = cxsur"), Period system robust/PCSE ("cov = persur"), Cross-section heteroskedasticity robust/PCSE ("cov = cxdiag"), Period heteroskedasticity robust ("cov = perdiag").
keepwgts	Keep full set of GLS/GMM weights used in estimation with object, if applicable (by default, only weights which take up little memory are saved).
coef = *arg*	Specify the name of the coefficient vector (if specified by list); the default behavior is to use the "C" coefficient vector.

iter = *arg* *(default* = "onec"*)*	Iteration control for GLS and GMM weighting specifications: perform one weight iteration, then iterate coefficients to convergence ("iter = onec"), iterate weights and coefficients simultaneously to convergence ("iter = sim"), iterate weights and coefficients sequentially to convergence ("iter = seq"), perform one weight iteration, then one coefficient step ("iter = oneb").
m = *integer*	Maximum number of iterations.
c = *number*	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients. The criterion will be set to the nearest value between 1e-24 and 0.2.
l = *number*	Set maximum number of iterations on the first-stage iteration to get the one-step weighting matrix.
unbalsur	Compute SUR factorization in unbalanced data using the subset of available observations for a cluster.
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.

Note that some options are only available for a subset of specifications.

Examples

In a non-panel workfile, we may estimate equations using the standard GMM options. The specification:

```
gmm(instwgt=white,gmmiter=2,nodf) cons c y y(-1) w @ c p(-1) k(-1)
    x(-1) tm wg g t
```

estimates the Klein equation for consumption using GMM with a White diagonal weighting matrix (two steps and no degree of freedom correction). The command:

```
gmm(method=cue,instwgt=hac,instlag=1,instkern=thann,
    instbw=andrews,nodf,deriv=aa) i c y y(-1) k(-1) @  c p(-1) k(-
    1) x(-1) tm wg g t
```

estimates the Klein equation for investment using a Newey-West HAC weighting matrix, with pre-whitening with 1 lag, a Tukey-Hanning kernel and the Andrews automatic bandwidth routine. The estimation is performed using continuously updating weight iterations.

When working with a workfile that has a panel structure, you may use the panel equation estimation options. The command

```
gmm(cx=fd, per=f) dj dj(-1) @ @dyn(dj)
```

estimates an Arellano-Bond "1-step" estimator with differencing of the dependent variable DJ, period fixed effects, and dynamic instruments constructed using DJ with observation specific lags from period $t-2$ to 1.

To perform the "2-step" version of this estimator, you may use:

```
gmm(cx=fd, per=f, gmm=perwhite, iter=oneb) dj dj(-1) @ @dyn(dj)
```

where the combination of the options "gmm = perwhite" and (the default) "iter = oneb" instructs EViews to estimate the model with the difference weights, to use the estimates to form period covariance GMM weights, and then re-estimate the model.

You may iterate the GMM weights to convergence using:

```
gmm(cx=fd, per=f, gmm=perwhite, iter=seq) dj dj(-1) @ @dyn(dj)
```

Alternately:

```
gmm(cx=od, gmm=perwhite, iter=oneb) dj dj(-1) x y @ @dyn(dj,-2,-6)
    x(-1) y(-1)
```

estimates an Arellano-Bond "2-step" equation using orthogonal deviations of the dependent variable, dynamic instruments constructed from DJ from period $t-6$ to $t-2$, and ordinary instruments X(-1) and Y(-1).

Cross-references

See "Generalized Method of Moments" on page 67 and Chapter 23. "Panel Estimation," on page 759 of *User's Guide II* for discussion of the various GMM estimation techniques.

See also `tsls` (p. 453).

See `Equation::gmm` (p. 87) in the *Object Reference* for the equivalent equation object command.

heckit	Interactive Use Commands

Estimate a selection equation using the Heckman ML or 2-step method.

Syntax

heckit(*options*) *response_eqn @ selection_eqn*

The response equation should be the dependent variable followed by a list of regressors. The selection equation should be a binary dependent variable followed by a list of regressors.

Options

General Options

2step	Use the Heckman 2-step estimation method. Note that this option is incompatible with the maximum likelihood options below.
coef = *arg*	Specify the name of the coefficient vector (if specified by list); the default behavior is to use the "C" coefficient vector.
prompt	Force the dialog to appear from within a program.
p	Print the estimation results.

ML Options

Note these options are not available if the "2step" option, above, is used.

cov = *arg*	Covariance matrix choice. *arg* may be any of the following: "opg" (outer product of gradients), "hessian" (observed Hessian matrix), or "white" (Huber/White sandwich). Note if this option is not used, EViews will default to "opg" when the BHHH optimizer is used, and "hessian" otherwise.
m = *integer*	Set maximum number of iterations.
c = *number*	Set convergence criteria.
deriv = n	Use numerical derivatives.
showopts / -showopts	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
s = *number*	Scale EViews' starting values by *number*.
r	Use Newton-Raphson optimizer.
b	Use BHHH optimizer.

Examples

```
wfopen http://www.stern.nyu.edu/~wgreene/Text/Edition7/TableF5-
    1.txt
heckit ww c ax ax^2 we cit @ lfp c wa wa^2 faminc we (k618+kl6)>0
heckit(2step) ww c ax ax^2 we cit @ lfp c wa wa^2 faminc we
    (k618+kl6)>0
```

This example replicates the Heckman Selection example given in Greene (2008, page 888), which uses data from the Mroz (1987) study to estimate a selection model. The first line of this example downloads the data set, the second line creates an equation object and estimates it using the default maximum likelihood estimation method of Heckman Selection, which replicates the first pane of Table 24.3 in Greene. The third line estimates the same model, using the two-step approach, which replicates the second pane of Table 24.3.

Cross-references

hconvert	Object Container, Data, and File Commands

Convert an entire Haver Analytics Database into an EViews database.

Syntax

hconvert *haver_path 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 10. "EViews Databases," on page 303 of *User's Guide I* for a discussion of EViews and Haver databases.

See also `dbcreate` (p. 320), `db` (p. 317), `hfetch` (p. 356) and `hlabel` (p. 357).

hfetch	Object Container, Data, and File Commands

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

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 status line. 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

See also Chapter 10. "EViews Databases," on page 303 of *User's Guide I* for a discussion of EViews and Haver databases. Additional information on EViews frequency conversion is provided in "Frequency Conversion" on page 151 of *User's Guide I*.

See also `dbcreate` (p. 320), `db` (p. 317), `hconvert` (p. 355) and `hlabel` (p. 357).

hist	Interactive Use Commands

Histogram and descriptive statistics of a series.

The `hist` command computes descriptive statistics and displays a histogram for the series.

Syntax

hist*(options) series_name*

Options

p	Print the histogram.

Examples

```
hist lwage
```

Displays the histogram and descriptive statistics of LWAGE.

Cross-references

See "Histogram and Stats" on page 358 of *User's Guide I* for a discussion of the descriptive statistics reported in the histogram view.

See `distplot` (p. 813) in the *Object Reference* for a more full-featured and customizable method of constructing histograms, and `Series::hist` (p. 502) in the *Object Reference* for the equivalent series object view command.

hlabel

Object Container, Data, and File Commands

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

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 10. "EViews Databases," on page 303 of *User's Guide I* for a discussion of EViews and Haver databases.

See also `hfetch` (p. 356) and `hconvert` (p. 355).

hpf	Interactive Use Commands

Smooth a series using the Hodrick-Prescott filter.

Syntax

hpf*(options) series_name filtered_name [@ cycle_name]*

Smoothing Options

The degree of smoothing may be specified as an option. You may specify the smoothing as a value, or using a power rule:

lambda = *arg*	Set smoothing parameter value to *arg*; a larger number results in greater smoothing.
power = *arg* *(default = 2)*	Set smoothing parameter value using the frequency power rule of Ravn and Uhlig (2002) (the number of periods per year divided by 4, raised to the power *arg*, and multiplied by 1600). Hodrick and Prescott recommend the value 2; Ravn and Uhlig recommend the value 4.

If no smoothing option is specified, EViews will use the power rule with a value of 2.

Other Options

prompt	Force the dialog to appear from within a program.
p	Print the graph of the smoothed series and the original series.

Examples

```
hpf(lambda=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 453 of *User's Guide I* for details.

See `Series::hpf` (p. 503) of the *Object Reference* for the equivalent series object command.

import	Object Container, Data, and File Commands

Imports data from a foreign file or a previously saved workfile into the current default workfile. The import command lets you perform four different types of data importing: dated reads, match-merge reads, sequential reads, and appends.

Dated imports can only be performed if the destination workfile is a dated workfile. You must specify the date structure of the source data as part of the import command. EViews will then match the date structure of the source with that of the destination, and perform frequency conversion if necessary.

Match-merge imports require both a source ID series and a destination ID series. EViews will read the source data into the destination workfile based upon matched values of the two ID series.

Sequential imports will read the source data into the destination workfile by matching the first observation of the source file to the first observation in the destination workfile's current sample, then the second observation of the source with the second observation in the destination's current sample, and so on.

Appended imports simply append the source data to the end of the destination workfile.

Syntax

The general form of the **import** command is:

import*([type=], options) source_description import_specification [***@smpl** *smpl_string] [***@genr** *genr_string] [***@rename** *rename_string]*

where the syntax for *import_specification* depends on whether the read is a dated ("Dated Imports" on page 361), match-merge ("Match-Merge Import" on page 362), sequential ("Sequential Read" on page 364), or appended import ("Appended Read" on page 365).

- *Source_description* should contain a description of the file from which the data is to be imported is specified as the first argument following the import command and its options. The specification of the description is usually just the path and file name of the file, however you can also specify more precise information. See `wfopen` (p. 472) for more details on the specification of *source_description*.
- The optional "type=" option may be used to specify a source type. For the most part, you should not need to specify a "type=" option as EViews will automatically deter-

mine the type from the filename. The following table summaries the various source formats and along with the corresponding "type = " keywords:

	Option Keywords
Access	"access"
Aremos-TSD	"a", "aremos", "tsd"
Binary	"binary"
dBASE	"dbase"
Excel (through 2003)	"excel"
Excel 2007 (xml)	"excelxml"
EViews Workfile	---
Gauss Dataset	"gauss"
GiveWin/PcGive	"g", "give"
HTML	"html"
Lotus 1-2-3	"lotus"
ODBC Dsn File	"dsn"
ODBC Query File	"msquery"
ODBC Data Source	"odbc"
MicroTSP Workfile	"dos", "microtsp"
MicroTSP Mac Workfile	"mac"
RATS 4.x	"r", "rats"
RATS Portable / TROLL	"l", "trl"
SAS Program	"sasprog"
SAS Transport	"sasxport"
SPSS	"spss"
SPSS Portable	"spssport"
Stata	"stata"
Text / ASCII	"text"
TSP Portable	"t", "tsp"

- The optional **@smpl** keyword may be used to specify that data is only imported for the observations specified by *smpl_string*. By default, the import will use all of the observations in the workfile. If **@smpl** is included, but no *smpl_string* is included, then the current default workfile sample is used.
- The optional **@genr** keyword may be used to generate a new series in the workfile as part of the import procedure. *genr_string* may be any valid series creation expression, and can be an expression based upon one of the imported series. See genr (p. 342)

for information on valid expressions.

- The optional **@rename** keyword may be used to rename some of the imported series where *rename_string* contains pairs of old object names followed by the new names. See `rename` (p. 421) for additional discussion.

In the remainder of this discussion, we examine each of the different import types in greater depth.

Dated Imports

The syntax for a dated import command is:

import*([type =], options) source_description* **@freq** *frequency start_date [***@smpl** *smpl_string] [***@genr** *genr_string] [***@rename** *rename_string]*

The *import_specification* consists of the **@freq** keyword followed by a *frequency* argument specifying the frequency of the source data and a *start_date* to specify the starting date of the source data. See `wfcreate` (p. 467) for details on the forms that *frequency* may take.

Basic Options

resize	Extend the destination workfile (if necessary) to include the entire range of the source data.
mode = *arg* (*default* = "o")	Set the behavior for handling name conflicts when an imported series already exists in the destination workfile. *arg* can be "o" (Completely replace existing series with source series. Note that values outside of the range of the source data will be overwritten with NAs), "u" (Overwrite existing series only for values within the range of the source data. Destination values outside of the source range will be unchanged), "ms" (Overwrite existing series, unless source value is an NA, in which case keep destination values), "md" (Only overwrite NA values in destination series), "r" (rename any conflicts), or "p" (do not import any series which have a name conflict).
page = *page_name*	Optional name for the page into which the data should be imported.
prompt	Force the dialog to appear from within a program.

Frequency Conversion Options

When importing data from a lower frequency source into a higher frequency destination:

c = *arg*	Low to high conversion methods: "r" (constant match average), "d" (constant match sum), "q" (quadratic match average), "t" (quadratic match sum), "i" (linear match last), "c" (cubic match last).

When importing data from a higher frequency source into a lower frequency destination:

c = *arg*	*High to low conversion methods removing NAs:* "a" (average of the nonmissing observations), "s" (sum of the nonmissing observations), "f" (first nonmissing observation), "l" (last nonmissing observation), "x" (maximum nonmissing observation), "m" (minimum nonmissing observation). *High to low conversion methods propagating NAs*: "an" or "na" (average, propagating missings), "sn" or "ns" (sum, propagating missings), "fn" or "nf" (first, propagating missings), "ln" or "nl" (last, propagating missings), "xn" or "nx" (maximum, propagating missings), "mn" or "nm" (minimum, propagating missings).

Examples

```
import c:\temp\quarterly.xls @freq q 1990
```

will import the file QUARTERLY.XLS into the current default workfile. The source file has a quarterly frequency, starting at 1990.

```
import(c=s) c:\temp\quarterly.xls range="GDP_SHEET" @freq q 1990
    @rename gdp_per_capita gdp
```

will import from same file, but instead will use the data on the Excel sheet called "GDP_SHEET", and will rename the series GDP_PER_CAPITA to GDP. A frequency conversion method using the sum of the nonmissing observations is used rather than the default average.

```
import(mode=p) c:\temp\annual.txt @freq a 1990 @smpl 1994 1996
```

will import data from a text file called annual.txt, into the current default workfile. Any data in the text file that already exists in the destination workfile will be ignored, and for the remaining series, only the dates between 1994 and 1996 will be imported.

Match-Merge Import

Syntax

import*(options) source_description* **@id** *id* **@destid** *destid [*@**smpl** *smpl_string] [*@**genr** *genr_string] [*@**rename** *rename_string]*

The *import_specification* consists of the **@id** keyword and at least one ID series in the source file, followed by the **@destid** keyword and at least one ID series in the destination workfile. The two sets of ID series should be compatible, in that they should contain a subset of identical values that provide information on how observations from the two files should be matched. If one of the ID series is a date series, you should surround it with the **@date( )** keyword.

Options

resize	Extend the destination workfile (if necessary) to include the entire range of the source data.
mode = *arg* *(default* = "o"*)*	Set the behavior for handling name conflicts when an imported series already exists in the destination workfile. *arg* can be "o" (Completely replace existing series with source series. Note that values outside of the range of the source data will be overwritten with NAs), "u" (Overwrite existing series only for values within the range of the source data. Destination values outside of the source range will be unchanged), "ms" (Overwrite existing series, unless source value is an NA, in which case keep destination values), "md" (Only overwrite NA values in destination series), "r" (rename any conflicts), or "p" (do not import any series which have a name conflict).
nacat	Treat "NA" values as a category when copying using general match merge operations.
propnas	Propogate NAs / partial periods evaluate to NAs when converting.
c = *arg*	Set the match merge contraction method. If you are importing a numeric source series by general match merge, the argument can be one of: "mean", "med" (median), "max", "min", "sum", "sumsq" (sum-of squares), "var" (variance), "sd" (standard deviation), "skew" (skewness), "kurt" (kurtosis), "quant" (quantile, used with "quant = " option), "obs" (number of observations), "nas" (number of NA values), "first" (first observation in group), "last" (last observation in group), unique" (single unique group value, if present), "none" (disallow contractions). If importing an alpha series, only the non-summary methods "max", "min", "obs", "nas", first", "last", "unique" and "none" are supported. For importing of numeric series, the default contraction method is "c = mean"; for copying of alpha series, the default is "c = unique".
page = *page_name*	Optional name for the page into which the data should be imported.
prompt	Force the dialog to appear from within a program.

Most of the conversion options should be self-explanatory. As for the others: "first" and "last" give the first and last non-missing observed for a given group ID; "obs" provides the number of non-missing values for a given group; "nas" reports the number of NAs in the

group; "unique" will provide the value in the source series if it is the identical for all observations in the group, and will return NA otherwise; "none" will cause the import to fail if there are multiple observations in any group—this setting may be used if you wish to prohibit all contractions.

Examples

```
import(c=max, type=excel) c:\data\stateunemp.xls @id states
    @destid states
```

will import the file STATEUNEMP.XLS using the ID series STATES in both files as the match merge criteria. The maximum value is used as a contraction method. Note that although the "type = excel" option was used, it was not necessary since EViews will automatically detect the file type based on the file's extension (.xls).

```
import c:\data\stategdp.txt colhead=3 delim=comma @id states
    @date(year) @destid states @date
```

will import the file STATEGDP.TXT, specifying that there are three lines of column headers, and the delimiter for the text file is a comma. The series STATES is used as an ID series in both files, along with a date series (YEAR for the source file, and the default EViews date series, @DATE, for the destination workfile). Note that this type of import, with both a cross-section ID and a date ID is most commonly employed for importing data into panel workfiles.

```
import c:\data\cagdp.xls @id states @date(year) @destid states
    @date @genr states="CA"
```

will import the file CAGDP.XLS into the current workfile. In this particular case the source file is a time series file for a single state, California. Since the importing is being done into a panel workfile, the `@genr` keyword is used to generate a series containing the state identifier, CA, which is then used as the source ID.

Sequential Read

Syntax

import*(options) source_description [@***smpl** *smpl_string] [@***genr** *genr_string] [@***rename** *rename_string]*

No *import_specification* is required for a sequential read.

Options

resize	Extend the destination workfile (*if necessary*) to include the entire range of the source data.
mode = *arg* *(default* = "o"*)*	Set the behavior for handling name conflicts when an imported series already exists in the destination workfile. *arg* can be "o" (Completely replace existing series with source series. Note that values outside of the range of the source data will be overwritten with NAs), "u" (Overwrite existing series only for values within the range of the source data. Destination values outside of the source range will be unchanged), "ms" (Overwrite existing series, unless source value is an NA, in which case keep destination values), "md" (Only overwrite NA values in destination series), "r" (rename any conflicts), or "p" (do not import any series which have a name conflict).
page = *page_name*	Optional name for the page into which the data should be imported.
prompt	Force the dialog to appear from within a program.

Examples

```
import(resize) sales.dta @smpl @all
```

will import the Stata file SALES.DTA into the current workfile, using the entire workfile range. If the sales.dta file contains more observations that the current workfile, the current workfile is resized to accommodate the extra rows of data.

Appended Read

Syntax

import*(options) source_description* **@append** *[***@genr** *genr_string] [***@rename** *rename_string]*

The *import_specification* consists of the **@append** keyword. Note that the **@smpl** keyword is not supported for appended import.

Options

mode = *arg* *(default* = "o"*)*	Set the behavior for handling name conflicts when an imported series already exists in the destination workfile. *arg* can be "o" (Completely replace existing series with source series. Note that values outside of the range of the source data will be overwritten with NAs), "r" (rename any conflicts), or "p" (do not import any series which have a name conflict).

page = *page_name*	Optional name for the page into which the data should be imported.
prompt	Force the dialog to appear from within a program.

Examples

```
import(page=demand) demand.txt @append
```

will append the text file, DEMAND.TXT, to the bottom of the page "demand" in the current workfile.

Cross-references

See Chapter 3. "Workfile Basics," on page 41 of *User's Guide I* for a discussion of workfiles.

See also `wfopen` (p. 472), `copy` (p. 306), `pageload` (p. 400), `read` (p. 418), `fetch` (p. 332), `wfsave` (p. 485), and `pagesave` (p. 402).

importattr	Object Container, Data, and File Commands

Imports observation values stored inside one or more series in a second workfile page into the attribute fields of objects within the current workfile page.

Syntax

importattr(*options*) *source_page* [**@keep** *keeplist* **@drop** *droplist*]

The source page should contain one series whose values are the names of the objects into which the attributes should be imported, and one or more additional series containing the values of each attribute to be imported. By default, the procedure will assume there is an alpha series called NAME in the source page that contains the object names. The "name = " option can be used to specify that a different series in the source page contains the object names.

Values in the name column will always be converted into legal EViews names before matching to object names in the current workfile. (For example, any spaces will be replaced by underscores.)

Typically, the name of the series in the source workfile will be used as the attribute name inside the current workfile. However, if an object in the source workfile has a display name set, this value will be used for the attribute name instead. Display names must be used to specify attribute names if your attribute names contain spaces since spaces are not allowed within EViews object names.

For numeric series, attribute values will be imported using the formatting currently set for the series spreadsheet view of the source series.

By default, all series in the source page will be used as attribute values (except the series containing object names). To import only a subset of series as attributes, name patterns can be provided in the **@keep** and **@drop** arguments to restrict which series will be used.

`importattr` is most often useful when importing custom attributes from an external file. It is common for foreign data files, such as Excel files, to have one file (or sheet in Excel) containing the data, and a separate file (or sheet) containing the attributes, or meta-data, of each series. In such cases the `pageload` command can be used to read in the attribute file as a separate page in your workfile, and then `importattr` can be used to assign them to the data page.

Options

mode = *arg*	Specify how the procedure treats existing attribute values in the current workfile page. *arg* may be "o" or "overwrite" (clears all existing attribute values in the object before applying the imported attribute value), "u" or "update" (clears existing attribute values only for the attributes that are being imported), "m" or "merge" (keeps existing values if the imported value of the attribute is empty), "md" (keeps all existing non-empty values. only empty values will be replaced with the imported values).
name = *arg*	Specify the name of the alpha series in the source page containing the names of objects in the target page.

Examples

```
importattr(name=objnames) attributes @keep attr1 attr2
```

Imports values from series ATTR1 and ATTR2 in the page "Attributes" into attributes "attr1" and "attr2" of objects in the current workfile. The series OBJNAMES in the page "Attributes" specifies which objects in the current workfile should have their attribute values updated.

Cross-references

liml	Interactive Use Commands

Limited Information Maximum Likelihood and K-class Estimation.

Syntax

liml*(options) y c x1 [x2 x3 ...] @ z1 [z2 z3 ...]*

liml*(options) specification @ z1 [z2 z3 ...]*

To use the `liml` command, 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.

You may estimate nonlinear equations or equations specified with formulas by first providing a specification, then listing the instrumental variables after an "*@*"-sign. 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, unless the `noconst` option is specified.

Options

noconst	Do not include a constant in the instrumental list. Without this option, a constant will always be included as an instrument, even if not specified explicitly.
w = *arg*	Weight series or expression.
wtype = *arg* *(default* = "istdev"*)*	Weight specification type: inverse standard deviation ("istdev"), inverse variance ("ivar"), standard deviation ("stdev"), variance ("var").
wscale = *arg*	Weight scaling: EViews default ("eviews"), average ("avg"), none ("none"). The default setting depends upon the weight type: "eviews" if "wtype = istdev", "avg" for all others.
kclass = *number*	Set the value of k in the K-class estimator. If omitted, LIML is performed, and k is calculated as part of the estimation procedure.
se = *arg* *(default* = "iv"*)*	Set the standard-error calculation type: IV based ("se = iv"), K-Class based ("se = kclass"), Bekker ("se = bekk"), or Hansen, Hausman, and Newey ("se = hhn").
m = *integer*	Set maximum number of iterations.
c = *number*	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients. The criterion will be set to the nearest value between 1e-24 and 0.2.
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.

showopts / -showopts	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
coef = *arg*	Specify the name of the coefficient vector (if specified by list); the default behavior is to use the "C" coefficient vector.
prompt	Force the dialog to appear from within a program.
p	Print estimation results.

Examples

```
liml gdp c cpi inc @ lw lw(-1)
```

calculates a LIML estimation of GDP on a constant, CPI, and INC, using a constant, LW, and LW(-1) as instruments.

Cross-references

See "Limited Information Maximum Likelihood and K-Class Estimation" on page 63 of *User's Guide II* for discussion.

See `Equation::liml` (p. 101) for the equivalent equation object method.

load	Object Container, Data, and File Commands

Load a workfile.

Provided for backward compatibility. Same as `wfopen` (p. 472).

logclear	Programming Commands

Clears the log window corresponding to the program.

Use this command anytime from within a program to clear the log of the program.

Syntax

logclear

Cross-references

See "Program Message Logging" on page 123 for details.

See also `logmode` (p. 370), `logmsg` (p. 372), and `logsave` (p. 372).

logit	Interactive Use Commands

Estimate binary models with logistic errors.

Provide for backward compatibility. Equivalent to issuing the command, `binary` with the option *"(d = l)"*.

See `binary` (p. 279).

logmode	Programming Commands

Set the log settings of specified message types.

Activates or deactivates the logging of specified message types if message types are set to be *program controlled*.

Syntax

logmode *msgtype_list*

where *msgtype_list* is a list of the message types to update.

Options

Message type options

all/-all	[Show/Do not show] all messages.
error/-error, e/-e	[Show/Do not show] error messages.
logmsg /-logmsg, l/ -l	[Show/Do not show] logmsg messages.
program/-program, p/-p	[Show/Do not show] program lines.
statusline/-statusline, s / -s	[Show/Do not show] status line messages.

hideprogline/-hideprogline	[Hide/Do not hide] the program line number when reporting errors encountered during execution.

addin/-addin	[Show/Do not show] messages generally appropriate for addin error reporting. `addin` is equivalent to the command and program mode ("Program Modes" on page 122) statements:
	`logmode hideprogline -error`
	`mode quiet`

debug	Show messages generally appropriate for debugging of programs. Equivalent to the command: `logmode -hideprogline error`

Note that using `logmode` with `debug` will override all subsequent (in either the current program or any program run using `exec` or `run`), `logmode` with `hideprog` and `-error` specifications. In particular, debug will override subsequent `logmode addin` statements.

Examples

```
logmode p
```

turns on logging of program lines of code. Note that by default all message types are initially turned off.

```
logmode error -p
```

Building on the first command, this activates the logging of errors and deactivates the logging of program lines.

```
logmode -all s
```

turns off logging of all types, with exception to status line messages. Note the order of message types is important. For example,

```
logmode p -all
```

will initially activate the logging of program lines, but following `p` with `-all` will deactivate program lines as well as any other messages that have been previously activated.

Cross-references

See "Program Message Logging" on page 123 for details. See also "Program Modes" on page 122.

See also `logclear` (p. 369), `logmsg` (p. 372), and `logsave` (p. 372).

logmsg	Programming Commands

Adds a line of text to the program log.

Syntax

logmsg *text*

Example

```
logmsg About to iterate through list of states
```

appends the text "About to iterate through list of states" to the program log.

Cross-references

See "Program Message Logging" on page 123 for details.

See also `logclear` (p. 369), `logmode` (p. 370), and `logsave` (p. 372).

logsave	Programming Commands

Saves the program log to a text file.

Syntax

logsave *filepath*

where *filepath* is the location and filename for saving the log file.

Options

type = rtf	Save the log as an RTF file (thus preserving syntax coloring).

Example

```
logsave c:\EViews\myprog.text
```

saves the contents of the program log to the text file MYPROG, in the "C:\EViews" directory.

Cross-references

See "Program Message Logging" on page 123 for details.

See also `logclear` (p. 369), `logmode` (p. 370), and `logmsg` (p. 372).

ls	Interactive Use Commands

Estimation by linear or nonlinear least squares regression.

When the current workfile has a panel structure, `ls` also estimates cross-section weighed least squares, feasible GLS, and fixed and random effects models.

Syntax

ls*(options) y x1 [x2 x3 ...]*

ls*(options) specification*

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 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 may also be specified by explicit equation. You should specify the equation as a formula. The parameters to be estimated should be included explicitly: "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.

Options

Non-Panel LS Options

w = *arg*	Weight series or expression. Note: *we recommend that, absent a good reason, you employ the default settings* **Inverse std. dev. weights** *("wtype = istdev") with* **EViews default** *scaling ("wscale = eviews") for backward compatibility with versions prior to EViews 7.*
wtype = *arg* *(default* = "istdev"*)*	Weight specification type: inverse standard deviation ("istdev"), inverse variance ("ivar"), standard deviation ("stdev"), variance ("var").

wscale = *arg*	Weight scaling: EViews default ("eviews"), average ("avg"), none ("none"). The default setting depends upon the weight type: "eviews" if "wtype = istdev", "avg" for all others.
cov = *keyword*	Covariance type *(optional)*: "white" (White diagonal matrix), "hac" (Newey-West HAC).
nodf	Do not perform degree of freedom corrections in computing coefficient covariance matrix. The default is to use degree of freedom corrections.
covlag = *arg* *(default* = 1*)*	Whitening lag specification: *integer* (user-specified lag value), "a" (automatic selection).
covinfo = *arg* *(default* = "aic"*)*	Information criterion for automatic selection: "aic" (Akaike), "sic" (Schwarz), "hqc" (Hannan-Quinn) (if "lag = a").
covmaxlag = *integer*	Maximum lag-length for automatic selection *(optional)* (if "lag = a"). The default is an observation-based maximum of $T^{1/3}$.
covkern = *arg* *(default* = "bart"*)*	Kernel shape: "none" (no kernel), "bart" (Bartlett, *default*), "bohman" (Bohman), "daniell" (Daniel), "parzen" (Parzen), "parzriesz" (Parzen-Riesz), "parzgeo" (Parzen-Geometric), "parzcauchy" (Parzen-Cauchy), "quadspec" (Quadratic Spectral), "trunc" (Truncated), "thamm" (Tukey-Hamming), "thann" (Tukey-Hanning), "tparz" (Tukey-Parzen).
covbw = *arg* *(default* = "fixednw" *)*	Kernel Bandwidth: "fixednw" (Newey-West fixed), "andrews" (Andrews automatic), "neweywest" (Newey-West automatic), *number* (User-specified bandwidth).
covnwlag = *integer*	Newey-West lag-selection parameter for use in nonparametric kernel bandwidth selection (if "covbw = neweywest").
covbwint	Use integer portion of bandwidth.
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. The criterion will be set to the nearest value between 1e-24 and 0.2.
s	Use the current coefficient values in "C" as starting values for equations specified by list with AR or MA terms (see also `param` (p. 413)).

s = *number*	Determine starting values for equations specified by list with AR or MA terms. Specify a number between zero and one representing the fraction of preliminary least squares estimates computed without AR or MA terms to be used. Note that out of range values are set to "s = 1". Specifying "s = 0" initializes coefficients to zero. By default EViews uses "s = 1".
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.
z	Turn off backcasting in ARMA models.
coef = *arg*	Specify the name of the coefficient vector (if specified by list); the default behavior is to use the "C" coefficient vector.
prompt	Force the dialog to appear from within a program.
p	Print basic estimation results.

Note: not all options are available for all equation methods. See the *User's Guide II* for details on each estimation method.

Panel LS Options

cx = *arg*	Cross-section effects: (default) none, fixed effects ("cx = f"), random effects ("cx = r").
per = *arg*	Period effects: (default) none, fixed effects ("per = f"), random effects ("per = r").
wgt = *arg*	GLS weighting: (default) none, cross-section system weights ("wgt = cxsur"), period system weights ("wgt = persur"), cross-section diagonal weighs ("wgt = cxdiag"), period diagonal weights ("wgt = perdiag").

cov = *arg*	Coefficient covariance method: (default) ordinary, White cross-section system robust ("cov = cxwhite"), White period system robust ("cov = perwhite"), White heteroskedasticity robust ("cov = stackedwhite"), Cross-section system robust/PCSE ("cov = cxsur"), Period system robust/PCSE ("cov = persur"), Cross-section heteroskedasticity robust/PCSE ("cov = cxdiag"), Period heteroskedasticity robust/PCSE ("cov = perdiag").
keepwgts	Keep full set of GLS weights used in estimation with object, if applicable (by default, only small memory weights are saved).
rancalc = *arg* *(default* = "sa*")*	Random component method: Swamy-Arora ("rancalc = sa"), Wansbeek-Kapteyn ("rancalc = wk"), Wallace-Hussain ("rancalc = wh").
nodf	Do not perform degree of freedom corrections in computing coefficient covariance matrix. The default is to use degree of freedom corrections.
coef = *arg*	Specify the name of the coefficient vector (if specified by list); the default behavior is to use the "C" coefficient vector.
iter = *arg (default* = "onec")	Iteration control for GLS specifications: perform one weight iteration, then iterate coefficients to convergence ("iter = onec"), iterate weights and coefficients simultaneously to convergence ("iter = sim"), iterate weights and coefficients sequentially to convergence ("iter = seq"), perform one weight iteration, then one coefficient step ("iter = oneb"). Note that random effects models currently do not permit weight iteration to convergence.
unbalsur	Compute SUR factorization in unbalanced data using the subset of available observations for a cluster.
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. The criterion will be set to the nearest value between 1e-24 and 0.2.
s	Use the current coefficient values in "C" as starting values for equations specified with AR terms (see also `param` (p. 413)).

s = *number*	Determine starting values for equations specified with AR terms. Specify a number between zero and one representing the fraction of preliminary least squares estimates computed without AR terms to be used. Note that out of range values are set to "s = 1". Specifying "s = 0" initializes coefficients to zero. By default EViews uses "s = 1".
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.
prompt	Force the dialog to appear from within a program.
p	Print basic estimation results.

Examples

```
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.

```
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. The "*z*" option turns off backcasting.

```
coef(2) beta
param beta(1) .2 beta(2) .5 c(1) 0.1
ls(cov=white) q = beta(1)+beta(2)*(l^c(1) + k^(1-c(1)))
```

estimates the nonlinear regression starting from the specified initial values. The "cov = white" option reports heteroskedasticity consistent standard errors.

```
ls r = c(1)+c(2)*r(-1)+div(-1)^c(3)
```

estimates an UNTITLED nonlinear equation.

```
ls(cx=f, per=f) n w k ys c
```

estimates an UNTITLED equation in a panel workfile using both cross-section and period fixed effects.

```
ls(cx=f, wgt=cxdiag) n w k ys c
```

estimates an UNTITLED equation in a panel workfile with cross-section weights and fixed effects.

Cross-references

Chapter 1. "Basic Regression Analysis," on page 5 and Chapter 2. "Additional Regression Tools," on page 23 of *User's Guide II* discuss the various regression methods in greater depth.

Chapter 16. "Special Expression Reference," on page 557 describes special terms that may be used in `ls` specifications.

See Chapter 23. "Panel Estimation," on page 759 of *User's Guide II* for a discussion of panel equation estimation.

See `Equation::ls` (p. 103) for the equivalent equation object method command.

open	Object Container, Data, and File Commands || Programming Commands

Opens a program file, or text (ASCII) file.

This command should be used to open program files or text (ASCII) files for editing.

You may also use the command to open workfiles or databases. This use of the `open` command for this purposes is provided for backward compatibility. We recommend instead that you use the new commands `wfopen` (p. 472) and `pageload` (p. 400) to open a workfile, and `dbopen` (p. 322) to open databases.

Syntax

open*(options) [path\]file_name*

You should provide the name of the file to be opened including the extension (and optionally a path), or the file name without an extension but with an option identifying its type. Specified types always take precedence over automatic type identification. If a path is not provided, EViews will look for the file in the default directory.

Files with the ".PRG" extension will be opened as program files, unless otherwise specified. Files with the ".TXT" extension will be opened as text files, unless otherwise specified.

For backward compatibility, files with extensions that are recognized as database files are opened as EViews databases, unless an explicit type is specified. Similarly, files with the extensions ".WF" and ".WF1", and foreign files with recognized extensions will be opened as workfiles, unless otherwise specified.

All other files will be read as text files.

Options

p	Open file as program file.
t	Open file as text file.
type = *arg* ("prg" or "txt")	Specify text or program file type using keywords.

Examples

```
open finfile.txt
```

opens a text file named "Finfile.TXT" in the default directory.

```
open "c:\program files\my files\test1.prg"
```

opens a program file named "Test1.PRG" from the specified directory.

```
open a:\mymemo.tex
```

opens a text file named "Mymemo.TEX" from the A: drive.

Cross-references

See `wfopen` (p. 472) and `pageload` (p. 400) for commands to open files as workfiles or workfile pages, and `dbopen` (p. 322) for opening database files.

optimize — Programming Commands

Find the solution to a user-defined optimization problem.

The `optimize` command calls the EViews optimizer to find the optimal control values for a subroutine defined elsewhere in the program file or files.

Syntax

optimize*(options) subroutine_name(arguments)*

You should follow the optimize command keyword with options and *subroutine_name*, the name of a defined subroutine in your program (or included programs).

The subroutine must contain at least two arguments.

By default EViews will interpret the first argument as the output of the subroutine and will use it as the value to optimize. If the objective contains more than one value, as in a series or vector, EViews will optimize the sum of the values. The second argument is, by default, used to define the control parameters for which EViews will find the optimal values.

Since the objective and controls are defined by a standard EViews subroutine, the arguments of the subroutine may correspond to numbers, strings, and various EViews objects such as series, vectors, scalars.

Options

Optimization Objective and Controls

The default optimization objective is to maximize the first argument of the subroutine. You may use the following optimization options to change the optimization objective and to specify the coefficients (control parameters):

max *[= integer]*	Maximize the subroutine objective or sum of the values of the subroutine objective *(default)*. By default the first argument of the subroutine is used as the maximization objective. You may change the objective to another argument by specifying an *integer* argument location.
min *[= integer]*	Minimize the subroutine objective or sum of the values of the subroutine objective. By default the first argument of the subroutine is used as the minimization objective. You may change the objective to another argument by specifying an *integer* argument location.
ls *[= integer]*	Perform least squares minimization of the sum of squared values of the subroutine objective. *(The objective argument cannot be a scalar value when using this option.)* By default the first argument of the subroutine is used as the minimization objective. You can change the objective to another argument by specifying an *integer* argument location.
ml *[= integer]*	Perform maximum likelihood estimation of the sum of the values of the subroutine objective. *(The objective argument cannot be a scalar value when using this option.)* By default the first argument of the subroutine is used as the maximization objective. You can change the objective to another argument by specifying an *integer* argument location. Note that the "ml" option specifies the same optimization as when using the "max" option, but permits a different set of Hessian matrix choices.

coef = *integer* *(default = 2)*	Specify the argument number of the function that contains the coefficient controls for the optimization. If the argument is a vector or matrix, each element of the vector or matrix will be treated as a coefficient. If the argument is a series, each element of the series within the current workfile sample will be treated as a coefficient. The default value is 2 so that the second argument is assumed to contain the coefficient controls.

Optimization Options

grads = *integer*	Specifies an argument number corresponding to analytic gradients for each of the coefficients. If this option is not specified, gradients are evaluated numerically. Available for "ls" and "ml" estimation. • If the objective argument is a scalar, the gradient argument should be a vector of length equal to the number of elements in the coefficient argument. • If the objective argument is a series, the gradient argument should be a group object containing one series per element of the coefficient argument. The series observations should contain the corresponding derivatives for each observation in the current workfile sample. • For a vector objective, the gradient argument should be a matrix with number of rows equal to the length of the objective vector, and columns equal to the number of elements in the coefficient argument. • "grad = " may not be specified if the objective is a matrix.
hess = *arg*	Specify the type of Hessian approximation: "numeric" (numerical approximation), "bfgs" (Broyden–Fletcher–Goldfarb–Shanno), or "opg" (outer product of gradients, or BHHH). "opg" is only available when using "ls" or "ml" optimization. The default value is "bfgs" unless using "ls" optimization, which defaults to "opg".
step = *arg* *(default_ = "marquardt")*	Set the step method: "marquardt", "dogleg" or "linesearch".
scale = *arg*	Set the scaling method: "maxcurve" (default), or "none".
m = *int*	Set the maximum number of iterations
c = *number*	Set the convergence criteria.

trust = *number* (*default* = 0.25)	Sets the initial trust region size as a proportion of the initial control values. Smaller values of this parameter may be used to provide a more cautious start to the optimization in cases where larger steps immediately lead into an undesirable region of the objective. Larger values may be used to reduce the iteration count in cases where the objective is well behaved but the initial values may be far from the optimum values.
deriv = high	Always use high precision numerical derivatives. Without this option, EViews will start by using lower precision derivatives, and switch to higher precision as the optimization progresses.
feps = *number* (*default* = 2.2e-16)	Set the expected relative accuracy of the objective function. The value indicates what fraction of the observed objective value should be considered to be random noise.
noerr	Turn off error reporting.
finalh = *name*	Save the final Hessian matrix into the workfile with name *name*. For "hess = bfgs", the final Hessian will be based on numeric derivatives rather than the BFGS approximation used during the optimization since the BFGS approximation need not converge to the true Hessian.

Examples

The first example estimates a regression model using maximum likelihood. The subroutine LOGLIKE first computes the regression residuals using the coefficients in the vector BETA along with the dependent variable series given by DEP and the regressors in the group REGS.

```
subroutine loglike(series logl, vector beta, series dep, group regs)
    series r = dep - beta(1) - beta(2)*regs(1) - beta(3)*regs(2) -
       beta(4)*regs(3)
    logl = @log((1/beta(5))*@dnorm(r/beta(5)))
endsub

series LL
vector(5) MLCoefs
MLCoefs = 1
MLCoefs(5) = 100
optimize(ml=1, finalh=mlhess, hess=numeric) loglike(LL, MLCoefs, y,
    xs)
```

The `optimize` command instructs EViews to use the LOGLIKE subroutine for purposes of maximization, and to use maximum likelihood to maximize the sum (over the workfile sample) of the LL series with respect to the five elements of the vector MLCOEFS. EViews employs a numeric approximation to the Hessian in optimization, and saves the final estimate in the workfile in the sym object MLHESS.

Notice that we specify initial values for the MLCOEFS coefficients prior to calling the optimization routine.

Our second example recasts the estimation above as a least squares optimization problem, and illustrates the use of the "grads = " option to employ analytically computed gradients defined in the subroutine.

```
subroutine leastsquareswithgrads(series r, vector beta, group grads,
    series dep, group regs)
    r = dep - beta(1) - beta(2)*regs(1) - beta(3)*regs(2) -
       beta(4)*regs(3)
    grads(1) = 1
    grads(2) = regs(1)
    grads(3) = regs(2)
    grads(4) = regs(3)
endsub

series LSresid
vector(4) LSCoefs
lscoefs = 1
series grads1
series grads2
series grads3
series grads4
group grads grads1 grads2 grads3 grads4
optimize(ls=1, grads=3) leastsquareswithgrads(LSresid, lscoefs,
    grads, y, xs)
```

Note that for a linear least squares problem, the derivatives of the coefficients are trivial - the regressors themselves (and a series of ones for the constant).

The next example uses matrix computation to obtain an optimizer objective that finds the solution to the same least squares problem. While the optimizer is not a solver, we can trick it into solving that equation by creating a vector of residuals equal to $(X'X)\beta - X'Y$, and asking the optimizer to find the values of β that minimize the square of those residuals:

```
subroutine local matrixsolve(vector rvec, vector beta, series dep,
    group regs)
    stom(regs, xmat)
    xmat = @hcat(@ones(100), xmat)
    stom(dep, yvec)
```

```
    rvec = @transpose(xmat)*xmat*beta - @transpose(xmat)*yvec
    rvec = @epow(rvec,2)
endsub
vector(4) MSCoefs
MSCoefs = 1
vector(4) rvec
optimize(min=1) matrixsolve(rvec, mscoefs, y, xs)
```

The first few lines of the subroutine convert the input dependent variable series and regressor group into matrices. Note that the regressor group does not contain a constant term upon input, so we append a column of ones to the regression matrix XMAT, using the `@hcat` command.

Lastly, we define a subroutine containing the quadratic form, and use the optimize command to find the value that minimizes the function:

```
subroutine f(scalar !y, scalar !x)
    !y = 5*!x^2 - 3*!x - 2
endsub
scalar in = 0
scalar out = 0
optimize(min) f(out, in)
```

The subroutine F calculates the simple quadratic formula:

$$Y = 5X^2 - 3X - 2 \tag{12.1}$$

which attains a minimum of -2.45 at an IN value of 0.3.

Cross-references

For discussion, see Chapter 10. “User-Defined Optimization,” beginning on page 221.

optsave	Object Container, Data, and File Commands

Save the current EViews global options settings “.INI” files into a directory.

Syntax

optsave *directory*

Save a copy of the current global options settings into the specified directory. Usually this command will be used in conjuncture with a later use of the `optset` command. Any existing option settings in the directory will be overwritten.

“General Options” and “Graphics Defaults” will always be saved. “Database registry settings” and “Database object aliases” will only be saved if the file location setting for the

"Database Registry" and "Alias Map Path" is the same as the file location of the INI File Path.

If the directory name is omitted, the option settings currently in effect will be used to replace the default global options. (This can be used to copy option settings back into your default settings after the `optset` command has switched to using options in a different directory).

Note that this command does not change which set of options are active. You must follow this command with the `optset` command if you would like to switch to using the saved copy as your active set of options.

Cross-references

See Appendix A. "Global Options," beginning on page 761 of *User's Guide I* for discussion of the global options.

See also `optset` (p. 385).

optset	Object Container, Data, and File Commands

Replace the current EViews global options settings ".INI" files with ones based in a different directory.

Syntax

optset *directory*

Temporarily switch to using the global options settings stored within ".INI" files in the specified directory. These will typically have been previously saved by using the `optsave` command.

"General Options" and "Graphics Defaults" will always be switched. "Database registry settings" and Database object aliases" will only be switched if the file location setting for the "Database Registry" and "Alias Map Path" is the same as the file location of the INI File Path.

The new options will stay in effect until EViews is restarted or until the `optset` command is executed again with a different directory. After the `optset` command has been issued, changing settings using the **Options** menu will modify settings in the new directory.

If the directory name is omitted, the global options settings will be reset to use the settings from the default location (the same as restarting EViews).

Note that you can use the command "`optset .\`" in a program to switch to using the global options saved in the same directory as the program file. This can be used to ensure that multiple users always use the same global options settings when running a shared program.

Cross-references

See Appendix A. "Global Options," beginning on page 761 of *User's Guide I* for discussion of the global options.

See also `optsave` (p. 384).

ordered	Interactive Use Commands

Estimation of ordered dependent variable models.

Syntax

equation name.**ordered***(options) y x1 [x2 x3 ...]*

equation name.**ordered***(options) specification*

Options

d = *arg (default* = "n"*)*	Specify likelihood: normal likelihood function, ordered probit ("n"), logistic likelihood function, ordered logit ("l"), Type I extreme value likelihood function, ordered Gompit ("x").
q *(default)*	Use quadratic hill climbing as the maximization algorithm.
r	Use Newton-Raphson as the maximization algorithm.
b	Use Berndt-Hall-Hall-Hausman as 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. The criterion will be set to the nearest value between 1e-24 and 0.2.
s	Use the current coefficient values in "C" as starting values (see also `param` (p. 413)).
s = *number*	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 = *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.
coef = *arg*	Specify the name of the coefficient vector (if specified by list); the default behavior is to use the "C" coefficient vector.
prompt	Force the dialog to appear from within a program.
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=l,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 = @coefs
```

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.

Cross-references

See "Ordered Dependent Variable Models" on page 278 of the *User's Guide II* for additional discussion.

See also `binary` (p. 279).

See `Equation::ordered` (p. 119) for the equivalent equation object command.

output

Programming Commands

Redirect printer output.

You may specify that any procedure that would normally send output to the printer puts output in a text, Rich Text Format (RTF), or comma-separated value (CSV) file, in a spool object, or into frozen table or graph objects in the current workfile.

Syntax

output*[(f)] base_name*

output*(options) [path\]file_name*

output*[(s)] spool_name*

output off

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 OUT01, the second print command will generate OUT02, and so on.

You can also use the optional settings, described below, to redirect table and text output to a text, RTF, or CSV file, or all output (including graphs) to an RTF file. If you elect to redirect output to a file, you must specify a filename.

You can use the "s" option, described below, to redirect all output to a spool object.

When followed by the optional keyword `off`, the `output` command turns off output redirection. Subsequent print commands will be directed to the printer.

Options

f	Redirect all output to frozen objects in the default workfile, using *base_name*.
t	Redirect table and text output to a text file. Graphic output will still be sent to the printer.
r	Redirect all output to an Rich Text Format (RTF) file.
v	Redirect all output to an comma-separated value (CSV) file.

s	Redirect all output to a spool object.
o	Overwrite file if necessary. If the specified filename for the text, RTF, or CSV file exists, overwrite the file. The default is to append to the file. Only applicable for text, RTF, or CSV file output (specified using options "t", "r", or "v").
c	Command logging. Output both the output, and the command used to generate the output. This option is only applicable in a program when used in conjunction with the `pon` (p. 664) command which enables automatic printing of output. The use of `pon` is not required when `output` is specified in a command window. Only applicable for text, RTF, or CSV file output (specified using options "t", "r", or "v").

Examples

```
output print_
```

causes the first print command to generate a table or graph object named PRINT_01, the second print command to generate an object named PRINT_02, 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 first line redirects printing to the RESULTS.TXT file, while the print option of the second and third lines sends the graph output to the printer. The last line turns output redirection off and restores normal printer use.

If instead, the first line read:

```
output(r) c:\data\results
```

all subsequent output would be sent to the RTF file RESULTS.RTF.

```
output(s) mySpool
```

redirects all output to the MYSPOOL spool. If the spool already exists, printed objects will be appended to the end of the spool.

Cross-references

See "Print Setup," beginning on page 779 of *User's Guide I* for further discussion.

See also `pon` (p. 664), `poff` (p. 664).

pageappend	Object Container, Data, and File Commands

Append contents of the specified workfile page to the active workfile page.

Syntax

pageappend*(options) wfname[\pgname] [object_list]*

where *wfname* is the name of a workfile that is currently in memory. You may optionally provide the name of a page in *wfname* that you wish to used as a source, and the list of objects to be read from the source workfile page. If no *wfname* is provided, EViews will use the default page in the source workfile.

The command appends the contents of the source page to the active page in the default workfile. The target page is first unstructured (if necessary) and its range is expanded to encompass the combined range of the sample from the source workfile page, and the destination page.

The default behavior is to append all series and alpha objects (but not other types) from the source page, but the optional *object_list* may be provided to specify specific series, or to specify objects other than series or alpha objects to be included. Command options may also be used to modify the list of objects to be included.

Note that since this operation is performed in place, the original workfile page cannot be recovered. We recommend that you consider backing up your original page using pagecopy (p. 392).

Options

smpl = *smpl_spec*	Specifies an optional sample identifying which observations from the source page are to be appended. Either provide the sample range in double quotes or specify a named sample object. The default is "@all".
allobj	Specifies that all objects (including non-series and non-alpha objects) should be appended. For objects other than series and alphas, appending involves simply copying the objects from the source page to the destination page. This option may not be used with an explicit *object_list* specification.
match	Specifies that only series and alphas in the append page that match series and alphas of the same name in the active page should be appended. This option may not be used with "allobj" or with an explicit object_list specification.

sufix = *suffix_arg* *(default = "_a")*	Specifies a string to be added to the end of the source object name, if necessary, to avoid name collision when creating a new object in the target page.
obsid = *arg*	Provides the name of a series used to hold the date or observation ID of each observation in the destination workfile.
wfid = *arg*	Provides the name of a (boolean) series to hold an indicator of the source for each observation in the destination workfile (0, if from the destination; 1, if from the source).

Examples

```
pageappend updates
```

appends, to the default workfile page, all of observations in all of the series in the active page of the workfile UPDATES.

```
pageappend(match, smpl="1999 2003") updates
```

restricts the series to those which match (by name) those in the default workfile page, and restricts the observations to merge to those between 1999 and 2003.

```
pageappend newdat\page1 income cons
```

takes only the INCOME and CONS series from the PAGE1 of the NEWDATA workfile, and appends them to the current workfile page.

```
pageappend(alltypes, suffix="_1") mydata
```

appends all objects from MYDATA, merging series with matching names, and renaming other matching objects by appending "_1" to the end of the name.

Cross-references

See "Appending to a Workfile" on page 277 of *User's Guide I* for discussion.

See also pagecopy (p. 392).

pagecontract	Object Container, Data, and File Commands

Contract the active workfile page according to the specified sample.

Syntax

pagecontract *smpl_spec*

where *smpl_spec* is a sample specification. Contraction removes observations not in specified sample from the active workfile page. Note that since this operation is performed in place, you may wish to backup your original page (see pagecopy (p. 392)) prior to contract-

ing.

Examples

```
pagecontract if income<50000 and race=2
```

removes all observations with INCOME values greater than or equal to 50000 and RACE not equal to 2.

```
pagecontract 1920 1940 1945 2000
```

removes observations for the years 1941 to 1944.

Cross-references

See "Contracting a Workfile" on page 280 of *User's Guide I* for discussion.

See also `pagecopy` (p. 392).

pagecopy	Object Container, Data, and File Commands

Copies all or part of the active workfile page to a new workfile, or to a new page within the default workfile.

Syntax

pagecopy*(options) [object_list]*

where the optional *object_list* specifies the workfile objects to be copied. If *object_list* is not provided, all objects will be copied subject to the option restrictions discussed below.

If copying objects to a new page within the default workfile, you may choose to copy series objects (series, alphas, and links) by value or by link (by employing the "bylink" option). If you elect to copy by value, links in the source page will converted to ordinary series and alpha objects when they are copied. If you copy by link, series and alpha objects in the source page are converted to links when copied. The default is to copy by value.

If you copy objects to a new workfile, data objects must be copied by value.

Options

bylink	Specifies that series and alpha objects be copied as links to the source page. This option is not available if you use the "wf = " option, since linking requires that the destination page be in the same workfile as the source page. Automatically sets the "dataonly" option so that only series, alphas, links, and valmaps will be copied.
smpl = *smpl_spec*	Specifies an optional sample identifying which observations from the source page are to be appended. Either provide the sample range in double quotes or specify a named sample object. The default is "@all".
rndobs = *integer*	Copy only a random subsample of *integer* observations from the specified sample. Not available with "bylink," "rndpct," or "prob."
rndpct = *arg*	Copy only a random percentage subsample of *arg* (a number between 0 and 100) of the specified sample. Not available with "bylink," "rndobs," or "prob."
prob = *arg*	Copies a random subsample where each observation has a fixed probability, *prob*, of being selected. *prob* should be entered as a percentage value (a number between 0 and 100). Not available with "bylink," "rndobs," or "rndpc"
dataonly	Only series, alphas, links, and valmaps should be copied. The default is to copy all objects (unless the "bylink" option is specified, in which case only series objects are copied).
nolinks	Do not copy links from the source page.
wf = *wf_name*	Optional name for the destination workfile. If not provided, EViews will create a new untitled workfile. Not available if copying using the "bylink" option.
page = *page_name*	Optional name for the newly created page. If not provided, EViews will use the next available name of the form "Untitled##", where ## is a number.

Examples

```
pagecopy(page=allvalue, wf=newwf)
```

will first create a new workfile named NEWWF, with a page ALLVALUE that has the same structure as the current page. Next, all of the objects in the active workfile page will be copied into the new page, with the series objects copied by value. In contrast,

```
pagecopy(bylink, page=alllink)
```

will instead create a page ALLLINK in the existing workfile, and will copy all series objects by creating links in the new page.

```
pagecopy(page=partcopy, bylink, smpl="1950 2000 if
    gender=""male""") a* *z
```

will create a new page named PARTCOPY in the existing workfile containing the specified subsample of observations, and will copy all series objects in the current page beginning with the letter "A" or ending with the letter "Z". The objects will be copied by creating links in the new page.

```
pagecopy(page=rndcopy, smpl="1950 2000 if gender=""male""",
    rndobs=200, dataonly, nolinks)
```

creates a new workfile and page RNDCOPY containing a 200 observation random sample from the specified subsample. Series and alpha objects only will be copied by value from the source page.

Cross-references

See "Copying from a Workfile" on page 280 of *User's Guide I* for discussion.

See also `pageappend` (p. 390).

pagecreate	Object Container, Data, and File Commands

Create a new page in the default workfile. The new page becomes the active page.

Syntax

pagecreate*(options) freq[(subperiod_opts)] start_date end_date [num_cross_sections]*

pagecreate*(options)* **u** *num_observations*

pagecreate(*id_method[,options]*) *id_list [*@**srcpage** *page_list]*

pagecreate(**idcross***[,options]*) *id1 id2 [*@**srcpage** *page1 page2]*

pagecreate(**idcross***[,options]*) *id1* @**range**(*freq, start_date, end_date*) *[*@**srcpage** *page1]*

These different forms of the `pagecreate` command encompass three distinct approaches to creating a new workfile page: (1) regular frequency description or unstructured data description; (2) using the union or intersection of unique values from one or more ID series in existing workfile pages; (3) using the cross of unique values from two identifier series or from an identifier series and a date range. Each of these approaches is described in greater detail below.

Regular Frequency or Unstructured Description

The first two forms of the command permit you to create a new workfile page using a regular frequency or unstructured description:

- **pagecreate***(options) freq[(subperiod_opts)] start_date end_date [num_cross_sections]*
- **pagecreate***(options)* **u** *num_observations*

The first form of the command should be employed to create a regular frequency page with the specified frequency, start, and end date. If you include the optional argument *num_cross_sections,* EViews will create a balanced panel page using integer identifiers for each of the cross-sections. Note that more complex panel structures may be defined using `pagestruct` (p. 408).

The second form of the command creates an unstructured workfile with the specified number of observations.

Note that these forms of the command are analogous to `wfcreate` (p. 467) except that instead of creating a new workfile, we create a new page in the default workfile.

The *freq* argument should be specified using one of the following forms:

Sec*[opt]*, 5Sec*[opt]*, 15Sec*[opt]*, 30Sec*[opt]*	Seconds in intervals of: 1, 5, 15, or 30 seconds, respectively. You may optionally specify days of the week and start and end times during the day using the *opt* parameter. See explanation of subperiod options below.
Min*[opt]*, 2Min*[opt]*, 5Min*[opt]*, 10Min*[opt]*, 15Min*[opt]*, 20Min*[opt]*, 30Min[*[opt]*	Minutes in intervals of: 1, 2, 5, 10, 15, 20, or 30 minutes, respectively. You may optionally specify days of the week and start and end times during the day using the *opt* parameter. See explanation of subperiod options below.
H*[opt]*, 2H*[opt]*, 4H*[opt]*, 6H*[opt]*, 8H*[opt]*, 12H[*[opt]*	Hours in intervals of: 1, 2, 4, 6, 8, or 12 hours, respectively. You may optionally specify days of the week and start and end times during the day using the *opt* parameter. See explanation of subperiod options below.
D(*s, e*)	Daily with arbitrary days of the week. Specify the first and last days of the week with integers *s* and *e*, where Monday is given the number 1 and Sunday is given the number 7. (Note that the "D" option used to specify a 5-day frequency in versions prior to EViews 7).
D5 or 5	Daily with five days per week, Monday through Friday.
D7 or 7	Daily with seven days per week.

W	Weekly
T	Ten-day (daily in intervals of ten).
F	Fortnight
BM	Bimonthly
M	Monthly
Q	Quarterly
S	Semi-annual
A or Y	Annual
2Y, 3Y, 4Y, 5Y, 6Y, 7Y, 8Y, 9Y, 10Y, 20Y	Multi-year in intervals of: 2, 3, 4, 5, 6, 7, 8, 9, 10, or 20 years, respectively.

Subperiod options

EViews allows for setting the days of the week and the time of day within intraday frequencies, which include **second**s, **minutes**, and **hours**. For instance, you may specify hourly data between 8AM and 5PM on Monday through Wednesday. These subperiod options should follow the frequency keyword and be enclosed in parentheses.

To specify days of the week, use integers to indicate the days, where Monday is given the number 1 and Sunday is given the number 7. For example,

```
pagecreate(wf=strhours) 30MIN(1-6, 8:00-17:00) 1/3/2000 12/30/2000
```

indicates a half-hour frequency that includes Monday through Saturday from 8AM to 5PM.

To specify the start and end times, you may use either a 24 hour clock, including minutes and optionally seconds, or a 12 hour clock using AM and PM. For example, each of the following represents 8PM: 8PM, 8:00PM, 8:00:00PM, 20:00, and 20:00:00. Thus, our previous example could have been written:

```
pagecreate(wf=strhours) 30MIN(1-6, 8AM-5PM) 1/3/2000 12/30/2000
```

Note that day and time ranges may be delimited by either commas or dashes. So this command is also equivalent to:

```
pagecreate(wf=strhours) 30MIN(1,6, 8AM,5PM) 1/3/2000 12/30/2000
```

though you will likely find the dashes easier to read.

If you wish to include all days of the week but would like to specify a start and end time, set the date range to include all days and then specify the times. The day of the week parameter appears first and is required if you wish to supply the time of day parameters. For instance,

```
pagecreate(wf=storehours) 30MIN(1-7, 10AM-3PM) 1/3/2000 12/30/2000
```

indicates a half-hour frequency from 10AM to 3PM on all days of the week.

You may also include a time with the start and end date parameters to specify partial days at the beginning or end of the workfile. For example,

```
pagecreate(wf=strhours) 30MIN(1-6, 8AM-5PM) 1/3/2000 10AM
    12/30/2000 2PM
```

creates the same workfile page as above, but limits the first day, 1/3/2000, to 10AM - 5PM and the last day, 12/30/2000, to 8AM - 2PM.

Unique Values from a Set of Identifier Series

The next form of the command allows for creating pages from the unique values of one or more identifier series found in one or more workfile pages:

- **pagecreate**(*id_method[,options]*) *identifier_list [@***srcpage** *page_list]*

The *identifier_list* should include one or more ID series. If more than one ID series is provided, EViews will use the values that are unique across all of the series. If you wish to create a page with a date structure, you should specify *one* of your identifiers using the special "@DATE" keyword identifier, enclosing the series (or the date ID component series) inside parentheses. If you wish to use the date ID values from the source workfile page, you may use the "@DATE" keyword without arguments.

The *id_method* describes how to handle unique ID values that differ across multiple pages:

id	Use the observed values of the series in the *identifier_list* in specified page.
idunion	Use the union of the observed values of the series in the *identifier_list* in the specified pages.
idintersect	Use the intersection of the observed values of the series in the *identifier_list* in the specified pages.

If the optional source page or list of source pages is not provided, EViews will use the default workfile page. Note that if a single workfile page is used, the two ID methods yield identical results.

Cross of Unique Values from Two Identifier Series or from an Identifier Series and a Date Range

The last two forms of the command create a new page by crossing the unique values in two ID series located in one or more pages, or by crossing an ID series from one page with a date range. First, you may specify a pair of identifiers, and optionally source pages where they are located,

- **pagecreate**(**idcross***[,options]*) *id1 id2 [@***srcpage** *page1 page2]*

You may instruct EViews to create a date structured page by specifying one of your two identifiers using a "@DATE" specification as described above.

Alternately, you may provide a single identifier and a range specification using the "@RANGE" keyword with a *freq*, *start_date*, and *end_date*, and optionally, the location of the identifier series.

- **pagecreate**(**idcross***[,options]*) *id1* **@range**(*freq, start_date, end_date*) *[***@srcpage** *page1]*

Options

smpl = *smpl_spec*	Specifies an optional sample identifying which observations to use when creating a page using the *id_method* option. Either provide the sample range in double quotes or specify a named sample object. The default is "@all". When multiple source workfiles are involved, the specified sample must be valid for all workfiles.
page = *page_name*	Optional name for the newly created page. If not provided, EViews will use the next available name of the form "Untitled##", where ## is a number.
wf = *wf_name*	Optional name for the new workfile. If not provided, EViews will create a new page in the default workfile.
prompt	Force the dialog to appear from within a program.

Examples

Regular Frequency or Unstructured Description

The two commands:

```
pagecreate(page=annual) a 1950 2005
pagecreate(page=unstruct) u 1000
```

create new pages in the existing workfile. The first page is an annual page named ANNUAL, containing data from 1950 to 2005; the second is a 1000 observation unstructured page named UNSTRUCT.

```
pagecreate(page=mypanel) a 1935 1954 10
```

creates a new workfile page named MYPANEL, containing a 10 cross-section annual panel for the years 1935 to 1954.

```
pagecreate(page=fourday) D(1,4) 1/3/2000 12/31/2000
```

specifies a daily workfile page from January 3, 2000 to December 31, 2000, including only Monday through Thursday. The day range may be delimited by either a comma or a dash, such that

```
pagecreate(wf=fourday) D(1-4) 1/3/2000 12/31/2000
```

is equivalent to the previous command.

```
pagecreate(wf=captimes) 15SEC(2-4) 1/3/2000 12/30/2000
```

creates a workfile page with 15 second intervals on Tuesday through Thursday only, from 1/3/2000 to 12/30/2000.

Unique Values from a Set of Identifier Series

```
pagecreate(id, page=statepage) state
```

creates a new page STATEIND using the distinct values of STATE in the current workfile page.

```
pagecreate(id, page=statepage) state industry
```

creates a new page named STATEIND, using the distinct STATE/INDUSTRY values in the active page.

```
pagecreate(id, page=stateyear) state @date(year)
pagecreate(id, page=statemonth) @date(year, month)
```

use STATE, along with YEAR, and the YEAR and MONTH series respectively, to form identifiers that will be used in creating the new dated workfile pages.

```
pagecreate(id, smpl="if sex=1") crossid @date
```

creates a new page using CROSSID and existing date ID values of the active workfile page. Note that only observations in the subsample defined by "@all if sex = 1" are used to determine the unique values.

```
pagecreate(id, page=AllStates, smpl="if sex=""Female""") stateid
    @srcpage north south east west
```

creates a new page ALLSTATES structured using the union of the unique values of STATEID from the NORTH, SOUTH, EAST and WEST workfile pages that are in the sample "if sex = "Female"". Note the use of the double quote escape character for the double quotes in the sample string.

```
pagecreate(idintersect, page=CommonStates, smpl="1950 2005")
    stateid @srcpage page1 page2 page3
```

creates a new page name COMMONSTATES structured using the intersection of the unique values of STATEID taken from the pages PAGE1, PAGE2, and PAGE3.

Cross of Unique Values from Two Identifier Series or from an Identifier Series and a Date Range

```
pagecreate(idcross,page=UndatedPanel) id1 id2 @srcpage page1 page2
```

will add the new page UNDATEDPANEL to the current workfile. UNDATEDPANEL will be structured as an undated panel using values from the cross of ID1 from PAGE1 and ID2 from PAGE2.

To create a dated page using the "idcross" option, you must tag one of the identifiers using an "@DATE" specification:

```
pagecreate(idcross,page=AnnualPanel) id1 @date(year) @srcpage
    page1 page2
```

You may also specify the cross of an identifier with a date range:

```
pagecreate(idcross,page=QuarterlyPanel) id1 @range(q, 1950, 2006)
    @srcpage page1
```

creates a quarterly panel page named QUARTERLYPANEL using the cross of ID1 taken from PAGE1, and quarterly observations from 1950q1 to 2006q4.

Cross-references

See "Creating a Workfile," on page 42 of *User's Guide I* for discussion.

See also wfcreate (p. 467) and pagedelete (p. 400).

pagedelete

Object Container, Data, and File Commands

Delete the named pages from the default workfile.

Syntax

pagedelete *[pgname1 pgname2 pgname3...]*

By default pagedelete will delete the currently active page in the default workfile. You may specify other pages to delete by listing them, in a space delimited list, after the pagedelete command.

Examples

```
pagedelete page1 page2
```

Cross-references

See also pagecreate (p. 394).

pageload

Object Container, Data, and File Commands

Load one or more new pages in the default workfile.

Syntax

pageload *[path\]workfile_name*

pageload*(options) source_description [*@**keep** *keep_list] [*@**drop** *drop_list] [*@**keepmap** *keepmap_list] [*@**dropmap** *dropmap_list] [*@**selectif** *condition]*

pageload*(options)source_description table_description [*@**keep** *keep_list] [*@**drop** *drop_list] [*@**keepmap** *keepmap_list] [*@**dropmap** *dropmap_list] [*@**selectif** *condition]*

The basic syntax for pageload follows that of wfopen (p. 472). The difference between the two commands is that pageload creates a new page in the default workfile, rather than opening or creating a new workfile. If a page is loaded with a name that matches an existing page, EViews will rename the new page to the next available name (*e.g.*, "INDIVID" will be renamed "INDIVID1".

Examples

```
pageload "c:\my documents\data\panel1"
```

loads the workfile PANEL1.WF1 from the specified directory. All of the pages in the workfile will be loaded as new pages into the current workfile.

```
pageload f.wf1
```

loads all of the pages in the workfile F.WF1 located in the default directory.

See the extensive set of examples in wfopen (p. 472).

Cross-references

See "Creating a Page by Loading a Workfile or Data Source" on page 85 of *User's Guide I* for discussion.

See also wfopen (p. 472) and pagecreate (p. 394).

pagerefresh	Object Container, Data, and File Commands

Refresh all links and auto-series in the active workfile page. Primarily used to refresh links that use external database data.

Syntax

pagerefresh

Cross-references

See "Creating a Database Link" on page 348 and "Understanding the Cache" on page 349 of *User's Guide I*.

See also Chapter 8. "Series Links," on page 219 of *User's Guide I* for a description of link objects, and "Auto-Updating Series" on page 189 of *User's Guide I* for a discussion of auto-updating series.

See wfrefresh (p. 485) to reference an entire workfile.

See also Link::link (p. 318) and Link::linkto (p. 319) in the *Object Reference*.

pagerename	Object Container, Data, and File Commands

Rename the specified workfile page.

Syntax

pagerename *old_name new_name*

renames the *old_name* page in the default workfile to *new_name*. Page names are case-insensitive for purposes of comparison, even though they are displayed using the input case.

Examples

```
pagerename Page1 StateByYear
```

Cross-references

See also `pagecreate` (p. 394).

pagesave	Object Container, Data, and File Commands

Save the active page in the default workfile as an EViews workfile (.WF1 file) or as a foreign data source.

Syntax

pagesave*(options) [path\]filename*

pagesave*(options) source_description* [@**keep** *keep_list]* [@**drop** *drop_list]* [@**keepmap** *keepmap_list]* [@**dropmap** *dropmap_list]* [@**smpl** *smpl_spec]*

pagesave*(options) source_description table_description* [@**keep** *keep_list]* [@**drop** *drop_list]* [@**keepmap** *keepmap_list]* [@**dropmap** *dropmap_list]* [@**smpl** *smpl_spec]*

The command saves the active page in the specified directory using *filename*. By default, the page is saved as an EViews workfile, but options may be used to save all or part of the page in a foreign file or data source. See `wfopen` (p. 472) for details on the syntax of *source_descriptions* and *table_descriptions*. Note, in particular, that you may use the *byrow table_description* for Excel 2007 files to instruct EViews to save the series by row (instead of the standard by column).

Options

type = *arg*, t = *arg*	Optional type specification: (*see table below*). Note that ODBC support is provided only in the *EViews Enterprise Edition*.
mode = *arg*	Specify whether to create a new file, overwrite an existing file, or update an existing file. *arg* may be "create" (create new file only; error on attempt to overwrite) or "update" (only available for Excel files, and only if Excel is installed) (update an existing file, only overwriting the area specified by the range = *table_description*). If the "mode = " option is not used, EViews will create a new file, unless the file already exists in which case it will overwrite it.
maptype = *arg*	Write selected maps as: numeric ("n"), character ("c"), both numeric and character ("b").
nomapval	Do not write mapped values for series with attached value labels (the default is to write mapped values)
noid	Do not write observation identifiers to foreign data files (by default, EViews will include a column with the date or observation identifier).

The following table summaries the various foreign formats, along with the corresponding "type = " keywords:

	Option Keywords
Access	"access"
Aremos-TSD	"a", "aremos", "tsd"
Binary	"binary"
dBASE	"dbase"
Excel (through 2003)	"excel"
EViews Workfile	---
Gauss Dataset	"gauss"
GiveWin/PcGive	"g", "give"
HTML	"html"
Lotus 1-2-3	"lotus"
ODBC Dsn File	"dsn"
ODBC Data Source	"odbc"
MicroTSP Workfile	"dos", "microtsp"
MicroTSP Mac Workfile	"mac"

RATS 4.x	“r”, “rats”
RATS Portable / TROLL	“l”, “trl”
SAS Program	“sasprog”
SAS Transport	“sasxport”
SPSS	“spss”
SPSS Portable	“spssport”
Stata (Version 7 Format)	“stata”
Text / ASCII	“text”
TSP Portable	“t”, “tsp”

Note that if you wish to save your Excel 2007 XML file with macros enabled, you should specify an explicit filename extension of “.XLSM”.

Examples

```
pagesave new_wf
```

saves the current EViews workfile page as “New_wf.WF1” in the default directory.

```
pagesave "c:\documents and settings\my data\consump"
```

saves the current workfile page as “Consump.WF1” in the specified path.

To save the current page in a macro-enabled Excel 2007 file, you should specify the explicit filename extension “.XLSM”:

```
pagesave(type=excelxml, mode=update) range=”Sheet2!a1” byrow
    macro.xlsm @keep gdp unemp
```

Cross-references

See “Saving a Workfile” on page 75 in the *User’s Guide I*.

See also wfopen (p. 472) and wfsave (p. 485).

pageselect — Object Container, Data, and File Commands

Make the specified page in the default workfile the active page.

Syntax

pageselect *pgname*

where *pgname* is the name of a page in the default workfile.

Examples

```
pageselect page2
```

changes the active page to PAGE2.

Cross-references

See also wfselect (p. 488).

pagestack	Object Container, Data, and File Commands

Create a panel structured workfile page using series, alphas, or links from the default workfile page (convert repeated series to repeated observations).

Series in the new panel workfile may be created by stacking series, alphas, and links whose names contain a pattern (series with names that differ only by a "stack identifier"), or by repeating a single series, alpha, or link, for each value in a set of stack identifiers.

Syntax

pagestack*(options) stack_id_list [@ series_to_stack]*

pagestack*(options) pool_name [@ series_to_stack]*

pagestack*(options) series_name_pattern [@ series_to_stack]*

The resulting panel workfile will use the identifiers specified in one of the three forms of the command:

- *stack_id_list* includes a list of the ID values (*e.g.*, "US UK JPN").
- *pool_name* is the name of a pool object that contains the ID values.
- *series_name_pattern* contains an expression from which the ID values may be determined. The pattern should include the "?" character as a stand in for the parts of the series names containing the stack identifiers. For example, if "CONS?" is the *series_name_pattern*, EViews will find all series with names beginning with "CONS" and will extract the IDs from the trailing parts of the observed names.

The *series_to_stack* list may contain two types of entries: stacked series (corresponding to sets of series, alphas, and links whose names contain the stack IDs) and simple series (other series, alphas, and links).

To stack a set of series whose names differ only by the stack IDs, you should enter an expression that includes the "?" character in place of the IDs. You may list the names of a single stacked series (*e.g.*, "GDP?" or "?CONS"), or you may use expressions containing the wildcard character "*" (*e.g.*, "*?" and "?C*") to specify multiple series.

By default, the stacked series will be named in the new workfile using the base portion of the series name (if you specify "?CONS" the stacked series will be named "CONS"), and will contain the values of the individual series stacked one on top of another. If one of the indi-

vidual series associated with a particular stack ID does not exist, the corresponding stacked values will be assigned the value NA.

Individual (simple) series may also be stacked. You may list the names of individual simple series (*e.g.*, "POP INC"), or you can specify your series using expressions containing the wildcard character "*" (*e.g.*, "*", "*C", and "F*"). A simple series will be stacked on top of itself, once for each ID value. If the target workfile page is in the same workfile, EViews will create a link in the new page; otherwise, the stacked series will contain (repeated) copies of the original values.

When evaluating wildcard expressions, stacked series take precedence over simple series. This means that simple series wildcards will be processed using the list of series not already included as a stacked series.

If the *series_to_stack* list is not specified, the expression "*? *", is assumed.

Options

?=*name_patt*, idreplace = *name_patt*	Specifies the characters to use instead of the identifier, "?", in naming the stacked series. By default, the *name_patt* is blank, indicating, for example, that the stacked series corresponding to the pattern "GDP?" will be named "GDP" in the stacked workfile page. If pattern is set to "STK", the stacked series will be named GDPSTK.
interleave	Interleave the observations in the destination stacked workfile (stack by using all of the series values for the first source observation, followed by the values for the second observation, and so on). The default is to stack observations by identifier (stack the series one on top of each other).
wf=*wf_name*	Optional name for the new workfile. If not provided, EViews will create a new page in the default workfile.
page=*page_name*	Optional name for the newly created page. If not provided, EViews will use the next available name of the form "Untitled##", where ## is a number.

Examples

Consider a workfile that contains the seven series: GDPUS, GDPUK, GDPJPN, CONSUS, CONSUK, CONSJPN, CONSFR, and WORLDGDP.

```
pagestack us uk jpn @ *?
```

creates a new, panel structured workfile page with the series GDP and CONS, containing the stacked GDP? series (GDPUS, GDPUK, and GDPJPN) and stacked CONS? series (CONSUS,

CONSUK, and CONSJPN). Note that CONSFR and WORLDGDP will not be copied or stacked.

We may specify the stacked series list explicitly. For example:

```
pagestack(page=stackctry) gdp? @ gdp? cons?
```

first determines the stack IDs from the names of series beginning with "GDP", the stacks the GDP? and CONS? series. Note that this latter example also names the new workfile page STACKCTRY.

If we have a pool object, we may instruct EViews to use it to specify the IDs:

```
pagestack(wf=newwf, page=stackctry) countrypool @ gdp? cons?
```

Here, the panel structured page STACKCTRY will be created in the workfile NEWWF.

Simple series may be specified by adding them to the stack list, either directly, or using wildcard expressions. Both commands:

```
pagestack us uk jpn @ gdp? cons? worldgdp consfr
pagestack(wf=altwf) us uk jpn @ gdp? cons? *
```

stack the various GDP? and CONS? series on top of each other, and stack the simple series GDPFR and WORLDGDP on top of themselves.

In the first case, we create a new panel structured page in the same workfile containing the stacked series GDP and CONS and link objects CONSFR and WORLDGDP, which repeat the values of the series. In the second case, the new panel page in the workfile ALTWF will contain the stacked GDP and CONS, and series named CONSFR and WORLDGDP containing repeated copies of the values of the series.

The following two commands are equivalent:

```
pagestack(wf=newwf) us uk jpn @ *? *
pagestack(wf=newwf) us uk jpn
```

Here, every series, alpha, and link in the source workfile is stacked and copied to the destination workfile, either by stacking different series containing the *stack_id* or by stacking simple series on top of themselves.

The "?=" option may be used to prevent name collision.

```
pagestack(?="stk") us uk jpn @ gdp? gdp
```

stacks GDPUS, GDPUK and GDPJPN into a series called GDPSTK and repeats the values of the simple series GDP in the destination series GDP.

Cross-references

For additional discussion, see "Stacking a Workfile" on page 293 in *User's Guide I*. See also pageunstack (p. 411).

pagestruct	Object Container, Data, and File Commands

Assign a structure to the active workfile page.

Syntax

pagestruct*(options) [id_list]*

pagestruct*(options)* *

where *id_list* is an (optional) list of ID series. The "*" may be used as shorthand for the indices currently in place.

If an *id_list* is provided, EViews will attempt to auto determine the workfile structure. Auto-determination may be overridden through the use of options.

If you do not provide an *id_list*, the workfile will be restructured as a regular frequency workfile. In this case, either the "none" or the "freq = " and "start = " options described below must be provided.

Options

none	Remove the existing workfile structure.
freq = *arg*	Specifies a regular frequency; if not provided EViews will auto-determine the frequency. The frequency may be specified as "a" (annual), "s" (semi-annual), "q" (quarterly), "m" (monthly), "w" (weekly), "d" (5-day daily), "7" (7-day daily), or "u" (unstructured/undated).
start = *arg*	Start date of the regular frequency structure; if not specified, defaults to "@FIRST". Legal values for *arg* are described below.
end = *arg*	End date of the regular frequency structure; if not specified, defaults to "@LAST". Legal values for *arg* are described below.
regular, reg	When used with a date ID, this option informs EViews to insert observations (if necessary) to remove gaps in the date series so that it follows the regular frequency calendar. The option has no effect unless a date index is specified.

create	Allow creation of a new series to serve as an additional ID series when duplicate ID values are found in any group. EViews will use this new series as the observation ID. The default is to prompt in interactive mode and to fail in programs.
balance = *arg*, bal = *arg*	Balance option (for panel data) describing how EViews should handle data that are unbalanced across ID (cross-section) groups. The *arg* should be formed using a combination of starts ("s"), ends ("e"), and middles ("m"), as in "balance = se" or "balance = m". If balancing starts (*arg* contains "s"), EViews will (if necessary) add observations to your workfile so that each cross-section begins at the same observation (the earliest date or observation observed). If balancing ends (*arg* contains "e"), EViews will add any observations required so that each cross-section ends at the same point (the last date or observation observed). If balancing middles (*arg* contains "m") EViews will add observations to ensure that each cross-section has consecutive observations from the earliest date or observation for the cross-section to the last date or observation for the cross-section. Note that "balance = m" is implied by the "regular" option.
dropna	Specifies that observations which contain missing values (NAs or blank strings) in any ID series (including the date or observation ID) be removed from the workfile. If "dropna" is not specified and NAs are found, EViews will prompt in interactive mode and fail in programs.
dropbad	Specifies that observations for which any of the date index series contain values that do not represent dates be removed from the workfile. If "dropbad" is not provided and bad dates are present, EViews will prompt in interactive mode and fail in programs.

The values for start and end dates should contain date literals (actual dates or periods), *e.g.*, "1950q1" and "2/10/1951", "@FIRST", or "@LAST" (first and last observed dates in the date ID series). Date literals must be used for the "start = " option when restructuring to a regular frequency.

In addition, offsets from these date specifications can be specified with a " + " or "–" followed by an integer: "@FIRST-5", "@LAST + 2", "1950m12 + 6". Offsets are most often used when resizing the workfile to add or remove observations from the endpoints.

Examples

```
pagestruct state industry
```

structures the workfile using the IDs in the STATE and INDUSTRY series.

A date ID series (or a series used to form a date ID) should be tagged using the "@DATE" keyword. For example:

```
pagestruct state @date(year)
pagestruct(regular) @date(year, month)
```

A "*" may be used to indicate the indices defined in the current workfile structure.

```
pagestruct(end=@last+5) *
```

adds 5 observations to the end of the current workfile.

When you omit the *id_list*, EViews will attempt to restructured the workfile to a regular frequency. In this case you must either provide the "freq = " and "start = " options to identify the regular frequency structure, or you must specify "none" to remove the existing structure:

```
pagestruct(freq=a, start=1950)
pagestruct(none)
```

Cross-references

For extensive discussion, see "Structuring a Workfile," beginning on page 249 in the *User's Guide I*.

pageunlink	Object Container, Data, and File Commands

Break links in all link objects and auto-updating series (formulae) in the active workfile page.

You should use some caution with this command as you will not be prompted before the links and auto-updating series are converted.

Syntax

pageunlink

Examples

```
pageunlink
```

breaks links in all pages of the active workfile page.

Cross-references

See Chapter 8. "Series Links," on page 219 of the *User's Guide I* for a description of link objects, and "Auto-Updating Series" on page 189 of the *User's Guide I* for a discussion of auto-updating series.

See also `Link::link` (p. 318) and `Link::linkto` (p. 319) in the *Object Reference*.

See `unlink` (p. 458) and `wfunlink` (p. 489) for object and workfile based unlinking, respectively.

pageunstack	Object Container, Data, and File Commands

Unstack workfile page (convert repeated observations to repeated series).

Create a new workfile page by taking series objects (series, alphas, or links) in the default workfile page and breaking them into multiple series (or alphas), one for each distinct value found in a user supplied list of series objects. Typically used on a page with a panel structure.

Syntax

pageunstack*(options) stack_id obs_id [@ series_to_unstack]*

where *stack_id* is a single series containing the unstacking ID values used to identify the individual unstacked series, *obs_id* is a series containing the observation IDs, and *series_to_unstack* is an optional list of series objects to be copied to the new workfile.

Options

namepat = *name_pattern*	Specifies the pattern from which unstacked series names are constructed, where "*" indicates the original series name and "?" indicates the stack ID. By default the *name_pattern* is "*?", indicating, for example, that if we have the IDs "US", "UK", "JPN", the unstacked series corresponding to the series GDP should be named "GDPUS", "GDPUK", "GDPJPN" in the unstacked workfile page.
wf = *wf_name*	Optional name for the new workfile. If not provided, EViews will create a new page in the default workfile.
page = *page_name*	Optional name for the newly created page. If not provided, EViews will use the next available name of the form "Untitled##", where ## is a number.

Examples

Consider a workfile that contains the series GDP and CONS which contain the values of Gross Domestic Product and consumption for three countries stacked on top of each other. Suppose further there is an alpha object called COUNTRY containing the observations "US", "UK", and "JPN", which identify which from which country each observation of GDP and CONS comes. Finally, suppose there is a date series DATEID which identifies the date for each observation. The command:

```
pageunstack country dateid @ gdp cons
```

creates a new workfile page using the workfile frequency and dates found in DATEID. The page will contain the 6 series GDPUS, GDPUK, GDPJPN, CONSUS, CONSUK, and CONSJPN corresponding to the unstacked GDP and CONS.

Typically the source workfile described above would be structured as a dated panel with the cross-section ID series COUNTRY and the date ID series DATEID. Since the panel has built-in date information, we may use the "@DATE" keyword as the DATEID. The command:

```
pageunstack country @date @ gdp cons
```

uses the date portion of the current workfile structure to identify the dates for the unstacked page.

The *stack_id* must be an ordinary, or an alpha series that uniquely identifies the groupings to use in unstacking the series. *obs_id* may be one or more ordinary series or alpha series, the combination of which uniquely identify each observation in the new workfile.

You may provide an explicit list of series to unstack following an "*@*" immediately after the *obs_id*. Wildcards may be used in this list. For example:

```
pageunstack country dateid @ g* c*
```

unstacks all series and alphas that have names that begin with "G" or "C'.

If no *series_to_unstack* list is provided, all series in the source workfile will be unstacked. Thus, the two commands:

```
pageunstack country dateid @ *
pageunstack country dateid
```

are equivalent.

By default, series are named in the destination workfile page by appending the *stack_id* values to the original series name. Letting "*" stand for the original series name and "?" for the *stack_id*, names are constructed as "*?". This default may be changed using the "namepat = " option. For example:

```
pageunstack(namepat="?_*") country dateid @ gdp cons
```

creates the series US_GDP, UK_GDP, JPN_GDP, etc.

Cross-references

For additional discussion and examples, see "Unstacking a Workfile" on page 287 of the *User's Guide I*. See also pagestack (p. 405).

param	Global Commands

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

param *coef_name1 number1 [coef_name2 number2 coef_name3 number3…]*

List, in pairs, the names of the coefficient vector and its element number followed by the corresponding starting values for any 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 that is initialized with zeros. The second line sets the second and third elements of BETA to 0.1 and 0.5, respectively.

Cross-references

See "Starting Values" on page 46 of the *User's Guide II* for a discussion of setting initial values in nonlinear estimation.

plot	Object Creation Commands

Line graph.

Provided for backward compatibility. See line (p. 828).

print	Command Actions

Sends views of objects to the default printer.

Syntax

print*(options) object1 [object2 object3 …]*

print*(options) object_name.view_command*

`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 an object, `print` will print the default view for the object.

Options

p	Print in portrait orientation.
l	Print in landscape orientation.

The default orientation is set by clicking on **Print Setup**.

Examples

```
print gdp log(gdp) d(gdp) @pch(gdp)
```

sends a table of GDP, log of GDP, first difference of GDP, and the 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 orientation, you may use the commands:

```
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 779 of the *User’s Guide II* for a discussion of print options and the **Print Setup** dialog.

See `output` (p. 387) for print redirection.

probit	Interactive Use Commands

Estimation of binary dependent variable models with normal errors.

Equivalent to "`binary(d=n)`".

See `binary` (p. 279).

program	Programming Commands

Declare a program.

Syntax

program *[path\]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 105 of the *User's Guide I* for further details, and examples of writing EViews programs.

See also `open` (p. 378).

qreg	Interactive Use Commands

Estimate a quantile regression specification.

Syntax

qreg*(options) y x1 [x2 x3 ...]*
qreg*(options) linear_specification*

Options

quant = *number* *(default = 0.5)*	Quantile to be fit (where *number* is a value between 0 and 1).
w = *arg*	Weight series or expression. Note: *we recommend that, absent a good reason, you employ the default settings* **Inverse std. dev. weights** *("wtype = istdev") with* **EViews default** *scaling ("wscale = eviews") for backward compatibility with versions prior to EViews 7.*
wtype = *arg* *(default = "istdev")*	Weight specification type: inverse standard deviation ("istdev"), inverse variance ("ivar"), standard deviation ("stdev"), variance ("var").
wscale = *arg*	Weight scaling: EViews default ("eviews"), average ("avg"), none ("none"). The default setting depends upon the weight type: "eviews" if "wtype = istdev", "avg" for all others.
cov = *arg* (*default* = "sandwich")	Method for computing coefficient covariance matrix: "iid" (ordinary estimates), "sandwich" (Huber sandwich estimates), "boot" (bootstrap estimates). When "cov = iid" or "cov = sandwich", EViews will use the sparsity nuisance parameter calculation specified in "spmethod = " when estimating the coefficient covariance matrix.
bwmethod = *arg* *(default = "hs")*	Method for automatically selecting bandwidth value for use in estimation of sparsity and coefficient covariance matrix: "hs" (Hall-Sheather), "bf" (Bofinger), "c" (Chamberlain).
bw = *number*	Use user-specified bandwidth value in place of automatic method specified in "bwmethod = ".
bwsize = *number* *(default = 0.05)*	Size parameter for use in computation of bandwidth (used when "bw = hs" and "bw = bf").
spmethod = *arg* *(default = "kernel")*	Sparsity estimation method: "resid" (Siddiqui using residuals), "fitted" (Siddiqui using fitted quantiles at mean values of regressors), "kernel" (Kernel density using residuals) Note: "spmethod = resid" is not available when "cov = sandwich".
btmethod = *arg* *(default = "pair")*	Bootstrap method: "resid" (residual bootstrap), "pair" (xy-pair bootstrap), "mcmb" (MCMB bootstrap), "mcmba" (MCMB-A bootstrap).

btreps = *integer* *(default* = 100*)*	Number of bootstrap repetitions
btseed = *positive integer*	Seed the bootstrap random number generator. If not specified, EViews will seed the bootstrap random number generator with a single integer draw from the default global random number generator.
btrnd = *arg* *(default* = "kn" or method previously set using `rndseed` (p. 423)*)*.	Type of random number generator for the bootstrap: improved Knuth generator ("kn"), improved Mersenne Twister ("mt"), Knuth's (1997) lagged Fibonacci generator used in EViews 4 ("kn4") L'Ecuyer's (1999) combined multiple recursive generator ("le"), Matsumoto and Nishimura's (1998) Mersenne Twister used in EViews 4 ("mt4").
btobs = *integer*	Number of observations for bootstrap subsampling (when "bsmethod = pair"). Should be significantly greater than the number of regressors and less than or equal to the number of observations used in estimation. EViews will automatically restrict values to the range from the number of regressors and the number of estimation observations. If omitted, the bootstrap will use the number of observations used in estimation.
btout = *name*	*(optional)* Matrix to hold results of bootstrap simulations.
k = *arg* *(default* = "e"*)*	Kernel function for sparsity and coefficient covariance matrix estimation (when "spmethod = kernel"): "*e*" (Epanechnikov), "*r*" (Triangular), "*u*" (Uniform), "*n*" (Normal–Gaussian), "*b*" (Biweight–Quartic), "*t*" (Triweight), "*c*" (Cosinus).
m = *integer*	Maximum number of iterations.
s	Use the current coefficient values in "C" as starting values (see also `param` (p. 413)).
s = *number* *(default* = 0*)*	Determine starting values for equations. Specify a number between 0 and 1 representing the fraction of preliminary least squares coefficient estimates. Note that out of range values are set to the default.
showopts / -showopts	[Do / do not] display the starting coefficient values and estimation options in the estimation output.

coef = *arg*	Specify the name of the coefficient vector (if specified by list); the default behavior is to use the "C" coefficient vector.
prompt	Force the dialog to appear from within a program.
p	Print estimation results.

Examples

```
qreg y c x
```

estimates the default least absolute deviations (median) regression for the dependent variable Y on a constant and X. The estimates use the Huber Sandwich method for computing the covariance matrix, with individual sparsity estimates obtained using kernel methods. The bandwidth uses the Hall and Sheather formula.

```
qreg(quant=0.6, cov=boot, btmethod=mcmba) y c x
```

estimates the quantile regression for the 0.6 quantile using MCMB-A bootstrapping to obtain estimates of the coefficient covariance matrix.

Cross-references

See Chapter 14. "Quantile Regression," on page 423 of the *User's Guide II* for a discussion of the quantile regression.

range

Object Container, Data, and File Commands

Reset the workfile range for a regular frequency workfile.

No longer supported. See the replacement command `pagestruct` (p. 408).

read

Object Container, Data, and File Commands

Import data from a foreign disk file into series.

May be used to import data into an existing workfile from a text, Excel, or Lotus file on disk.

Unless you need to merge data into an *existing* workfile page, we recommend that you use the more powerful, easy-to-use tools for reading data (see "Creating a Workfile by Reading from a Foreign Data Source" on page 47 of the *User's Guide I*). See `pageload` (p. 400), and `wfopen` (p. 472) for command details.

Syntax

read*(options) [path\]file_name name1 [name2 name3 ...]*

read*(options) [path\]file_name n*

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. 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.

The input specification follows the source file name. 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 series in the file instead of a list of names. EViews will name the *n* series using the names 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.

To import data into alpha series, you must specify the names of your series, and should enter the tag "$" following the series name (*e.g.*, "NAME $ INCOME CONSUMP").

Options

prompt	Force the dialog to appear from within a program.

File type options

t = dat, txt	ASCII (plain text) files.
t = wk1, wk3	Lotus spreadsheet files.
t = xls	Excel spreadsheet files.

If you do not specify the "*t*" option, EViews uses the file name extension to determine the file type. If you specify the "*t*" option, the file name extension will not be used to determine the file type.

Options for ASCII text files

t	Read data organized by series. Default is to read by observation with series in columns.
na = *text*	Specify text for NAs. Default is "NA".
d = t	Treat tab as delimiter (note: you may specify multiple delimiter options). The *default* is "d = c" only.
d = c	Treat comma as delimiter.
d = s	Treat space as delimiter.
d = a	Treat alpha numeric characters as delimiter.
custom = *symbol*	Specify symbol/character to treat as delimiter.
mult	Treat multiple delimiters as one.

name	Series names provided in file.
label = *integer*	Number of lines between the header line and the data. Must be used with the "name" option.
rect *(default)* / norect	[Treat / Do not treat] file layout as rectangular.
skipcol = *integer*	Number of columns to skip. Must be used with the "*rect*" option.
skiprow = *integer*	Number of rows to skip. Must be used with the "rect" option.
comment = *symbol*	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 = *symbol*	Specify symbol/character for currency data.

Options for spreadsheet (Lotus, Excel) files

t	Read data organized by series. Default is to read by observation with series in columns.
letter_number (default = "b2"*)*	Coordinate of the upper-left cell containing data.
s = *sheet_name*	Sheet name for Excel 5–8 Workbooks.

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 in the file are listed by observation, the missing value NA is coded as a "." (dot or period), and there are three series, which are to be named ID, LWAGE, HRS (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 using names provided in the file.

```
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 129 of the *User's Guide I* for a discussion and examples of importing data from external files.

See also `write` (p. 491).

rename	Object Utility Commands

Rename an object in the active workfile or database.

Syntax

rename *old_name new_name [old_name1 new_name1 [old_name2 new_name2 [...]]]*

After the `rename` keyword, list the pairs of old object names followed by the new names. Note that the name specifications may include matching wildcard patterns.

Examples

```
rename temp_u u2
```

renames an object named TEMP_U as U2.

```
rename aa::temp_u aa::u2
```

renames the object TEMP_U to U2 in database AA.

```
rename a* b*
```

renames all objects beginning with the letter "A" to begin with the letter "B".

```
rename a1 a2 b1 b2
```

renames A1 to A2 and B1 to B2.

Cross-references

See Chapter 4. "Object Basics," on page 93 of the *User's Guide I* for a discussion of working with objects in EViews.

reset	Interactive Use Commands

Compute Ramsey's regression specification error test.

Syntax

reset*(n, options)*

You must provide the number of powers of fitted terms *n* to include in the test regression.

Options

prompt	Force the dialog to appear from within a program.
p	Print the test result.

Examples

```
ls lwage c edu race gender
reset(2)
```

carries out the RESET test by including the square and the cube of the fitted values in the test equation.

Cross-references

See "Ramsey's RESET Test" on page 188 of the *User's Guide II* for a discussion of the RESET test.

rndint

Object Assignment Commands

Generate uniform random integers.

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

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 with integers drawn randomly from 0 to 10. Note that unlike standard series assignment using `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 "Statistical Distribution Functions" on page 525 of the *User's Guide I*.

See also nrnd (p. 561), rnd (p. 563) and rndseed (p. 423).

rndseed	Global Commands

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.

Note that EViews 5 has updated the seeding routines of two of our pseudo-random number generators (backward compatible options are provided). It is strongly recommended that you use new generators.

Syntax

rndseed*(options) integer*

Follow the `rndseed` keyword with the optional generator type and an integer for the seed.

Options

type = *arg* *(default* = "kn"*)*	Type of random number generator: improved Knuth generator ("kn"), improved Mersenne Twister ("mt"), Knuth's (1997) lagged Fibonacci generator used in EViews 4 ("kn4)", L'Ecuyer's (1999) combined multiple recursive generator ("le"), Matsumoto and Nishimura's (1998) Mersenne Twister used in EViews 4 ("mt4").

When EViews starts up, the default generator type is set to the improved Knuth lagged Fibonacci generator. Unless changed using `rndseed`, Knuth's generator will be used for subsequent pseudo-random number generation.

	Knuth ("kn4")	L'Ecuyer ("le")	Mersenne Twister ("mt4")
Period	2^{129}	2^{319}	$2^{19937} - 1$
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 "Statistical Distribution Functions" on page 525 of the *User's Guide I*.

At press time, further information on the improved seeds may be found on the web at the following addresses:

Knuth generator: http://sunburn.stanford.edu/~knuth/news02.html#rng

Mersenne twister: http://www.math.keio.ac.jp/~matumoto/MT2002/emt19937ar.html

See also `nrnd` (p. 561), `rnd` (p. 563) and `rndint` (p. 422).

References

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 2, Semi-numerical Algorithms, 3rd edition*, Reading, MA: Addison-Wesley Publishing Company. *Note: the C implementation of the lagged Fibonacci generator is described in the errata to the 2nd edition, downloadable from Knuth's web site.*

L'Ecuyer, P. (1999). "Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators," *Operations Research*, 47(1), 159-164

Matsumoto, M. and T. Nishimura (1998). "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator," *ACM Transactions on Modeling and Computer Simulation*, 8(1), 3-30.

robustls	Equation Methods

Estimates an equation using robust least squares.

You may perform three different types of robust estimation: M-estimation, S-estimation and MM-estimation.

Syntax:

robustls*(options) y x1 [x2 x3…]*

Enter the `robustls` keyword, followed by the dependent variable and a list of the regressors.

Options

method = *arg* (*default* = "m")	Robust estimation method: "m" (M-estimation), "s" (S-estimation) or "mm" (MM-estimation).
cov = *arg* (*default* = "type1")	Covariance method type: "type1", "type2", or "type3".
tuning = *number*	Specify a value for the tuning parameter. If a value is not specified, EViews will use the default tuning parameter for the type of estimation and weighting function (if applicable).
c = *s*	Convergence criterion. The criterion will be set to the nearest value between 1e-24 and 0.2.
coef = *arg*	Specify the name of the coefficient vector (if specified by list); the default behavior is to use the "C" coefficient vector.
m = *integer*	Maximum number the number of iterations.
prompt	Force the dialog to appear from within a program.
p	Print results.

M-estimation Options

fn = *arg* (*default* = "bisquare")	Weighting function used during M-estimation: "andrews" (Andrews), "bisquare" (Bisquare), "cauchy" (Cauchy), "fair", "huber", "huberbi" (Huber-bisquare), "logistic" (Logistic), "median", "tal" (Talworth), "Welsch" (Welsch).
scale = *arg* (*default* = "madzero")	Scaling method used for calculating the scalar parameter during M estimation: "madzero" (median absolute deviation, zero centered), "madmed" (median absolute deviation, median centered), "huber" (Huber scaling).
hmat	Use the hat-matrix to down-weight observations with high leverage.

S and MM estimation options

compare = *integer* (*default* = 4)	Number of comparison sets.
refine = *integer* (*default* = 2)	Number of refinements.
trials = *integer* (*default* = 200)	Number of trials.
subsmpl = *integer*	Specifies the size of the subsamples. Note, the default is number of coefficients in the regression.
seed = *number*	Specifies the random number generator seed
rng = *arg*	Specifies the type of random number generator. The key can be; improved Knuth generator ("kn"), improved Mersenne Twister ("mt"), Knuth's (1997) lagged Fibonacci generator used in EViews 4 ("kn4") L'Ecuyer's (1999) combined multiple, recursive generator ("le"), Matsumoto and Nishimura's (1998) Mersenne Twister used in EViews 4 ("mt4").

MM estimation options

mtuning = *arg*	M-estimator tuning parameter. Note the S-estimator tuning parameter is set with the "tuning = " option outlined above.
hmat	Use the hat-matrix to down-weight observations with high leverage during m-estimation.

Examples

The following examples use the "Rousseeuw and Leroy.wf1" file located in the EViews application data directory.

```
robustls salinity c lagsal trend discharge
```

This line estimates a simple M-type robust estimation, with SALINITY as the dependent variable, and a constant, LAGSAL, TREND and DISCHARGE as independent variables.

The line:

```
robustls(method=mm, tuning=2.937, mtuning=3.44, cov=type2)
    salinity c lagsal trend discharge
```

estimates the same model, but using MM-estimation, with an S tuning constant of 2.937, an M tuning constant of 3.44, and using Huber Type II standard errors.

Cross-references

See Chapter 11. "Robust Least Squares," beginning on page 349 of *User's Guide II* for discussion.

run	Programming Commands

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

run*(options) [path\]prog_name(prog_options) [%0 %1 …]*

If you wish to pass one or more options to the program, you should enclose them in parentheses immediately after the filename. If the program has arguments, you should list them after the filename.

EViews first checks to see if the specified program is in memory. If the program is not located, EViews then looks for the program on disk in *the current working directory*, or in the specified path. The program file should have a ".PRG" extension, which you need not specify in the *prog_name*.

Options

integer (default = 1)	Set maximum errors allowed before halting the program.
c	Run program file without opening a window for display of the program file.
verbose / quiet	Verbose mode in which messages will be sent to the status line at the bottom of the EViews window (slower execution), or quiet mode which suppresses workfile display updates (faster execution).
v / q	Same as [verbose / quiet].
ver4 / ver5	Execute program in [version 4 / version 5] compatibility mode.
this = *object_name*	Set the `_this` object for the executed program. If omitted, the executed program will inherit the `_this` object from the parent program, or from the current active workfile object when the `exec` command is issued from the command window.

Examples

```
run(q) simul(h=2) x xhat
```

quietly runs a program named "Simul.prg" from the default directory using options string "h = 2" and arguments %0 = X and %1 = 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 `exec` (p. 328) command or an `include` (p. 661) statement.

Cross-references

See "Executing a Program" on page 108 of the *User's Guide I* for further details.

See also `exec` (p. 328) and `include` (p. 661).

save	Object Container, Data, and File Commands

Save current workfile to disk.

This usage is provided only for backward compatibility, as it has been replaced with the equivalent `wfsave` (p. 485) command.

Syntax

save *[path\]file_name*

Follow the keyword with a name for the file. If an explicit path is not specified, the file will be stored in the default directory, as set in the **File Locations** global options.

Examples

```
save MyWorkfile
```

saves the current workfile with the name MYWORKFILE.WF1 in the default directory.

```
save c:\data\MyWF1
```

saves the current workfile with the name MYWF1.WF1 in the specified directory.

Cross-references

See `wfsave` (p. 485).

seas	Interactive Use Commands

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 `Series::x11` (p. 547) and `Series::x12` (p. 549) in the *Object Reference*.

Syntax

seas*(options) series_name name_adjust [name_fac]*

List the name of the original series and the name to be given to the seasonally adjusted series. You may optionally include an additional name for the seasonal factors. `seas` will display the seasonal factors using the convention of the Census X11 program.

Options

m	Multiplicative (ratio to moving average) method.
a	Additive (difference from moving average) method.
prompt	Force the dialog to appear from within a program.

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.

Cross-references

See "Seasonal Adjustment" on page 394 of the *User's Guide I* for a discussion of seasonal adjustment methods.

See also `seasplot` (p. 849), `Series::x11` (p. 547), and `Series::x12` (p. 549) in the *Object Reference*.

setcell	Table Commands

Insert contents into cell of a table.

The `setcell` command puts a string or number into a cell of a table.

Syntax

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.

Note that in general, that unless you wish to control the formatting, assignment statements of the form

```
Mytable(1,1) = "hello"
```

are easier than using the default `setcell` statement:

```
Setcell(mytable,1,1,"hello")
```

Cross-references

Chapter 3. "Working with Tables and Spreadsheets," on page 45 describes table formatting using commands. See "Table Objects," beginning on page 691 of the *User's Guide I* for a discussion and examples of table formatting in EViews.

See also `Table::setjust` (p. 709) and `Table::setformat` (p. 703) in the *Object Reference*. Note that this command is supported primarily for backward compatibility. There is a more extensive set of table procs for working with and customizing tables. See "Table Procs," on page 688 in *Object Reference*.

setcolwidth	Table Commands

Set width of a column of a table.

Provided for backward compatibility. See `Table::setwidth` (p. 715) in the *Object Reference* for the new method of setting the width of table and spreadsheet columns.

Syntax

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. By default, each column is approximately 10 characters wide.

Examples

```
setcolwidth(mytab,2,20)
```

sets the second column of table MYTAB to fit approximately 20 characters.

Cross-references

Chapter 3. "Working with Tables and Spreadsheets," on page 45 of the *User's Guide I* describes table formatting using commands. See also "Table Objects" on page 691 of the *User's Guide I* for a discussion and examples of table formatting in EViews.

Note that this command is supported primarily for backward compatibility. There is a more extensive set of table procs for working with and customizing tables. See "Table Procs," on page 688 in the *Object Reference*.

See also `Table::setwidth` (p. 715) and `Table::setheight` (p. 707) in the *Object Reference*.

setline	Table Commands

Place a double horizontal line in a table.

Provided for backward compatibility. For a more general method of setting the line characteristics and borders for a set of table cells, see the table proc `Table::setlines` (p. 710) See also `Table::setwidth` (p. 715) and `Table::setheight` (p. 707) in the *Object Reference*.

Syntax

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 3. "Working with Tables and Spreadsheets," on page 45 of the *User's Guide I* describes table formatting using commands. See also "Table Objects" on page 691 of the *User's Guide I* for a discussion and examples of table formatting in EViews.

See `Table::setlines` (p. 710) in the *Object Reference* for more flexible line drawing tools. Note that this command is supported primarily for backward compatibility. There is a more extensive set of table procs for working with and customizing tables. See "Table Procs," on page 688 in the *Object Reference*.

shell	Programming Commands

Start the Windows command shell, optionally executing a command.

Syntax

shell*(options) [arg1 arg2 arg3…]*

See `spawn` (p. 440) for available options. By default, the Windows command shell will be started in hidden mode with the exit code for success set to zero.

Examples

```
shell mkdir c:\newdir
```

makes a new directory "c:\newdir".

```
shell(out=flist) dir /b *.wf1
```

lists all workfiles in the current directory, saving output in a table named FLIST.

Cross-references

See `spawn` (p. 440) for details on spawning a new process.

show	Command Actions

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

show object_name.*view_command*

show *object1 [object2 object3 ...]*

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 "Object Commands" on page 9 for discussion.

Chapter 1. "Object View and Procedure Reference," on page 2 in the *Object Reference* provides a complete listing of the views of the various objects.

See also `close` (p. 289) and `do` (p. 327).

smooth	Interactive Use Commands

Exponential smoothing.

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

smooth*(method) series_name smooth_name [freq]*

You should follow the `smooth` keyword with the name of the series and a name for the smoothed series. You must also specify the smoothing method in parentheses. 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* and *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*, and *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*, and *z*, between zero and one for the mean, trend, and seasonal parameters, respectively.

Other Options:

prompt	Force the dialog to appear from within a program.
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.

```
smpl @first @last-10
smooth(m,.1,.1,.1) order order_hw
smpl @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 427 of the *User's Guide I* for a discussion of exponential smoothing methods.

smpl	Global Commands

Set sample range.

The `smpl` command sets the workfile sample to use for statistical operations and series assignment expressions.

Syntax

smpl *smpl_spec*

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 "Dates" on page 82. The sample spec 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 keyword.

Special keywords for smpl

The following "@-keywords" can be used in a `smpl` command:

@all	The entire workfile range.
@first	The first observation in the workfile.
@last	The last observation in the workfile.

The following frequency functions may also be used:

@year	The four digit year in which the current observation begins.
@quarter	The quarter of the year in which the current observation begins.
@month	The month of the year in which the current observation begins.
@day	The day of the month in which the current observation begins.
@weekday	The day of the week in which the current observation begins, where Monday is given the number 1 and Sunday is given the number 7.
@hour	The observation hour as an integer. (9:30AM returns 9, 5:15PM returns 17.)
@minute	The observation minute as an integer. (9:30PM returns 30.)
@second	The observation second as an integer.
@hourf	The observation time as a floating point hour. (9:30AM returns 9.5, 5:15PM returns 17.25.)

In panel settings, you may use the additional keywords:

@firstmin	The earliest of the first observations (computed across cross-sections).
@firstmax	The latest of the first observations.
@lastmin	The earliest of the last observations.
@lastmax	The latest of the last observations.

Examples

```
smpl 1955m1 1972m12
```

sets the workfile sample from 1955M1 to 1972M12.

```
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.

The sample may be set for intraday data using optional times after the dates. For example,

```
smpl 1/3/2000 10AM 12/30/2000 2PM
```

removes any observations before 10AM on 1/3/2000 and after 2PM on 12/30/2000.

```
smpl if @hourf<=9.5 and @hourf<=14.5
```

sets the sample to include only observations between and including 9:30AM and 2:30PM.

```
smpl if @minute=0 or @minute=30
```

selects only observations that appear on the half hour.

```
smpl if @weekday=1 and @hourf=10
```

sets the sample to include only observations that appear on Mondays at 10AM.

Cross-references

See "Samples" on page 119 of the *User's Guide I* for a discussion of samples in EViews.

See also `Sample::set` (p. 470) and `Sample::sample` (p. 469) in the *Object Reference*.

solve	Interactive Use Commands

Solve the model.

`solve` finds the solution to a simultaneous equation model for the set of observations specified in the current workfile sample.

Syntax

solve*(options) model_name*

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. 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 `Model::solveopt` (p. 399) in the *Object Reference*.

Examples

```
solve mod1
```

solves the model MOD1 using the default solution method.

```
solve(m=500,e) nonlin2
```

solves the model NONLIN2 with an extended search of up to 500 iterations.

Cross-references

See Chapter 20. "Models," on page 627 of the *User's Guide II* for a discussion of models.

See also `Model::model` (p. 389), `Model::msg` (p. 390) and `Model::solveopt` (p. 399) in the *Object Reference*.

sort	Object Container, Data, and File Commands

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 sort order.

EViews will first remove any workfile structures and then will sort the workfile using the specified settings.

Syntax

sort*(options) arg1 [arg2 arg3…]*

List the name of the series or groups 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 a Workfile" on page 301 of the *User's Guide I*.

spawn	Programming Commands

Spawn a new process.

Syntax

spawn*(options) filename [arg1 arg2 arg3…]*

Follow the keyword with a filename indicating the process to spawn, and optional arguments to be passed to the process.

Options

"n" or "normal"	Create process in normal mode. The process will typically create a maximized window and may wait for user input.
"m" or "minimized"	Create process in a minimized window. Note that some applications may not accept requests to run in minimized mode.
"h" or "hidden"	Create process in hidden mode (without a visible window). Note that some applications may not accept requests to run in hidden mode.

t = *isecs*	Specifies the maximum time in seconds that EViews should wait for the process to complete. If the timeout interval is reached and the process has not completed, EViews will generate an error. A timeout setting of zero may be used to indicate that EViews should not wait for the spawned process to complete. If no timeout option is provided, EViews will wait indefinitely for the process to complete.
exit = *icode*	Specifies the exit code that the process will return if it is completed successfully. If the process returns an exit code other than the specified value, EViews will generate an error. If the exit code option is not specified, EViews will not generate an error no matter what exit code is returned by the process.
out = *tablename*	If an output table name is specified, EViews will capture any data written to standard output by the spawned process and store it into a table object with the specified name in the workfile. Note that this option will have no effect unless either the minimized or hidden option is used and the timeout value is not zero.

Examples

```
spawn "c:\program files\microsoft office\office11\excel.exe"
    test.xls
```

starts a new Excel process, passing it the command line argument "test.xls".

Cross-references

See `shell` (p. 432) for information on starting a Windows command shell.

stats	Interactive Use Commands

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.

`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.

Syntax

stats*(options) ser1 [ser2 ser3 …]*

Options

p	Print the stats 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.

Cross-references

See "Descriptive Statistics & Tests" on page 358 and page 486 of the *User's Guide I* for a discussion of the descriptive statistics views of series and groups.

See also `boxplot` (p. 811) and `hist` (p. 356).

statusline — Programming Commands

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

statusline *message_string*

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 105 of the *User's Guide I* for a discussion and examples of programs, control variables and string variables.

stepls — Interactive Use Commands

Estimation by stepwise least squares.

Syntax

stepls*(options) y x1 [x2 x3 ...] @ z1 z2 z3*

Specify the dependent variable followed by a list of variables to be included in the regression, but not part of the search routine, followed by an "**@**" symbol and a list of variables to be part of the search routine. If no included variables are required, simply follow the dependent variable with an "**@**" symbol and the list of search variables.

Options

method = *arg*	Stepwise regression method: "stepwise" (default), "uni" (uni-directional), "swap" (swapwise), "comb" (combinatorial).
nvars = *int*	Set the number of search regressors. Required for swapwise and combinatorial methods, optional for uni-directional and stepwise methods.
w = *arg*	Weight series or expression. Note: *we recommend that, absent a good reason, you employ the default settings* **Inverse std. dev. weights** *("wtype = istdev") with* **EViews default** *scaling ("wscale = eviews") for backward compatibility with versions prior to EViews 7.*
wtype = *arg* *(default* = "istdev"*)*	Weight specification type: inverse standard deviation ("istdev"), inverse variance ("ivar"), standard deviation ("stdev"), variance ("var").
wscale = *arg*	Weight scaling: EViews default ("eviews"), average ("avg"), none ("none"). The default setting depends upon the weight type: "eviews" if "wtype = istdev", "avg" for all others.
coef = *arg*	Specify the name of the coefficient vector (if specified by list); the default behavior is to use the "C" coefficient vector.
prompt	Force the dialog to appear from within a program.
p	Print estimation results.

Stepwise and uni-directional method options

back	Set stepwise or uni-directional method to run backward. If omitted, the method runs forward.
tstat	Use *t*-statistic values as a stopping criterion. (*Default* uses *p*-values).
ftol = *number* *(default* = 0.5*)*	Set forward stopping criterion value.
btol = *number* *(default* = 0.5*)*	Set backward stopping criterion value.

fmaxstep = *int* *(default = 1000)*	Set the maximum number of steps forward.
bmaxstep = *int* *(default = 1000)*	Set the maximum number of steps backward.
tmaxstep = *int* *(default = 2000)*	Set the maximum total number of steps.

Swapwise method options

minr2	Use minimum R-squared increments. (*default* uses maximum R-squared increments.)

Combinatorial method options

force	Suppress the warning message issued when a large number of regressions will be performed.

Examples

```
stepls(method=comb,nvars=3) y c @ x1 x2 x3 x4 x5 x6 x7 x8
```

performs a combinatorial search routine to search for the three variables from the set of X1, X2, ..., X8, yielding the largest R-squared in a regression of Y on a constant and those three variables.

Cross-references

See "Stepwise Least Squares Regression," beginning on page 46.

store	Object Container, Data, and File Commands

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.

Syntax

store*(options) object_list*

Follow the `store` command keyword with a list of 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 2 and earlier where the default was to store objects in individual databank files)*.

You may 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 parenthe-

ses, in which case all objects without an explicit database name will be stored in the specified database.

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 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

d = *db_name*	Store to the specified database.
i	Store to individual databank files.
1 / 2	Store series in [single / double] precision to save space.
o	Overwrite object in database (default is to merge data, where possible).
g = *arg*	Group store from workfile to database: "s" (copy group definition and series as separate objects), "t" (copy group definition and series as one object), "d" (copy series only as separate objects), "l" (copy group definition only).

If you do not specify the precision option (1 or 2), the global option setting will be used. See "Database Storage Defaults" on page 770 of the *User's Guide II*.

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 109 of the *User's Guide I* discusses exporting data in other file formats. See Chapter 10. "EViews Databases," on page 303 of the *User's Guide I* for a discussion of EViews databases and databank files.

For additional discussion of wildcards, see Appendix A. "Wildcards," on page 683 of the *User's Guide I*.

See also `fetch` (p. 332) and `copy` (p. 306).

switchreg	Interactive Use Commands

Estimate a switching regression model (simple exogenous or Markov).

Syntax

switchreg(*options*) *dependent_var list_of_varying_regressors [* **@nv** *list_of_nonvarying_regressors] [* **@prv** *list_of_probability_regressors]*

List the `switchreg` keyword, followed by options, then the dependent variable and a list of the regressors with regime-varying coefficients, following optionally by the keyword **@nv** and a list of regressors with regime-invariant coefficients, and by the keyword **@prv** and a list of regressors that enter into the transition probability specification.

The dependent variable in `switchreg` may not be an expression. Dynamics may be specified by including lags of the dependent variable as regressors, or by specifying AR errors using the `AR` keyword. The latter incorporate mean adjusted lags of the form specified by the "Hamilton-model."

Options

type = *arg*	Type of switching: simple exogenous ("simple"), Markov ("markov").
nstates = *integer* *(default* = 2*)*	Number of regimes.
heterr	Allow for heterogeneous error variances across regimes
fprobmat = *arg*	Name of fixed transition probability matrix allows for fixing specific elements of the time-invariant transition matrix. Leave NAs in elements of the matrix to estimate. The (i, j) element of the matrix corresponds to $P(s_t = j \mid s_{t-1} = i)$.

initprob = *arg* *(default* = "ergodic"*)*	Method for determining initial Markov regime probabilities: ergodic solution ("ergodic"), estimated parameter ("est"), equal probabilities ("uniform"), user-specified probabilities ("user"). If "initprob = user" is specified, you will need to specify the "userinit = " option.
userinit = *arg*	Name of vector containing user-specified initial Markov probabilities. The vector should have rows equal to the number of states; we expand this to the size of the initial lag state vector where necessary for AR specifications. For use in specifications containing both the "type = markov" and "initprob = user" options.
startnum = *arg* *(default* = 0 or 25*)*	Number of random starting values tried. The default is 0 for user-supplied coefficients (option "s") and 25 in all other cases.
startiter = *arg* *(default* = 10*)*	Number of iterations taken after each random start before comparing objective to determine final starting value.
searchnum = *arg* *(default* = 0*)*	Number of post-estimation perturbed starting values tried.
searchstds = *arg* *(default* = 1*)*	Number of standard deviations to use in perturbed starts (if "searchnum = ") is specified.
seed = *positive_integer* from 0 to 2,147,483,647	Seed the random number generator. If not specified, EViews will seed random number generator with a single integer draw from the default global random number generator.
rnd = *arg* *(default* = "kn" or method previously set using `rndseed` (p. 423) in the *Command and Programming Reference)*.	Type of random number generator: improved Knuth generator ("kn"), improved Mersenne Twister ("mt"), Knuth's (1997) lagged Fibonacci generator used in EViews 4 ("kn4") L'Ecuyer's (1999) combined multiple recursive generator ("le"), Matsumoto and Nishimura's (1998) Mersenne Twister used in EViews 4 ("mt4").

In addition to the specification options, there are options for estimation and covariance calculation.

Additional Options

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. The criterion will be set to the nearest value between 1e-24 and 0.2.
s	Use the current coefficient values in "C" as starting values (see also `param` (p. 413) of the *Command and Programming Reference*).
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.
cov = *arg* *(default* = "invinfo"*)*	Coefficient covariance method: Inverse information matrix ("invinfo"), Huber-White sandwich ("white" or "sandwich").
covinfo = *arg* *(default* = "hessian"*)*	Information matrix method: Hessian ("hessian"), outer-product of gradients ("opg").
nodf	Do not degree-of-freedom correct the coefficient covariance estimate.
coef = *arg*	Specify the name of the coefficient vector (if specified by list); the default behavior is to use the "C" coefficient vector.
prompt	Force the dialog to appear from within a program.
p	Print results.

Examples

```
switchreg(type=markov) y c @nv ar(1) ar(2) ar(3) ar(4)
```

estimates a Hamilton-type Markov switching regression model with four non-regime varying autoregressive terms implying mean adjustment for the lagged endogenous.

```
switchreg(type=markov) y c @nv y(-1) y(-2) y(-3) y(-4)
```

specifies an alternate dynamic model in which the lags enter directly into the contemporaneous equation without mean adjustment.

```
switchreg(type=markov) yy_dalt c @nv ar(1) ar(2) ar(3) ar(4) @prv c
    yy_ldalt
```

estimates a 2 state model with non-varying AR(4) and transition matrix probability regressor YY_LDALT.

Cross-references

See Chapter 13. "Switching Regression," beginning on page 389 of *User's Guide II* for a description of the switching regression methodology.

See also `Equation::rgmprobs` (p. 134), `Equation::transprobs` (p. 146), `Equation::makergmprobs` (p. 114) and `Equation::maketransprobs` (p. 115), all in the *Object Reference*, for routines that allow you to work with the regime probabilities and transition probabilities.

tabplace	Table Commands

Copies a portion of one table to the specified location in another table.

Syntax

tabplace(*desttable,sourcetable,d1,s1,s2)*

tabplace(*desttable,sourcetable,dr1,dc1,sr1,sc1,sr2,sc2*)

The `tabplace` command can be specified either using coordinates where columns are signified with a letter, and rows by a number (for example "A3" represents the first column, third row), or by row number and column number.

The first syntax represents coordinate form, where *sourcetable* is the name of the table from which to copy, *s1* specifies the upper-left coordinate portion of the section of the source table to be copied, *s2* specifies the bottom-right coordinate, *desttable* specifies the name of the table to copy to, and *d1* specifies the upper-left coordinate of the destination table.

The second syntax represents the row/column number form, where *sourcetable* is the name of the table from which to copy, *sr1* specifies the source table upper row number, *sc1* specifies the source table left most column number, *sr2* specifies the source table bottom row number, *sc2* specifies the source table right most column number. *desttable* specifies the name of the table to copy to, and *dr1* and *dr2* specify the upper and left most row and column of the destination table, respectively.

Examples

```
tabplace(table2,table1,"d1","B9","E17")
```

places a copy of the data from cell range B9 to E17 in TABLE1 to TABLE2 at cell D1

```
tabplace(table3,table1,10,3,9,2,17,5)
```

copies 8 rows of data (from row 9 to row 17) and 3 columns (from 2 to 5)of data in TABLE1 to the tenth row and 3rd column of TABLE3.

Cross-references

See also `Table::copytable` (p. 692).

For additional discussion of table commands see Chapter 3. "Working with Tables and Spreadsheets," on page 45 of the *Command and Programming Reference*.

See also Chapter 16. "Table and Text Objects," on page 691 of the *User's Guide I* for a discussion and examples of table formatting in EViews.

testadd	Interactive Use Commands

Test whether to add regressors to an estimated equation.

Tests the hypothesis that the listed variables were incorrectly omitted from an estimated equation (only available for equations estimated by list). The test displays some combination of Wald and LR test statistics, as well as the auxiliary regression.

Syntax

testadd*(options) arg1 [arg2 arg3 ...]*

List the names of the series or groups of series to test for omission after the keyword. The test is applied to the default equation, if defined.

Options

prompt	Force the dialog to appear from within a program.
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:

Cross-references

See "Coefficient Diagnostics" on page 140 of the *User's Guide II* for further discussion.

See also `testdrop` (p. 451).

testdrop	Interactive Use Commands

Test whether to drop regressors from a regression.

Tests the hypothesis that the listed variables were incorrectly included in the estimated equation (only available for equations estimated by list). The test displays some combination of F and LR test statistics, as well as the test regression.

Syntax

testdrop*(options) arg1 [arg2 arg3 ...]*

List the names of the series or groups of series to test for omission after the keyword. The test is applied to the default equation, if defined.

Options

prompt	Force the dialog to appear from within a program.
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:

Cross-references

See "Coefficient Diagnostics" on page 140 of the *User's Guide II* for further discussion of testing coefficients.

See also `testadd` (p. 450) and `Equation::wald` (p. 155) in the *Object Reference*.

tic	Programming Commands

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 status line. 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. 452) and `@toc` (p. 671).

toc	Programming Commands

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 status line. The set of commands:

```
tic
[some commands]
!elapsed = @toc
[more commands]
toc
```

resets the time, executes commands, saves the elapsed time in the control variable !ELAPSED, executes additional commands, and displays the total elapsed time in the status line.

Cross-references

See also `tic` (p. 451) and `@toc` (p. 671).

tsls	Interactive Use Commands

Two-stage least squares.

Carries out estimation using two-stage least squares.

Syntax

tsls*(options) y x1 [x2 x3 ...] @ z1 [z2 z3 ...]*

tsls*(options) specification @ z1 [z2 z3 ...]*

To use the `tsls` command, 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. You may estimate nonlinear equations or equations specified with formulas by first providing a specification, then listing the instrumental variables after an "@"-sign.

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.

Options

Non-Panel TSLS Options

nocinst	Do not automatically include a constant as an instrument.
w = *arg*	Weight series or expression. Note: *we recommend that, absent a good reason, you employ the default settings* **Inverse std. dev. weights** *("wtype = istdev") with* **EViews default** *scaling ("wscale = eviews") for backward compatibility with versions prior to EViews 7.*
wtype = *arg* *(default* = "istdev"*)*	Weight specification type: inverse standard deviation ("istdev"), inverse variance ("ivar"), standard deviation ("stdev"), variance ("var").
wscale = *arg*	Weight scaling: EViews default ("eviews"), average ("avg"), none ("none"). The default setting depends upon the weight type: "eviews" if "wtype = istdev", "avg" for all others.
cov = *keyword*	Covariance type *(optional)*: "white" (White diagonal matrix), "hac" (Newey-West HAC).

nodf	Do not perform degree of freedom corrections in computing coefficient covariance matrix. The default is to use degree of freedom corrections.
covlag = *arg* *(default* = 1*)*	Whitening lag specification: *integer* (user-specified lag value), "a" (automatic selection).
covinfo = *arg* *(default* = "aic"*)*	Information criterion for automatic selection: "aic" (Akaike), "sic" (Schwarz), "hqc" (Hannan-Quinn) (if "lag = a").
covmaxlag = *integer*	Maximum lag-length for automatic selection *(optional)* (if "lag = a"). The default is an observation-based maximum of $T^{1/3}$.
covkern = *arg* *(default* = "bart"*)*	Kernel shape: "none" (no kernel), "bart" (Bartlett, *default*), "bohman" (Bohman), "daniell" (Daniel), "parzen" (Parzen), "parzriesz" (Parzen-Riesz), "parzgeo" (Parzen-Geometric), "parzcauchy" (Parzen-Cauchy), "quadspec" (Quadratic Spectral), "trunc" (Truncated), "thamm" (Tukey-Hamming), "thann" (Tukey-Hanning), "tparz" (Tukey-Parzen).
covbw = *arg* *(default* = "fixednw" *)*	Kernel Bandwidth: "fixednw" (Newey-West fixed), "andrews" (Andrews automatic), "neweywest" (Newey-West automatic), *number* (User-specified bandwidth).
covnwlag = *integer*	Newey-West lag-selection parameter for use in nonparametric kernel bandwidth selection (if "covbw = neweywest").
covbwint	Use integer portion of bandwidth.
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. The criterion will be set to the nearest value between 1e-24 and 0.2.
s	Use the current coefficient values in "C" as starting values for equations specified by list with AR or MA terms (see also `param` (p. 413)).
s = *number*	Determine starting values for equations specified by list with AR or MA terms. Specify a number between zero and one representing the fraction of preliminary least squares estimates computed without AR or MA terms to be used. Note that out of range values are set to "s = 1". Specifying "s = 0" initializes coefficients to zero. By default EViews uses "s = 1".

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.
z	Turn off backcasting in ARMA models.
coef = *arg*	Specify the name of the coefficient vector (if specified by list); the default behavior is to use the "C" coefficient vector.
prompt	Force the dialog to appear from within a program.
p	Print basic estimation results.

Panel TSLS Options

cx = *arg*	Cross-section effects. For fixed effects estimation, use "cx = f"; for random effects estimation, use "cx = r".
per = *arg*	Period effects. For fixed effects estimation, use "cx = f"; for random effects estimation, use "cx = r".
wgt = *arg*	GLS weighting: (default) none, cross-section system weights ("wgt = cxsur"), period system weights ("wgt = persur"), cross-section diagonal weighs ("wgt = cxdiag"), period diagonal weights ("wgt = perdiag").
cov = *arg*	Coefficient covariance method: (default) ordinary, White cross-section system robust ("cov = cxwhite"), White period system robust ("cov = perwhite"), White heteroskedasticity robust ("cov = stackedwhite"), Cross-section system robust/PCSE ("cov = cxsur"), Period system robust/PCSE ("cov = persur"), Cross-section heteroskedasticity robust/PCSE ("cov = cxdiag"), Period heteroskedasticity robust ("cov = perdiag").
keepwgts	Keep full set of GLS weights used in estimation with object, if applicable (by default, only small memory weights are saved).
rancalc = *arg* (*default* = "sa")	Random component method: Swamy-Arora ("rancalc = sa"), Wansbeek-Kapteyn ("rancalc = wk"), Wallace-Hussain ("rancalc = wh").

nodf	Do not perform degree of freedom corrections in computing coefficient covariance matrix. The default is to use degree of freedom corrections.
coef = *arg*	Specify the name of the coefficient vector (if specified by list); the default is to use the "C" coefficient vector.
iter = *arg* *(default* = "onec"*)*	Iteration control for GLS specifications: perform one weight iteration, then iterate coefficients to convergence ("iter = onec"), iterate weights and coefficients simultaneously to convergence ("iter = sim"), iterate weights and coefficients sequentially to convergence ("iter = seq"), perform one weight iteration, then one coefficient step ("iter = oneb"). Note that random effects models currently do not permit weight iteration to convergence.
s	Use the current coefficient values in "C" as starting values for equations with AR terms (see also `param` (p. 413)).
s = *number*	Determine starting values for equations specified with AR terms. Specify a number between zero and one representing the fraction of preliminary least squares estimates computed without AR terms. Note that out of range values are set to "s = 1". Specifying "s = 0" initializes coefficients to zero. By default, EViews uses "s = 1".
m = *integer*	Set maximum number of iterations.
c = *number*	Set convergence criterion. The criterion is based upon the maximum of the percentage changes in the scaled coefficients. The criterion will be set to the nearest value between 1e-24 and 0.2.
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.
unbalsur	Compute SUR factorization in unbalanced data using the subset of available observations for a cluster.
showopts / -showopts	[Do / do not] display the starting coefficient values and estimation options in the estimation output.
prompt	Force the dialog to appear from within a program.
p	Print estimation results.

Examples

```
tsls y_d c cpi inc ar(1) @ lw(-1 to -3)
```

estimates an UNTITLED equation using TSLS regression of Y_D on a constant, CPI, INC with AR(1) using a constant, LW(-1), LW(-2), and LW(-3) as instruments.

```
param c(1) .1 c(2) .1
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.

Cross-references

See Chapter 2. "Additional Regression Tools," on page 23 and "Two-Stage Least Squares" on page 515 of the *User's Guide II* for details on two-stage least squares estimation in single equations and systems, respectively. "Instrumental Variables" on page 728 of the *User's Guide II* discusses estimation using pool objects, while "Instrumental Variables Estimation" on page 762 of the *User's Guide II* discusses estimation in panel structured workfiles.

See also `ls` (p. 373).

ubreak	Interactive Use Commands

Andrews-Quandt test for unknown breakpoint.

Carries out the Andrews-Quandt test for parameter stability at some unknown breakpoint.

Syntax

ubreak*(options) trimlevel @ x1 x2 x3*

You must provide the level of trimming of the data. The level must be one of the following: 49, 48, 47, 45, 40, 35, 30, 25, 20, 15, 10, or 5. If the equation is specified by list and contains no linear terms, you may specify a subset of the regressors to be tested for a breakpoint after an "@" sign.

Options

wfname = *series_name*	Store the individual Wald F-statistics into the series *series_name*.
lfname = *series_name*	Store the individual likelihood ratio F-statistics into the series *series_name*.
prompt	Force the dialog to appear from within a program.
p	Print the result of the test.

Examples

```
ls log(spot) c log(p_us) log(p_uk)
ubreak 15
```

regresses the log of SPOT on a constant, the log of P_US, and the log of P_UK, and then carries out the Andrews-Quandt test, trimming 15% of the data from each end.

To test whether only the constant term and the coefficient on the log of P_US are subject to a structural break, use:

```
ubreak @ c log(p_us)
```

Cross-references

See "Quandt-Andrews Breakpoint Test" on page 172 of the *User's Guide II* for further discussion.

See also `Equation::chow` (p. 54) and `Equation::rls` (p. 135) in the *Object Reference*.

unlink	Object Container, Data, and File Commands

Break links and auto-updating series (formulae) in the specified series objects.

Syntax

unlink *link_names*

`unlink` converts link objects and auto-updating series to ordinary series or alphas. Follow the keyword with a list of names of links and auto-updating series to be converted to ordinary series (values). The list of links may include wildcard characters.

Examples

```
unlink gdp income
```

converts the link series GDP and INCOME to ordinary series.

```
unlink *
```

breaks all links in the active workfile page.

Cross-references

See Chapter 8. "Series Links," on page 219 of the *User's Guide I* for a description of link objects, and "Auto-Updating Series" on page 189 of the *User's Guide I* for a discussion of auto-updating series. See also `Link::link` (p. 318) and `Link::linkto` (p. 319).

See also `pageunlink` (p. 410) and `wfunlink` (p. 489) for page and workfile based unlinking, respectively.

uroot	Interactive Use Commands

Carries out unit root tests on a single series, pool series, group of series, or panel structured series.

The ordinary, single series unit root tests include 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).

If used on a series in a panel structured workfile, or with a pool series, or group of series, the procedure will perform panel unit root testing. The panel unit root tests include Levin, Lin and Chu (LLC), Breitung, Im, Pesaran, and Shin (IPS), Fisher - ADF, Fisher - PP, and Hadri tests on levels, or first or second differences.

Syntax

uroot*(options) object_name*

where *object_name* can be the name of a series, group or pool object.

Options

Basic Specification Options

You should specify the exogenous variables and order of dependent variable differencing in the test equation using the following options:

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).
dif = *integer* *(default = 0)*	Order of differencing of the series prior to running the test. Valid values are {0, 1, 2}.

For backward compatibility, the shortened forms of these options, “c”, “t”, and “n”, are presently supported. For future compatibility we recommend that you use the longer forms.

For ordinary (non-panel) unit root tests, you should specify the test type using one of the following keywords:

adf *(default)*	Augmented Dickey-Fuller.
dfgls	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.

For panel testing, you may use one of the following keywords to specify the test:

sum *(default)*	Summary of all of the panel unit root tests.
llc	Levin, Lin, and Chu.
breit	Breitung.
ips	Im, Pesaran, and Shin.
adf	Fisher - ADF.
pp	Fisher - PP.
hadri	Hadri.

Options for ordinary (non-panel) unit root tests

In addition, the following panel specific options are available:

hac = *arg*	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). Applicable to PP, KPSS, ERS, and NP tests. *The default settings are test specific* ("bt" for PP and KPSS, "ar" for ERS, "argls" for NP).
band = *arg*, b = *arg* *(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, and NP tests when using kernel sums-of-covariances estimators (where "hac = " is one of {bt, pz, qs}).

lag = *arg* (*default* = "a")	Method of selecting lag length (number of first difference terms) to be included in the regression: "a" (automatic information criterion based selection), or *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}).
info = *arg* (*default* = "sic")	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}).
maxlag = *integer*	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}).

Options for panel unit root tests

The following panel specific options are available:

balance	Use balanced (across cross-sections or series) data when performing test.
hac = *arg* (*default* = "bt")	Method of estimating the frequency zero spectrum: "bt" (Bartlett kernel), "pr" (Parzen kernel), "qs" (Quadratic Spectral kernel). Applicable to "Summary", LLC, Fisher-PP, and Hadri tests.
band = *arg*, b = *arg* (*default* = "nw")	Method of selecting the bandwidth: "nw" (Newey-West automatic variable bandwidth selection), "a" (Andrews automatic selection), *number* (user-specified common bandwidth), *vector_name* (user-specified individual bandwidth). Applicable to "Summary", LLC, Fisher-PP, and Hadri tests.

lag = *arg*	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 common lag length), *vector_name* (user-specific individual lag length). If the "balance" option is used, $$default = \begin{cases} 1 \text{ if } (T_{\min} \leq 60) \\ 2 \text{ if } (60 < T_{\min} \leq 100) \\ 4 \text{ if } (T_{\min} > 100) \end{cases}$$ where $T_{\min}$ is the length of the shortest cross-section or series, otherwise *default* = "a". Applicable to "Summary", LLC, Breitung, IPS, and Fisher-ADF tests.
info = *arg* (*default* = "sic")	Information criterion to use when computing automatic lag length selection: "aic" (Akaike), "sic" (Schwarz), "hqc" (Hannan-Quinn). Applicable to "Summary", LLC, Breitung, IPS, and Fisher-ADF tests.
maxlag = *arg*	Maximum lag length to consider when performing automatic lag length selection, where *arg* is an *integer* (common maximum lag length) or a *vector_name* (individual maximum lag length) $$default = \text{int}(\min_i(12,\ T_i/3) \cdot (T_i/100)^{1/4})$$ where T_i is the length of the cross-section or series.

Other options

prompt	Force the dialog to appear from within a program.
p	Print output from the test.

Examples

The command:

```
uroot(adf,const,lag=3,save=mout) gdp
```

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.

```
uroot(dfgls,trend,info=sic) ip
```

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.

```
uroot(kpss,const,hac=pr,b=2.3) unemp
```

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.

```
uroot(np,hac=ardt,info=maic) sp500
```

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 Testing" on page 471 of the *User's Guide II* for discussion of standard unit root tests performed on a single series, and "Panel Unit Root Testing" on page 483 of the *User's Guide II* for discussion of unit roots tests performed on panel structured workfiles, groups of series, or pooled data.

varest	Interactive Use Commands

Specify and estimate a VAR or VEC.

Syntax

varest*(options) lag_pairs endog_list [@ exog_list*]

varest(method = ec, *trend, n, options) lag_pairs endog_list [@ exog_list*]

The first form of the command estimates a VAR. It is the interactive command equivalent of using `ls` to estimate a named VAR object (see `Var::ls` (p. 764) in the *Object Reference* for syntax and details).

The second form of the command estimates a VEC. It is the interactive command equivalent of using `ec` to estimate a named VAR object (see `Var::ec` (p. 756) in the *Object Reference* for syntax and details).

Examples

prompt	Force the dialog to appear from within a program.
p	Print output from the test.

Examples

```
varest 1 3 m1 gdp
```

estimates an unnamed unrestricted VAR with two endogenous variables (M1 and GDP), a constant and 3 lags (lags 1 through 3).

```
varest(noconst) 1 3 ml gdp
```

estimates the same VAR, but with no constant term included in the specification.

```
varest(method=ec) 1 4 m1 gdp tb3
```

estimates a VEC with four lagged first differences, three endogenous variables and one cointegrating equation using the default trend option "*c*".

```
varest(method=ec,b,2) 1 2 4 4 tb1 tb3 tb6 @ d2 d3 d4
```

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 also `Var::var` (p. 776), `Var::ls` (p. 764) and `Var::ec` (p. 756) in the *Object Reference*.

wfclose

Object Container, Data, and File Commands

Close the active or specified workfile.

Syntax

wfclose(*options*) *[name]*

wfclose allows you to close the currently active workfile. You may optionally provide the name of a workfile if you do not wish to close the active workfile. If more than one workfile is found with the same name, the one most recently opened will be closed.

Options

noerr	Do not error if a *name* is provided and no workfile with that name is found.

Examples

```
wfclose
```

closes the workfile that was most recently active.

```
wfclose basics
```

closes the workfile named BASICS.

Cross-references

See also `wfcreate` (p. 467), `wfopen` (p. 472) and `wfsave` (p. 485).

wfcompare	Object Container, Data, and File Commands

Compare the contents of the current workfile or page with the contents of a different workfile, page or database.

Syntax

wfcompare*(options) targetspec baselinespec [***@keep** *keeplist* **@drop** *droplist]*

The command shows a list of any differences between the objects specified by *targetspec* and those specified by *baselinespec*.

targetspec should be a specification of objects in the current workfile. It should take the form of *[page\]name_pattern*, where the optional *page* may be used to specify a specific page of the current workfile. *name_pattern* is used to list the objects you wish to compare. The typical form of *targetspec* is simply "*", meaning to compare all objects in the current workfile.

baselinespec should be a specification of the list of objects to compare against. It should take the form of *[container::page\]name_pattern*, where the optional *container::page* may be used to specify the name of the workfile or database or page containing the objects to compare against. *name_pattern* is used to list the objects you wish to compare against. If *baselinespec* is blank, the version of the current workfile stored on disk is used as the baseline.

The optional **@keep** and **@drop** lists allow you to narrow further the list of objects in *targetspec* by listing specific objects to compare (using **@keep**), or drop from the *targetspec* (using **@drop**).

Options

tol = *arg*	Specifies the threshold below which differences between the target and baseline values should be ignored. The threshold is specified as a fraction of the baseline value. For example, if tol = 1%, one or more observations in the target object must differ from their values in the baseline object by at least one percent for the objects to be reported as different. The default tolerance is 1e-15.
seterr	Set an error if any differences exceeding the specified tolerance were found. This option may be useful in batch programming to alert the user if a program causes unexpectedly large changes to data values.

list = *key*	Selects which objects should be included in the output list. *key* may be "a" (list objects that exist in the target but not baseline), "d" (list objects that exist in the baseline, but not target), "r" (list objects that exist in both but have different objects types), "c" (list objects that exist in both but have different frequencies), "m" (list objects that exist in both but have values that differ by more than the tolerance), "u" (list objects that exist in both and are unchanged), "s" (list objects that exist in both but cannot be compared).
out = *name*	Create a table object *name*, containing the comparison table.
nohead	Used with "out = " option to suppress the header row at the top of the frozen table.

Examples

```
wfcompare
```

Compares the current workfile with the previously saved version of the workfile on disk. All pages are compared.

```
wfcompare *_0 *_1
```

compares all in the current page with names ending with "_0" with all objects whose names ends in "_1".

```
wfcompare page1\* page2\*
```

compares all objects in page1 of the current workfile with those in page2.

```
wfcompare page1\* page2\* @drop GDP UNEMP
```

compares all objects except for the objects "GDP" and "UNEMP" in page1 of the current workfile with those in page2.

```
wfcompare * myfile.wf1
```

compares the contents of the current page with the contents of the same page in the saved workfile 'myfile.wf1'

```
wfcompare *\* jun2012.wf1
```

compares the contents of all pages in the current workfile in memory with all pages of the saved workfile "Jun2012.WF1".

```
wfcompare page2\x* test.edb::y*
```

compares all objects in page2 of the current workfile whose name begins with the letter "x" to all objects in the database test.edb whose name begins with the letter "y".

Cross-references

See "Comparing Workfiles," on page 88 of *User's Guide I* for discussion.

wfcreate	Object Container, Data, and File Commands

Create a new workfile. The workfile becomes the active workfile.

Syntax

wfcreate*(options) frequency[(subperiod_opts)] start_date end_date [num_cross_sections]*

wfcreate*(options)* **u** *num_observations*

The first form of the command may be used to create a new regular frequency workfile with the specified frequency, start, and end date. Subperiod options may also be specified for intraweek or intraday data. See table below for a complete description for each frequency. If you include the optional *num_cross_sections*, EViews will create a balanced panel page using integer identifiers for each of the cross-sections. Note that more complex panel structures may be created using `pagestruct` (p. 408). You may use the `@now` keyword to specify the current date/time as either the *start_date* or *end_date*.

The second form of the command is used to create an unstructured workfile with the specified number of observations.

Options

wf = *wf_name*	Optional name for the new workfile.
page = *page_name*	Optional name for the page in the new workfile.
prompt	Force the dialog to appear from within a program.

Arguments

The *frequency* argument should be specified using one of the following forms:

Sec*[opt]*, 5Sec*[opt]*, 15Sec*[opt]*, 30Sec*[opt]*	Seconds in intervals of: 1, 5, 15, or 30 seconds, respectively. You may optionally specify days of the week and start and end times during the day using the *opt* parameter. See explanation of subperiod options below.

Min*[opt]*, 2Min*[opt]*, 5Min*[opt]*, 10Min*[opt]*, 15Min*[opt]*, 20Min*[opt]*, 30Min[*[opt]*	Minutes in intervals of: 1, 2, 5, 10, 15, 20, or 30 minutes, respectively. You may optionally specify days of the week and start and end times during the day using the *opt* parameter. See explanation of subperiod options below.
H*[opt]*, 2H*[opt]*, 4H*[opt]*, 6H*[opt]*, 8H*[opt]*, 12H[*[opt]*	Hours in intervals of: 1, 2, 4, 6, 8, or 12 hours, respectively. You may optionally specify days of the week and start and end times during the day using the *opt* parameter. See explanation of subperiod options below.
D(*s, e*)	Daily with arbitrary days of the week. Specify the first and last days of the week with integers *s* and *e*, where Monday is given the number 1 and Sunday is given the number 7. (Note that the "D" option used to specify a 5-day frequency prior to EViews 7.)
D5 or 5	Daily with five days per week, Monday through Friday.
D7 or 7	Daily with seven days per week.
W	Weekly
T	Ten-day (daily in intervals of ten).
F	Fortnight
BM	Bimonthly
M	Monthly
Q	Quarterly
S	Semi-annual
A or Y	Annual
2Y, 3Y, 4Y, 5Y, 6Y, 7Y, 8Y, 9Y, 10Y, 20Y	Multi-year in intervals of: 2, 3, 4, 5, 6, 7, 8, 9, 10, or 20 years, respectively.

Subperiod options

EViews allows for setting the days of the week and the time of day within intraday frequencies, which include *seconds*, *minutes*, and *hours*. For instance, you may specify hourly data between 8AM and 5PM on Monday through Wednesday. These subperiod options should follow the frequency keyword and be enclosed in parentheses.

To specify days of the week, use integers to indicate the days, where Monday is given the number 1 and Sunday is given the number 7. For example,

```
wfcreate(wf=storehours) 30MIN(1-6, 8:00-17:00) 1/3/2000 12/30/2000
```

indicates a half-hour frequency that includes Monday through Saturday from 8AM to 5PM.

To specify the start and end times, you may use either a 24 hour clock, including minutes and optionally seconds, or a 12 hour clock using AM and PM. For example, each of the following represents 8PM: 8PM, 8:00PM, 8:00:00PM, 20:00, and 20:00:00. Thus, our previous example could have been written:

```
wfcreate(wf=storehours) 30MIN(1-6, 8AM-5PM) 1/3/2000 12/30/2000
```

If you wish to include all days of the week but would like to specify a start and end time, set the date range to include all days and then specify the times. The day of the week parameter appears first and is required if you wish to supply the time of day parameters. For instance,

```
wfcreate(wf=storehours) 30MIN(1-7, 10AM-3PM) 1/3/2000 12/30/2000
```

indicates a half-hour frequency from 10AM to 3PM on all days of the week.

You may also include a time with the start and end date parameters to specify partial days at the beginning or end of the workfile. For example,

```
wfcreate(wf=storehours) 30MIN(1-6, 8AM-5PM) 1/3/2000 10AM
   12/30/2000 2PM
```

creates the same workfile as above, but limits the first day, 1/3/2000, to 10AM - 5PM and the last day, 12/30/2000, to 8AM - 2PM.

Alignment options

Certain frequencies optionally allow you to specify the starting point of the frequency period. Weekly and biweekly frequencies allow you to set the day at which the week begins. Annual, semiannual and quarterly frequencies allow you to set the month at which the quarter or year begins. Setting the starting period is important if you wish to use frequency conversion to convert data from a different frequency.

To specify the start period, simply add an extra term to the frequency symbol, surrounded in parenthesis, containing the day, or month, upon which you wish the frequency to start. For example:

```
wfcreate w(monday) 2000 2010
```

creates a weekly workfile from 2000 to 2010, where each week starts on a Monday.

```
wfcreate a(july) 2001 2007
```

creates an annual workfile where each year starts in July.

Note that by default, if you do not specify a starting point, EViews will use the period of the specified *start_date*. To make this difference concrete, consider the commands:

```
wfcreate w 2000 2010
```

and

```
wfcreate w(monday) 2000 2010
```

Since January 1st, 2000 was a Saturday, the first command will create a weekly workfile where each week starts on a Saturday, and the first observation in the workfile will span the period January 1st-January 7th 2000.

The second command will force EViews to start weeks on a Monday, and thus the first observation will actually span the period December 27th 1999 - January 2nd 2000.

Examples

```
wfcreate(wf=annual, page=myproject) a 1950 2005
wfcreate(wf=unstruct, page=undated) u 1000
```

creates two workfiles. The first is a workfile named ANNUAL containing a single page named MYPROJECT containing annual data from 1950 to 2005; the second is a workfile named UNSTRUCT containing a single page named UNDATED with 1000 unstructured observations.

```
wfcreate(wf=griliches_grunfeld, page=annual) a 1935 1954 10
```

creates the GRILICHES_GRUNFELD workfile containing a page named "ANNUAL" with 10 cross-sections of annual data for the years 1935 to 1954.

```
wfcreate(wf=fourday) D(1,4) 1/3/2000 12/31/2000
```

specifies a daily workfile from January 3, 2000 to December 31, 2000, including only Monday through Thursday. The day range may be delimited by either a comma or a dash, such that

```
wfcreate(wf=fourday) D(1-4) 1/3/2000 12/31/2000
```

is equivalent to the previous command.

```
wfcreate(wf=captimes) 15SEC(2-4) 1/3/2000 12/30/2000
```

creates a workfile with 15 second intervals on Tuesday through Thursday only, from 1/3/2000 to 12/30/2000.

Cross-references

See "Creating a Workfile," on page 42 of *User's Guide I* for discussion.

See also `pagecreate` (p. 394) and `pagedelete` (p. 400).

wfdetails	Object Container, Data, and File Commands

Change the details displayed in the current workfile window, optionally freezing the results into a table object in the current workfile.

Syntax

wfdetails(*options*) *col1[(width1)] [col2[(width2)].....] @sort sortcol*

Specify the names of the attribute columns you would like to display in the workfile details view, optionally including the width of the column in parenthesis. Widths can be specified with positive numbers, indicating a width in pixels, or negative numbers if you wish to specify the width in approximate characters.

You may use the @sort keyword at the end of the specification to indicate by which column to sort the details view.

Options

out = *name*	Create a table object *name*, containing the details view table.
nohead	Used with "out = " option to suppress the header row at the top of the frozen table.

Examples

```
wfdetails start end
```

shows the workfile details view, with the attribute columns "start" and "end".

```
wfdetails(out=dettable) start(30) end(-10)
```

shows the same view, but setting the "start" column's width to 30 pixels, and the "end" column's width to 10 characters, and freezing the view into a table called DETTABLE.

```
wfdetails start end @sortcol type
```

sorts the details view by object type.

Cross-references

See "Workfile Details Display," on page 62 of *User's Guide I* for discussion.

wfopen	Object Container, Data, and File Commands

Open a workfile. Reads in a previously saved workfile from disk, or reads the contents of a foreign data source into a new workfile.

The opened workfile becomes the default workfile; existing workfiles in memory remain on the desktop but become inactive.

Syntax

wfopen *[path\]source_name*

wfopen*(options) source_description [table_description] [variables_description]*

wfopen*(options) source_description [table_description] [dataset_modifiers]*

where *path* is an optional local path or URL.

There are three basic forms of the **wfopen** command:

- the first form is used by EViews native ("EViews and MicroTSP" on page 474) and time series database formats ("Time Series Formats" on page 475).
- the second form used for raw data files—Excel, Lotus, ASCII text, and binary files ("Raw Data Formats" on page 476).
- the third form is used with the remaining source formats, which we term *dataset formats,* since the data have already been arranged in named variables ("Datasets" on page 483).

(See "Options" on page 473 for a description of the supported source formats and corresponding types.)

In all three cases, the workfile or external data source should be specified as the first argument following the command keyword and options.

- In most cases, the external data source is a file, so the *source_description* will be the description of the file (including local path or URL information, if necessary). Alternatively, the external data source may be the output from a web server, in which case the URL should be provided. Similarly, when reading from an ODBC query, the ODBC DSN (data source name) should be used as the *source_description*.

  If the *source_description* contains spaces, it must be enclosed in (double) quotes.

For raw and dataset formats, you may use *table_description* to provide additional information about the data to be read:

- Where there is more than one table that could be formed from the specified external data source, a *table_description* may be provided to select the desired table. For exam-

ple, when reading from an Excel file, an optional cell range may be provided to specify which data are to be read from the spreadsheet. When reading from an ODBC data source, a SQL query or table name must be used to specify the table of data to be read.

- In raw data formats, the *table_description* allows you to provide additional information regarding names and descriptions for variables to be read, missing values codes, settings for automatic format, and data range subsetting.
- When working with text or binary files, the *table_description* must be used to describe how to break up the file into columns and rows.

For raw and dataset formats, you may use the *dataset_modifiers* specification to select the set of variables, maps (value labels), and observations to be read from the source data. The dataset_modifiers consists of the following keyword delimited lists:

*[@***keep** *keep_list] [@***drop** *drop_list] [@***keepmap** *keepmap_list] [@***dropmap** *dropmap_list] [@***selectif** *condition]*

- The **@keep** and **@drop** keywords, followed by a list of names and patterns, are used to specify variables to be retain or dropped. Similarly, the **@keepmap** and **@dropmap** keywords followed by lists of name patterns controls the reading of value labels. The keyword **@selectif**, followed by an if condition (e.g., "if age > 30 and gender = 1") may be used to select a subset of the observations in the original data. By default, all variables, value labels, and observations are read.

By default, all variables, maps and observations in the source file will be read.

Options

type = *arg* / t = *arg*	Optional type specification: *(see table below)*. Note that ODBC support is provided only in the *EViews Enterprise Edition*.
wf = *wf_name*	Optional name for the new workfile.
page = *page_name*	Optional name for the page in the new workfile.
prompt	Force the dialog to appear from within a program.

For the most part, you should not need to specify a "type = " option as EViews will automatically determine the type from the filename.

The following table summaries the various source formats and types, along with the corresponding "type = " keywords:

	Source Type	Option Keywords
Access	dataset	“access”
Aremos-TSD	time series database	“a”, “aremos”, “tsd”
Binary	raw data	“binary”
dBASE	dataset	“dbase”
Excel (through 2003)	raw data	“excel”
Excel 2007 (xml)	raw data	“excelxml”
EViews Workfile	native	---
Gauss Dataset	dataset	“gauss”
GiveWin/PcGive	time series database	“g”, “give”
HTML	raw data	“html”
Lotus 1-2-3	raw data	“lotus”
ODBC Dsn File	dataset	“dsn”
ODBC Query File	dataset	“msquery”
ODBC Data Source	dataset	“odbc”
MicroTSP Workfile	native	“dos”, “microtsp”
MicroTSP Mac Workfile	native	“mac”
RATS 4.x	time series database	“r”, “rats”
RATS Portable / TROLL	time series database	“l”, “trl”
SAS Program	dataset	“sasprog”
SAS Transport	dataset	“sasxport”
SPSS	dataset	“spss”
SPSS Portable	dataset	“spssport”
Stata	dataset	“stata”
Text / ASCII	raw data	“text”
TSP Portable	time series database	“t”, “tsp”

EViews and MicroTSP

The syntax for EViews and MicroTSP files is:

wfopen *[path\]workfile_name*

where *path* is an option local path or URL.

Examples

```
wfopen c:\data\macro
```

loads a previously saved EViews workfile "Macro.WF1" from the "data" directory in the C drive.

```
wfopen 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 specifies the type explicitly:

```
wfopen(type=dos) nipa
```

Time Series Formats

The syntax for time series format files (Aremos-TSD, GiveWin/PcGive, RATS, RATS Portable/TROLL, TSP Portable) is:

wfopen*(options) [path\]source_name*

where *path* is an optional local path or URL.

If the source files contain data of multiple frequencies, the resulting workfile will be of the lowest frequency, and higher frequency data will be converted to this frequency. If you wish to obtain greater control over the workfile creation, import, or frequency conversion processes, we recommend that you open the file using `dbopen` (p. 322) and use the database tools to create your workfile.

Aremos Example

```
wfopen dlcs.tsd
wfopen(type=aremos) dlcs.tsd
```

open the AREMOS-TSD file DLCS.

GiveWin/PcGive Example

```
wfopen "f:\project\pc give\data\macrodata.in7"
wfopen(type=give) "f:\project\pc give\data\macrodata"
```

open the PcGive file MACRODATA.

Rats Examples

```
wfopen macrodata.rat"
wfopen macrodata.trl
```

read the native RATS 4.x file MACRODATA.RAT and the RATS Portable/TROLL file "Macrodata.TRL".

TSP Portable Example

```
wfopen macrodata.tsp
```

reads the TSP portable file "Macrodata.TSP".

Raw Data Formats

The command for reading raw data (Excel 97-2003, Excel 2007, HTML, ASCII text, Binary, Lotus 1-2-3) is

wfopen*(options) source_description [table_description] [variables_description] [@***keep** *keep_list] [@***drop** *drop_list] [@***keepmap** *keepmap_list] [@***dropmap** *dropmap_list] [@***selectif** *condition]*

where the syntax of the *table_description* and *variables_description* differs slightly depending on the type of file.

Excel and Lotus Files

The syntax for reading Excel and Lotus files is:

wfopen*(options) source_description [table_description] [variables_description]*

The following *table_description* elements may be used when reading Excel and Lotus data:

- "range = *arg*", where *arg* is a range of cells to read from the Excel workbook, following the standard Excel format *[worksheet!][topleft_cell[:bottomright_cell]]*.

  If the worksheet name contains spaces, it should be placed in single quotes. If the worksheet name is omitted, the cell range is assumed to refer to the currently active sheet. If only a top left cell is provided, a bottom right cell will be chosen automatically to cover the range of non-empty cells adjacent to the specified top left cell. If only a sheet name is provided, the first set of non-empty cells in the top left corner of the chosen worksheet will be selected automatically. As an alternative to specifying an explicit range, a name which has been defined inside the excel workbook to refer to a range or cell may be used to specify the cells to read.

- "byrow", transpose the incoming data. This option allows you to read files where the series are contained in rows (one row per series) rather than columns.

The optional *variables_description* may be formed using the elements:

- "colhead = *int*", number of table rows to be treated as column headers.
- "namepos = [first|last|all|none]", which row(s) of the column headers should be used to form the column name. The setting "first" refers to first line, "last" is last line, "all" is all lines and "none" is no lines. The remaining column header rows will be used to form the column description. The default setting is "all".
- "Nonames", the file does not contain a column header (same as "colhead = 0").
- "names = ("*arg1*","*arg2*",…)", user specified column names, where *arg1*, *arg2*, … are names of the first series, the second series, *etc.* when names are provided, these override any names that would otherwise be formed from the column headers.

- "descriptions = ("*arg1*","*arg2*",…)", user specified descriptions of the series. If descriptions are provided, these override any descriptions that would otherwise be read from the data.
- "types = ("*arg1*","*arg2*",…)", user specified data types of the series. If types are provided they will override the types automatically detected by EViews. You may use any of the following format keywords: "a" (character data), "f" (numeric data), "d" (dates), or "w"(EViews automatic detection).
- "na = "*arg1*"", text used to represent observations that are missing from the file. The text should be enclosed on double quotes.
- "scan = [*int*| **all**]", number of rows of the table to scan during automatic format detection ("scan = all" scans the entire file).
- "firstobs = *int*", first observation to be imported from the table of data (default is 1). This option may be used to start reading rows from partway through the table.
- "lastobs = *int*", last observation to be read from the table of data (default is last observation of the file). This option may be used to read only part of the file, which may be useful for testing.

Excel Examples

```
wfopen "c:\data files\data.xls"
```

loads the active sheet of DATA.XLS into a new workfile.

```
wfopen(page=mypage) "c:\data files\data.xls" range="GDP data"
    @drop X
```

reads the data contained in the "GDP data" sheet of "Data.XLS" into the MYPAGE page of a new workfile. The data for the series X is dropped, and the name of the new workfile page is "GDP".

To load the Excel file containing US Macro Quarterly data from Stock and Watson's *Introduction to Econometrics* you may use the command:

```
wfopen
    http//wps.aw.com/wps/media/objects/3254/3332253/datasets2e/dat
    asets/USMacro_Quarterly.xls
```

which will load the Excel file directly into EViews from the publisher's website (as of 08/2009).

HTML Files

The syntax for reading HTML pages is:

wfopen*(options) source_description [table_description] [variables_description]*

The following *table_description* elements may be used when reading an HTML file or page:

- "table = *arg*", where *arg* specifies which table to read in an HTML file/page containing multiple tables.

  When specifying *arg,* you should remember that tables are named automatically following the pattern "Table01", "Table02", "Table03", *etc.* If no table name is specified, the largest table found in the file will be chosen by default. Note that the table numbering may include trivial tables that are part of the HTML content of the file, but would not normally be considered as data tables by a person viewing the page.

- "skip = *int*", where *int* is the number of rows to discard from the top of the HTML table.

- "byrow", transpose the incoming data. This option allows you to import files where the series are contained in rows (one row per series) rather than columns.

The optional *variables_description* may be formed using the elements:

- "colhead = *int*", number of table rows to be treated as column headers.

- "namepos = [first|last|all|none]", which row(s) of the column headers should be used to form the column name. The setting "first" refers to first line, "last" is last line, "all" is all lines and "none" is no lines. The remaining column header rows will be used to form the column description. The default setting is "all".

- "Nonames", the file does not contain a column header (same as "colhead = 0").

- "names = ("*arg1*","*arg2*",…)", user specified column names, where *arg1*, *arg2*, … are names of the first series, the second series, *etc.* when names are provided, these override any names that would otherwise be formed from the column headers.

- "descriptions = ("*arg1*","*arg2*",…)", user specified descriptions of the series. If descriptions are provided, these override any descriptions that would otherwise be read from the data.

- "types = ("*arg1*","*arg2*",…)", user specified data types of the series. If types are provided they will override the types automatically detected by EViews. You may use any of the following format keywords: "a" (character data), "f" (numeric data), "d" (dates), or "w"(EViews automatic detection).

- "na = "*arg1*"", text used to represent observations that are missing from the file. The text should be enclosed on double quotes.

- "scan = [*int*|**all**]", number of rows of the table to scan during automatic format detection ("scan = all" scans the entire file).

- "firstobs = *int*", first observation to be imported from the table of data (default is 1). This option may be used to start reading rows from partway through the table.

- "lastobs = *int*", last observation to be read from the table of data (default is last observation of the file). This option may be used to read only part of the file, which may be useful for testing.

HTML Examples

```
wfopen "c:\data.html"
```

loads into a new workfile the data located on the HTML file "Data.HTML" located on the C:\ drive

```
wfopen(type=html)
    "http://www.tradingroom.com.au/apps/mkt/forex.ac" colhead=3,
    namepos=first
```

loads into a new workfile the data with the given URL located on the website site "http://www.tradingroom.com.au". The column header is set to three rows, with the first row used as names for columns, and the remaining two lines used to form the descriptions.

Text and Binary Files

The syntax for reading text or binary files is:

wfopen*(options) source_description [table_description] [variables_description]*

If a *table_description* is not provided, EViews will attempt to read the file as a free-format text file. The following *table_description* elements may be used when reading a text or binary file:

- "ftype = [ascii|binary]" specifies whether numbers and dates in the file are stored in a human readable text (ASCII), or machine readable (Binary) form.
- "rectype = [crlf|fixed|streamed]" describes the record structure of the file:

  "crlf", each row in the output table is formed using a fixed number of lines from the file (where lines are separated by carriage return/line feed sequences). This is the default setting.

  "fixed", each row in the output table is formed using a fixed number of characters from the file (specified in "reclen = *arg*"). This setting is typically used for files that contain no line breaks.

  "streamed", each row in the output table is formed by reading a fixed number of fields, skipping across lines if necessary. This option is typically used for files that contain line breaks, but where the line breaks are not relevant to how rows from the data should be formed.
- "reclines = *int*", number of lines to use in forming each row when "rectype = crlf" (default is 1).
- "reclen = *int*", number of bytes to use in forming each row when "rectype = fixed".

- "recfields = *int*", number of fields to use in forming each row when "rectype = streamed".
- "skip = *int*", number of lines (if rectype is "crlf") or bytes (if rectype is not "crlf") to discard from the top of the file.
- "comment = *string*", where *string* is a double-quoted string, specifies one or more characters to treat as a comment indicator. When a comment indicator is found, everything on the line to the right of where the comment indicator starts is ignored.
- "emptylines = [keep|drop]", specifies whether empty lines should be ignored ("drop"), or treated as valid lines ("keep") containing missing values. The default is to ignore empty lines.
- "tabwidth = *int*", specifies the number of characters between tab stops when tabs are being replaced by spaces (default = 8). Note that tabs are automatically replaced by spaces whenever they are not being treated as a field delimiter.
- "fieldtype = [delim|fixed|streamed|undivided]", specifies the structure of fields within a record:

  "Delim", fields are separated by one or more delimiter characters

  "Fixed", each field is a fixed number of characters

  "Streamed", fields are read from left to right, with each field starting immediately after the previous field ends.

  "Undivided", read entire record as a single series.
- "quotes = [single|double|both|none]", specifies the character used for quoting fields, where "single" is the apostrophe, "double" is the double quote character, and "both" means that either single or double quotes are allowed (default is "both"). Characters contained within quotes are never treated as delimiters.
- "singlequote", same as "quotes = single".
- "delim = [comma|tab|space|dblspace|white|dblwhite]", specifies the character(s) to treat as a delimiter. "White" means that either a tab or a space is a valid delimiter. You may also use the abbreviation "d = " in place of "delim = ".
- "custom = "*arg1*"", specifies custom delimiter characters in the double quoted string. Use the character "t" for tab, "s" for space and "a" for any character.
- "mult = [on|off]", to treat multiple delimiters as one. Default value is "on" if "delim" is "space", "dblspace", "white", or "dblwhite", and "off" otherwise.
- "endian = [big|little]", selects the endianness of numeric fields contained in binary files.
- "string = [nullterm|nullpad|spacepad]", specifies how strings are stored in binary files. If "nullterm", strings shorter than the field width are terminated with a single

zero character. If “nullpad”, strings shorter than the field width are followed by extra zero characters up to the field width. If “spacepad”, strings shorter than the field width are followed by extra space characters up to the field width.

- “byrow”, transpose the incoming data. This option allows you to import files where the series are contained in rows (one row per series) rather than columns.

A central component of the *table_description* element is the format statement. You may specify the data format using the following table descriptors:

- Fortran Format:

  fformat = (*[n1]Type[Width][.Precision]*, *[n2]Type[Width][.Precision]*, ...)

  where *Type* specifies the underlying data type, and may be one of the following,

  I - integer

  F - fixed precision

  E - scientific

  A - alphanumeric

  X - skip

  and *n1*, *n2*, ... are the number of times to read using the descriptor (*default* = 1). More complicated Fortran compatible variations on this format are possible.

- Column Range Format:

  rformat = (*[n1]Type[Width][.Precision]*, *[n2]Type[Width][.Precision]*, ...)

  where optional type is “$” for string or “#” for number, and *n1*, *n2*, *n3*, *n4*, *etc.* are the range of columns containing the data.

- C printf/scanf Format:

  cformat = "*fmt*"

  where *fmt* follows standard C language (printf/scanf) format rules.

The optional *variables_description* may be formed using the elements:

- “colhead = *int*”, number of table rows to be treated as column headers.
- “namepos = [first|last|all|none]”, which row(s) of the column headers should be used to form the column name. The setting “first” refers to first line, “last” is last line, “all” is all lines and “none” is no lines. The remaining column header rows will be used to form the column description. The default setting is “all”.
- “Nonames”, the file does not contain a column header (same as “colhead = 0”).
- “names = ("*arg1*", "*arg2*",...)”, user specified column names, where *arg1*, *arg2*, ... are names of the first series, the second series, *etc.* when names are provided, these override any names that would otherwise be formed from the column headers.

- "descriptions = ("*arg1*", "*arg2*",…)", user specified descriptions of the series. If descriptions are provided, these override any descriptions that would otherwise be read from the data.
- "types = ("*arg1*","*arg2*",…)", user specified data types of the series. If types are provided they will override the types automatically detected by EViews. You may use any of the following format keywords: "a" (character data), "f" (numeric data), "d" (dates), or "w"(EViews automatic detection).
- "na = "*arg1*"", text used to represent observations that are missing from the file. The text should be enclosed on double quotes.
- "scan = [*int*|**all**]", number of rows of the table to scan during automatic format detection ("scan = all" scans the entire file).
- "firstobs = *int*", first observation to be imported from the table of data (default is 1). This option may be used to start reading rows from partway through the table.
- "lastobs = *int*", last observation to be read from the table of data (default is last observation of the file). This option may be used to read only part of the file, which may be useful for testing.

Text and Binary File Examples (.txt, .csv, etc.)

```
wfopen c:\data.csv skip=5, names=(gdp, inv, cons)
```

reads "Data.CSV" into a new workfile page, skipping the first 5 rows and naming the series GDP, INV, and CONS.

```
wfopen(type=text) c:\date.txt delim=comma
```

loads the comma delimited data DATE.TXT into a new workfile.

```
wfopen(type=raw) c:\data.txt skip=8, rectype=fixed,
    format=(F10,X23,A4)
```

loads a text file with fixed length data into a new workfile, skipping the first 8 rows. The reading is done as follows: read the first 10 characters as a fixed precision number, after that, skip the next 23 characters (X23), and then read the next 4 characters as strings (A4).

```
wfopen(type=raw) c:\data.txt rectype=fixed, format=2(4F8,2I2)
```

loads the text file as a workfile using the specified explicit format. The data will be a repeat of four fixed precision numbers of length 8 and two integers of length 2. This is the same description as "format = (F8,F8,F8,F8,I2,I2,F8,F8,F8,F8,I2,I2)".

```
wfopen(type=raw) c:\data.txt rectype=fixed, rformat=(GDP 1-2 INV 3
    CONS 6-9)
```

loads the text file as a workfile using column range syntax. The reading is done as follows: the first series is located at the first and second character of each row, the second series

occupies the 3rd character, the third series is located at character 6 through 9. The series will named GDP, INV, and CONS.

Datasets

The syntax for reading data from the remaining sources (Access, Gauss, ODBC, SAS program, SAS transport, SPSS, SPSS portable, Stata) is:

wfopen*(options) source_description table_description [@***keep** *keep_list] [@***drop** *drop_list] [@***selectif** *condition]*

Note that for this purpose we view Access and ODBC as datasets.

ODBC or Microsoft Access

The syntax for reading from an ODBC or Microsoft Access data source is

wfopen*(options) source_description table_description [@***keep** *keep_list] [@***drop** *drop_list] [@***selectif** *condition]*

When reading from an ODBC or Microsoft Access data source, you must provide a *table_description* to indicate the table of data to be read. You may provide this information in one of two ways: by entering the name of a table in the data source, or by including an SQL query statement enclosed in double quotes.

Note that ODBC support is provided only in the *EViews Enterprise Edition*.

ODBC Examples

```
wfopen c:\data.dsn CustomerTable
```

opens in a new workfile the table named CUSTOMERTABLE from the ODBC database described in the DATA.DSN file.

```
wfopen(type=odbc) "my server" "select * from customers where id>30"
    @keep p*
```

opens in a new workfile with SQL query from database using the server “MY SERVER”, keeping only variables that begin with P. The query selects all variables from the table CUSTOMERS where the ID variable takes a value greater than 30.

Other Dataset Types

The syntax for reading data from the remaining sources (Gauss, SAS program, SAS transport, SPSS, SPSS portable, Stata) is:

wfopen*(options) source_description [@***keep** *keep_list] [@***drop** *drop_list] [@***selectif** *condition]*

Note that no *table_description* is required.

SAS Program Example

If a data file, "Sales.DAT", contains the following space delimited data:

AZ 110 1002
CA 200 2003
NM 90 908
OR 120 708
WA 113 1123
UT 98 987

then the following SAS program file can be read by EViews to open the data:

```
Data sales;
    infile sales.dat';
    input state $ price sales;
run;
```

SAS Transport Examples

```
wfopen(type=sasxport) c:\data.xpt
```

loads a SAS transport file "data.XPT" into a new workfile.

```
wfopen c:\inst.sas
```

creates a workfile by reading from external data using the SAS program statements in "Inst.SAS". The program may contain a limited set of SAS statements which are commonly used in reading in a data file.

Stata Examples

To load a Stata file "Data.DTA" into a new workfile, dropping maps MAP1 and MAP2, you may enter:

```
wfopen c:\data.dta @dropmap map1 map2
```

To download the sports cards dataset from Stock and Watson's *Introduction to Econometrics* you may use the command:

```
wfopen
    http://wps.aw.com/wps/media/objects/3254/3332253/datasets2e/da
    tasets/Sportscards.dta
```

which will load the Stata dataset directly into EViews from the publisher's website (as of 08/2009).

Cross-references

See Chapter 3. "Workfile Basics," on page 41 of *User's Guide I* for a discussion of workfiles.

See also pageload (p. 400), read (p. 418), fetch (p. 332), wfsave (p. 485), wfclose (p. 464) and pagesave (p. 402).

wfrefresh	Object Container, Data, and File Commands

Refresh all links and auto-series in the active workfile. Primarily used to refresh links that use external database data.

Syntax

wfrefresh

Examples

```
wfrefresh
```

Cross-references

See "Creating a Database Link" on page 348 and "Understanding the Cache" on page 349 of the *User's Guide I*.

See also Chapter 8. "Series Links," on page 219 of the *User's Guide I* for a description of link objects, and "Auto-Updating Series" on page 189 of the *User's Guide I* for a discussion of auto-updating series.

See also pagerefresh (p. 401), Link::link (p. 318) and Link::linkto (p. 319) in the *Object Reference*.

wfsave	Object Container, Data, and File Commands

Save the default workfile as an EViews workfile (.wf1 file) or as a foreign file or data source.

Syntax

wfsave*(options) [path\]filename*

wfsave*(options) source_description [***@keep** *keep_list] [***@drop** *drop_list] [***@keepmap** *keepmap_list] [***@dropmap** *dropmap_list] [***@smpl** *smpl_spec]*

wfsave*(options) source_description table_description [***@keep** *keep_list] [***@drop** *drop_list] [***@keepmap** *keepmap_list] [***@dropmap** *dropmap_list] [***@smpl** *smpl_spec]*

saves the active workfile in the specified directory using *filename*. By default, the workfile is saved as an EViews workfile, but options may be used to save all or part of the *active* page in a foreign file or data source. See wfopen (p. 472) for details on the syntax for *source_descriptions* and *table_descriptions*. Note, in particular, that you may use the *byrow*

table_description for Excel 2007 files to instruct EViews to save the series by row (instead of the standard by column).

Options

Workfile Save Options

1	Save using single precision.
2	Save using double precision.
c	Save compressed workfile (not compatible with EViews versions prior to 5.0).

The default workfile save settings use the global options.

Foreign Source Save Options

type = *arg*, t = *arg*	Optional type specification: *(see table below)*. Note that ODBC support is provided only in the *EViews Enterprise Edition*.
mode = *arg*	Specify whether to create a new file, overwrite an existing file, or update an existing file. *arg* may be "create" (create new file only; error on attempt to overwrite) or "update" (only available for Excel files, and only if Excel is installed) (update an existing file, only overwriting the area specified by the range = *table_description*). If the "mode = " option is not used, EViews will create a new file, unless the file already exists in which case it will overwrite it.
maptype = *arg*	Write selected maps as: numeric ("n"), character ("c"), both numeric and character ("b").
nomapval	Do not write mapped values for series with attached value labels (the default is to write the mapped values if available).
noid	Do not write observation identifiers to foreign data files (by default, EViews will include a column with the date or observation identifier).

The following table summaries the various formats along with the corresponding "type = " keywords:

	Option Keywords
Access	"access"
Aremos-TSD	"a", "aremos", "tsd"
Binary	"binary"
dBASE	"dbase"

Excel (through 2003)	“excel”
Excel 2007 (xml)	“excelxml”
EViews Workfile	---
Gauss Dataset	“gauss”
GiveWin/PcGive	“g”, “give”
HTML	“html”
Lotus 1-2-3	“lotus”
ODBC Dsn File	“dsn”
ODBC Data Source	“odbc”
MicroTSP Workfile	“dos”, “microtsp”
MicroTSP Mac Workfile	“mac”
RATS	“r”, “rats”
RATS Portable / TROLL	“l”, “trl”
SAS Program	“sasprog”
SAS Transport	“sasxport”
SPSS	“spss”
SPSS Portable	“spssport”
Stata	“stata”
Text / ASCII	“text”
TSP Portable	“t”, “tsp”

Note that if you wish to save your Excel 2007 XML file with macros enabled, you should specify an explicit filename extension of “.XLSM”.

Examples

```
wfsave new_wf
```

saves the current EViews workfile as “New_wf.WF1” in the default directory.

```
wfsave "c:\documents and settings\my data\consump"
```

saves the current workfile as “Consump.WF1” in the specified path.

```
wfsave macro @keep gdp unemp
```

saves the two series GDP and UNEMP in a separate workfile, “macro.WF1” in the default directory.

```
wfsave macro @dropmap gdp*
```

saves all of the series in the current workfile, other than those that match the name pattern “gdp*” in a workfile, “macro.WF1” in the default directory.

```
wfsave(type=excel) macro @keep gdp unemp
```

saves the two series GDP and UNEMP as an Excel 2003 file, "macro.xls" in the default directory.

```
wfsave(type=excelxml, mode=update) range="Sheet2!a1" byrow
    macro.xlsx @keep gdp unemp
```

will save the two series GDP and UNEMP into the existing Excel 2007 file "macro.xlsx", specifying that the series should be written by row, starting in cell A1 on sheet Sheet2.

To save the latter file in a macro-enabled Excel 2007 file, you should specify the explicit filename extension ".XLSM":

```
wfsave(type=excelxml, mode=update) range="Sheet2!a1" byrow
    macro.xlsm @keep gdp unemp
```

Cross-references

See Chapter 3. "Workfile Basics," on page 41 of the *User's Guide I* for a discussion of workfiles.

See also `pagesave` (p. 402), `wfopen` (p. 472), and `pageload` (p. 400).

wfselect

Object Container, Data, and File Commands

Make the selected workfile page the active workfile page.

Syntax

wfselect *wfname[\pgname]*

where *wfname* is the name of a workfile that has been loaded into memory. You may optionally provide the name of a page in the new default workfile that you wish to be made active.

Examples

```
wfselect myproject
wfselect myproject\page2
```

both change the default workfile to MYPROJECT. The first command uses the default active page, while the second changes the page to PAGE2.

Cross-references

See Chapter 3. "Workfile Basics," on page 41 of the *User's Guide I* for a discussion of workfiles.

See also `pageselect` (p. 404).

wfstats	Object Container, Data, and File Commands

Display the workfile statistics and summary view.

Syntax

wfstats *[wfname]*

displays the workfile statistics and summary view showing, for each page of the specified workfile, the structure of the page, and a summary of the objects contained in the page. The specified workfile must be open. If no workfile is specified, `wfstats` will display results for the active workfile.

Examples

```
wfstats
```

displays the statistics view for the active workfile.

```
wfstats wf2
```

displays the statistics for the open workfile WF2.

Cross-references

See "Workfile Summary View" on page 75 of the *User's Guide I* for a description of the workfile statistics and summary view.

wfunlink	Object Container, Data, and File Commands

Break links in all link objects and auto-updating series (formulae) in the active workfile.

You should use some caution with this command as you will not be prompted before the links and auto-updating series are converted.

Syntax

wfunlink

Examples

```
wfunlink
```

breaks links in all pages of the active workfile

Cross-references

See Chapter 8. "Series Links," on page 219 of *User's Guide I* for a description of link objects, and "Auto-Updating Series" on page 189 of *User's Guide I* for a discussion of auto-updating series.

See also `Link::link` (p. 318) and `Link::linkto` (p. 319) in the *Object Reference*.

See also `pageunlink` (p. 410) and `unlink` (p. 458) for page and object based unlinking, respectively.

wfuse	Object Container, Data, and File Commands

Activates a workfile. If the workfile is currently open in EViews, then it is selected to become the default workfile. If the workfile is not currently open, it is opened.

Syntax

wfuse *[path\]workfile_name[.wf1][\page_name]*

The name of the workfile is specified as the argument to `wfopen` and may be followed by an optional page name to specify a specific page in the workfile. If the workfile is not located in the current default directory, and is not currently open in EViews, you should specify the path of the workfile along with its name. If you do specify a path, you should also include the .WF1 extension.

Examples

```
wfuse basics
```

activates the BASICS workfile. If BASICS is not currently open in EViews and is not located in the current default directory, `wfuse` will error.

```
wfuse c:\mydata\basics.wf1
```

activates the BASICS workfile located in "c:\mydata".

```
wfuse c:\mydata\basics.wf1\page1
```

activates the page named PAGE1 in the BASICS workfile located in "C:\MYDATA".

Cross-references

See Chapter 3. "Workfile Basics," on page 41 of the *User's Guide I* for a discussion of workfiles.

See also `wfopen` (p. 472) and `wfsave` (p. 485).

workfile	Object Container, Data, and File Commands

Create or change workfiles.

No longer supported; provided for backward compatibility. This command has been replaced by `wfcreate` (p. 467) and `pageselect` (p. 404).

write	Object Container, Data, and File Commands

Write EViews data to a text (ASCII), Excel, or Lotus file on disk.

Creates a foreign format disk file containing EViews data. May be used to export EViews data to another program.

Unless you need to write your workfile data in transposed form, we recommend that you use the more powerful command for writing a workfile page documented in `pagesave` (p. 402).

Syntax

write*(options) [path\]filename arg1 [arg2 arg3 …]*

Follow the 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.

Note that EViews cannot, at present, write into an existing file. The file that you select will, if it exists, be replaced.

Options

Options are specified in parentheses after the keyword and are used to specify the format of the output file.

prompt	Force the dialog to appear from within a program.

File type

t = dat, txt	ASCII (plain text) files.
t = wk1, wk3	Lotus spreadsheet files.
t = xls	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 = *string*	Specify text string for NAs. Default is "NA".
names *(default)* / nonames	[Write / Do not write] series names.
dates / nodates	[Write / Do not write] dates/obs. "Dates" is the default unless the "t" option for writing by series is used, in which case "nodates" is the default.
d = *arg*	Specify delimiter *(default* is tab*)*: "s" (space), "c" (comma).
t	Write by series. Default is to write by obs with series in columns.

Spreadsheet (Lotus, Excel) files

letter_number	Coordinate of the upper-left cell containing data.
names *(default)* / nonames	[Write / Do not write] series names.
dates *(default)* / nodates	[Write / Do not write] dates/obs.
dates = *arg*	Excel format for writing date: "first" (convert to the first day of the corresponding observation if necessary), "last" (convert to the last day of the corresponding observation).
t	Write by series. Default is to write by obs with 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.

Cross-references

See "Exporting to a Spreadsheet or Text File" on page 144 of *User's Guide I* for a discussion. See `pagesave` (p. 402) for a superior method of exporting workfile pages to foreign formats.

See also read (p. 418).

xclose	External Interface Commands

Close an open connection to an external application.

Syntax

xclose

xclose is used to close an open COM session with an external application.

Examples

```
xclose
```

closes the open connection.

Cross-references

See "EViews COM Automation Client Support (MATLAB and R)," beginning on page 162 for discussion.

See also xopen (p. 496), xget (p. 493), xput (p. 498), xrun (p. 500), and xlog (p. 496).

xget	External Interface Commands

Retrieve data from an external application into an EViews object.

Syntax

xget*(options) object* **@smpl** *sample*

xget is used to retrieve data from an external COM application (support is currently provided for MATLAB and R). An existing connection to the external application must have already been opened using xopen (p. 496). The xget command should be followed by the name of the object to retrieve, followed by an optional **@smpl** keyword and a sample specification, or the name of a sample object. Including **@smpl** lets you specify the observations into which the data will be retrieved.

Options

name = *arg*	Specify the name of the object to be created in EViews. If the *name* option is not specified, the created object will have the same name as the external application object that is being retrieved.

wf = *arg*	Specify the workfile into which the retrieved objects will be placed. The specified workfile must be currently open inside EViews. If the *wf* option is not specified, the objects will be put into the current default workfile.
page = *arg*	Specify the workfile page into which the retrieved objects will be placed. If the *page* option is not specified, the objects will be put into the current default page.
type = *arg*	Specify the EViews object type to be created. *arg* can be "series", "alpha", "matrix", "vector", "svector", "coef", "sym", "scalar" or "string". If the *type* option is not set, EViews will automatically decide which object type to create.
mode = *arg* *(default* = "merge"*)*	Merge options: "protect" (protect destination – only retrieve values if destination does not already exist), "merge" (prefer source -– merge only if source value is non-missing), "mergedest" (prefer destination – merge only if destination value is missing), "update" (replace all destination values in the retrieval sample with source values), "overwrite" (replace all destination values in retrieval sample with source values, and NAs outside of sample).

R-specific options

fdata = *arg*	When reading a factor object, specifies how to read the factor data: as numbers *(default)*, as labels ("labels"), as both numbers and labels ("both"). If "fdata = both", the labels will be read into a valmap object, and the valmap will be attached to the destination series (the data target must be a series for this setting).
fmap = *arg*	Name of the valmap object to hold the factor labels (when "fdata = both").

fmode = *arg*	Specifies settings for reading factor label data into a valmap: merge with preference to the existing values in the map *(default)*, merge with preference to the factor map values ("merge"), overwrite existing valmap definitions ("overwrite"), do not alter an existing valmap ("protect"). The default method adds definitions from the factor to an existing valmap, retaining existing valmap entries that conflict with definitions from the factor. "Merge" adds definitions from the factor to an existing valmap, replacing conflicting valmap entries with definitions from the factor. "Overwrite" replaces an existing valmap specification with the factor definitions. "Protect" ensures that an existing valmap is not altered.

MATLAB-specific options

workspace = *arg* *(default* = "base"*)*	MATLAB environment in which to obtain the data: "base" (base workspace), "global" (global workspace).

Examples

```
xget X
```

retrieves an object "X" from the current open external application connection, and stores it into the current default workfile.

```
xget(type=vector, name=x1) Ymat
```

retrieves the object named "Ymat" and stores it into the current default workfile as a vector named X1.

```
xget(wf=mywf, type=series, name=x2) X @smpl 1990 1999
```

retrieves X and stores it in the MYWF workfile as a series called X2, where only the observations between 1990 and 1999 are filled in.

Cross-references

See "EViews COM Automation Client Support (MATLAB and R)," beginning on page 162 for discussion.

See also `xopen` (p. 496), `xclose` (p. 493), `xput` (p. 498), `xrun` (p. 500) and `xlog` (p. 496).

xlog	External Interface Commands

Switch on or off the external application log inside EViews.

Syntax

xlog*(arg)*

`xlog` is used to switch on or off the external COM application log window inside EViews. *arg* should be either "show" (to switch the log on), or "hide" (to switch it off).

Examples

```
xlog(hide)
```

switches off the log.

Cross-references

See "EViews COM Automation Client Support (MATLAB and R)," beginning on page 162 for discussion.

See also `xopen` (p. 496), `xclose` (p. 493), `xget` (p. 493), `xput` (p. 498), and `xrun` (p. 500).

xopen	External Interface Commands

Open a connection to an external application.

Syntax

xopen*(options)*

`xopen` is used to start a COM session with an external application, either R or MATLAB. EViews can only have a single session open at a time; a session must be closed (see `xopen` (p. 496)) before a new session can be opened.

Options

type = *arg*	Set the type of connection to be opened. *arg* may be "r" (R) or "m" (MATLAB).
progid = *arg*	(optional) Set the version of MATLAB or statconnDCOM to which EViews connects when opening a session. If not specified, EViews will use the default ProgID specified in the global options.

nolog	Do not open a session log window.
case = *arg*	Specify the default case for objects in R or MATLAB using `xput` (p. 498). If "case = " is not provided, the default case specified in the global options will be assumed. Note that once a connection has been opened, the case option cannot be changed; you may however, use the "name = " option when using `xput` (p. 498) to provide an explicit name.

Note that the MATLAB ProgIDs may be of particular interest as MATLAB (R2008a and later) offers several distinct ways in which to connect to the server. The relevant ProgIDs are:

1. "MATLAB.Application"— this ProgID starts a command window version of MATLAB that was most recently used as a server (might not be the latest installed version of MATLAB).
2. "MATLAB.Application.Single"— same as (1) but starts a dedicated server so that other programs looking to use MATLAB cannot connect to your instance.
3. "MATLAB.Autoserver"—starts a command window server using the most recent version of MATLAB.
4. "MATLAB.Autoserver.Single"—same as (3) but starts a dedicated server.
5. "MATLAB.Desktop.Application"—starts the full desktop MATLAB as a server using the most recent version of MATLAB.

Each ProgID may be amended to indicate a specific version of MATLAB. For example, using the ProgID:

MATLAB.Desktop.Application.7.6

instructs EViews to use the full desktop MATLAB GUI for version R2008a (v7.6) as the Automation server.

For additional details on the use of ProgIDs, see the whitepapers *Using EViews COM Automation Client for* MATLAB and *Using EViews COM Automation Client for* R.

Examples

```
xopen(type=m)
```

opens a connection to MATLAB.

```
xopen(type=r, case=lower)
```

opens a connection to R and sets the default name-case to lower.

```
xopen(type=m, progid=MATLAB.Desktop.Application.7.9)
```

opens a connection to MATLAB 7.9 running with the full desktop GUI.

Cross-references

See "EViews COM Automation Client Support (MATLAB and R)," beginning on page 162 for discussion. See also "External Program Interface" on page 774 of *User's Guide I* for global options setting.

See `xclose` (p. 493), `xget` (p. 493), `xput` (p. 498), `xrun` (p. 500), and `xlog` (p. 496).

xput	External Interface Commands

Send an EViews object to an external application.

Syntax

xput*(options) ob_list* **@drop** *ob1* **@smpl** *sample*

`xput` is used to push data from EViews to an COM external application (either R or MATLAB). An existing connection to the external application must have already been opened, using `xopen`.

ob_list should be a space delimited list of EViews objects to push into the application. An asterisk (*) can be used as a wildcard. Objects that can be pushed include series, alphas, matrices, syms, vectors, svectors, rowvectors, coefs, scalars and strings. if the object list contains an object you do not wish to be pushed to the external application, you can use **@drop** followed by the name of the object.

For series and alphas, you may enter a sample to specify which observations you wish to be pushed, using **@smpl** followed by the sample specification or the name of a sample object.

Options

name = *arg*	Specify the name or names of the objects to be created in the destination application. Multiple names may be specified using a wildcard or a space-delimited list of names. Names specified using the *name* option are case-sensitive so that destination objects will preserve the case of *arg*. If the *name* option is not specified, the created objects will have the same name as the EViews objects that are being pushed, with the case determined by the case established for the COM connection (see `xopen` (p. 496)).
mode = *arg* *(default* = "overwrite"*)*	Merge options: "protect" (protect destination – only put values if destination does not already exist), "overwrite" (replace all destination values with source values, and resize if necessary).

wf = *arg*	Specify the workfile containing the objects to be pushed. The specified workfile must be currently open inside EViews. If the "wf = " option is not specified, the objects will be taken from the current default workfile.
page = *arg*	Specify the workfile page containing the objects to be pushed. If the "page = " option is not specified, the objects will be taken from the current default page.
map	Write value-mapped series using map labels instead of underlying values.

R-specific options

rtype = *arg*	Specify the type of object to be created in R. *arg* can be "vector", "ts", "data.frame" or "list".

MATLAB-specific options

mtype = *arg*	Specify the type of object to be created in MATLAB: "matrix" (matrix object), "cell" (cell object).
workspace = *arg* *(default* = "base"*)*	MATLAB environment in which to place the data: "base" (base workspace), "global" (global workspace).

Examples

```
xopen(type=m)
xput(name=g) group01
```

opens a connection to MATLAB and then pushes the group GROUP01 to MATLAB, giving it the name G.

```
xopen(type=r)
xput(page=page2, rtype=vector) x @smpl 1990 1999
```

Opens a connection to R and pushes the series X from page PAGE2 of the current default workfile into a vector in R. Only the 10 observations for X from 1990 and 1999 are pushed into R. X will be named in R using the name "X" with the default case specified in the global options.

```
xopen(type=r, case=upper)
xput(rtype=data.frame, name=df1) x* @drop x2
```

Opens a connection to R and puts all objects whose name starts with "X", apart from the object X2, into a data frame, named "df1". The names of the "X*" objects will be uppercased in R.

Cross-references

See "EViews COM Automation Client Support (MATLAB and R)," beginning on page 162 for discussion. See also "External Program Interface" on page 774 of *User's Guide I* for global options setting of the default case for names.

See also `@xputnames` (p. 602), `xopen` (p. 496), `xclose` (p. 493), `xget` (p. 493), `xrun` (p. 500), and `xlog` (p. 496).

xrun	External Interface Commands

Run a command in an external application.

Syntax

xrun *command*

`xrun` is used to run a command in an external COM application (either R or MATLAB). An existing connection to the external application must have already been opened using `xopen`. `xrun` should be followed by the command you wish to run, surrounded in quotes.

Examples

```
xopen(type=m, case=upper)
xput Y
xput XS
xrun "beta = inv(XS'*XS)*XS'*Y"
```

opens a connection to MATLAB, sends the series Y and the group XS to MATLAB, then runs a command which will compute the least squares coefficients from a regression of Y on XS.

The commands

```
xopen(type=r, case=upper)
xput(rtype=data.frame, name=cancer) age drug2 drug3 studytim
xrun z<-glm(STUDYTIM~AGE+1+DRUG2+DRUG3,
    family=Gamma(link=log),data=cancer)
```

send data to R and estimate GLM model.

Note that the statconnDCOM package does not always automatically capture all of your R output. Consequently, you may find that using `xrun` to issue a command that displays output in R may only return a subset of the usual output to your log window. In the most extreme case, you may see a simple "OK" message displayed in your log window. To instruct statconnDCOM to show all of the output, you should use enclose your command inside an explicit `print` statement in R. Thus, to display the contents of a matrix X, you must issue the command

```
xrun print(X)
```

instead of the more natural

```
xrun X
```

Cross-references

See "EViews COM Automation Client Support (MATLAB and R)," beginning on page 162 for a discussion.

See also xopen (p. 496), xclose (p. 493), xget (p. 493), xput (p. 498), and xlog (p. 496).

Chapter 13. Operator and Function Reference

The reference material in this section describes basic 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 6. "Working with Data," beginning on page 165 of *User's Guide I*.

This material is divided into several topics:

- Operators.
- Basic mathematical functions.
- Time series functions.
- Financial functions.
- Descriptive statistics.
- Cumulative statistics functions.
- Moving statistics functions.
- Group row functions.
- By-group statistics.
- Additional and special functions.
- Trigonometric functions.
- Statistical distribution functions.
- String functions.
- Date functions.
- Indicator functions.
- Workfile and informational functions.
- Value map functions.

Extensive documentation on libraries of more specialized functions is provided elsewhere:

- For a description of functions related to string and date manipulation, see "String Function Summary" on page 567 and "Date Function Summary" on page 568.
- For discussion of the workfile functions that provide information about each observation of a workfile based on information contained in the structure of the workfile, see Chapter 15. "Workfile Functions," on page 549.

- Functions for working with value maps are documented in "Valmap Functions" on page 214 of the *User's Guide I*.
- For a list of functions specific to matrices, see "Matrix Command and Function Summary" on page 605.

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 165. 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.
^	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 nonzero, and 0 otherwise.

In addition, EViews provides special functions to perform comparisons using special rules for handling missing values (see "Missing Values" on page 173):

`@eqna(x,y)`	equal to	takes the value 1 if X and Y are equal, and 0 otherwise. NAs are treated as ordinary values for purposes of comparison.
`@isna(x)`	equal to NA	takes the value 1 if X is equal to NA and 0 otherwise.
`@neqna(x,y)`	not equal to	takes the value 1 if X and Y are not equal, and 0 if they are equal. NAs are treated as ordinary values for purposes of comparison.

Basic Mathematical Functions

The following 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.

Note: `@iff`, `@inv`, and `@recode` do not work with matrix objects.

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) = 1.79176, @factlog(0) = 0.
`@floor(x)`	largest integer not greater than	@floor(1.23) = 1, @floor(-3.1) = -4.
`@iff(s,x,y)`	recode by condition	returns x if condition s is true; otherwise returns y. Note this is the same as `@recode`.
`@inv(x)`	reciprocal, $1/x$	inv(2) = 0.5 (For series only; you should use `@einv` to obtain the element inverse of a matrix).
`@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(@exp(1)) = 1.
`@log10(x)`	base-10 logarithm, $\log_{10}(x)$	@log10(100) = 2.

`@logx(x,b)`	base-*b* logarithm, $\log_b(x)$	@logx(256,2) = 8.
`@nan(x,y)`	recode NAs in X to Y	returns x if $x<>\text{NA}$, and y if $x = \text{NA}$.
`@recode(s,x,y)`	recode by condition	returns x if condition s is true; otherwise returns y.
`@round(x)`	round to the nearest integer	@round(-97.5) = -98, @round(3.5) = 4.
`@sqrt(x), sqr(x)`	square root	@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
`d(x)`	first difference	$(1-L)X = X - X(-1)$ where L is the lag operator.
`d(x,n)`	*n*-th order difference	$(1-L)^n X$.
`d(x,n,s)`	*n*-th order difference with a seasonal difference at s	$(1-L)^n(1-L^s)X$.
`dlog(x)`	first difference of the logarithm	$(1-L)\log(X) = \log(X) - \log(X(-1))$.
`dlog(x,n)`	*n*-th order difference of the logarithm	$(1-L)^n\log(X)$.
`dlog(x,n,s)`	*n*-th order difference of the logarithm with a seasonal difference at s	$(1-L)^n(1-L^s)\log(X)$.
`@pc(x)`	one-period percentage change (in percent)	equals `@pch(x)*100`
`@pch(x)`	one-period percentage change (in decimal)	$(X - X(-1))/X(-1)$.
`@pca(x)`	one-period percentage change—annualized (in percent)	equals `@pcha(x)*100`

`@pcha(x)`	one-period percentage change—annualized (in decimal)	$@\text{pcha}(x) = (1 + @\text{pch}(x))^n - 1$ where n is the lag associated with one-year ($n = 4$) for quarterly data, etc.).
`@pcy(x)`	one-year percentage change (in percent)	equals `@pchy(x)*100`
`@pchy(x)`	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.).

Financial Functions

The following series-only functions permit you to perform calculations commonly employed in financial analysis. The functions are evaluated for each observation in the workfile sample.

Name	Function	Description
`@fv(r,n,x[,fv,t])`	future value	future value of an n-period annuity with rate r, payments x, and optional final lump sum payment fv. A non-zero value for the optional t indicates that the payments are made at the beginning of periods instead of ends.
`@nper(r,x,pv[,fv,t])`	number of periods	number of annuity periods required to produce the present value pv with rate r, payments x and optional final lump sum payment fv. A non-zero value for the optional t indicates that the payments are made at the beginning of periods.
`@pv(r,n,x[,fv,t])`	present value	present value of an n-period annuity with rate r, payments x, and optional final lump sum payment fv. A non-zero value for the optional t indicates that the payments are made at the beginning of periods.

`@pmt(r,n,pv[,fv,t])`	payment amount	payment amount for an n-period, pv present value annuity with rate r and optional final lump sum payment fv. A non-zero value for the optional t indicates that the payments are made at the beginning of periods.
`@rate(n,x,pv[,fv,t])`	interest	interest rate for an n period annuity with payments x, present value pv, and optional final lump sum payment fv. A non-zero value for the optional t indicates that the payments are made at the beginning of periods.

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, "1945m01 1979m12")
```

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, "1945m01 1979m12")
```

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:

```
smpl 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.

If the desired sample expression contains the double quote character, it may be entered using the double quote as an escape character. Thus, if you wish to use the equivalent of,

```
smpl if name = "Smith"
```

in your @MEAN function, you should enter the sample condition as:

```
series y = @mean(x, "if name=""Smith""")
```

The pairs of double quotes in the sample expression are treated as a single double quote.

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 (division by n).
`@covp(x,y[,s])`	population covariance	the covariance between X and Y (division by n).
`@covs(x,y[,s])`	sample covariance	the covariance between X and Y (division by $n-1$).
`@gmean(x[,s])`	geometric mean	the geometric mean of X. The geometric mean is calculated as the exponential of the sum of the logs of X.
`@hmean(x[,s])`	harmonic mean	computes the harmonic mean of the values of X. The harmonic mean is calculated as the reciprocal of the mean of the reciprocals of X.
`@imax(x)`	maximum index	workfile index of the maximum of the values in X for the current sample.
`@imin(x)`	minimum index	workfile index of the maximum of the values in X for the current sample.
`@inner(x,y[,s])`	inner product	the inner product of X and Y.
`@kurt(x[,s])`	kurtosis	kurtosis of values in X.

`@mae(x,y[,s])`	mean absolute error	the mean of the absolute value of the difference between X and Y.
`@mape(x,y[,s])`	mean absolute percentage error	100 multiplied by the mean of the absolute difference between X and Y, divided by Y.
`@max(x[,s])`	maximum	maximum of the values in X.
`@mean(x[,s])`	mean	average of the values in X.
`@median(x[,s])`	median	computes the median of the X (uses the average of middle two observations if the number of observations is even).
`@min(x[,s])`	minimum	minimum of the values in X.
`@nas(x[,s])`	number of NAs	the number of missing observations for X in the current sample.
`@prod(x[,s])`	product	the product of the elements of X (note this function is prone to numerical overflows).
`@obs(x[,s])`	number of observations	the number of non-missing observations for X in the current sample.
`@quantile(x,q[,m,s])`	quantile	the *q*-th quantile of the series X. *m* is an optional string argument for specifying the quantile method: "b" (Blom), "r" (Rankit-Cleveland), "o" (Ordinary), "t" (Tukey), "v" (van der Waerden), "g" (Gumbel). The default value is "r".
`@ranks(x[,o,t,s])`	ranks	the ranking of each observation in X. The order of ranking is set using *o*: "a" (ascending - default) or "d" (descending). Ties are broken according to the setting of *t:* "i" (ignore), "f" (first), "l" (last), "a" (average - default), "r" randomize. If you wish to specify tie-handling options, you must also specify the order option (*e.g.* '@ranks(x, "a", "i")').
`@rmse(x,y[,s])`	root mean square error	the square root of the mean of the squared difference between X and Y.
`@skew(x[,s])`	skewness	skewness of values in X.

`@stdev(x[,s])`	standard deviation	square root of the unbiased sample variance (sum-of-squared residuals divided by $n-1$).
`@stdevp(x[,s])`	population standard deviation	square root of the population variance (sum-of-squared residuals divided by n).
`@stdevs(x[,s])`	sample standard deviation	square root of the unbiased sample variance. Note this is the same calculation as `@stdev`.
`@sum(x[,s])`	sum	the sum of X.
`@sumsq(x[,s])`	sum-of-squares	sum of the squares of X.
`@theil(x,y[,s])`	Theil inequality coefficient	the root mean square error divided by the sum of the square roots of the means of X squared and Y squared.
`@var(x[,s])`	variance	variance of the values in X (division by n).
`@varp(x[,s])`	population variance	variance of the values in X. Note this is the same calculation as `@var`.
`@vars(x[,s])`	sample variance	sample variance of the values in X (division by $n-1$).

Cumulative Statistic Functions

These functions perform basic running or cumulative statistics for a series and may be used as part of a series expression. The functions are split into two types, those that cumulate forwards and those that cumulate backwards. The forwards cumulating functions return the running values of a statistic from the start of the workfile (or optionally a sample) to the current observation. The backwards cumulating functions return the running values from the end of the workfile (or sample) to the current observation.

By default, EViews will use the *entire workfile range* when computing the statistics. You may provide the optional sample *s* as a literal (quoted) sample expression or a named sample.

Missing values, NAs, do not propagate through these functions. Thus the cumulative sums of the numbers 1, 3, 4, NA, 5 are 1, 4, 8, 8, 13.

Name	**Function**	**Description**
`@cumsum(x[,s])`	cumulative sum	cumulative sum of the values in X from the start of the workfile/sample.

`@cumprod(x[,s])`	cumulative product	cumulative product of the values in X from the start of the workfile/sample (note this function could be subject to numerical overflows).
`@cummean(x[,s])`	cumulative mean	mean of the values in X from the start of the workfile/sample to the current observation.
`@cumstdev(x[,s])`	cumulative standard deviation	sample standard deviation of the values in X from the start of the workfile/sample to the current observation. Note this calculation involves division by $n-1$.
`@cumstdevp(x[,s])`	cumulative population standard deviation	population standard deviation of the values in X from the start of the workfile/sample to the current observation. Note this calculation involves division by n.
`@cumstdevs(x[,s])`	cumulative sample standard deviation	sample standard deviation of the values in X from the start of the workfile/sample. Note this performs the same calculation as `@cumstdev`.
`@cumvar(x[,s])`	cumulative variance	population variance of the values in X from the start of the workfile/sample to the current observation. Note this calculation involves division by n.
`@cumvarp(x[,s])`	cumulative population variance	population variance of the values in X from the start of the workfile/sample to the current observation. Note this performs the same calculation as `@cumvar`.
`@cumvars(x[,s])`	cumulative sample variance	sample variance of the values in X from the start of the workfile/sample to the current observation. Note this calculation involves division by $n-1$.
`@cummax(x[,s])`	cumulative maximum	maximum of the values in X from the start of the workfile/sample to the current observation.
`@cummin(x[,s])`	cumulative minimum	minimum of the values in X from the start of the workfile/sample to the current observation.
`@cumsumsq(x[,s])`	cumulative sum-of-squares	sum of squares of the values in X from the start of the workfile/sample to the current observation.

`@cumobs(x[,s])`	cumulative nmber of non-NA observations	the number of non-missing observations in X from the start of the workfile/sample to the current observation.
`@cumnas(x[,s])`	cumulative number of NA observations	the number of missing observations in X from the start of the workfile/sample to the current observation.
`@cumbsum(x[,s])`	backwards cumulative sum	cumulative sum of the values in X from the end of the workfile/sample.
`@cumbprod(x[,s])`	backwards cumulative product	cumulative product of the values in X from the end of the workfile/sample (note this function could be subject to numerical overflows).
`@cumbmean(x[,s])`	backwards cumulative mean	mean of the values in X from the end of the workfile/sample to the current observation.
`@cumbstdev(x[,s])`	backwards cumulative standard deviation	sample standard deviation of the values in X from the end of the workfile/sample to the current observation. Note this calculation involves division by $n-1$.
`@cumbstdevp(x[,s])`	backwards cumulative population standard deviation	population standard deviation of the values in X from the end of the workfile/sample to the current observation. Note this calculation involves division by n.
`@cumbstdevs(x[,s])`	backwards cumulative sample standard deviation	sample standard deviation of the values in X from the end of the workfile/sample. Note this performs the same calculation as `@cumstdev`.
`@cumbvar(x[,s])`	backwards cumulative variance	population variance of the values in X from the end of the workfile/sample to the current observation. Note this calculation involves division by n
`@cumbvarp(x[,s])`	backwards cumulative population variance	population variance of the values in X from the end of the workfile/sample to the current observation. Note this performs the same calculation as `@cumvar`.
`@cumbvars(x[,s])`	backwards cumulative sample variance	sample variance of the values in X from the end of the workfile/sample to the current observation. Note this calculation involves division by $n-1$.
`@cumbmax(x[,s])`	backwards cumulative maximum	maximum of the values in X from the end of the workfile/sample to the current observation.

`@cumbmin(x[,s])`	backwards cumulative minimum	minimum of the values in X from the end of the workfile/sample to the current observation.
`@cumbsumsq(x[,s])`	backwards cumulative sum-of-squares	sum of squares of the values in X from the start of the workfile/sample to the current observation.
`@cumbobs(x[,s])`	backwards cumulative nmber of non-NA observations	the number of non-missing observations in X from the end of the workfile/sample to the current observation.
`@cumbnas(x[,s])`	backwards cumulative nmber of NA observations	the number of missing observations in X from the end of the workfile/sample to the current observation.

Moving Statistic Functions

These functions perform basic rolling or moving statistics. They may be used as part of a series expression.

The moving statistic functions are in two types, those that propagate missing observations (NAs) and those that don't. The functions that do not propagate NAs, which start with "@m", skip observations which are NA. The functions that do propagate NAs will return NA when an NA is encountered.

For example if the series X contains {1, 3, 4, NA, 5, 3, 2} then "@movav(x,2)" will give {NA, 2, 3.5, NA, NA, 4, 2.5}, whereas "@mav(x,2)" will give {1, 2, 3.5, 4, 5, 4, 2.5}.

Name	Function	Description
`@movsum(x,n)`	n-period backward moving sum	$\text{@movsum(x,3)} = (X + X(-1) + X(-2))$. NAs are propagated.
`@movav(x,n)`	n-period backward moving average	$\text{@movav(x, 3)} = (X + X(-1) + X(-2))/3$ NAs are propagated.
`@movavc(x,n)`	n-period centered moving average	centered moving average of X. Note if n is even then the window length is increased by one and the two endpoints are weighted by 0.5. NAs are propagated.

`@movstdev(x,n)`	n-period backwards moving standard deviation	sample standard deviation (division by $n-1$) of X for the current and previous $n-1$ observations. NAs are propagated.
`@movstdevs(x,n)`	n-period backwards moving sample standard deviation	sample standard deviation (division by $n-1$) of X for the current and previous $n-1$ observations. Note this is the same calculation as @movstdev. NAs are propagated.
`@movstdevp(x,n)`	n-period backwards moving population standard deviation	population standard deviation (division by n) of X for the current and previous $n-1$ observations. NAs are propagated.
`@movvar(x,n)`	n-period backwards moving variance	population variance (division by n) of X for the for the current and previous $n-1$ observations. NAs are propagated.
`@movvars(x,n)`	n-period backwards moving sample variance	sample variance (division by $n-1$) of X for the current and previous $n-1$ observations. NAs are propagated.
`@movvarp(x,n)`	n-period backwards moving population variance	population variance (division by n).of X for the current and previous $n-1$ observations. Note this is the same calculation as `@movvar`. NAs are propagated.
`@movcov(x,y,n)`	n-period backwards moving covariance	population covariance (division by n) between X and Y of the current and previous $n-1$ observations. NAs are propagated.
`@movcovs(x,y,n)`	n-period backwards moving sample covariance	sample covariance (division by $n-1$) between X and Y of the current and previous $n-1$ observations. NAs are propagated.
`@movcovp(x,y,n)`	n-period backwards moving population covariance	population covariance (division by n) between X and Y of the current and previous $n-1$ observations. Note this is the same calculation as @movcov. NAs are propagated.
`@movcor(x,y,n)`	n-period backwards moving correlation	correlation between X and Y of the current and previous $n-1$ observations. NAs are propagated. NAs are propagated.

`@movmax(x,n)`	*n*-period backwards moving maximum	the maximum of X for the current and previous $n-1$ observations. NAs are propagated.
`@movmin(x,n)`	*n*-period backwards moving minimum	the minimum of X for the current and previous $n-1$ observations. NAs are propagated.
`@movsumsq(x,n)`	*n*-period backwards sum-of-squares	the sum-of-squares of X for the current and previous $n-1$ observations. NAs are propagated.
`@movskew(x,n)`	*n*-period backwards skewness	the skewness of X for the current and previous $n-1$ observations. NAs are propagated.
`@movkurt(x,n)`	*n*-period backwards kurtosis	the kurtosis of X for the current and previous $n-1$ observations. NAs are propagated.
`@movobs(x,n)`	*n*-period backwards nmber of non-NA observations	the number of non-missing observations in X for the current and previous $n-1$ observations. Note this function always returns the same value as `@mobs`.
`@movnas(x,n)`	*n*-period backwards nmber of NA observations	the number of missing observations in X for the current and previous *n-1* observations. Note this function always returns the same value as `@mnas`.
`@movinner(x,y,n)`	*n*-period backwards inner product of X and Y	inner product of X and Y for the current and previous $n-1$ observations. NAs are propagated.
`@msum(x,n)`	*n*-period backward moving sum	$\text{@msum(x,3)} = (X + X(-1) + X(-2))$ NAs are not propagated.
`@mav(x,n)`	*n*-period backward moving average	$\text{@mav(x, 3)} = (X + X(-1) + X(-2))/3$ NAs are not propagated.
`@mavc(x,n)`	*n*-period centered moving average	centered moving average of X. Note if n is even then the window length is increased by one and the two endpoints are weighted by 0.5. NAs are not propagated.

`@mstdev(x,n)`	n-period backwards moving standard deviation	sample standard deviation (division by $n-1$) of X for the current and previous $n-1$ observations. NAs are not propagated.
`@mstdevs(x,n)`	n-period backwards moving sample standard deviation	sample standard deviation (division by $n-1$) of X for the current and previous $n-1$ observations. Note this is the same calculation as `@movstdev`. NAs are not propagated.
`@mstdevp(x,n)`	n-period backwards moving population standard deviation	population standard deviation (division by n) of X for the current and previous $n-1$ observations. NAs are not propagated.
`@mvar(x,n)`	n-period backwards moving variance	population variance (division by n) of X for the for the current and previous $n-1$ observations. NAs are not propagated.
`@mvars(x,n)`	n-period backwards moving sample variance	sample variance (division by $n-1$) of X for the current and previous $n-1$ observations. NAs are not propagated.
`@mvarp(x,n)`	n-period backwards moving population variance	population variance (division by n) of X for the current and previous $n-1$ observations. Note this is the same calculation as `@movvar`. NAs are not propagated.
`@mcov(x,y,n)`	n-period backwards moving covariance	population covariance (division by n) between X and Y of the current and previous $n-1$ observations. NAs are not propagated.
`@mcovs(x,y,n)`	n-period backwards moving sample covariance	sample covariance (division by $n-1$) between X and Y of the current and previous $n-1$ observations. NAs are not propagated.
`@mcovp(x,y,n)`	n-period backwards moving population covariance	population covariance (division by n) between X and Y of the current and previous $n-1$ observations. Note this is the same calculation as `@movcov`. NAs are not propagated.
`@mcor(x,y,n)`	n-period backwards moving correlation	correlation between X and Y of the current and previous $n-1$ observations. NAs are not propagated.

`@mmax(x,n)`	n-period backwards moving maximum	the maximum of X for the current and previous $n-1$ observations. NAs are not propagated.
`@mmin(x,n)`	n-period backwards moving minimum	the minimum of X for the current and previous $n-1$ observations. NAs are not propagated.
`@msumsq(x,n)`	n-period backwards sum-of-squares	the sum-of-squares of X for the current and previous $n-1$ observations. NAs are not propagated.
`@mskew(x,n)`	n-period backwards skewness	the skewness of X for the current and previous $n-1$ observations. NAs are not propagated.
`@mkurt(x,n)`	n-period backwards kurtosis	the kurtosis of X for the current and previous $n-1$ observations. NAs are not propagated.
`@mobs(x,n)`	n-period backwards number of non-NA observations	the number of non-missing observations in X for the current and previous $n-1$ observations. Note this function always returns the same value as `@movobs`.
`@mnas(x,n)`	n-period backwards number of NA observations	the number of missing observations in X for the current and previous $n-1$ observations. Note this function always returns the same value as `@movnas`.
`@minner(x,y,n)`	n-period backwards inner product of X and Y	inner product of X and Y for the current and previous $n-1$ observations. NAs are not propagated.

Group Row Functions

These functions work on a group object. They may be used to create a new series where each row of that series is a function of the corresponding row of a group. Thus, if group G contains the series X, Y and Z, the `@rmean` function will return (X + Y + Z)/3.

These functions do not propagate NAs, *i.e.*, they generate data based upon the observations within the group row that are non-NA.

As an example, the `@rsum` function will return a series where each row is equal to the sum of the non-NA elements in the row of a group.

Note that in the following descriptions, G is a named group in your workfile.

Name	Function	Description
`@columns(g)`	number of columns.	returns the number of columns in G. Note this is the same as "G.@count", and will return the same number for every row.
`@rnas(g)`	row-wise number of NAs.	the number of missing observations in the rows of G.
`@robs(g)`	row-wise number of non-NAs.	the number of non-missing observations in the rows of G.
`@rvalcount(g,v)`	row-wise count of observations matching *v*.	the number of observations with a value equal to *v* in the rows of G. Note *v* can be a scalar or a series
`@rsum(g)`	row-wise sum.	the sum of the rows of G.
`@rmean(g)`	row-wise mean.	the mean of the rows of G.
`@rstdev(g)`	row-wise standard deviation.	the sample standard deviation of the rows of G. Note division is by $n-1$.
`@rstdevp(g)`	row-wise population standard deviation.	the population standard deviation of the rows of G. Note division is by n.
`@rstdevs(g)`	row-wise sample standard deviation.	the sample standard deviation of the rows of G. Note division is by $n-1$, and this calculation is the same as `@rstdev`.
`@rvar(g)`	row-wise variance.	the population variance of the rows of G. Note division is by n.
`@rvarp(g)`	row-wise population variance	the population variance of the rows of G. Note division is by n, and this calculation is the same as `@rvar`.
`@rvars(g)`	row-wise sample variance.	the sample variance of the rows of G. Note division is by $n-1$.
`@rsumsq(g)`	row-wise sum-of-squares.	the sum of the squares of the rows of G.
`@rfirst(g)`	row-wise first non-NA value.	the value of the first non-missing observation in the rows of G.
`@rfirsti(g)`	row-wise first non-NA index.	the index (*i.e.*, column number) of the first non-missing observation in the rows of G.

`@rlast(g)`	row-wise last non-NA value.	the value of the last non-missing observation in the rows of G
`@rlasti(g)`	row-wise last non-NA index.	the index (*i.e.*, column number) of the last non-missing observation in the rows of G.
`@rmax(g)`	row-wise maximum value.	the value of the maximum observation in the rows of G.
`@rmaxi(g)`	row-wise maximum index.	the index (*i.e.*, column number) of the maximum observation in the rows of G. In the case of ties, the first matching index is given.
`@rmin(g)`	row-wise minimum value.	the value of the minimum observation in the rows of G.
`@rmini(g)`	row-wise minimum index.	the index (*i.e.*, column number) of the minimum observation in the rows of G. In the case of ties, the first matching index is given.

By-Group Statistics

The following "by group"-statistics are available in EViews. They may be used as part of a series expression statements to compute the statistic for subgroups, and to assign the value of the relevant statistic to each observation.

The functions take a series expression *arg1* for which we wish to compute group statistics and one or more series expressions *arg2*, defining the groups over which we wish to compute the statistics. In the leading case, you will provide a single series expression containing categories. More generally, you can provide several series expressions separated by commas (*arg2*, *arg3*, ...) that jointly define the categories. For brevity, the table entries below only depict a single categorical variable *arg2*.

Function	**Description**
`@obsby(`*arg1*, *arg2[, s]*`)`	Number of non-NA *arg1* observations for each *arg2* group.
`@nasby(`*arg1*, *arg2[, s]*`)`	Number of *arg1* NA values for each *arg2* group.
`@sumsby(`*arg1*, *arg2[, s]*`)`	Sum of *arg1* observations for each *arg2* group.
`@meansby(`*arg1*, *arg2[, s]*`)`	Mean of *arg1* observations for each *arg2* group.
`@minsby(`*arg1*, *arg2[, s]*`)`	Minimum value of *arg1* observations for each *arg2* group.
`@maxsby(`*arg1*, *arg2[, s]*`)`	Maximum value of *arg1* observations for each *arg2* group.

`@mediansby`(*arg1*, *arg2[, s]*)	Median of *arg1* observations for each *arg2* group.
`@varsby`(*arg1*, *arg2[, s]*)	Variance of *arg1* observations for each *arg2* group (division by n).
`@varssby`(*arg1*, *arg2[, s]*)	Sample variance of *arg1* observations for each *arg2* group (division by $n-1$).
`@varpsby`(*arg1*, *arg2[, s]*)	Variance of *arg1* observations for each *arg2* group (division by n).
`@stdevsby`(*arg1*, *arg2[, s]*)	Sample standard deviation of *arg1* observations for each *arg2* group (division by $n-1$).
`@stdevssby`(*arg1*, *arg2[, s]*)	Sample standard deviation of *arg1* observations for each *arg2* group (division by $n-1$).
`@stdevpsby`(*arg1*, *arg2[, s]*)	Standard deviation of *arg1* observations for each *arg2* group (division by n).
`@sumsqsby`(*arg1*, *arg2[, s]*)	Sum of squares of *arg1* observations for each *arg2* group.
`@quantilesby`(*arg1*, *arg2*, *arg3[, s]*)	Quantiles of *arg1* observations for each *arg2* group where *arg1* is the desired quantile.
`@skewsby`(*arg1*, *arg2[, s]*)	Skewness of *arg1* observations for each *arg2* group.
`@kurtsby`(*arg1*, *arg2[, s]*)	Kurtosis of *arg1* observations for each *arg2* group.
`@firstsby`(*arg1*, *arg2[, s]*)	First non-missing value in *arg1* for each *arg2* group
`@lastsby`(*arg1*, *arg2[, s]*)	Last non-missing value in *arg1* for each *arg2* group

With the exception of `@quantileby`, all of the functions take the form:

@STATBY(*arg1*, *arg2[, arg3, arg4, …, argn, s]*)

where *@STATBY* is one of the by-group function keyword names, *arg1* is a series expression, *arg2* and the optional *arg3* to *argn* are numeric or alpha series expression identifying the subgroups, and *s* is an optional sample literal (a quoted sample expression) or a named sample. With the exception of `@obsby`, *arg1* must be a numeric series.

By default, EViews will use the workfile sample when computing the descriptive statistics. You may provide the optional sample *s* as a literal (quoted) sample expression or a named sample.

The `@quantileby` function requires an additional argument for the quantile value that you wish to compute:

`@quantileby`(*arg1*, *arg2[, arg3, arg4, …, argn]*, *q[, s]*)

For example, to compute the first quartile, you should use the *q* value of 0.25.

There are two related functions of note,

`@groupid`(*arg1[, arg2, arg3, arg4, …, argn, smpl]]*)

returns an integer indexing the group ID for each observation of the series or alpha expression *arg*.

Special Functions

EViews provides a number of 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
`@beta(a,b)`	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 $a,\ b > 0$.
`@betainc(x,a,b)`	incomplete beta integral: $$\frac{1}{B(a,\ b)}\int_0^x t^{a-1}(1-t)^{b-1}dt$$ for $0 \le x \le 1$ and $a,\ b > 0$.
`@betaincder(x,a,b,s)`	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: $$\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix}\ =\ \begin{bmatrix}\frac{\partial B}{\partial x} & \frac{\partial B}{\partial a} & \frac{\partial B}{\partial b}\\ \frac{\partial^2 B}{\partial x^2} & \frac{\partial^2 B}{\partial x \partial a} & \frac{\partial^2 B}{\partial x \partial b}\\ \frac{\partial^2 B}{\partial a^2} & \frac{\partial^2 B}{\partial a \partial b} & \frac{\partial^2 B}{\partial b^2}\end{bmatrix}$$
`@betaincinv(p,a,b)`	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 $0 \le p \le 1$ and $a,\ b > 0$.
`@betalog(a,b)`	natural logarithm of the beta integral: $$\log B(a,\ b)\ =\ \log\Gamma(a) + \log\Gamma(b) - \log\Gamma(a+b).$$

`@binom(n,x)`	binomial coefficient: $$\binom{n}{x} = \frac{n!}{x!(n-x)!}$$ for n and x positive integers, $0 \le x \le n$.
`@binomlog(n,x)`	natural logarithm of the binomial coefficient: $$\log(n!) - \log(x!) - \log((n-x)!)$$
`@cloglog(x)`	complementary log-log function: $$\log(-\log(1-x))$$ See also `@qextreme`.
`@digamma(x), @psi(x)`	first derivative of the log gamma function: $$\psi(x) = \frac{d\log\Gamma(x)}{dx} = \frac{1}{\Gamma(x)}\frac{d\Gamma(x)}{dx}$$
`@erf(x)`	error function: $$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$$ for $x \ge 0$.
`@erfc(x)`	complementary error function: $$\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}}\int_x^\infty e^{-t^2}\,dt = 1 - \mathrm{erf}(x).$$ for $x \ge 0$.
`@gamma(x)`	(complete) gamma function: $$\Gamma(x) = \int_0^\infty e^{-t}t^{x-1}\,dt$$ for $x \ge 0$.
`@gammader(x)`	first derivative of the gamma function: $$\Gamma'(x) = d\Gamma(x)/(dx)$$ Note: Euler's constant, $\gamma \approx 0.5772$, may be evaluated as $\gamma = -@\mathrm{gammader}(1)$. See also `@digamma` and `@trigamma`.
`@gammainc(x,a)`	incomplete gamma function: $$G(x,\,a) = \frac{1}{\Gamma(a)}\int_0^x e^{-t}t^{a-1}\,dt$$ for $x \ge 0$ and $a > 0$.

`@gammaincder(x,a,n)`	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 $0 \le p < 1$ 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)	$@\mathrm{acos}(-1) = \pi$
`@asin(x)`	arc sine (real results in radians)	$@\mathrm{asin}(-1) = \pi/2$
`@atan(x)`	arc tangent (results in radians)	$@\mathrm{atan}(1) = \pi/4$
`@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: $F_X(k) = \Pr(X \le k)$. 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 $F_X(q^*) \ge p$. 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.

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 $0 \le p \le 1$ 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 $x = 0, 1, \ldots, n, \ldots$, and 0 otherwise, for $0 \le p \le 1$.

Chi-square	`@cchisq(x,v)`, `@dchisq(x,v)`, `@qchisq(p,v)`, `@rchisq(v)`	$f(x,\ v)\ =\ \frac{1}{2^{v/2}\Gamma(v/2)}x^{v/2-1}e^{-x/2}$ where $x \geq 0$, and $v > 0$. Note that the degrees of freedom parameter v need not be an integer. In addition, the `@chisq(x,v)` function may be used to obtain the p-values directly.
Exponential	`@cexp(x,m)`, `@dexp(x,m)`, `@qexp(p,m)`, `@rexp(m)`	$f(x,\ m)\ =\ \frac{1}{m}e^{-x/m}$ for $x \geq 0$, and $m > 0$.
Extreme Value (Type I-minimum)	`@cextreme(x)`, `@dextreme(x)`, `@qextreme(p)`, `@cloglog(p)`, `@rextreme`	$f(x)\ =\ \exp(x - e^{x})$ for $-\infty < x < \infty$.
F-distribution	`@cfdist(x,v1,v2)`, `@dfdist(x,v1,v2)`, `@qfdist(p,v1,v2)`, `@rfdist(v1,v1)`	$f(x,\ v_1,\ v_2)\ =\ \frac{v_1^{v_1/2}\,v_2^{v_2/2}}{B(v_1/2,\ v_2/2)}$ $x^{(v_1-2)/2}(v_2 + v_1 x)^{-(v_1+v_2)/2}$ 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	`@cgamma(x,b,r)`, `@dgamma(x,b,r)`, `@qgamma(p,b,r)`, `@rgamma(b,r)`	$f(x,\ b,\ r)\ =\ b^{-r}x^{r-1}e^{-x/b}/\Gamma(r)$ where $x \geq 0$, and $b,\ r > 0$.
Generalized Error	`@cged(x,r)`, `@dged(x,r)`, `@qged(p,r)`, `@rged(r)`	$f(x,\ r)\ =\ \frac{r\Gamma\left(\frac{3}{r}\right)^{1/2}}{2\Gamma\left(\frac{1}{r}\right)^{3/2}}\exp\left(-\lvert x\rvert^{r}\left(\frac{\Gamma\left(\frac{3}{r}\right)}{\Gamma\left(\frac{1}{r}\right)}\right)^{r/2}\right)$ where $-\infty < x < \infty$, and $r > 0$.
Laplace	`@claplace(x)`, `@dlaplace(x)`, `@qlaplace(x)`, `@rlaplace`	$f(x)\ =\ \frac{1}{2}e^{-\lvert x\rvert}$ for $-\infty < x < \infty$.
Logistic	`@clogistic(x)`, `@dlogistic(x)`, `@qlogistic(p)`, `@rlogistic`	$f(x)\ =\ \frac{1}{1+e^{-x}}\ =\ \frac{e^{x}}{1+e^{x}}$ for $-\infty < x < \infty$.

Log-normal	`@clognorm(x,m,s)`, `@dlognorm(x,m,s)`, `@qlognorm(p,m,s)`, `@rlognorm(m,s)`	$f(x, m, s) = \frac{1}{x\sqrt{2\pi s^2}} e^{-(\log x - m)^2/(2s^2)}$ $x > 0$, $0 < m < \infty$, and $s > 0$.
Negative Binomial	`@cnegbin(x,n,p)`, `@dnegbin(x,n,p)`, `@qnegbin(s,n,p)`, `@rnegbin(n,p)`	$\Pr(x, n, p) = \frac{\Gamma(x+n)}{\Gamma(x+1)\Gamma(n)} p^n (1-p)^x$ if $x = 0, 1, \ldots, n, \ldots$, and 0 otherwise, for $0 \le x \le 1$.
Normal (Gaussian)	`@cnorm(x)`, `@dnorm(x)`, `@qnorm(p)`, `@rnorm, nrnd`	$f(x) = (2\pi)^{-1/2} e^{-x^2/2}$ for $-\infty < x < \infty$.
Poisson	`@cpoisson(x,m)`, `@dpoisson(x,m)`, `@qpoisson(p,m)`, `@rpoisson(m)`	$\Pr(x, m) = m^x e^{-m}/x!$ if $x = 0, 1, \ldots, n, \ldots$, and 0 otherwise, for $m > 0$.
Pareto	`@cpareto(x,k,a)`, `@dpareto(x,k,a)`, `@qpareto(p,k,a)`, `@rpareto(k,a)`	$f(x, k, a) = (ak^a)/x^{a+1}$ for location parameter $0 \le k \le x$ and shape parameter $a > 0$.
Student's t-distribution	`@ctdist(x,v)`, `@dtdist(x,v)`, `@qtdist(p,v)`, `@rtdist(v)`	$f(x, v) = \frac{\Gamma\left(\frac{v+1}{2}\right)}{(v\pi)^{\frac{1}{2}}\Gamma\left(\frac{v}{2}\right)} \left(1 + \left(\frac{x^2}{v}\right)\right)^{\frac{-(v+1)}{2}}$ for $-\infty < x < \infty$, and $v > 0$. Note that $v = 1$, yields the Cauchy distribution.
Uniform	`@cunif(x,a,b)`, `@dunif(x,a,b)`, `@qunif(p,a,b)`, `@runif(a,b), rnd`	$f(x) = \frac{1}{b-a}$ for $a < x < b$ and $b > a$.
Weibull	`@cweib(x,m,a)`, `@dweib(x,m,a)`, `@qweib(p,m,a)`, `@rweib(m,a)`	$f(x, m, a) = am^{-a} x^{a-1} e^{-(x/m)^a}$ where $0 < x < \infty$, 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
`@chisq(x,v)`	Chi-square	Returns the probability that a Chi-squared statistic with v degrees of freedom exceeds x: @chisq(x,v) = 1–@cchisq(x,d)
`@fdist(x,v1,v2)`	F-distribution	Probability that an F-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)
`@tdist(x,v)`	t-distribution	Probability that a t-statistic with v degrees of freedom exceeds x in absolute value (two-sided p-value): @tdist(x,v) = 2*(1–@ctdist(@abs(x),v))

String Functions

EViews offers an extensive library of functions for working with strings. For a complete list of relevant functions, see "String Function Summary," on page 567. For discussion, see "Strings," on page 65.

Date Functions

EViews provides a powerful collection of functions for working with dates. For a list of functions see "Date Function Summary," on page 568. For discussion, see "Dates," on page 82.

Indicator Functions

These functions produce indicators for whether each observation satisfies a specific condition:

Function	Description
`@inlist(`*series*`, "`*list*`")`	Creates a dummy variable equal to 1 for observations where *series* is equal to one of the values specified in *list*, and 0 otherwise. *list* should be a quoted, space delimited list of values. This function works on both numerical and alpha series.
`@between(`*series*`, `*val1*`, `*val2*`)`	Creates a dummy variable equal to 1 for observations where *series* is greater than or equal to *val1* and less than or equal to *val2*.

Workfile & Informational Functions

The following list summarizes the utility functions that are designed to provide information about the currently active workfile and the EViews environment. These functions are described in greater detail in Chapter 15. "Workfile Functions," on page 549.

General Information

Function	Name	Description
`@env(str)`	environment variables	returns a string containing the value of the Windows environment variable *str*. For example @env("username") returns the user-name of the current logged in user. @env("computername") returns the name of the computer.
`@fileexist(str)`	file exist	returns a 0 or 1 depending on whether the filename specified by *str* exists on disk.
`@folderexist(str)`	folder exist	returns a binary scalar, equal to 1 if the folder specified by *str* exists on disk, and 0 otherwise.
`@tablenames(str)`	table/sheet names	returns a space delimited string containing the names of the table names of a foreign file. *str* should be the name (and path) of the foreign file, enclosed in quotes. For an Excel file, the names of the Excel sheets will be returned.
`@getnextname(str)`	get next name	returns a string containing the next available variable name in the workfile, starting with *str*. *e.g.* entering "result" will return "result01" unless there is already a RESULT01, in which case it will return "result02".
`@vernum`	version number	returns a scalar containing the EViews version number.
`@verstr`	version string	returns a string containing the EViews version string (*e.g.* "EViews Enterprise Edition").

`@wdir(`*str*`)`	directory listing	returns a string list containing a list of all of the files in the directory *str*.
`@wquery(`*"database"*, *"search_expression"*, *"attribute_list"*`)`	database query	returns a string list containing the *attribute_list* of all objects in *database* that match the query given by *search_expression* (see `@wquery` (p. 680) for details).

Element Information

Function	Name	Description
`@elem(x,"v")`	element	returns the value of a series at an observation or date. The observation or date should be entered in double quotes.
`@first(x)`	first element	returns the value of the first non-missing value in the series for the current sample.
`@ifirst(x)`	index of first element	returns the workfile index of the first non-missing value in the series for the current sample.
`@ilast(x)`	last element	returns the workfile index of the last non-missing value in the series for the current sample.
`@last(x)`	index of last element	returns the value of the last non-missing value in the series for the current sample.

Basic Workfile Functions

Function	Name	Description
`@ispanel`	panel workfile	returns a 0 or 1 depending on whether the workfile is panel structured.
`@obsrange`	observations in range	returns the number of observations in the workfile range.
`@obssmpl`	observations in sample	returns the number of observations in the workfile sample.
`@pagecount`	page count	returns a scalar containing the number of pages in the current workfile.

`@pageexist(`*str*`)`	page exists	returns a 0 or 1 depending on whether the page specified by *str* exists in the current workfile.
`@pagefreq`	page frequency	returns a string containing the frequency of the active page.
`@pageids`	workfile page identifiers	returns a string containing a space delimited list of the id series for the current workfile page.
`@pagelist`	page names list	returns a space delimited string containing the names of all the pages in the current active workfile.
`@pagename`	page name	returns a string containing the name of the active page.
`@pagesmpl`	workfile page sample	returns a string containing the current sample for the active page.
`@wfname`	workfile name	returns a string containing the current default workfile name.
`@wfpath`	workfile path	returns a string containing the current default workfile path.

Dated Workfile Information

The following functions return information about each observation in the workfile. They are used to generate new series.

Function	**Name**	**Description**
`@after(`*arg1*`)`	date is after	creates a dummy variable equal to 1 if the observation is after or on the date given by *arg1*. *arg1* should be enclosed in quotes.
`@before(`*arg1*`)`	date is before	creates a dummy variable equal to 1 if the observation is before the date given by *arg1*. *arg1* should be enclosed in quotes.
`@date`	date	returns the date value.
`@day`	day of month	returns the day of the month.
`@during(`*arg1*	date is during	creates a dummy variable equal to 1 if the observation lies between the dates given by the date pair contained in *arg1*, and 0 otherwise. *arg1* should be given in quotes.

@enddate	observations in range	returns the end date of the period of time associated with the observation in the workfile.
@event(*arg1*[, *basis*])	event proportion	proportion of a one-off event that lies in each observation.
@holiday(*arg1*[, *basis*])	holiday proportion	proportion of an annual event that lies in each observation.
@hour	hour	returns the hour of each observation as an integer.
@hourf	hour floating point	returns the hour of each observation as a floating point number (*e.g.*, 9:30 a.m. returns 9.5, and 5:15 p.m. returns 17.25).
@isperiod(x)	period dummy	returns a dummy variable for whether the observation is in the specified period.
@minute	minutes	returns the minute of each observation as an integer.
@month	month	returns the month of an observation.
@quarter	quarter	returns the quarter of an observation.
@seas(x)	seasonal dummy	returns a seasonal dummy variable.
@second	seconds	returns the second of each observation as an integer.
@strdate(x)	string dates	returns a string representing the date for every observation.
@trend([*x*])	time trend	returns a time trend using the workfile calendar.
@trendc([*x*])	calendar time trend	returns a time trend where the trend uses calendar time.
@weekday	day of the week	returns the day of the week.
@year	year	returns the year

Panel Workfile Functions

Function	Name	Description
@cellid	element	returns the inner dimension identifier index.
@crossid	observations in range	returns the cross-section identifier index.

`@obsid`	cross-section observation number	returns the observation number within each panel cross section.
`@trend`	time trend	returns a time trend for each cross-section using the observed dates in the panel.
`@trendc`	calendar time trend	returns a time trend for each cross-section where the trend uses calendar time.

Valmap Functions

The following utility functions were designed to facilitate working with value mapped series. Additional detail is provided in "Valmap Functions" on page 214 of the *User's Guide I*

Function	Name	Description
`@map(`*x[, mapname]*`)`	mapped value	returns the mapped values of a series.
`@unmap(`*x, mapname)*	unmapped numeric value	returns the numeric values associated with a set of string value labels.
`@unmaptxt(`*x, mapname*`)`	unmapped text value	returns the string values associated with a set of string value labels.

References

Abramowitz, M. and I. A. Stegun (1964). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, New York: Dover Publications.

Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (1992). *Numerical Recipes in C, 2nd edition*, Cambridge University Press.

Temme, Nico M. (1996). *Special Functions: An Introduction to the Classical Functions of Mathematical Physics*, New York: John Wiley & Sons.

Chapter 14. Operator and Function Listing

The following reference is an alphabetical listing of operators and functions which may be used in series assignment and generation, and in many cases, in matrix operations or element evaluation. Additional details on these operators and functions is provided in Chapter 13. "Operator and Function Reference," on page 503.

Operator / Function	Description
+	add
–	subtract
*	multiply
/	divide
^	raise to the power
<	less than
< =	less than or equal to
< >	not equal to
=	equal to
>	greater than
> =	greater than or equal to
@abs(*x*), abs(*x*)	absolute value
@acos(*x*)	arc cosine (real results in radians)
@after(arg1)	Creates a dummy variable equal to 1 if the observation is after or on the date given by *arg1*. *arg1* should be enclosed in quotes.
and	logical and
@asin(x)	arc sine (real results in radians)
@atan(x)	arc tangent (results in radians)
@before(arg1)	Creates a dummy variable equal to 1 if the observation is before the date given by *arg1*. *arg1* should be enclosed in quotes.
@beta(a,b)	beta integral
@betainc(x,a,b)	incomplete beta integral
@betaincder(x,a,b,s)	derivative of the incomplete beta integral
@betaincinv(p,a,b)	inverse of the incomplete beta integral
@betalog(a,b)	natural logarithm of the beta integral

Operator / Function	Description
@between(x, val1, val2)	Creates a dummy variable equal to 1 for observations where the series is greater than or equal to *val1* and less than or equal to *val2*.
@binom(n,x)	binomial coefficient
@binomlog(n,x)	natural logarithm of the binomial coefficient
@cbeta(x,a,b)	beta cumulative distribution (CDF)
@cbinom(x,n,p)	binomial cumulative distribution (CDF)
@cchisq(x,v)	chi-square cumulative distribution (CDF)
@ceiling(x)	smallest integer not less than
@cellid	element
@cexp(x,m)	exponential cumulative distribution (CDF)
@cextreme(x)	extreme value cumulative distribution (CDF)
@cfdist(x,v1,v2)	F-distribution cumulative distribution (CDF)
@cgamma(x,b,r)	gamma cumulative distribution (CDF)
@cged(x,r)	generalized error cumulative distribution (CDF)
@chisq(x,v)	chi-square p-value
@claplace(x)	Laplace cumulative distribution (CDF)
@clogistic(x)	logistic cumulative distribution (CDF)
@cloglog(x)	complementary log-log function
@clognorm(x,m,s)	log normal cumulative distribution (CDF)
@cnegbin(x,n,p)	negative binomial cumulative distribution (CDF)
@cnorm(x)	normal cumulative distribution (CDF)
@columns(g)	number of columns.
@cor(x,y[,s])	correlation
@cos(x)	cosine (argument in radians)
@cov(x,y[,s])	population covariance
@covp(x,y[,s])	population covariance
@covs(x,y[,s])	sample covariance
@cpareto(x,a,k)	Pareto cumulative distribution (CDF)
@cpoisson(x,m)	Poisson cumulative distribution (CDF)
@crossid	observations in range
@ctdist(x,v)	Student's t-distribution cumulative distribution (CDF)
@cumbmax(x[,s])	backwards cumulative maximum
@cumbmean(x[,s])	backwards cumulative mean

Operator / Function	Description
@cumbmin(x[,s])	backwards cumulative minimum
@cumbnas(x[,s])	backwards cumulative nmber of NA observations
@cumbobs(x[,s])	backwards cumulative nmber of non-NA observations
@cumbprod(x[,s])	backwards cumulative product
@cumbstdev(x[,s])	backwards cumulative sample standard deviation
@cumbstdevp(x[,s])	backwards cumulative population standard deviation
@cumbstdevs(x[,s])	backwards cumulative sample standard deviation
@cumbsum(x[,s])	backwards cumulative sum
@cumbsumsq(x[,s])	backwards cumulative sum-of-squares
@cumbvar(x[,s])	backwards cumulative population variance
@cumbvarp(x[,s])	backwards cumulative population variance
@cumbvars(x[,s])	backwards cumulative sample variance
@cummax(x[,s])	cumulative maximum
@cummean(x[,s])	cumulative mean
@cummin(x[,s])	cumulative minimum
@cumnas(x[,s])	cumulative nmber of NA observations
@cumobs(x[,s])	cumulative nmber of non-NA observations
@cumprod(x[,s])	cumulative product
@cumstdev(x[,s])	cumulative sample standard deviation
@cumstdevp(x[,s])	cumulative population standard deviation
@cumstdevs(x[,s])	cumulative sample standard deviation
@cumsum(x[,s])	cumulative sum
@cumsumsq(x[,s])	cumulative sum-of-squares
@cumvar(x[,s])	cumulative population variance
@cumvarp(x[,s])	cumulative population variance
@cumvars(x[,s])	cumulative sample variance
@cunif(x,a,b)	uniform cumulative distribution (CDF)
@cweib(x,m,a)	Weibull cumulative distribution (CDF)
d(x)	first difference
d(x,n)	n-th order difference
d(x,n,s)	n-th order difference with a seasonal difference at s
@date	element
@dateadd(d, offset[,u])	calculates date number offsets
@datediff(d1,d2[,u])	difference between two dates

Operator / Function	Description
@datefloor(d1, u[, step])	calculates a date floor
@datepart(d1, u)	calculates the date part of a number
@datestr(d[, fmt])	converts date to string
@dateval(str1[, fmt])	converts string to date
@day	day of month
@dbeta(x,a,b)	beta density function
@dbinom(x,n,p)	binomial density function
@dchisq(x,v)	chi-square density function
@dexp(x,m)	exponential density function
@dextreme(x)	extreme value density function
@dfdist(x,v1,v2)	F-distribution density function
@dgamma(x,b,r)	gamma density function
@dged(x,r)	generalized error density function
@digamma(x), @psi(x)	first derivative of the log gamma function
@dlaplace(xb)	Laplace density function
dlog(x)	first difference of the logarithm
dlog(x,n)	n-th order difference of the logarithm
dlog(x,n,s)	n-th order difference of the logarithm with a seasonal difference at s
@dlogistic(x)	logistic density function
@dlognorm(x,m,s)	log normal density function
@dnegbin(x,n,p)	negative binomial density function
@dnorm(x)	normal density function
@dpareto(x,a,k)	Pareto density function
@dpoisson(x,m)	Poisson density function
@dtdist(x,v)	Student's t-distribution density function
@dtoo(str)	converts string to observation number
@dunif(x,a,b)	uniform density function
@during(arg1)	Creates a dummy variable equal to 1 if the observation lies between the dates given by the date pair contained in *arg1*, and 0 otherwise. *arg1* should be given in quotes.
@dweib(x,m,a)	Weibull density function
@elem(x,"v")	element
@enddate	observations in range

Operator / Function	Description
@env(str)	returns a string containing the value of the Windows environment variable *str*.
@eqna(str1, str2)	test for equality of strings
@eqna(x,y)	equal to
@erf(x)	error function
@erfc(x)	complementary error function
@event(arg1[, basis])	Proportion of a one-off event that lies in each observation.
@exp(x), exp(x)	exponential, e^x
@fact(x)	factorial, $x!$
@factlog(x)	natural logarithm of the factorial, $\log_e(x!)$
@fdist(x,v1,v2)	F-distribution p-value
@fileexist(str)	returns a 0 or 1 depending on whether the filename specified by *str* exists on disk.
@first(x)	returns the value of the first non-missing value in the series for the current sample.
@firstsby(arg1, arg2[, s])	First non-missing value in *arg1* for each *arg2* group.
@floor(x)	largest integer not greater than.
@folderexist(str)	returns a binary scalar, equal to 1 if the folder specified by *str* exists on disk, and 0 otherwise.
@fv(r,n,x*[*,fv,t*]*)	future value.
@gamma(x)	(complete) gamma function
@gammader(x)	first derivative of the gamma function
@gammainc(x,a)	incomplete gamma function
@gammaincder(x,a,n)	derivative of the incomplete gamma function
@gammaincinv(p,a)	inverse of the incomplete gamma function
@gammalog(x)	logarithm of the gamma function
@getnextname(str)	returns a string containing the next available variable name in the workfile, starting with *str*. *e.g.* entering "result" will return "result01" unless there is already a RESULT01, in which case it will return "result02".
@gmean(x[,s])	geometric mean
@groupid(arg1[, arg2, ..., argn, s])	group index for the groups defined by *arg1* to *argn*.
@hmean(x[.s])	harmonic mean
@holiday(arg1[, basis])	Proportion of an annual event that lies in each observation.
@hour	returns the hour of each observation as an integer.

Operator / Function	Description
@hourf	returns the hour of each observation as a floating point number.
@iff(s,x,y)	recode by condition
@ifirst(x)	returns the workfile index of the first non-missing value in the series for the current sample.
@ilast(x)	returns the workfile index of the last non-missing value in the series for the current sample.
@imax(x)	returns the workfile index of the maximum value in the series for the current sample.
@imin(x)	returns the workfile index of the minimum value in the series for the current sample.
@inlist(x, "list")	Creates a dummy variable equal to 1 for observations where the series is equal to one of the values specified in *list*, and 0 otherwise. *list* should be a quoted, space delimited list of values. This function works on both numerical and alpha series.
@inner(x,y[,s])	inner product
@insert(str1, str2, n])	string insertion
@instr(str1, str2[, n])	string location
@inv(x)	reciprocal, $1 / x$
@isempty(str)	tests for blank strings
@isna(x)	equal to NA
@ispanel	returns indicator for whether the workfile is panel structured.
@isperiod(x)	period dummy
@kurt(x[,s])	kurtosis
@kurtsby(arg1, arg2[,arg3, arg4..., argn, s])	kurtosis of *arg1* observations for each *arg2 to argn* defined group.
@last(x)	returns the value of the last non-missing value in the series for the current sample.
@lastsby(arg1, arg2[, s])	Last non-missing value in *arg1* for each *arg2* group.
@left(str, n)	returns the left part of a string
@len(str)	length of a string
@length(str)	length of a string
@log(x), log(x)	natural logarithm, $\log_e(x)$
@log10(x)	base-10 logarithm, $\log_{10}(x)$
@logit(x)	logistic transform
@logx(x,b)	base-*b* logarithm, $\log_b(x)$

Operator / Function	Description
@lower(str)	lowercases a string
@ltrim(str)	removes spaces from a string
@makedate(arg1[,arg2[,arg3]], fmt])	converts number to date
@map(x[, mapname])	mapped value
@mav(x,n)	n-period backward moving average. NAs are not propagated.
@mavc(x,n)	n-period centered moving average. NAs are not propagated.
@max(x[,s])	maximum
@maxsby((arg1, arg2[, arg3, arg4..., argn, s)	maximum value of *arg1* observations for each *arg2 to argn* group.
@mcor(x,y,n)	n-period backwards moving correlation. NAs are not propagated.
@mcov(x,y,n)	n-period backwards population moving covariance. NAs are not propagated.
@mcovp(x,y,n)	n-period backwards moving population covariance. NAs are not propagated.
@mcovs(x,y,n)	n-period backwards moving sample covariance. NAs are not propagated.
@mean(x[,s])	mean
@meansby(arg1, arg2[,arg3, arg4..., argn, s])	mean of *arg1* observations for each *arg2 to argn* defined group.
@median(x[,s])	median
@mediansby(arg1, arg2[, arg3, arg4..., argn, s])	median of *arg1* observations for each *arg2 to argn* defined group
@mid(str, n1[, n2])	returns the middle part of a string
@min(x[,s])	minimum
@minner(x,y,n)	n-period backwards inner product of X and Y. NAs are not propagated.
@minsby(arg1, arg2[,arg3, arg4..., argn, s])	minimum value of *arg1* observations for each *arg2 to argn* defined group.
@minute	returns the minute of each observation as an integer.
@mkurt(x,n)	n-period backwards kurtosis. NAs are not propagated.
@mmax(x,n)	n-period backwards moving maximum. NAs are not propagated.
@mmin(x,n)	n-period backwards moving minimum. NAs are not propagated.

Operator / Function	Description
@mnas(x,n)	*n*-period backwards number of NA observations. NAs are not propagated.
@mobs(x,n)	*n*-period backwards number of non-NA observations. NAs are not propagated.
@mod(x,y)	floating point remainder
@month	month
@movav(x,n)	*n*-period backward moving average. NAs are propagated.
@movavc(x,n)	*n*-period centered moving average. NAs are propagated.
@movcor(x,y,n)	*n*-period backwards moving correlation. NAs are propagated.
@movcov(x,y,n)	*n*-period backwards moving population covariance. NAs are propagated.
@movcovp(x,y,n)	*n*-period backwards moving population covariance. NAs are propagated.
@movcovs(x,y,n)	*n*-period backwards moving sample covariance. NAs are propagated.
@movinner(x,y,n)	*n*-period backwards inner product of X and Y. NAs are propagated.
@movkurt(x,n)	*n*-period backwards kurtosis. NAs are propagated.
@movmax(x,n)	*n*-period backwards moving maximum. NAs are propagated.
@movmin(x,n)	*n*-period backwards moving minimum. NAs are propagated.
@movnas(x,n)	*n*-period backwards nmber of NA observations
@movobs(x,n)	*n*-period backwards nmber of non-NA observations
@movskew(x,n)	*n*-period backwards skewness. NAs are propagated.
@movstdev(x,n)	*n*-period backwards moving sample standard deviation. NAs are propagated.
@movstdevp(x,n)	*n*-period backwards moving population standard deviation. NAs are propagated.
@movstdevs(x,n)	*n*-period backwards moving sample standard deviation. NAs are propagated.
@movsum(x,n)	*n*-period backward moving sum. NAs are propagated.
@movsumsq(x,n)	*n*-period backwards sum-of-squares. NAs are propagated.
@movvar(x,n)	*n*-period backwards moving population variance. NAs are propagated.
@movvarp(x,n)	*n*-period backwards moving population variance. NAs are propagated.
@movvars(x,n)	*n*-period backwards moving sample variance. NAs are propagated.

Operator / Function	Description
@mskew(x,n)	*n*-period backwards skewness. NAs are not propagated.
@mstdev(x,n)	*n*-period backwards moving sample standard deviation. NAs are not propagated.
@mstdevp(x,n)	*n*-period backwards moving population standard deviation. NAs are not propagated.
@mstdevs(x,n)	*n*-period backwards moving sample standard deviation. NAs are not propagated.
@msum(x,n)	*n*-period backward moving sum. NAs are not propagated.
@msumsq(x,n)	*n*-period backwards sum-of-squares. NAs are not propagated.
@mvar(x,n)	*n*-period backwards moving population variance. NAs are not propagated.
@mvarp(x,n)	*n*-period backwards moving population variance. NAs are not propagated.
@mvars(x,n)	*n*-period backwards moving sample variance. NAs are not propagated.
@nan(x,y)	recode NAs in X to Y
@nas(x[,s])	number of NAs
@nasby(arg1, arg2[,arg3, arg4..., argn, s])	number of *arg1* NA values for each *arg2 to argn* defined group.
@neqna(str1, str2)	tests for inequality of strings
@neqna(x,y)	not equal to
@now	returns current time
@nper(r,x,pv*[*,fv,t*]*)	annuity number of periods
nrnd	normal random number generator
@obs(x[,s])	number of observations
@obsby(arg1, arg2[,arg3, arg4..., argn, s])	number of non-NA *arg1* observations for each *arg2 to argn* defined group.
@obsid	cross-section observation number
@obsrange	observations in range
@obssmpl	observations in sample
or	logical or
@otod(n)	converts observation number to string
@otods(n)	converts sample observation number to string
@pagecount	returns a scalar containing the number of pages in the current workfile.

Operator / Function	Description
@pageexist(str)	returns a 0 or 1 depending on whether the page specified by *str* exists in the current workfile.
@pagefreq	returns a string containing the current page's frequency.
@pageids	returns a string containing a space delimited list of the id series for the current workfile page.
@pagelist	returns a space delimited string containing the names of all the pages in the current active workfile.
@pagename	returns a string containing the current default page name.
@pagesmpl	returns a string containing the current sample for the active page.
@pc(x)	one-period percentage change (in percent)
@pca(x)	one-period percentage change—annualized (in percent)
@pch(x)	one-period percentage change (in decimal)
@pcha(x)	one-period percentage change—annualized (in decimal)
@pchy(x)	one-year percentage change (in decimal)
@pcy(x)	one-year percentage change (in percent)
@pmt(r,n,pv*[*,fv,t*]*)	payment amount
@prod(x[,s])	product
@psi(x)	see @digamma
@pv(r,n,x*[*,fv,t*]*)	present value
@qbeta(s,a,b)	beta quantile function (inverse CDF)
@qbinom(s,n,p)	binomial quantile function (inverse CDF)
@qchisq(p,v)	chi-square quantile function (inverse CDF)
@qexp(p,m)	exponential quantile function (inverse CDF)
@qextreme(p)	extreme value quantile function (inverse CDF)
@qfdist(p,v1,v2)	F-distribution quantile function (inverse CDF)
@qgamma(p,b,r)	gamma quantile function (inverse CDF)
@qged(p,r)	generalized error quantile function (inverse CDF)
@qlaplace(x)	Laplace quantile function (inverse CDF)
@qlogistic(p)	logistic quantile function (inverse CDF)
@qlognorm(p,m,s)	log normal quantile function (inverse CDF)
@qnegbin(s,n,p)	negative binomial quantile function (inverse CDF)
@qnorm(p)	normal quantile function (inverse CDF)
@qpareto(p,a,k)	Pareto quantile function (inverse CDF)
@qpoisson(p,m)	Poisson quantile function (inverse CDF)

Operator / Function	Description
@qtdist(p,v)	Student's *t*-distribution quantile function (inverse CDF)
@quantile(x,q[,m,s])	quantile
@quantilesby(arg1, arg2[,arg3, arg4..., argn], q[, s])	q-quantiles of *arg1* observations for each *arg2 to argn* defined group.
@quarter	quarter of the year in a dated workfile
@qunif(p,a,b)	uniform quantile function (inverse CDF)
@qweib(p,m,a)	Weibull quantile function (inverse CDF)
@ranks(x[,o,t,s])	rank
@rate(n,x,pv*[*,fv,t*]*)	interest
@rbeta(a,b)	beta random number generator
@rbinom(a,b)	binomial random number generator
@rchisq(v)	chi-square random number generator
@recode(s,x,y)	recode by condition
@replace(str1, str2, str3[,n])	replaces a string with another
@rexp(m)	exponential random number generator
@rextreme(p)	extreme value random number generator
@rfdist(v1,v2)	F-distribution random number generator
@rfirst(g)	row-wise first non-NA value.
@rfirsti(g)	row-wise first non-NA index.
@rgamma(b,r)	gamma random number generator
@rged(r)	generalised error random number generator
@right(str,n)	returns the right parts of a string
@rlaplace	Laplace random number generator
@rlast(g)	row-wise last non-NA value.
@rlasti(g)	row-wise last non-NA index.
@rlogistic	logistic random number generator
@rlognorm(m,s)	log normal random number generator
@rmax(g)	row-wise maximum value.
@rmaxi(g)	row-wise maximum index.
@rmean(g)	row-wise mean.
@rmin(g)	row-wise minimum value.
@rmini(g)	row-wise minimum index.
@rnas(g)	row-wise number of NAs.
@rnegbin(n,p)	negative binomial random number generator

Operator / Function	Description
@rnorm	normal random number generator
@robs(g)	row-wise number of non-NAs.
@round(x)	round to the nearest integer
@rpareto(a,k)	Pareto random number generator
@rpoisson(m)	Poisson random number generator
@rstdev(g)	row-wise sample standard deviation.
@rstdevp(g)	row-wise population standard deviation.
@rstdevs(g)	row-wise sample standard deviation.
@rsum(g)	row-wise sum.
@rsumsq(g)	row-wise sum-of-squares.
@rtdist(v)	Student's t-distribution random number generator
@rtrim(str)	removes spaces from a string
@runif(a,b)	uniform random number generator
@rvalcount(g,v)	row-wise count of observations matching v.
@rvar(g)	row-wise population variance.
@rvarp(g)	row-wise population variance
@rvars(g)	row-wise sample variance.
@rweib(m,a)	Weibull random number generator
@seas(x)	seasonal dummy
@second	returns the second of each observation as an integer.
@sin(x)	sine (argument in radians)
@skew(x[,s])	skewness
@skewsby(arg1, arg2[,arg3, arg4..., argn, s)	skewness of *arg1* observations for each *arg2 to argn* defined group.
@sqrt(x), sqr(x)	square root
@stdev(x[,s])	standard deviation
@stdevp(x[,s])	population standard deviation
@stdevs(x[,s])	sample standard deviation
@stdevpsby(arg1, arg2[,arg3, arg4..., argn, s])	standard deviation of *arg1* observations for each *arg2 to argn* defined group (division by n).
@stdevsby(arg1, arg2[,arg3, arg4..., argn, s])	sample standard deviation of *arg1* observations for each *arg2 to argn* defined group (division by $n-1$).
@stdevssby((arg1, arg2[,arg3, arg4..., argn, s])	sample standard deviation of *arg1* observations for each *arg2 to argn* defined group (division by $n-1$).
@str(d,[fmt])	converts a number to a string

Operator / Function	Description
@strdate(fmt)	string corresponding to each element in workfile
@strdate(x)	string dates
@strlen(str)	length of a string
@strnow(fmt)	returns current time as a string
@sum(x[,s])	sum
@sumsby(arg1, arg2[, s])	sum of *arg1* observations for each *arg2 to argn* defined group.
@sumsq(x[,s])	sum-of-squares
@sumsqsby(arg1, arg2[, s])	sum of squares of *arg1* observations for each *arg2 to argn* defined group.
@tablenames(str)	returns a space delimited string containing the names of the table names of a foreign file.
@tan(x)	tangent (argument in radians)
@tdist(x,v)	t-distribution p-value
@trend	time trend
@trend([x])	time trend
@trendc	calendar time trend
@trendc([x])	calendar time trend
@trigamma(x)	second derivative of the log gamma function
@trim(str)	removes spaces from a string
@unmap(x, mapname)	unmapped numeric value
@unmaptxt(x, mapname)	unmapped text value
@upper(str)	uppercases a string
@val(str[, fmt])	converts a string to a number
@var(x[,s])	population variance
@varp(x[,s])	population variance
@varpsby(arg1, arg2[,arg3, arg4..., argn, s])	variance of *arg1* observations for each *arg2 to argn* defined group (division by n).
@vars(x[,s])	sample variance
@varsby(arg1, arg2[,arg3, arg4..., argn, s])	variance of *arg1* observations for each *arg2 to argn* defined group (division by n).
@varssby(arg1, arg2[,arg3, arg4..., argn, s])	sample variance of *arg1* observations for each *arg2 to argn* defined group (division by $n-1$).
@vernum	returns a scalar containing the EViews version number.
@verstr	returns a string containing the EViews version string (EViews Enterprise Edition).

Operator / Function	Description
@wdir(str)	returns a string list containing a list of all of the files in the directory *str*.
@weekday	day of the week
@wfname	returns a string containing the current default workfile name.
@wfpath	returns a string containing the current default workfile path.
@wquery("database", "search_expression", "attribute_list")	returns a string list containing the *attribute_list* of all objects in *database* that match the query given by *search_expression* .
@year	year

Chapter 15. Workfile Functions

EViews workfile functions provide information about each observation of a workfile based on the structure of the workfile.

These functions may be viewed in two ways. First, they may be thought of as virtual series available within each workfile that can be used wherever a regular series may be used. Alternatively, they may be thought of as special functions that depend on two implicit arguments: the workfile within which the function is being used, and the observation number within this workfile for which to return a value.

Since workfile functions are based on the structure of a workfile, they may only be used in operations where a workfile is involved. Workfile functions may be used in statements that generate series in a workfile, in statements that set the workfile sample, and in expressions used in estimating an equation using data in a workfile. These functions may not be used when manipulating scalar variables or vectors and matrices in EViews programs.

Basic Workfile Information

The `@obsnum` function provides information on observation numbering for the workfile:

- `@obsnum`: returns the observation number of the current observation in the workfile.

The observation number starts at one for the first observation in the workfile, and increments by one for each subsequent observation.

For example:

```
series idnum = @obsnum
```

creates a series IDNUM that contains sequential values from one to the number of observations in the workfile.

Other functions return scalar values associated with the workfile and default workfile sample:

- `@elem`(*x*, *arg*): returns the value of the series *x* at observation or date *arg*. If the workfile is undated, *arg* should be an integer corresponding to the observation ID as given in `@obsnum`. If the workfile is dated, *arg* should be a string representation of a date in the workfile. In both cases, the argument must be enclosed in double quotes. Note that `@elem` is not available in panel structured workfiles.
- `@ispanel`: returns indicator for whether the workfile is panel structured (0 if no workfile in memory).
- `@obsrange`: returns number of observations in the current active workfile range (0 if no workfile in memory).

- `@obssmpl`: returns number of observations in the current active workfile sample (0 if no workfile in memory).
- `@pagecount`: returns the number of pages in the current active workfile (0 if no workfile in memory)
- `@pageexist`: returns a 0 or 1 depending on whether the page specified by *str* exists in the current workfile.

The following functions return a string value associated with the current workfile:

- `@pagefreq`: returns the frequency of the active page.
- `@pagename`: returns the name of the active page.
- `@pagelist`: returns a space delimited string containing the names of all the pages in the current active workfile.
- `@pagesmpl`: returns the current sample for the active page.
- `@pageids`: returns the id series for the current active page. If the page is a regular dated page, "@date" is returned. If the page is irregular, or structured by an id series, names of the id series are returned. If the page is completely unstructured, empty string is returned. In a panel it returns the cross-section identifiers followed by the date identifier.
- `@wfname`: returns the current default workfile name.
- `@wfpath`: returns the current default workfile path.

Dated Workfile Information

Basic Date Functions

EViews offers a set of functions that provide information about the dates in your dated workfiles. The first two functions return the start and end date of the period of time (interval) associated with each observation in the workfile:

- `@date`: returns the start date of the period of time of each observation of the workfile.
- `@enddate`: returns the end date of the period of time associated with each observation of the workfile.

Each date is returned in a number using standard EViews date representation (fractional days since 1st Jan A.D. 1; see "Dates," beginning on page 82).

A period is considered to end during the last millisecond contained within the period. In a regular frequency workfile, each period will end immediately before the start of the next period. In an irregular workfile there may be gaps between the end of one period and the start of the following period due to observations that were omitted in the workfile.

The `@date` and `@enddate` functions may be combined with EViews date manipulation functions to provide a wide variety of calendar information about a dated workfile.

For example, if we had a monthly workfile containing sales data for a product, we might expect the total sales that occurred in a given month to be related to the number of business days (Mondays to Fridays) that occurred within the month. We could create a new series in the workfile containing the number of business days in each month by using:

```
series busdays = @datediff(@date(+1), @date, "B")
```

If the workfile contained irregular data, we would need to use a more complicated expression since in this case we cannot assume that the start of the next period occurs immediately after the end of the current period. For a monthly irregular file, we could use:

```
series busdays = @datediff(@dateadd(@date, 1, "M"), @date, "B")
```

Similarly, when working with a workfile containing daily share price data, we might be interested in whether price volatility is different in the days surrounding a holiday for which the market is closed. We may use the first formula given above to determine the number of business days between adjacent observations in the workfile, then use this result to create two dummy variables that indicate whether each observation is before or after a holiday day.

```
series before_holiday = (busdays > 1)
series after_holiday = (busdays(-1) > 1)
```

We could then use these dummy variables as exogenous regressors in the variance equation of a GARCH estimation to estimate the impact of holidays on price volatility.

In many cases, you may wish to transform the date numbers returned by `@date` so that the information is contained in an alternate format. EViews provides workfile functions that bundle common translations of date numbers to usable information. These functions include:

- `@year`: returns the four digit year in which each observation begins. It is equivalent to "@datepart(@date, "YYYY")".
- `@quarter`: returns the quarter of the year in which each observation begins. It is equivalent to "@datepart(@date, "Q")".
- `@month`: returns the month of the year in which each observation begins. It is equivalent to "@datepart(@date, "MM")".
- `@day`: returns the day of the month in which each observation begins. It is equivalent to "@datepart(@date, "DD")".
- `@weekday`: returns the day of the week in which each observation begins, where Monday is given the number 1 and Sunday is given the number 7. It is equivalent to "@datepart(@date, "W")".

- `@hour`: returns the hour associated with each observation as an integer. For example, 9:30AM returns 9, and 5:15PM returns 17.
- `@minute`: returns the minute associated with each observation as an integer. For example, 9:30PM returns 30.
- `@second`: returns the second associated with each observation as an integer.
- `@hourf`: returns the time associated with observation as a floating point hour. For example, 9:30AM returns 9.5, and 5:15PM returns 17.25.
- `@strdate`(*fmt*): returns the set of observation dates as strings, using the date format string *fmt*. See "Date Formats" on page 85 for a discussion of date format strings.
- `@seas`(*season_number*): returns dummy variables based on the period within the current year in which each observation occurs, where the year is divided up according to the workfile frequency. For example, in a quarterly file, "@seas(1)", "@seas(2)", "@seas(3)", and "@seas(4)" correspond to the set of dummy variables for the four quarters of the year. these expressions are equivalent (in the quarterly workfile) to "@quarter = 1", "@quarter = 2", "@quarter = 3", and "@quarter = 4", respectively.
- `@isperiod`(*arg*): returns dummy variables for whether each observation is in the specified period, where *arg* is a double quoted date or period number. Note that in dated workfiles, *arg* is rounded down to the workfile frequency prior to computation.

Additional information on working with dates is provided in "Dates," beginning on page 82.

Trend Functions

One common task in time series analysis is the creation of variables that represent time trends. EViews provides two distinct functions for this purpose:

- `@trend`(*["base_date"]*): returns a time trend that increases by one for each observation of the workfile. The optional *base_date* may be provided to indicate the starting date for the trend.
- `@trendc`(*["base_date"]*): returns a calendar time trend that increases based on the number of calendar periods between successive observations. The optional *base_date* may be provided to indicate the starting date for the trend.

The functions `@trend` and `@trendc` are used to represent two different types of time trends that differ in some circumstances:

- In a regular frequency workfile, `@trend` and `@trendc` both return a simple trend that increases by one for each observation of the workfile.
- In an irregular workfile, `@trend` provides an observation-based trend as before, but `@trendc` now returns a calendar trend that increases based on the number of calendar periods between adjacent observations. For example, in a daily irregular file

where a Thursday has been omitted because it was a holiday, the `@trendc` value would increase by two between the Wednesday before and the Friday after the holiday, while the `@trend` will increase by only one.

Both `@trend` and `@trendc` functions may be used with an argument consisting of a string containing the date at which the trend has the value of zero. If this argument is omitted, the first observation in the workfile will be given the value of zero.

The choice of which type of time trend is appropriate in a particular context should be based on what role the time trend is playing in the analysis. When used in estimation, a time trend is usually used to represent some sort of omitted variable. If the omitted variable is something that continues to increase independently of whether the workfile data is observed or not, then the appropriate trend would be the calendar trend. If the omitted variable is something that increases only during periods when the workfile data is observed, then the appropriate trend would be the observation trend.

An example of the former sort of variable would be where the trend is used to represent population growth, which continues to increase whether, for example, financial markets are open on a particular day or not. An example of the second sort of variable might be some type of learning effect, where learning only occurs when the activity is actually undertaken.

Note that while these two trends are provided as built in functions, other types of trends may also be generated based on the calendar data of the workfile. For example, in a file containing monthly sales data, a trend based on either the number of days in the month or the number of business days in the month might be more appropriate than a trend that increments by one for each observation.

These sorts of trends can be readily generated using `@date` and the `@datediff` functions. For example, to generate a trend based on the number of days elapsed between the start date of the current observation and the base date of 1st Jan 2000, we can use:

```
series daytrend = @datediff(@date, @dateval("1/1/2000"), "d")
```

When used in a monthly file, this series would provide a trend that adjusts for the fact that some months contain more days than others.

Note that trends in panel structured workfiles follow special rules. See "Panel Trend Functions" on page 556 for details.

Event Functions

These functions return information about each observation's relationship with a specified date, or date range:

- `@daycount(`["*weekday_range*"]): Returns the number of calendar days within each observation of the workfile. The optional *weekday_range* argument lets you specify certain days of the week to count.

If only one weekday is provided, `@daycount` returns the number of times that particular weekday occurs within the observation. If two weekdays are provided, `@daycount` returns the number of times that any weekday between (and including) the two weekdays occurs within the observation.

- `@before`("*date*"): returns a dummy variable with a value of 1 for each observation prior to *date*, and zero for every other observation. If an observation is partially before *date* (for example if you have a monthly workfile and specify a day as *date*), the fraction of the observation before *date* is returned.
- `@after`("*date*"): returns a dummy variable with a value of 1 for each observation after, *and including*, *date*, and zero for every other observation. If an observation is partially before *date* (for example if you have a monthly workfile and specify a day as *date*), the fraction of the observation after and including *date* is returned.
- `@during`("*date1 date2*"): returns a dummy variable with a value of 1 for each observation between *date1* and *date2* (inclusive), and zero for every other observation. If an observation partially covers the specified dates (for example if you have a monthly workfile and specify a pair of days in the same month), the fraction is returned.
- `@event`("*date1* [*date2*]", [*basis*]): returns a variable containing the proportion of a one-off event covered by each observation. The event can be specified by a single date, or by a pair of dates to denote a date range.

  If the workfile has a regular frequency and spans the entire event, the returned series will sum to one over all observations. If the workfile is irregular or does not span the entire event, the series may sum to less than one due to the observations that have been omitted.

  The optional argument, *basis,* may be used to specify that only certain days of the week, or times of the day should be included as part of the holiday event. *basis* should be enclosed in quotes and should follow a syntax of "*start_weekday-end_weekday, start_time-end_time*". eg. use "Mon-Fri" to indicate that only days from Monday to Friday (inclusive) should be counted when proportioning up the event between observations.
- `@holiday`("*date1* [*date2*]", [*basis*]): returns a variable containing the proportion of an annual event covered by each observation. The event can be specified by a single date, a date pair to denote a date range. When specifying a date, you may use one of the following special terms:

  Day in a month eg. "Dec25".

  Day in a month, with Saturday moved back to Friday and Sunday moved forward to Monday (by using " ~ " after the day) *e.g.,* "Dec25 ~ ".

  N-th weekday within a month (using negative values for last weekday in month). *e.g.,* "Nov4Thu" for the fourth Thursday in November, and "May-1Mon" for

the last Monday in May.

A specific holiday name: “Easter”, “Christmas”, “Thanksgiving”, “cny” (Chinese New Year).

You may optionally supply an offset to the specified holiday by including a number in parenthesis after the holiday. For example to specify a holiday of a range of dates covering three days before Easter to four days after Easter each year, use “Easter(-3) Easter(4).

The optional argument, *basis,* may be used In the `@event` and `@holiday` functions to specify that only certain days of the week, or times of the day should be included as part of the holiday event. *basis* should be enclosed in quotes and should follow a syntax of “*start_weekday-end_weekday, start_time-end_time*”. eg. use “Mon-Fri” to indicate that only days from Monday to Friday (inclusive) should be counted when proportioning up the event between observations.

Indicator Functions

These functions produce indicators for whether each observation satisfies a specific condition:

- `@inlist(`*series, “list”*): returns a dummy variable with a value of 1 for each observation of *series* equal to one of the values specified in *list*. *list* should be a quoted, space delimited list of values. For example, `@inlist(name, "John Jack Susan")` returns a dummy variable equal to 1 for each observation where name matches either “John”, “Jack” or “Susan”.
- `@between`(*series, val1, val2*): returns dummy variable equal to 1 for observations where series is greater than or equal to *val1* and less than or equal to *val2*. For example, `@between(X, 0.4, 0.6)` returns a dummy variable equal to 1 for each observation of X satisfying $0.4 \le X \le 0.6$.

Panel Workfile Functions

Panel Identifier Functions

Additional information is available in panel structured workfiles. EViews provides workfile functions that provide information about the cross-section, cell, and observation IDs associated with each observation in a panel workfile:

- `@crossid`: returns the cross-section index (cross-section number) of each observation.
- `@cellid`: returns the inner dimension index value for each observation. The index numbers identify the unique values of the inner dimension observed across all cross-sections. Thus, if the first cross-section has annual observations for 1990, 1992, 1994,

and 1995, and the second cross-section has observations for 1990, 1995, and 1997, the corresponding `@cellid` values will be (1, 2, 3, 4) and (1, 4, 5), respectively.

- `@obsid`: returns the observation number within each panel cross section for each observation. `@obsid` is similar to `@obsnum` except that it resets to one whenever it crosses the seam between two adjacent cross sections.

See "Identifier Indices" on page 740 of the *User's Guide II* for additional discussion.

Panel Trend Functions

Central to the notion of a panel trend is the notion that the trend values are initialized at the start of a cross-section, increase for successive observations in the specific cross-section, and are reset at the start of the next-cross section.

Beyond that, there are several notions of a time trend that may be employed. EViews provides four different functions that may be used to create a trend series: `@obsid`, `@trendc`, `@cellid`, and `@trend`.

The `@obsid` function may be used to obtain the simplest notion of a trend in which the values for each cross-section begin at one and increase by one for successive observations in the cross-section. To begin your trends at zero, simply use the expression "@OBSID-1". Note that such a trend does not use information about the cell ID values in determining the value increment.

The calendar trend function, `@trendc`, computes trends in which values for observations with the earliest observed date are normalized to zero, and successive observations are incremented based on the calendar for the workfile frequency.

Lastly, `@cellid` and `@trend` compute trends using the observed dates in the panel:

- `@cellid`, which returns an index into the unique values of the cell ID, returns a form of time trend in which the values increase based on the number of cell ID values between successive observations.
- `@trend` function is equivalent to "@cellid-1".

In fully balanced workfiles (workfiles with the same set of cell identifier in each cross-section), the expressions "@obsid-1", "@cellid-1", and "@trend" all return the same values. Additionally, if the workfile follows a regular frequency, then the `@trendc` function returns the same values as `@trend`.

Note that because of the way they employ information computed across cross-sections, `@trend` and `@trendc` may not take the optional *base_date* argument in panel structured workfiles (see "Trend Functions" on page 552).

Chapter 16. Special Expression Reference

The following reference is an alphabetical listing of special expressions that may be used in series assignment and generation, or as terms in estimation specifications.

Special Expression Summary

Special Expression Entries

The following section provides an alphabetical listing of the commands and functions associated with the EViews programming language. Each entry outlines the basic syntax and provides examples and cross references.

ar	Special Expression

Autoregressive error specification.

The AR specification can appear in an `ls` (p. 373) or `tsls` (p. 453) specification to indicate an autoregressive component. `ar(1)` indicates the first order component, `ar(2)` indicates the second order component, and so on.

You may express a range of AR terms using the "to" keyword between a starting and ending integer.

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.

```
tsls sale c adv ar(1 to 4) @ c gdp
```

estimates an equivalent specification.

Cross-references

See Chapter 4. "Time Series Regression," on page 85 of *User's Guide II* for details on ARMA and seasonal ARMA modeling.

See also `sar` (p. 563), `ma` (p. 560), and `sma` (p. 564).

@expand	Special Expression

Automatic dummy variables.

The `@expand` expression may be added in estimation to indicate the use of one or more automatically created dummy variables.

Syntax

@expand(*ser1[, ser2, ser3, ...][, drop_spec]*)

creates a set of dummy variables that span the unique values of the input series *ser1*, *ser2*, *etc.*

The optional *drop_spec* may be used to drop one or more of the dummy variables. *drop_spec* may contain the keyword **@dropfirst** (indicating that you wish to drop the first category), **@droplast** (to drop the last category), or a description of an explicit category, using the syntax:

@drop(*val1[, val2, val3,...]*)

where each argument corresponds to a category in @expand. You may use the wild card "*" to indicate all values of a corresponding category.

Example

For example, consider the following two variables:

- SEX is a numeric series which takes the values 1 and 0.
- REGION is an alpha series which takes the values "North", "South", "East", and "West".

The command:

```
eq.ls income @expand(sex) age
```

regresses INCOME on two dummy variables, one for "SEX = 0" and one for "SEX = 1" as well as the simple regressor AGE.

The @EXPAND statement in,

```
eq.ls income @expand(sex, region) age
```

creates 8 dummy variables corresponding to:

sex = 0, region = "North"

sex = 0, region = "South"

sex = 0, region = "East"

sex = 0, region = "West"

sex = 1, region = "North"

sex = 1, region = "South"

sex = 1, region = "East"

sex = 1, region = "West"

The expression:

```
@expand(sex, region, @dropfirst)
```

creates the set of dummy variables defined above, but no dummy is created for "SEX = 0, REGION = "North"". In the expression:

```
@expand(sex, region, @droplast)
```

no dummy is created for "SEX = 1, REGION = "WEST"".

The expression:

```
@expand(sex, region, @drop(0,"West"), @drop(1,"North")
```

creates a set of dummy variables from SEX and REGION pairs, but no dummy is created for "SEX = 0, REGION = "West"" and "SEX = 1, REGION = "North"".

```
@expand(sex, region, @drop(1,*))
```

specifies that dummy variables for all values of REGION where "SEX = 1" should be dropped.

```
eq.ls income @expand(sex)*age
```

regresses INCOME on regressor AGE with category specific slopes, one for "SEX = 0" and one for "SEX = 1".

Cross-references

See "Automatic Categorical Dummy Variables" on page 28 of *User's Guide II* for further discussion.

ma	Special Expression

Moving average error specification.

The `ma` specification may be added in an `ls` (p. 373) or `tsls` (p. 453) 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.

You may express a range of MA terms using the "to" keyword between a starting and ending integer.

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.

```
ls(z) m1 c tb3 tb3(-1) ma(1 to 4)
```

estimates the same model but with MA terms from 1 to 4.

Cross-references

See "Time Series Regression" on page 85 of *User's Guide II* for details on ARMA and seasonal ARMA modeling.

See also `sma` (p. 564), `ar` (p. 557), and `sar` (p. 563).

na	Special Expression

Not available code. "NA" is used to represent missing observations.

Examples

```
smpl if y >= 0
series z = y
smpl 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 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 `smpl` statement as in the first example above, or the `@recode` or `@nan` function.

Cross-references

See "Missing Values" on page 173 of *User's Guide I* for a discussion of working with missing values in EViews.

nrnd	Special 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

```
smpl @first @first
series y = 0
smpl @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 525 for a list of other random number generating functions from various distributions.

See also rnd (p. 563), rndint (p. 422) and rndseed (p. 423).

pdl	Special 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

pdl(*series_name*, *lags*, *order[,options]*)

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 23 of *User's Guide II* for further discussion.

rnd

Special 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 "Statistical Distribution Functions" on page 525.

See also `nrnd` (p. 561), `rndint` (p. 422) and `rndseed` (p. 423).

sar

Special 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 prod-

uct 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 93 of *User's Guide II* for details on ARMA and seasonal ARMA modeling.

See also sma (p. 564), ar (p. 557), and ma (p. 560).

sma	Special 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 93 of *User's Guide II* for details on ARMA and seasonal ARMA modeling.

See also sar (p. 563), ar (p. 557), and ma (p. 560).

@wexpand	Special Expression

String list containing automatic dummy variable expressions.

Return the string list associated with automatically created dummy variables from numeric or alpha series.

Syntax

@wexpand(*ser1[, ser2, ser3, ...][, "sample string"]*))

returns the string corresponding to the set of dummy variables that span the unique values of the input series *ser1*, *ser2*, *etc.* These may be numerical series with integer only values, alpha series or links.

Example

For example, consider the following two variables:

- SEX is a numeric series which takes the values 1 and 0.
- REGION is an alpha series which takes the values "North", "South", "East", and "West".

The command:

```
string slist = @wexpand(sex)
```

yields the string SEXLIST containing "SEX = 0 SEX = 1".

The `@wexpand` statement in,

```
series sregion = @wexpand(sex, region)
```

yields a space delimited string list SREGION with elements:

"SEX = 0 AND REGION = ""North"""

"SEX = 0 AND REGION = ""South"""

"SEX = 0 AND REGION = ""East"""

"SEX = 0 AND REGION = ""West"""

"SEX = 1 AND REGION = ""North"""

"SEX = 1 AND REGION = ""South"""

"SEX = 1 AND REGION = ""East"""

"SEX = 1 AND REGION = ""West"""

Cross-references

See "Automatic Categorical Dummy Variables" on page 28 of *User's Guide II* for further discussion.

Chapter 17. String and Date Function Reference

EViews provides a full library of string and date functions for use with alphanumeric and date values. Chapter 5. "Strings and Dates," on page 65 contains a discussion of the use of strings and dates in EViews, and provides a description of the string and date functions.

String Function Summary

String Functions

Date Function Summary

Date Functions

Date and String Entries

The following is an alphabetical listing of the functions used when working with strings and dates in EViews.

@addquotes	String Functions

Syntax: @addquotes(*str*)
Argument: string, *str*
Return: string

Returns the string *str* with quotation marks added to both the left and the right.

Example:

```
@addquotes(ss1)
```

Say the string SS1 contains text without quotes: Chicken Marsala. This command returns SS1 with quotes added to each end: “Chicken Marsala”.

See also `@stripquotes` (p. 588).

@asc	String Functions

Syntax: @asc(*str*)
Argument: string, *str*
Return: integer

Returns the integer ASCII value for the first character in the string *str*.

Example:

```
scalar s1 = @asc("C")
```

```
scalar s2 = @asc("c")
```

returns the scalars S1 = 67 and S2 = 99.

See also @chr (p. 570).

@chr	String Functions

Syntax: @chr(*i*)
Argument: integer, *i*
Return: string

Returns the character string corresponding to the ASCII value *i*.

Valid inputs are integers running from 0 to 255. Any integer higher than 255 will return the same thing as 0 (i.e. an empty string).

Example:

```
string s1 = @chr(67)
string s2 = @chr(99)
```

returns the strings S1 = “C” and S2 = “c”.

See also @asc (p. 569).

@dateadd	Date Functions

Syntax: @dateadd(*d, offset, u]*
Argument 1: date number, *d*
Argument 2: number of time units, *offset*
Argument 3: time unit, *u*
Return: date number

Returns the date number given by *d* offset by *offset* time units as specified by the time unit string *u*.

The valid time unit string values are: “A” or “Y” (annual), “S” (semi-annual), “Q” (quarters), “MM” (months), “WW” (weeks), “DD” (days), “B” (business days), “HH” (hours), “MI” (minutes), “SS” (seconds).

Example:

Suppose that the value of *d* is 730088.0 (midnight, December 1, 1999). Then we can add and subtract 10 days from the date by using the functions

```
@dateadd(730088.0, 10, "dd")
@dateadd(730088.0, -10, "dd")
```

which return 730098.0 (December 11, 1999) and (730078.0) (November 21, 1999). Note that these results could have been obtained by taking the original numeric value plus or minus 10.

To add 5 weeks to the existing date, simply specify "W" or "WW" as the time unit string:

```
@dateadd(730088.0, 5, "ww")
```

returns 730123.0 (January 5, 2000).

The valid time unit string values are: "A" or "Y" (annual), "S" (semi-annual), "Q" (quarters), "MM" (months), "WW" (weeks), "DD" (days), "B" (business days), "HH" (hours), "MI" (minutes), "SS" (seconds).

@datediff	Date Functions

Syntax: @datediff(*d1*, *d2*, *u)*
Argument 1: date number, *d1*
Argument 2: date number, *d2*
Argument 3: time unit, *u*
Return: date number

Returns the difference between two date numbers *d1* and *d2*, measured by time units specified by the time unit string *u*.

The valid time unit string values are: "A" or "Y" (annual), "S" (semi-annual), "Q" (quarters), "MM" (months), "WW" (weeks), "DD" (days), "B" (business days), "HH" (hours), "MI" (minutes), "SS" (seconds).

Example:

Suppose that *date1* is 730088.0 (December 1, 1999) and *date2* is 729754.0 (January 1, 1999), then,

```
@datediff(730088.0, 729754.0, "dd")
```

returns 334 for the number of days between the two dates. Note that this is result is simply the difference between the two numbers.

The following expressions calculate differences in months and weeks:

```
@datediff(730088.0, 729754.0, "mm")
@datediff(730088.0, 729754.0, "ww")
```

return 11 and 47 for the number of months and weeks between the dates.

@datefloor	Date Functions

Syntax: @datefloor(*d1, u[, step]*)
Argument 1: date number, *d1*
Argument 2: time unit, *u*
Argument 3: step, *step*
Return: date number

Finds the first possible date number defined by *d1* in the regular frequency defined by the given time unit *u* and optional step *step*. The frequency is defined relative to the basedate of Midnight, January 2001.

The valid time unit string values are: “A” or “Y” (annual), “S” (semi-annual), “Q” (quarters), “MM” (months), “WW” (weeks), “DD” (days), “B” (business days), “HH” (hours), “MI” (minutes), “SS” (seconds).

If *step* is omitted, the frequency will use a step of 1 so that by default, `@datefloor` will find the beginning of the period defined by the time unit.

Example:

Suppose that *date1* is 730110.8 (7 PM, December 23, 1999). Then the `@datefloor` values

```
@datefloor(730110.8, "dd")
@datefloor(730110.8, "mm")
```

yield 730110.0 (midnight, December 23, 1999) and 730088.0 (midnight, December 1, 1999), the date numbers for the beginning of the day and month associated with 730110.8, respectively.

```
@datefloor(730110.8, "hh", 6)
@datefloor(730110.8, "hh", 12)
```

return 730110.75 (6 PM, December 23, 1999), and 730110.5 (Noon, December 23, 1999). which are the earliest date numbers for 6 and 12 hour regular frequencies anchored at Midnight, January 1, 2001.

@datepart	Date Functions

Syntax: @datepart(*d1, fmt*)
Argument 1: date number, *d1*
Argument 2: date format string, *fmt*
Return: number

Returns a numeric part of a date number given by *fmt*, where *fmt* is a date format string component.

Example:

Consider the *d1* date value 730110.5 (noon, December 23, 1999). The `@datepart` values for

```
@datepart(730110.5, "dd")
@datepart(730110.5, "w")
@datepart(730110.5, "ww")
@datepart(730110.5, "mm")
@datepart(730110.5, "yy")
```

are 23 (day of the month), 1 (day of the week), 52 (week in the year), 12 (month in the year), and 99 (year), respectively).

See "Dates" on page 82 and "Date Formats" on page 85 for additional details on date numbers and date format strings.

@datestr	Date Functions | String Functions

Syntax: @datestr(*d[, fmt]*)
Argument 1: date number, *d*
Argument 2: date format string, *fmt*
Return: string

Convert the numeric date value, *d,* into a string representation of a date, *str,* using the optional format string *fmt.*

Example:

```
@datestr(730088,"mm/dd/yy")
```

will return "12/1/99",

```
@datestr(730088,"DD/mm/yyyy")
```

will return "01/12/1999", and

```
@datestr(730088,"Month dd, yyyy")
```

will return "December 1, 1999", and

```
@datestr(730088,"w")
```

will produce the string "3", representing the weekday number for December 1, 1999. See "Dates" on page 82 and "Date Formats" on page 85 for additional details on date numbers and date format strings. See also `@strdate` (p. 587).

@dateval	Date Functions | String Functions

Syntax: @dateval(*str1[, fmt]*)
Argument 1: string, *str1*
Argument 2: date format string, *fmt*
Return: date number

Convert the string representation of a date, *str,* into a date number using the optional format string *fmt.*

Example:

```
@dateval("12/1/1999", "mm/dd/yyyy")
```

will return the date number for December 1, 1999 (730088) while

```
@dateval("12/1/1999", "dd/mm/yyyy")
```

will return the date number for January 12, 1999 (729765).

See "Dates" on page 82 and "Date Formats" on page 85 for additional details on date numbers and date format strings. See also @strdate (p. 587).

@dtoo	Date Functions | String Functions

Syntax: @dtoo(*str*)
Argument: string, *str*
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 (p. 642)).

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.

Note that @dtoo will generate an error if used in a panel structured workfile.

See also `@otod` (p. 580).

@eqna	String Functions

Syntax: @eqna(*str1*, *str2*)
Argument 1: string, *str1*
Argument 2: string, *str2*
Return: integer

Tests for equality of *str1* and *str2*, treating null strings as ordinary blank strings, and not as missing values. Strings which test as equal return a 1, and 0 otherwise.

Example:

```
@eqna("abc", "abc")
```

returns a 1, while

```
@eqna("", "def")
```

returns a 0.

See also `@isempty` (p. 576) and `@neqna` (p. 579).

@insert	String Functions

Syntax: @insert(*str1*, *str2*, *n*)
Argument 1: string, *str1*
Argument 2: string, *str2*
Argument 3: integer, *n*
Return: string

Inserts the string *str2* into the base string *str1* at the position given by the integer *n*.

Example:

```
@insert("I believe it can be done", "not ", 16)
```

returns “I believe it cannot be done”.

See also `@replace` (p. 581), `@instr` (p. 576) and `@mid` (p. 579).

@instr	String Functions

Syntax: @instr(*str1*, *str2[*, *n]*)
Argument 1: string, *str1*
Argument 2: string, *str2*
Argument 3: integer, *n*
Return: string

Finds the starting position of the target string *str2* in the base string *str1*. By default, the function returns the location of the first occurrence of *str2* in *str1*. You may provide an optional integer *n* to change the occurrence. If the occurrence of the target string is not found, @INSTR will return a 0.

Example:

```
@instr("1.23415", "34")
```

return the value 4, since the substring “34” appears beginning in the fourth character of the base string.

See also `@mid` (p. 579).

@isempty	String Functions

Syntax: @isempty(*str*)
Argument: string, *str*
Return: integer

Tests for whether *str* is a blank string, returning a 1 if *str* is a null string, and 0 otherwise.

Example:

```
@isempty("1.23415")
```

returns a 0, while

```
@isempty("")
```

return the value 1.

See also `@len, @length` (p. 577).

@left	String Functions

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 sc1 = @left("I did not do it",5)
```

returns “I did”.

See also `@right` (p. 581), and `@mid` (p. 579).

@len, @length	String Functions

Syntax: @len(*str*), @length(*str*)
Argument: string, *str*
Return: integer

Returns an integer value for the length of the string *str*.

Example:

```
@length("I did not do it")
```

returns the value 15.

See also `@mid` (p. 579).

@lower	String Functions

Syntax: @lower(*str*)
Argument: string, *str*
Return: string

Returns the lowercase representation of the string *str*.

Example:

```
@lower("I did NOT do it")
```

returns the string “i did not do it”.

See also `@upper` (p. 590).

@ltrim

String Functions

Syntax: @ltrim(*str*)
Argument: string, *str*
Return: string

Returns the string *str* with spaces trimmed from the left.

Example:

```
@ltrim(" I doubt that I did it. ")
```

returns “I doubt that I did it. ”. Note that the spaces on the right remain.

See also `@rtrim` (p. 582) and `@trim` (p. 589).

@makedate

Date Functions

Syntax: @makedate(*arg1[, arg2[,arg3]], fmt*)
Argument 1: number, *arg1*
Argument 2: number(s) *arg2, arg3*
Argument 3: date format, *fmt*
Return: date number

Takes the numeric values given by the arguments *arg1*, and optionally, *arg2*, etc. and returns a date number using the required format string, *fmt*.

Example:

The expressions,

```
@makedate(1999, "yyyy")
@makedate(99, "yy")
```

both return the date number 729754.0 corresponding to 12 midnight on January 1, 1999.

```
@makedate(199003, "yyyymm")
@makedate(1990.3, "yyyy.mm")
@makedate(1031990, "ddmmyyyy")
@makedate(30190, "mmddyy")
```

all return the value 726526.0, representing March 1, 1990.

See "Date Formats" on page 85 and "Translating Ordinary Numbers into Date Numbers" on page 94 for additional details. See also `@strdate` (p. 587).

@mid	String Functions

Syntax: @mid(*str, n1[, n2]*)
Argument 1: string, *str*
Argument 2: integer, *n1*
Argument 3: integer, *n2*
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. 577) and `@right` (p. 581).

@neqna	String Functions

Syntax: @neqna(*str1, str2*)
Argument 1: string, *str1*
Argument 2: string, *str2*
Return: integer

Tests for inequality of *str1* and *str2*, treating null strings as ordinary blank strings, and not as missing values. Strings which test as equal return a 0, and 1 otherwise.

Example:

```
@neqna("abc", "abc")
```

returns a 0, while

```
@neqna("", "def")
```

returns a 1.

See also `@isempty` (p. 576) and `@eqna` (p. 575).

@now	Date Functions

Syntax: @now
Return: date number

Returns the date number associated with the current time.

@otod	Date Functions | String Functions

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 `@otods` (p. 580) and `@dtoo` (p. 574).

@otods	Date Functions | String Functions

Syntax: @otods(*n*)
Argument: integer, *n*
Return: string

Observation-TO-Date in Sample. Returns a string containing the date or observation corresponding to observation number *n*, counted from the beginning of the current workfile sample. Thus

```
@otods(2)
```

will return the date associated with the second observation in the current sample. Note that if *n* is negative, or is greater than the number of observations in the current sample, an empty string will be returned.

See also `@otod` (p. 580) and `@dtoo` (p. 574).

@replace	String Functions

Syntax: @replace(*str1*, *str2*, *str3[,n]*)
Argument 1: string, *str1*
Argument 2: string, *str2*
Argument 3: string, *str3*
Argument 4: integer, *n*
Return: string

Returns the base string *str1*, with the replacement *str3* substituted for the target string *str2*. By default, all occurrences of *str2* will be replaced, but you may provide an optional integer *n* to specify the number of occurrences to be replaced.

Example:

```
@replace("Do you think that you can do it?", "you", "I")
```

returns the string “Do I think that I can do it?”, while

```
@replace("Do you think that you can do it?", "you", "I", 1)
```

returns “Do I think that you can do it?”.

See also `@insert` (p. 575) and `@mid` (p. 579).

@right	String Functions

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. 577), `@mid` (p. 579).

@rows	String Functions

Syntax: @rows(*str_vector*)
Argument: string vector, *str_vector*
Return: integer

Returns the number of rows in string vector *str_vector*.

Example:

```
@rows(SV1)
```

returns the integer 3 if the string vector SV1 contains three rows.

@rtrim	String Functions

Syntax: @rtrim(*str*)
Argument: string, *str*
Return: string

Returns the string *str* with spaces trimmed from the right.

Example:

```
@rtrim("  I doubt that I did it.  ")
```

returns the string " I doubt that I did it.". Note that the spaces on the left remain.

See also `@ltrim` (p. 578) and `@trim` (p. 589).

@str	String Functions

Syntax: @str(*d*[, *fmt*])
Argument 1: scalar, *d*
Argument 2: numeric format string, *fmt*
Return: string

Returns a string representing the given number. You may provide an optional format string. (See also `@val` (p. 590) to convert a string into a number.)

EViews offers a variety of ways to write your number as a string. By default, EViews will format the number string using 10 significant digits, with no leading or trailing characters, no thousands separators, and an explicit decimal point.

(The default conversion is equivalent to using `@str` with the format "g.10" as described below.)

If you wish to write your number in a different fashion, you must provide an explicit format string. A numeric format string has the format:

[type][t][+][(][$][#][< | = | >][0][width][[.|..]precision][%][)]

There are a large number of syntax elements in this format but we may simplify matters greatly by dividing them into four basic categories:

- format: *[type]*
- width: *[< | = | >][width]*
- precision: *[precision]*
- advanced modifiers: the remaining elements (leading and trailing characters, padding and truncation modifiers, separators, display of negative numbers)

The *type*, *width*, and *precision* components are the basic components of the format so we will focus on them first. We describe the advanced modifiers in "Modified Formats" on page 585.

Basic Formats

EViews offers formats designed for both real-valued and integer data.

Basic Real-Value Formats

The basic real-value format is:

[type][< | = | >][width][.][precision]

The *type* component is a single character indicating the basic type and the *width* and *precision* arguments are numbers indicating the number of output characters and the precision at which the numbers should be written. If specified, the precision should be separated from the *type* and width portion of the format by a "." character (or ".." as we will see in "Modified Formats" on page 585).

If you specify a format with neither *width* nor *precision*, EViews will format the number at full precision with matching string width.

The following types are for use with real-valued data:

g	significant digits
z	significant digits with trailing zeros
c	fixed characters with single leading space for positive numbers
f	fixed decimal places

e	scientific/float
p	percentage (same as "f" but values are multiplied by 100)
s	suppressed decimal point format
r	ratio, *e.g.*, "30 1/5"

The type character may be followed by a width specification, consisting of a *width* indicating the number of output characters, optionally preceded by a " > ", " = " or " < " modifier.

- If no *width* is provided, the number will be rendered in a string of the exact length required to represent the value (e.g., the number 1.23450 will return "1.2345", a string of length 6).
- If an unmodified *width* or one with the " > " modifier is provided, the specified number places a *lower-bound* on the width of the string: the output will be left-padded to the specified width, if necessary, otherwise the string will be lengthened to accommodate the full output. By default, padding will use spaces, but you may elect to use 0's by providing an advanced modifier ("Modified Formats" on page 585).
- If the" < " modifier is provided along with *width*, the *width* places an *upper-bound* on the width of the string: the output will *not* be padded to the specified width. If the number of output characters exceeds the width, EViews will return a *width*-length string filled with the "#" character.
- If the" = " modifier is provided along with *width*, the *width* provides an *exact-bound* for the width of the string: the output will be padded to specified width, if necessary. If the number of characters exceeds the width, EViews will return a *width*-length string filled with the "#" character.

If you specify a *precision*, the interpretation of the value will vary depending on the format type. For example, *precision* represents the number of significant digits in the "g" and "z" formats, the number of characters in the "c" format, and the number of digits to the right of the decimal in the "f", "e", "p", and "s" formats. For the "r" format, the precision determines maximum denominator for the fractional portion (as a power-of-10).

The following guidelines are used to determine the precision implied by a number format:

- If you specify a format with only a *precision* specification, the *precision* will implicitly determine the width at which your numbers are written.
- If you specify a format with only a *width* specification, the *width* will implicitly determine the precision at which your numbers are written. Bear in mind that only the modified width specifications " =*width*" and " <*width*" impose binding restrictions on the precision.

- If you specify a format using both *width* and *precision*, the precision at which your numbers are written will be determined by whichever setting is most restrictive (*i.e.*, "f=4.8" and "f=4.2" both imply a formatted number with two digits to the right of the decimal).)

Basic Integer Formats

The basic integer format is:

[type][<|=|>][width]

The *type* component is a single character indicating the basic type. The following types are for use with integer data:

i	integer
h	hexidecimal
o	octal
b	binary

If one of the integer formats is used with real-valued data, the non-integer portion of the number will be ignored. You may specify a *width* using the syntax and rules described in "Basic Real-Value Formats" on page 583.

Modified Formats

Recall that the syntax of a numeric format string is given by:

[type][t][+][()][$][#][<|=|>][0][width][[.|..]precision][%][)]

In addition to the basic *type*, *width*, and *precision* specifiers, the formats take a set of modifiers which provide you with additional control over the appearance of your output string:

- You may combine any of the real-value format characters ("g", "e", "f", *etc.*) with the letter "t" ("gt", "et", "ft", *etc.*) to display a thousands separator ("1,234.56"). By default, the separator will be a comma ",", but the character may be changed to a "." using the ".." format as described below.
- You may add a "+" symbol after the format character ("g+", "et+", "i+") to display positive numbers with a leading "+".
- To display negative numbers within parentheses, you should enclose the remaining portion of the format in parentheses "ft+($8.2)".
- Add "$" to the format to display a leading "$" currency character.
- You should include a "#" to display a trailing point in scientific notation (*e.g.*, "3.e+34").

- The width argument should include a leading zero (“0”) if you wish padded numbers to display leading zeros instead of spaces (“g08.2”, “i05”).
- If you added a “t” character to your real-value format *type*, you may replace the usual “.” *width-precision* separator with “..” (“ft < 08..2”, “e = 7..”, “g..9”, *etc.*) to reverse the thousands and decimal separators so that thousands will be separated by a “.” and decimal portion by a “,” (“1.234,56”).
- Adding a “%” character to the end of the format adds the “%” character to the end of the resulting string.

Examples

```
string num = @str(1.23)
```

assigns to the string NUM the text “1.23”.

```
alpha alpha1 = @str(-123.88)
```

assigns the string “-123.88” to the alpha series ALPHA1.

```
string num = @str(123456,”z.9”)
```

returns a string formatted to 9 significant digits with trailing zeros: “123456.000”.

```
string num = @str(123456,”z.4”)
```

returns a string formatted to 4 significant digits with trailing zeros: “1.235e + 05”. Note that since the input number has more than 4 significant digits, no trailing zeros are added and the resulting string is identical to one that would have been produced with a “g.4” format.

```
string num = @str(126.543,”c.7%”)
```

returns a string with exactly 7 characters, including a leading space, and a trailing percent sign: “ 126.5%”.

```
string num = @str(126.543,"p.7")
```

converts the number 126.543 into a percentage with 7 decimal places and returns “12654.3000000”. Note no percent sign is included in the string.

```
string num = @str(1.83542,"f$5.4")
```

returns “$1.8354”. The width selection of “5” with an implicit “ > ” modifier is irrelevant, since the precision setting of “4”, coupled with the insertion of the “$” symbol yields a string with more characters than “5”.

```
string num = @str(1.83542,"f$8.4")
```

returns “ $1.8354”. Here the width selection is binding, and a leading space has been added to the string.

```
string num = @str(1.83542,"f$=5.4")
```

returns “ $1.84”. The explicit “ = ” width modifier causes the width setting of “5” to be binding.

```
string num = @str(524.784,"r")
```

converts the number 524.784 into a ratio representation: “524 98/125”.

```
string num = @str(1784.321,"r=3")
```

will return “###”, since there is no way to represent 1784.321 as a string with only 3 characters.

```
string num = @str(543,"b")
```

converts the number 543 into a binary representation: “1000011111”.

See also @val (p. 590) to convert a string into a number.

@strdate	Date Functions | String Functions

Syntax: @strdate(*fmt*)
Argument: date format string, *fmt*
Return: string corresponding to each element in workfile

Return the set of workfile row dates as strings, using the date format string *fmt*.

See also @datestr (p. 573) and @strnow (p. 589). See “Date Formats” on page 85 for additional detail.

@stripcommas	String Functions

Syntax: @stripcommas(*str*)
Argument: string, *str*
Return: string

Returns the string *str* with leading and trailing commas stripped.

Example:

```
string ss1 = ", lettuce, tomato, onion, broccoli, "
ss1 = @stripcommas(ss1)
```

The first line creates the string SS1 and sets it equal to “, lettuce, tomato, onion, broccoli,”. The second command removes the commas resulting in: “lettuce, tomato, onion, broccoli”.

Cross-references

See also @stripparens (p. 588) and @stripquotes (p. 588).

@stripparens	String Functions

Syntax: @stripparens(*str*)
Argument: string, *str*
Return: string

Returns the string *str* with parentheses stripped from both the left and the right.

Example:

```
string ss1 = "(I don't know)"
ss1 = @stripparens(ss1)
```

The first line creates the string SS1 and sets it equal to "(I don't know)". The second command removes the parentheses from both ends, resulting in: "I don't know".

Cross-references

See also `@stripcommas` (p. 587) and `@stripquotes` (p. 588).

@stripquotes	String Functions

Syntax: @stripquotes(*str*)
Argument: string, *str*
Return: string

Returns the string *str* with quotation marks stripped from both the left and the right.

Example:

```
@stripquotes(ss1)
```

If the string SS1 initially contains the text "Chicken Marsala" in quotes, then this command strips the quotes from each end, returning:

```
Chicken Marsala
```

Cross-references

See also `@addquotes` (p. 569), `@stripcommas` (p. 587), and `@stripparens` (p. 588).

@strlen	String Functions

Syntax: @strlen(*s*)
Argument: string, *s*
Return: number *n*

Same as `@len, @length` (p. 577).

@strnow	Date Function | String Functions

Syntax: @strnow(*fmt*)
Argument: date format string, *fmt*
Return: string

Returns a string representation of the current date number (at the moment the function is evaluated) using the date format string, *fmt*.

Example:

```
@strnow("DD/mm/yyyy")
```

returns the date associated with the current time in string form with 2-digit days, months, and 4-digit years separated by a slash, “24/12/2003”.

See also `@strdate` (p. 587) and `@datestr` (p. 573). See “Date Formats” on page 85 for additional detail.

@trim	String Functions

Syntax: @trim(*str*)
Argument: string, *str*
Return: string

Returns the string *str* with spaces trimmed from both the left and the right.

Example:

```
@trim("  I doubt that I did it.  ")
```

returns the string “I doubt that I did it.”.

See also `@ltrim` (p. 578) and `@rtrim` (p. 582).

@upper	String Functions

Syntax: @upper(*str*)
Argument: string, *str*
Return: string

Returns the uppercase representation of the string *str*.

Example:

```
@upper("I did not do it")
```

returns the string "I DID NOT DO IT".

See also `@lower` (p. 577).

@val	String Functions

Syntax: @val(*str[, fmt]*)
Argument 1: string, *str*
Argument 2: numeric format string, *fmt*
Return: scalar

Returns the number that a string *str* represents. You may provide an optional numeric format string *fmt*.

In most cases, EViews will be able to convert your string into the corresponding numeric value without additional input. If EViews is unable to perform a conversion, it will return a missing (NA) value.

There are a few important conventions used in the conversion process:

- A leading "$" in the string will be ignored.
- Strings enclosed in "( )" or with a leading "–" will be treated as negative numbers. All other numeric strings, including those with a leading " + " will be treated as positive numbers. You may not have a leading " + " or "–" inside of the parentheses.
- A trailing "%" sign instructs EViews to treat the input string as a percentage—the resulting value will be divided by 100.

There are some situations where you must provide a numeric format string so that EViews can properly interpret your input. The syntax for the format string depends on the type of number the string represents.

Real-Value Formats

EViews will properly interpret non-delimited decimal and scientific notation numeric input strings as numbers.

If your string uses "," to separate thousands, you should specify the "t" format string to remove "," delimiters prior to conversion. If the string uses "." to separate thousands, you should use "t.." to instruct EViews to remove "." delimiters.

If your input string represents a number with suppressed decimal format, you should include a format string beginning with the letter "s":

s.*precision*	suppressed decimal point format (precision determines the number of digits to the right of the decimal)

EViews will divide the resulting number by 10 raised to the power of the specified *precision*. The "s" format specification may be followed by a "t." or a "t.." specification if necessary.

Integer Formats

r	ratio (*e.g.*, "30 1/5").
i	integer
h	hexidecimal
o	octal
b	binary

You should use the "r", "h", "o", or "b" formats to indicate that your input is in the specified format. The "i" format is generally not necessary unless you wish to produce a missing value for a non-integer input string.

Examples

```
scalar num = @val("$1.23")
```

assigns the scalar NUM the numeric value 1.23.

```
series ser1 = @val("-$123.88")
```

returns the value -123.88.

```
scalar sperct = @val("478%")
```

divides the value by 100, setting the scalar SPERCT to 4.78.

```
scalar sratio = @val("(321 1/5)", "r")
```

sets the scalar SRATIO equal to -321.2

```
scalar shexa = @val("f01a", "h")
```

treats the string “f01a” as a hexadecimal number, converts it into the decimal equivalent, 61466, and assigns it to the scalar object SHEXA.

```
scalar sbin = @val("11110101", "b")
```

interprets the string “11110101” as a binary number, converts it into the decimal equivalent, 245, and assigns it to the scalar SBIN.

To verify that a value is an integer, you may use the “i” option.

```
scalar sintna = @val("98.32", "i")
scalar sint = @val("96", "i")
```

SINTNA will contain a missing value NA since the input represents a non-integer value, while SINT is set to 96.

See also `@str` (p. 582).

@wcount	String Functions

Syntax: @wcount(*str_list*)
Argument: string list, *str_list*
Return: number *n*

Returns the number of elements in the string list *str_list*.

Example:

```
@wcount("I did not do it")
```

returns the number 5.

@wcross	String Functions

Syntax: @wcross(*str_list1*, *str_list2*[, "*pattern*"])
Argument 1: string list, *str_list1*
Argument 2: string list, *str_list2*
Argument 3: string, *pattern*
Return: string list

Returns *str_list1* crossed with *str_list2,* according to the string *pattern*. The default pattern is “??”, which indicates that each element of *str_list1* should be crossed individually with each element of *str_list2*.

Example:

```
@wcross("A B C", "1 2 3 4")
```

returns every combination of the elements “A B C” and “1 2 3 4”, using the default pattern “??”. This produces the string list: “A1 A2 A3 A4 B1 B2 B3 B4 C1 C2 C3 C4”.

```
@wcross("ABC DEF", "1 2 3", "?-?")
```

returns the string list “ABC-1 ABC-2 ABC-3 DEF-1 DEF-2 DEF-3”, inserting a dash (“-”) between each crossed element as the “?-?” pattern indicates.

See also `@winterleave` (p. 595).

@wdelim

String Functions

Syntax: @wdelim(*str_list*, "*src_delim*", "*dest_delim*")
Argument 1: string list, *str_list*
Argument 2: string delimiter, *src_delim*
Argument 3: string delimiter, *dest_delim*
Return: string list

Returns a string list, replacing every appearance of the *src_delim* delimiter in *str_list* with a *dest_delim* delimiter. Delimiters must be single characters.

Note that most other string list functions (those beginning with “@w”) require that the delimiter be a space (“ ”). Use this function to convert strings with other delimiters into those which can be used with the string list functions.

Example:

```
@wdelim("Arizona, California, Washington", ",", "-")
```

identifies the comma as the source delimiter and replaces each comma with a dash, returning the string “Arizona-California-Washington”.

See also `@wreplace` (p. 599) and `@replace` (p. 581).

@wdrop

String Functions

Syntax: @wdrop(*str_list*, "*pattern_list*")
Argument 1: string list, *str_list*
Argument 2: string pattern list, *pattern_list*
Return: string list

Returns a string list, dropping elements in *str_list* that match the string pattern *pattern_list*. The *pattern_list* is space delimited and may be made up of any number of “?” (indicates any

single character) or "*" (indicates any number of characters). The pattern is case-sensitive and must exactly match the *str_list* characters to be dropped.

Example:

```
@wdrop("D D A B C", "B D C")
```

removes each instant of the elements "B", "D", and "C", returning the string "A".

```
@wdrop("ABC DEF GHI JKL", "?B? D?? *I")
```

drops the first three elements of the string list, returning the string "JKL". The string pattern "?B?" drops "ABC", the pattern "D??" drops "DEF", and the pattern "*I" drops "GHI".

See also `@wkeep` (p. 596) and `@wreplace` (p. 599).

@wfind	String Functions

Syntax: @wfind(*str_list*, *str_cmp*)
Argument 1: string list, *str_list*
Argument 2: string, *str_cmp*
Return: number *n*

Looks for the string *str_cmp* in the string list *str_list,* and returns the position in the list or 0 if the string is not in the list. This function is case-sensitive; use the `@wfindnc` (p. 594) function to ignore case.

Example:

```
@wfind("I did it", "did")
```

returns the number 2, indicating the string "did" is second in the list.

```
@wfind("I did not do it", "i")
```

This function is case-sensitive and searches for the full string, so "i" will not be found and the function will return the number 0.

See also `@wfindnc` (p. 594).

@wfindnc	String Functions

Syntax: @wfindnc(*str_list*, *str_cmp*)
Argument 1: string list, *str_list*
Argument 2: string, *str_cmp*
Return: number *n*

Looks for the string *str_cmp* in the string list *str_list,* and returns the position in the list or 0 if the string is not in the list. This is the same as the `@wfind` (p. 594) function, but is not case-sensitive.

Example:

```
@wfindnc("I did it", "DID")
```

returns the number 2, indicating the caseless string “did” is second in the list.

```
@wfindnc("I did not do it", "i")
```

returns the number 1, since the first “i” or “I” encountered was in the first position.

See also `@wfind` (p. 594).

@winterleave	String Functions

Syntax:	@winterleave(*str_list1*, *str_list2[*, *count1*, *count2]*)
Argument 1:	string list, *str_list1*
Argument 2:	string list, *str_list2*
Argument 3:	number, *count1*
Argument 4:	number, *count2*
Return:	string list

Interleaves *str_list1* with *str_list2,* according to the pattern specified by *count1* and *count2.* The default uses counts of one.

Example:

```
@winterleave("A B C", "1 2 3")
```

interleaves “A B C” with “1 2 3” to produce the string list “A 1 B 2 C 3”.

```
@winterleave("A B C", "1 2 3 4 5 6")
```

interleaves “A B C” with “1 2 3 4 5 6”, returning the string list “A 1 B 2 C 3 “” 4”. Since there are more elements in the second string, empty strings (“”) are inserted when the elements of the first string run out.

Alternately, you may specify a count pattern to control the spacing of elements:

```
@winterleave("A B C", "1 2 3 4 5 6", 1, 2)
```

This command combines one element of the first string for every two elements of the second string, returning “A 1 2 B 3 4 C 5 6”.

See also `@wcross` (p. 592).

@wintersect	String Functions

Syntax: @wintersect(*str_list1*, *str_list2*)
Argument 1: string list, *str_list1*
Argument 2: string list, *str_list2*
Return: string list

Returns the intersection of *str_list1* and *str_list2*. This function is case-sensitive.

Example:

```
@wintersect("John and Greg won", "Greg won with Mark's help")
```

returns the string “Greg won”.

Since this function is case-sensitive,

```
@wintersect("John and Greg won", "greg won with Mark's help")
```

just returns the string “won”.

See also `@wunion` (p. 601).

@wkeep	String Functions

Syntax: @wkeep(*str_list*, *pattern_list*)
Argument 1: string list, *str_list*
Argument 2: string pattern list, *pattern_list*
Return: string list

Returns the list of elements in *str_list* that match the string pattern *pattern_list*. The *pattern_list* is space delimited and may be made up of any number of “?” (indicates any single character) or “*” (indicates any number of characters). The pattern is case-sensitive and must exactly match the *str_list* characters to be kept.

Example:

```
@wkeep("D D A B C", "B D C")
```

returns all matching elements in *pattern_list* that are found in *str_list:* “D D B C”.

```
@wkeep("ABC DEF GHI JKL", "?B? D?? *I")
```

keeps the first three elements of the string list, returning the string “ABC DEF GHI”. The string pattern “?B?” keeps “ABC”, the pattern “D??” keeps “DEF”, and the pattern “*I” keeps “GHI”.

See also @wdrop (p. 593) and @wreplace (p. 599).

@wleft	String Functions

Syntax: @wleft(*str_list*, *n*)
Argument 1: string list, *str_list*
Argument 2: number, *n*
Return: string list

Returns a string list containing the left *n* elements of *str_list*. If the string list *str_list* is shorter than *n* elements, this function returns all of the elements in the source string.

Example:

```
@wleft("I did not do it", 2)
```

returns the left two elements, “I did”.

See also @wright (p. 600) and @wmid (p. 597).

@wmid	String Functions

Syntax: @wmid(*str_list*, *n1*[, *n2*])
Argument 1: string list, *str*
Argument 2: number, *n1*
Argument 3: number, *n2*
Return: string list

Returns *n2* elements from *str_list*, starting at element *n1* and continuing to the right. If you omit *n2*, it will return all of the remaining elements in the string.

Example:

```
@wmid("I doubt you did it", 2, 3)
```

starts at the second element of the string list and returns the three elements “doubt you did”.

```
@wmid("I doubt it", 9)
```

returns the full string list “I doubt it”.

See also @wleft (p. 597) and @wright (p. 600).

@wnotin	String Functions

Syntax: @wnotin(*str_list1*, *str_list2*)
Argument 1: string list, *str_list1*
Argument 2: string list, *str_list2*
Return: string list

Returns elements of *str_list1* that are not in *str_list2*. This function is case-sensitive.

Example:

```
string employee1 = @wnotin("Full Name: John Smith", "Full Name:")
string employee2 = @wnotin("Full Name: Mary Jones", "Full Name:")
```

assigns the string “John Smith” to the string object EMPLOYEE1, and the string “Mary Jones” to the string object EMPLOYEE2.

```
@wnotin("John and Greg won", "and Greg")
```

returns the string “John won”.

See also `@wunique` (p. 602).

@word	String Functions

Syntax: @word(*str_list*, *n*)
Argument 1: string list, *str_list*
Argument 2: number, *n*
Return: string list

Returns the *n*-th element from the string list *str_list*.

Example:

```
@word("I don't think so", 2)
```

returns the second element, “don’t”.

See also `@wordq` (p. 599).

@wordq	String Functions

Syntax: @wordq(*str_list*, *n*)
Argument 1: string list, *str_list*
Argument 2: number, *n*
Return: string list

Returns the *n*-th element from the string list *str_list*. This function is the same as the `@word` (p. 598) function, while preserving quotes.

Example:

```
@wordq("""Chicken Marsala"" ""Beef Stew""", 2)
```

returns the second element and includes quotes: “Beef Stew”. The `@word` (p. 598) function would return the same elements, but would not include quotation marks in the string.

See also `@word` (p. 598).

@wread	String Functions

Syntax: @wread(*“file”*)
Argument 1: file name of a text file on disk.
Return: string

Returns a string containing the contents of the specified text file on disk. Note that any line breaks in the text file will be removed.

Example

```
@wread(“c:\temp\myfile.txt”)
```

Returns a string containing the contents of the “c:\temp\myfile.txt”.

@wreplace	String Functions

Syntax: @wreplace(*str_list*, "*src_pattern*", "*replace_pattern*")
Argument 1: string list, *str_list*
Argument 2: string pattern, *src_pattern*
Argument 3: string pattern, *replace_pattern*
Return: string list

Replaces instances of *src_pattern* in *str_list* with *replace_pattern*. The pattern lists may be made up of any number of "?" (indicates any single character) or "*" (indicates any number of characters). The pattern is case-sensitive and must exactly match the *str_list* characters to be replaced.

Example:

```
@wreplace("ABC AB", "*B*", "*X*")
```

replaces all instances of "B" with "X", returning the string "AXC AX".

```
@wreplace("ABC DDBC", "??B?", "??X?")
```

replaces all instances of "B" which have two leading characters and one following character, returning the string "ABC DDXC".

See also `@wdrop` (p. 593) and `@wkeep` (p. 596).

@wright

String Functions

Syntax: @wright(*str_list*, *n*)
Argument 1: string list, *str_list*
Argument 2: number, *n*
Return: string list

Returns a string list containing *n* elements from the right end of *str_list*. If the source is shorter than *n*, the entire string is returned.

Example:

```
@wright("I doubt it", 2)
```

returns the right two elements, "doubt it".

See also `@wleft` (p. 597) and `@wmid` (p. 597).

@wsort

String Functions

Syntax: @wsort(*str_list[*, "d"*]*)
Argument 1: string list, *str_list*
Argument 2: "d" to sort in descending order
Return: string list

Returns sorted elements of *str_list*. Use the "d" flag to sort in descending order.

Example:

```
@wsort("expr1 aa1 fr3")
```

returns the sorted string “aa1 expr1 fr3”

```
@wsort("fq8 Fq8 xpr1", "d")
```

sorts the string in descending order: “xpr1 Fq8 fq8”.

@wsplit	String Functions

Syntax: @wsplit(*str_list*)
Argument: string list, *str_list*
Return: svector

Returns an svector containing the elements of *str_list*.

Example:

If string list SS01 contains “A B C D E F”, then

```
@wsplit(ss01)
```

returns an untitled svector, placing an element of SS01 in each row. For example, row one of the svector contains “A”, row two contains “B”, etc.

@wunion	String Functions

Syntax: @wunion(*str_list1*, *str_list2*)
Argument 1: string list, *str_list1*
Argument 2: string list, *str_list2*
Return: string list

Returns the union of *str_list1* and *str_list2*. This function is case-sensitive.

Example:

```
@wunion("ABC DEF", "ABC G H def")
```

returns the string “ABC DEF G H def”. Each new element is added to the string list, skipping elements that have already been added to the list.

See also `@wintersect` (p. 596).

@wunique	String Functions

Syntax: @wunique(*str_list*)
Argument: string list, *str_list*
Return: string list

Returns *str_list* with duplicate elements removed from the list. This function is case-sensitive.

Example:

```
@wunique("fr1 fr2 fr1")
```

returns the string "fr1 fr2". Note that this function is case-sensitive, such that

```
@wunique("a A b B c c")
```

returns the string "a A b B c".

See also `@wnotin` (p. 598).

@xputnames	String Functions

Syntax: @xputnames
Return: string

returns a space delimited list containing the names of objects created in a foreign application using the last `xput` (p. 498) command. In most cases this list will contain EViews object names, but in cases where the object had a name that was not valid in the foreign application, it will return the name that was created.

For example if you issue the commands:

```
xopen(r)
xput X Y LOG(Z)
string a = @xputnames
```

The string object A will contain "X Y LOG_Z".

Cross-references

See "EViews COM Automation Client Support (MATLAB and R)," beginning on page 162 for discussion. See also "External Program Interface" on page 774 of *User's Guide I* for global options setting of the default case for names.

See also `xput` (p. 498).

Chapter 18. Matrix Language Reference

The following entries constitute a listing of the functions and commands used in the EViews matrix language. For a description of the EViews matrix language, see Chapter 11. "Matrix Language," on page 239.

Matrix Command and Function Summary

Matrix Utility Commands

Matrix Utility Functions

Matrix Algebra Functions

@inner .....Computes the inner product of two vectors or series, or the inner product of a matrix object (p. 628).
@inverse.....Returns the inverse of a square matrix object or sym (p. 629).
@issingular.....Returns 1 if the square matrix or sym is singular, and 0 otherwise (p. 630).
@norm .....Computes the norm of a matrix object or series (p. 634).
@outer .....Computes the outer product of two vectors or series, or the outer product of a matrix object (p. 635).
@pinverse.....Returns the Moore-Penrose pseudo-inverse of a square matrix object or sym (p. 636).
@qform .....Compute the quadratic form of a symmetric matrix and a matrix or vector (p. 636).
@rank .....Returns the rank of a matrix object (p. 637).
@solvesystem .....Solves system of linear equations, $Mx = v$, for x (p. 641).
@svd .....Performs singular value decomposition (p. 643).
@trace .....Computes the trace of a square matrix or sym (p. 644).

Matrix Element Functions

@ediv .....Computes element by element division of two matrices (p. 618).
@eeq .....Computes element by element equality comparison of two matrices (p. 618).
@eeqna .....Computes element by element equality comparison of two matrices with NAs treated as ordinary value for comparison (p. 619).
@ege .....Computes element by element tests for whether the elements in the matrix or sym are greater than or equal to corresponding elements in another matrix or sym (p. 619).
@egt.....Computes element by element tests for whether the elements in the matrix or sym are strictly greater than corresponding elements in another matrix or sym (p. 619).
@einv.....Computes element by element inversion of a matrix (p. 621).
@ele.....Computes element by element tests for whether the elements in the matrix or sym are less than or equal to corresponding elements in another matrix or sym (p. 621).
@elt.....Computes element by element tests for whether the elements in the matrix or sym are strictly less than corresponding elements in another matrix or sym (p. 621).
@emult.....Computes element by element multiplication of two matrices (p. 622).
@eneq .....Computes element by element inequality comparison of two matrices (p. 622).

@eneqna..... Computes element by element inequality comparison of two matrices with NAs treated as ordinary value for comparison (p. 622).

@epow..... Raises each element in a matrix to a power (p. 623).

Matrix Descriptive Statistics Functions

You may use any of the descriptive statistics functions that are described in "Descriptive Statistics" on page 508. These statistics will be applied to the matrix object as a whole so that, for example, using `@mean` computes the mean taken over all non NA values in the matrix.

In addition, the following functions have special forms that apply to matrix objects:

@cor..... Computes correlation between two vectors, or between the columns of a matrix (p. 616).

@cov ..... Computes covariance between two vectors, or between the columns of a matrix using n as the divisor (p. 616).

@covp..... Computes covariance between two vectors, or between the columns of a matrix using n as the divisor (p. 616).

@covs ..... Computes covariance between two vectors, or between the columns of a matrix using $n-1$ as the divisor (p. 616).

@inner..... Computes the inner product of two vectors or series, or the inner product of a matrix object (p. 628).

Matrix Column Functions

There are also a handful of column functions that return results computed for each column of the matrix:

@cfirst ..... Returns the value of the first non-missing value in each column of a matrix (p. 610).

@cifirst ..... Returns the index of the first non-missing value in each column of a matrix (p. 611).

@cilast..... Returns the index of the last non-missing value in each column of a matrix (p. 611).

@cimax..... Returns the index of the maximal value in each column of a matrix (p. 611).

@cimin ..... Returns the index of the minimal value in each column of a matrix (p. 612).

@cilast..... Returns the index of the last non-missing value in each column of a matrix (p. 611).

@clast..... Returns the value of the last non-missing value in each column of the matrix (p. 612).

@cmax..... Returns the maximal value in each column of a matrix (p. 612).

@cmean..... Returns the mean value in each column of a matrix (p. 613).

@cmin ..... Returns the minimal value for each column of the matrix (p. 613).

Additional Functions

In addition, EViews also supports matrix element versions of the following categories of functions:

Category	Matrix Element Support
Operators (p. 504)	addition, subtraction, multiplication, and comparison operators (note that the comparison operators perform the comparison for every element of the matrix object and return a 1 if true for all elements, and a 0 if the comparison fails for any element).
Basic Mathematical Functions (p. 505)	All, except for "@inv" and "@recode".
Special Functions (p. 522)	All
Trigonometric Functions (p. 524)	All
Statistical Distribution Functions (p. 525)	All

Matrix Language Entries

The following section provides an alphabetical listing of the commands and functions associated with the EViews matrix language. Each entry outlines the basic syntax and provides examples and cross references.

@capplyranks	Matrix Utility Functions

Syntax: @capplyranks(*m*, *v*[, *n*])
Argument 1: vector, matrix, or sym, *m*
Argument 2: vector, *v*
Argument 3: (optional) integer, *n*
Return: vector, matrix, or sym

Reorder the rows of a matrix *m* using the ranks in the vector *v*. If the optional argument *n* is specified, only the rows in column *n* will be reordered. *v* should contain unique integers from 1 to the number of rows of *m*.

```
matrix m2 = @capplyranks(m1, v1)
```

reorders the rows of the matrix M1 using the ranks in V1, while

```
matrix m3 = @cappyranks(m1, v1, 3)
```

reorders only the rows of column 3 of M1.

Note that you may use the `@ranks` function to obtain the ranks of a vector. Obtaining unique integer ranking for data with ties requires use of the "i" or "r" option in `@ranks`, as in

```
vector y = @ranks(x, "a", "i")
```

See "Descriptive Statistics" on page 508.

See also `@rapplyranks` (p. 637) and `@permute` (p. 635).

@cfirst — Matrix Column Functions

Syntax: @cfirst(*m*)
Argument: matrix, *m*
Return: vector

Returns a vector containing the first non-missing values of each column of the matrix *m*. Example:

```
vector v1 = @cmax(m1)
```

See also `@cifirst` (p. 611), `@first` (p. 625).

@cholesky — Matrix Algebra Functions

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)
```

@cifirst	Matrix Column Functions

Syntax: @cifirst(*m*)
Argument: matrix, *m*
Return: vector

Returns a vector containing the index (*i.e*, row number) of the first non-missing value of each column of the matrix *m*. Example:

```
vector v1 = @cifirst(m1)
```

See also `@cfirst` (p. 610), `@ifirst` (p. 627).

@cilast	Matrix Column Functions

Syntax: @cilast(*m*)
Argument: matrix, *m*
Return: vector

Returns a vector containing the index (*i.e*, row number) of the last non-missing value of each column of the matrix *m*. Example:

```
vector v1 = @cilast(m1)
```

See also `@clast` (p. 612), `@ilast` (p. 627).

@cimax	Matrix Column Functions

Syntax: @cimax(*m*)
Argument: matrix, *m*
Return: vector

Returns a vector containing the index (*i.e*, row number) of the maximal values of each column of the matrix *m*. Example:

```
vector v1 = @cimax(m1)
```

See also `@cimin` (p. 612), `@cmax` (p. 612), `@cmin` (p. 613).

@cimin	Matrix Column Functions

Syntax: @cimin(*m*)
Argument: matrix, *m*
Return: vector

Returns a vector containing the index (*i.e*, row number) of the minimal values of each column of the matrix *m*. Example:

```
vector v1 = @cimin(m1)
```

See also `@cimax` (p. 611), `@cmax` (p. 612), `@cmin` (p. 613).

@clast	Matrix Column Functions

Syntax: @clast(*m*)
Argument: matrix, *m*
Return: vector

Returns a vector containing the last non-missing value of each column of the matrix *m*. Example:

```
vector v1 = @clast(m1)
```

See also `@cilast` (p. 611), `@last` (p. 630).

@cmax	Matrix Column Functions

Syntax: @cmax(*m*)
Argument: matrix, *m*
Return: vector

Returns a vector containing the maximal values of each column of the matrix *m*. Example:

```
vector v1 = @cmax(m1)
```

See also `@cimax` (p. 611), `@cimin` (p. 612), `@cmin` (p. 613).

@cmean	Matrix Column Functions

Syntax: @cmean(*m*)
Argument: matrix, *m*
Return: vector

Returns a vector containing the mean values of each column of the matrix *m*. Example:

```
vector v1 = @cmean(m1)
```

See also `@csum` (p. 617).

@cmin	Matrix Column Functions

Syntax: @cmin(*m*)
Argument: matrix, *m*
Return: vector

Returns a vector containing the minimal values of each column of the matrix *m*. Example:

```
vector v1 = @cmin(m1)
```

See also `@cimax` (p. 611), `@cimin` (p. 612), `@cmax` (p. 612).

@cnas	Matrix Column Functions

Syntax: @cnas(*m*)
Argument: matrix, *m*
Return: vector

Returns a vector containing the number of missing values in each column of the matrix *m*. Example:

```
vector v1= @cnas(m1)
```

@cobs	Matrix Column Functions

Syntax: @cobs(*m*)
Argument: matrix, *m*
Return: vector

Returns a vector containing the number of non-missing values in each column of the matrix *m*. Example:

```
vector v1= @cobs(m1)
```

colplace — Matrix Utility Commands

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. 639).

@columnextract — Matrix Utility Functions

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. 639).

@columns — Matrix Utility Functions

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. 639).

@cond

Matrix Algebra Functions

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 sc1 = @cond(m1)
mat1(1,4) = @cond(s1,2)
```

@convert

Matrix Utility Functions

Syntax:	@convert(*o*, *smp*)
Argument 1:	series or group, *o*
Argument 2:	(optional) sample, *smp*
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. 642).

@cor	Matrix Descriptive Statistics Functions

Syntax: @cor(*v1*, *v2*)
Argument 1: vector, rowvector, or series, *v1*
Argument 2: vector, rowvector, or series, *v2*
Return: scalar

Syntax: @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. 616). See `Group::cor` (p. 267) in the *Object Reference* for more general computation of correlations.

@cov	Matrix Descriptive Statistics Functions

Syntax: @cov(*v1*, *v2*)
Argument 1: vector, rowvector, or series, *v1*
Argument 2: vector, rowvector, or series, *v2*
Return: scalar

Syntax: @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 sc1 = @cov(v1, v2)
s1(1,2) = @cov(v1, r1)
```

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. 616). See `Group::cov` (p. 271) in the *Object Reference* for more general computation of covariances.

@covp	Matrix Descriptive Statistics Functions

Compute covariances using n as the divisor in the moment calculation. Same as `@cov` (p. 616). See also `@covs` (p. 617). See `Group::cov` (p. 271) in the *Object Reference* for more general computation of covariances.

@covs	Matrix Descriptive Statistics Functions

Compute covariances using $n-1$ as the divisor in the moment calculation. See also `@cov` (p. 616) or `@covp` (p. 617). See also `@covs` (p. 617). See `Group::cov` (p. 271) in the *Object Reference* for more general computation of covariances.

@csum	Matrix Column Functions

Syntax:	@csum(m)
Argument:	matrix, m
Return:	vector

Returns a vector containing the summation of the rows in each column of the matrix m. Example:

```
vector v1= @csum(m1)
```

See also `@cmean` (p. 613).

@det	Matrix Algebra Functions

Syntax: @det(*m*)
Argument: matrix or sym, *m*
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 sc1 = @det(m1)
vec4(2) = @det(s2)
```

See also @rank (p. 637).

@ediv	Matrix Element Functions

Syntax: @ediv(*m1*,*m2*)
Argument 1: matrix, *m1*
Argument 2: matrix, *m2*
Return: matrix

Returns the element by element division of two matrix objects or syms. Each element of the returned matrix is equal to the corresponding element in *m1* divided by the corresponding element in *m2*. Note *m1* and *m2* must be of identical dimensions. Example:

```
matrix m3 = @ediv(m1,m2)
```

See also @einv (p. 621), @emult (p. 622), and @epow (p. 623).

@eeq	Matrix Element Functions

Syntax: @eeq(*m1*,*m2*)
Argument 1: matrix, *m1*
Argument 2: matrix, *m2*
Return: matrix

Returns the element by element test of equality between two matrix objects or syms. Each element of the returned matrix is equal to 1 or 0 depending on whether the corresponding element in *m1* is equal to the corresponding element in *m2*. Note *m1* and *m2* must be of identical dimensions. Example:

```
matrix m3 = @eeq(m1,m2)
```

See also @eneq (p. 622), @eeqna (p. 619), and @eneqna (p. 622).

@eeqna	Matrix Element Functions

Syntax: @eeqna(*m1*,*m2*)
Argument 1: matrix, *m1*
Argument 2: matrix, *m2*
Return: matrix

Returns the element by element test of equality between two matrix objects or syms. NAs are treated as ordinary values for purposes of comparison. Each element of the returned matrix is equal to 1 or 0 depending on whether the corresponding element in *m1* is equal to the corresponding element in *m2*. Note *m1* and *m2* must be of identical dimensions. Example:

```
matrix m3 = @eeqna(m1,m2)
```

See also @eneqna (p. 622), @eeq (p. 618), and @eneq (p. 622).

@ege	Matrix Element Functions

Syntax: @ege(*m1*,*m2*)
Argument 1: matrix, *m1*
Argument 2: matrix, *m2*
Return: matrix

Returns a test for whether the elements in the matrix or sym *m1* are greater than or equal to the corresponding elements in *m2*. Each element of the returned matrix is equal to 1 or 0 depending on the outcome of the comparison. Note *m1* and *m2* must be of identical dimensions. Example:

```
matrix m3 = @egt(m1,m2)
```

See also @egt (p. 619), @ele (p. 621), and @elt (p. 621).

@egt	Matrix Element Functions

Syntax: @egt(*m1*,*m2*)
Argument 1: matrix, *m1*
Argument 2: matrix, *m2*
Return: matrix

Returns a test for whether the elements in the matrix or sym *m1* are greater than the corresponding elements in *m2*. Each element of the returned matrix is equal to 1 or 0 depending on the outcome of the comparison. Note *m1* and *m2* must be of identical dimensions. Example:

```
matrix m3 = @egt(m1,m2)
```

See also `@ege` (p. 619), `@ele` (p. 621), and `@elt` (p. 621).

@eigenvalues	Matrix Algebra Functions

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 (`@eigenvectors` (p. 620)). The eigenvalues are arranged in ascending order.

Example:

```
vector v1 = @eigenvalues(s1)
```

@eigenvectors	Matrix Algebra Functions

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. 620).

@einv	Matrix Element Functions

Syntax: @einv(*m*)
Argument: matrix,*m*
Return: matrix

Returns the element by element inverse of a matrix object or sym. Each element of the returned matrix is equal to 1 divided by the corresponding element of the input matrix. Example:

```
matrix m2 = @einv(m1)
```

See also `@ediv` (p. 618), `@emult` (p. 622), and `@epow` (p. 623).

@ele	Matrix Element Functions

Syntax: @ege(*m1*,*m2*)
Argument 1: matrix, *m1*
Argument 2: matrix, *m2*
Return: matrix

Returns a test for whether the elements in the matrix or sym *m1* are greater than or equal to the corresponding elements in *m2*. Each element of the returned matrix is equal to 1 or 0 depending on the outcome of the comparison. Note *m1* and *m2* must be of identical dimensions. Example:

```
matrix m3 = @ele(m1,m2)
```

See also `@elt` (p. 621), `@ege` (p. 619), and `@egt` (p. 619).

@elt	Matrix Element Functions

Syntax: @egt(*m1*,*m2*)
Argument 1: matrix, *m1*
Argument 2: matrix, *m2*
Return: matrix

Returns a test for whether the elements in the matrix or sym *m1* are greater than the corresponding elements in *m2*. Each element of the returned matrix is equal to 1 or 0 depending on the outcome of the comparison. Note *m1* and *m2* must be of identical dimensions. Example:

```
matrix m3 = @elt(m1,m2)
```

See also @ele (p. 621), @ege (p. 619), and @egt (p. 619).

@emult	Matrix Element Functions

Syntax: @emult(*m1*,*m2*)
Argument 1: matrix, *m1*
Argument 2: matrix, *m2*
Return: matrix

Returns the element by element multiplication of two matrix objects or syms. Each element of the returned matrix is equal to the corresponding element in *m1* multiplied by the corresponding element in *m2*. Note *m1* and *m2* must be of identical dimensions. Example:

```
matrix m3 = @emult(m1,m2)
```

See also @ediv (p. 618), @einv (p. 621), and @epow (p. 623).

@eneq	Matrix Element Functions

Syntax: @eneq(*m1*,*m2*)
Argument 1: matrix, *m1*
Argument 2: matrix, *m2*
Return: matrix

Returns the element by element test of inequality between two matrix objects or syms. Each element of the returned matrix is equal to 1 or 0 depending on whether the corresponding element in *m1* is not equal to the corresponding element in *m2*. Note *m1* and *m2* must be of identical dimensions. Example:

```
matrix m3 = @eneq(m1,m2)
```

See also @eeq (p. 618), @eeqna (p. 619), and @eneqna (p. 622).

@eneqna	Matrix Element Functions

Syntax: @eneqna(*m1*,*m2*)
Argument 1: matrix, *m1*
Argument 2: matrix, *m2*
Return: matrix

Returns the element by element test of inequality between two matrix objects or syms. NAs are treated as ordinary values for purposes of comparison. Each element of the returned matrix is equal to 1 or 0 depending on whether the corresponding element in *m1* is not equal to the corresponding element in *m2*. Note *m1* and *m2* must be of identical dimensions. Example:

```
matrix m3 = @eneqna(m1,m2)
```

See also @eeqna (p. 619), @eeq (p. 618), and @eneq (p. 622).

@epow	Matrix Element Functions

Syntax: @epow(*m1*,*o*)
Argument 1: matrix, *m1*
Argument 2: matrix, *m2,* scalar or number, *n*
Return: matrix

Returns a matrix where every element is equal to the corresponding element in *m1* raised to the power *n*. If the second argument is a matrix, the each element of *m1* will be raised to the power of the corresponding element of *m2.* Example:

```
matrix m2 = @epow(m1,3)
```

See also @ediv (p. 618), @einv (p. 621), and @emult (p. 622).

@explode	Matrix Utility Functions

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. 628).

@fill	Matrix Algebra Functions

Syntax: @fill(*n1*, *n2*, *n3*, ...)
Arguments: scalar
Return: vector

Returns a vector containing the elements specified by the arguments to the function. The vector will have length equal to the number of arguments. The maximum number of arguments is 96.

Example

```
vector v = @fill(1,4,6,21.3)
```

Will return a 4 element vector, where the first element is set to 1, the second to 4, the third to 6 and the fourth to 21.3.

See Vector::fill (p. 786), Coef::fill (p. 20), Matrix::fill (p. 350), Rowvector::fill (p. 454), and Sym::fill (p. 639) for routines to perform general filling of matrix objects.

@filledmatrix

Matrix Utility Functions

Syntax: @filledmatrix(*n1*, *n2*, *n3*)
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 Matrix::fill (p. 350).

See Coef::fill (p. 20), Rowvector::fill (p. 454), Series::fill (p. 498), Sym::fill (p. 639), and Vector::fill (p. 786) for routines to perform general filling of matrix objects.

@filledrowvector

Matrix Utility Functions

Syntax: @filledrowvector(*n1*, *n2*)
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. 623).

See Coef::fill (p. 20), Matrix::fill (p. 350), Sym::fill (p. 639), and Vector::fill (p. 786) for routines to perform general filling of matrix objects.

@filledsym — Matrix Utility Functions

Syntax: @filledsym(*n1*, *n2*)
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 Sym::fill (p. 639).

See Coef::fill (p. 20), Matrix::fill (p. 350), Rowvector::fill (p. 454), and Vector::fill (p. 786) for routines to perform general filling of matrix objects.

@filledvector — Matrix Utility Functions

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. 623).

See Vector::fill (p. 786), Coef::fill (p. 20), Matrix::fill (p. 350), Rowvector::fill (p. 454), and Sym::fill (p. 639) for routines to perform general filling of matrix objects.

@first — Matrix Utility Functions

Syntax: @first(*o*)
Argument: vector or series, *o*
Return: scalar

Returns a vector containing the first non-missing values of the series or vector. The series version uses the current workfile sample. Example:

```
scalar s1 = @first(ser1)
```

See also @ifirst (p. 627), @cfirst (p. 610).

@getmaindiagonal — Matrix Utility Functions

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)
```

@hcat — Matrix Algebra Functions

Syntax: @hcat(*m1*, *m2*)
Argument 1: matrix, vector or sym
Argument 2: matrix, vector or sym
Return: matrix

@hcat performs horizontal concatenation of two matrix objects. *m1* and *m2* must have the same number of rows. If *m1* is a matrix with k rows and m columns, and *m2* is a matrix with k rows and n columns, then @hcat will return a matrix with k rows and $(m+n)$ columns.

See also @vcat (p. 646).

@identity — Matrix Utility Functions

Syntax: @identity(*n*)
Argument: integer, *n*
Return: matrix

Returns a square $n \times n$ identity matrix. Example:

```
matrix i1 = @identity(4)
```

@ifirst	Matrix Utility Functions

Syntax: @ifirst(*o*)
Argument: vector or series, *o*
Return: scalar

Returns a scalar containing the index of first non-missing values the series or vector. The series version uses the current workfile sample. Example:

```
scalar s1 = @ifirst(ser1)
```

See also `@first` (p. 625), `@cifirst` (p. 611).

@ilast	Matrix Utility Functions

Syntax: @ilast(*o*)
Argument: vector or series, *o*
Return: scalar

Returns a vector containing the first non-missing values the series or vector. The series version uses the current workfile sample. Example:

```
scalar s1 = @ilast(ser1)
```

See also `@last` (p. 630), `@cilast` (p. 611).

@imax	Matrix Utility Functions

Syntax: @imax(*o*)
Argument: vector or series, *o*
Return: scalar

Returns a scalar containing the index of the maximum of the series or vector. The series version uses the current workfile sample. Example:

```
scalar s1 = @imax(ser1)
```

@imin	Matrix Utility Functions

Syntax: @imin(*o*)
Argument: vector or series, *o*
Return: scalar

Returns a scalar containing the index of the minimum of the series or vector. The series version uses the current workfile sample. Example:

```
scalar s1 = @imin(ser1)
```

@implode	Matrix Utility Functions

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. 623).

@inner	Matrix Algebra Functions

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: sym

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. 635).

@inverse — Matrix Algebra Functions

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. 641).

@issingular	Matrix Algebra Functions

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 sc1 = @issingular(m1)
```

@kronecker	Matrix Algebra Functions

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)
```

@last	Matrix Utility Functions

Syntax: @last(*o*)
Argument: vector or series, *o*
Return: scalar

Returns a vector containing the last non-missing value of the series or vector. The series version uses the current workfile sample. Example:

```
scalar s1 = @first(ser1)
```

See also `@ilast` (p. 627), `@clast` (p. 612).

@makediagonal	Matrix Utility Functions

Syntax: @makediagonal(*v*, *k*)
Argument 1: vector or rowvector, *v*
Argument 2: (optional) integer, *k*
Return: sym or 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 sym 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*.

Examples:

```
sym s1 = @makediagonal(v1)
matrix m2 = @makediagonal(v1,1)
matrix m4 = @makediagonal(r1,-3)
```

S1 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)
```

Note that to make a diagonal matrix with the same elements on the diagonal, you may use `@identity`, multiplied by the scalar value.

See also `@identity` (p. 626).

matplace	Matrix Utility Commands

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)
```

@mnrnd	Matrix Utility Functions

Syntax: @mnrnd(*n1*,*n2*)
Argument 1: integer, *n1*
Argument 2: integer, *n2*
Return: vector, matrix or sym

Creates a vector, matrix or sym filled with normal random numbers. The size of the created matrix is given by the integers *n1* (number of rows) and *n2* (number of columns). Examples:

```
matrix m1=@mnrnd(3,2)
sym s1=@mnrnd(5,5)
vector v1=@mnrnd(18)
```

See also `@mrnd` (p. 632), `nrnd` (p. 634), and `rnd` (p. 638).

@mrnd	Matrix Utility Functions

Syntax: @mrnd(*n1*,*n2*)
Argument 1: integer, *n1*
Argument 2: integer, *n2*
Return: vector, matrix or sym

Creates a vector, matrix or sym filled with uniform random numbers. The size of the created matrix is given by the integers *n1* (number of rows) and *n2* (number of columns). Examples:

```
matrix m1=@mrnd(3,2)
sym s1=@mrnd(5,5)
vector v1=@mrnd(18)
```

See also `@mnrnd` (p. 632), `nrnd` (p. 634), and `rnd` (p. 638).

mtos	Matrix Utility Commands

Convert matrix to a series or group. Fills a series or group with the data from a vector or matrix.

Syntax

Vector Proc: **mtos**(vector, series*[, sample]*)

Matrix Proc: **mtos**(matrix, group*[, sample]*)

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. Example:

```
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 11. "Matrix Language," on page 239 for further discussion and examples of the use of matrices.

See also stom (p. 642), stomna (p. 642), and @convert (p. 615).

@norm	Matrix Algebra Functions

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)
```

nrnd	Matrix Utility Commands

Syntax: nrnd(*m*)
Argument 1: matrix, vector or sym, *m*

Fill the matrix object *m* with normal random numbers.

Example:

```
nrnd(m1)
```

See also @mnrnd (p. 632), @mrnd (p. 632), and rnd (p. 638).

@ones	Matrix Utility Functions

Syntax: @ones(*n1*,*n2*)
Argument 1: integer, *n1*
Argument 2: integer, *n2*
Return: vector, matrix or sym

Creates a vector, matrix or sym filled with the value 1. The size of the created matrix is given by the integers *n1* (number of rows) and *n2* (number of columns). Example:

```
matrix m1=@ones(3,2)
sym s1=@ones(5,5)
vector v1=@ones(18)
```

@outer	Matrix Algebra Functions

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. 628).

@permute	Matrix Utility Functions

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. 638).

See also `@capplyranks` (p. 609) and `@rapplyranks` (p. 637).

@pinverse	Matrix Algebra Functions

Syntax: @inverse(*m*)
Argument: matrix or sym, *m*
Return: matrix or sym

Returns the Moore-Penrose pseudo-inverse of a matrix object or sym. The pseudoinverse has the property that both pre-multiplying and post-multiplying the pseudo-inverse by the source matrix returns the source matrix, and that both pre-multiplying and post-multiplying the source matrix by the pseudo-inverse will return the inverse. The pseudo-inverse of a matrix returns a matrix, while the pseudo-inverse of a sym returns a sym. Note that pseudo-inverting a sym is much faster than inverting a matrix.

Examples:

```
matrix m2 = @pinverse(m1)
sym s2 = @pinverse(s1)
sym s3 = @pinverse(@implode(m2))
```

@qform	Matrix Algebra Functions

Syntax: @qform(*s*, *o*)
Argument 1: sym
Argument 2: vector
Return: scalar

Syntax: @qform(*s*, *o*)
Argument 1: sym
Argument 2: matrix or sym
Return: sym

`@qform` returns the quadratic form of a symmetric matrix with another matrix, or vector. If S is the symmetric matrix, and v is a vector, `@qform` will return v'S*v, which results in a scalar. If S is symmetric matrix, and m is a matrix, then `@qform` returns m'S*m, which is a symmetric matrix.

@rank	Matrix Algebra Functions

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. 643).

(You may use the `@ranks` function to obtain a ranking of the elements of a vector. See "Descriptive Statistics" on page 508.)

@rapplyranks	Matrix Utility Functions

Syntax: @rapplyranks(*m*, *v[, n]*)
Argument 1: vector, matrix, or sym, *m*
Argument 2: vector, *v*
Argument 3: (optional) integer, *n*
Return: vector, matrix, or sym

Reorder the columns of a matrix *m* using the ranks in the vector *v*. If the optional argument *n* is specified, only the columns in row *n* will be reordered. *v* should contain unique integers from 1 to the number of rows of *m*.

```
matrix m2 = @rapplyranks(m1, v1)
```

reorders the columns of the matrix M1 using the ranks in V1, while

```
matrix m3 = @rappyranks(m1, v1, 3)
```

reorders only the columns in row 3 of M1.

Note that you may use the `@ranks` function to obtain the ranks of a vector. Obtaining unique integer ranking for data with ties requires use of the "i" or "r" option in `@ranks`, as in

```
vector y = @ranks(x, "a", "i")
```

See "Descriptive Statistics" on page 508.

See also @capplyranks (p. 609) and @permute (p. 635).

@resample	Matrix Utility Functions

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. 635).

rnd	Matrix Utility Commands

Syntax: rnd(*m*)
Argument 1: matrix, vector or sym, *m*

Fill the matrix object *m* with uniform random numbers.

Example:

```
rnd(m1)
```

See also @mnrnd (p. 632), @mrnd (p. 632), and nrnd (p. 634).

@rowextract	Matrix Utility Functions

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. 614).

rowplace	Matrix Utility Commands

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. 614).

@rows	Matrix Utility Functions

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 sc1=@rows(m1)
scalar size=@rows(m1)*@columns(m1)
```

For series and groups `@rows` (p. 639) returns the number of observations in the workfile range. See also `@columns` (p. 614).

@scale	Matrix Utility Functions

Syntax: @scale(*m, v[,p]*)
Argument 1: matrix or sym, *m*
Argument 2: vector or rowvector, *v*
Argument 3: (optional) number, *p*
Return: matrix or sym

Scale the rows *or* columns of a matrix, or the rows *and* columns of a sym matrix.

- If *m* is a matrix and *v* is a vector, the i-th row of *m* will be scaled by the i-th element of *v*, optionally raised to the power *p* (row scaling). The length *v* must equal the number of rows of *m*.
- If *m* is a matrix and *v* is a rowvector, the i-th column of *m* will be scaled by the i-th element of *v*, optionally raised to the power *p* (column scaling). The length *v* must equal the number of columns of *m*.
- If *m* is a sym object, then *v* may either be a vector or a rowvector. The (i,j)-th element of *m* will be scaled by both the i-th and j-th elements of *v* (row and column scaling). The length *v* must equal the number of rows (and columns) of *m*.

Let M be the matrix object, V be the vector or rowvector, and $\Lambda = \mathrm{diag}(V_1, V_{2,\ldots}, V_p)$ be the diagonal matrix formed from the elements of V. Then `@scale(m, v, p)` returns:

- $\Lambda^p M$, if M is a matrix and V is a vector.
- $M\Lambda^p$, if M is a matrix and V is a rowvector.
- $\Lambda^p M \Lambda^p$, if M is a sym matrix.

Example:

```
sym covmat = @cov(grp1)
vector vars = @getmaindiagonal(covmat)
sym corrmat = @scale(covmat, vars, -0.5)
```

computes the covariance matrix for the series in the group object GRP1, extracts the variances to a vector VARS, then uses VARS to obtain a correlation matrix from the covariance matrix.

```
matrix covmat1 = covmat
matrix rowscaled = @scale(covmat, vars, -0.5)
matrix colscaled = @scale(covmat, @transpose(vars), -0.5)
```

```
sym corrmat1 = colscaled
```

performs the same scaling in multiple steps. Note that the COLSCALED matrix is square and symmetric, but not a sym object.

@solvesystem	Matrix Algebra Functions

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. 629).

@sort	Matrix Utility Functions

Syntax: @sort(*v*, *o*, *t*)
Argument 1: vector
Argument 2: (optional) option, *o*
Argument 3: (optional) option, *t*
Return: vector

Returns the sorted elements of the matrix or vector object *o*. The order of sorting is set using o: “a” (ascending) or “d” (descending). Ties are broken according to the setting of *t*: “i” (ignore), “f” (first), “l” (last), “a” (average), “r” randomize. The defaults are “a”, and “i”, respectively.

Note that sorting a matrix sorts every element of the matrix and arranges them by column, with the first element in the first row of the first column, and the last element in the last row of the last column.

Example:

```
vector rank1 = @sort(v1,"a","f")
```

stom	Matrix Utility Commands

Syntax: stom(*o1*, *o2*, *smp*)
Argument 1: series or group, *o1*
Argument 2: vector or matrix, *o2*
Argument 3: (optional) sample *smp*

*S*eries-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. 642). See also and `@convert` (p. 615).

stomna	Matrix Utility Commands

Syntax: stomna(*o1*, *o2*, *smp*)
Argument 1: series or group, *o1*
Argument 2: vector or matrix, *o2*
Argument 3: (optional) sample *smp*

*S*eries-TO-Matrix Object with *NA*s. 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(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. 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. 615) and `stom` (p. 642).

@subextract — Matrix Utility Functions

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 Functions

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:

$$\begin{aligned} m1 &= UWV' \\ U'U &= V'V = I \end{aligned} \tag{18.1}$$

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. 637) function for a related discussion.

Examples:

```
matrix m2
vector v1
matrix m3 = @svd(m1,v1,m2)
```

@trace	Matrix Algebra Functions

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 sc1 = @trace(m1)
```

@transpose	Matrix Algebra Functions

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)
```

@uniquevals

Matrix Utility Functions

Syntax: **@uniquevals**(*arg*)
Argument: series, alpha, vector or matrix
Return: vector or svector

Returns a vector or svector containing the list of unique values in the object specified by *arg*. If *arg* is a series, vector or matrix, `@uniquevals` will return a vector object containing the unique elements of the series, vector or matrix. If *arg* is an alpha series, `@uniquevals` will return an svector of the unique values in the alpha.

Examples

```
@uniquevals(X)
```

returns a vector containing the unique values in X.

@unitvector

Matrix Utility Functions

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 instead you wish to create a 10-element vector of ones, you should use a declaration statement of the form:

```
vector v1=@ones(10)
```

See also `@ones` (p. 634).

@unvec

Matrix Utility Functions

Syntax: @unvec(*v*, *n*)
Argument 1: vector, *v*
Argument 2: integer, *n*
Return: matrix

Creates an *n*-row matrix filled with the unstacked elements of the vector *v*. EViews will report a size mismatch if the number of elements of *v* is not evenly divisible by *n*.

Note that `@unvec` is the inverse of the `@vec` function.

Example:

```
vector v = @mrnd(12)
matrix m1 = @unvec(v1, 3)
```

creates a 12 element vector of uniform random numbers V1 and unstacks it into a 3×4 matrix M1.

See also `@vec` (p. 647).

@unvech	Matrix Utility Functions

Syntax: @unvec(*v*)
Argument: vector, *v*
Return: sym

Creates a sym matrix with lower triangle filled using the unstacked elements of the vector *v*. The sym will be sized automatically. EViews will report a size mismatch if the number of elements of *v* does not correspond to a valid sym matrix size (is not a triangular number).

Note that `@unvech` is the inverse of the `@vech` function.

Example:

```
vector v1 = @mnrnd(15)
sym s1 = @unvech(v1)
```

creates a 15 element vector of normal random numbers V1 and unstacks it into a 5×5 sym matrix S1.

See also `@vech` (p. 647).

@vcat	Matrix Algebra Functions

Syntax: @vcat(*m1*, *m2*)
Argument 1: matrix, vector or sym
Argument 2: matrix, vector or sym
Return: matrix

`@vcat` performs vertical concatenation of two matrix objects. *m1* and *m2* must have the same number of columns. If *m1* is a matrix with *m* rows and *k* columns, and *m2* is a matrix with *n* rows and *k* columns, then `@vcat` will return a matrix with $(m+n)$ rows and *k* columns.

See also `@hcat` (p. 626).

@vec	Matrix Utility Functions

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:

```
matrix m1 = @mrnd(10, 3)
vector v1 = @vec(m1)
```

creates a 10×3 matrix of uniform random numbers M1 and stacks it in a 30 element vector V1.

See also `@unvec` (p. 645).

@vech	Matrix Utility Functions

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:

```
sym s1 = @mrnd(5,5)
vector v1 = @vech(m1)
```

creates a 5×5 sym matrix S1 and stacks its lower triangle in a 15 element vector V1.

See also `@unvech` (p. 646).

Chapter 19. 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 105.

Programming Summary

Program Statements

Support Commands

Support Functions

There is also a set of functions that may be used to obtain information about the active workfile. See "Basic Workfile Functions" on page 530.

Dialog Display Functions

Programming Language Entries

The following section provides an alphabetical listing of the commands and functions associated with the EViews programming language. Each entry outlines the basic syntax and provides examples and cross references.

addin	Support Commands

Register a program file as an EViews Add-in.

Syntax

addin*(options) [path\]prog_name*

registers the specified program file as an EViews Add-in. Note that the program file should have a ".PRG" extension, which you need not specify in the *prog_name*.

If you do not provide the optional path specification, EViews looks for the program file in the default EViews Add-ins directory.

Explicit path specifications containing ".\" and "..\" (to indicate the current level and one directory level up) are evaluated relative the directory of the installer program in which the `addin` command is specified, or the EViews default directory if `addin` is run from the command line.

You may use the special " < addins > "directory keyword in your path specification.

Options

type = *arg*	Specify the Add-ins type, where *arg* is the name of a EViews object type. The *default* is to create a global Add-in. Specifying an object-specific Add-in using a matrix object as in "type = matrix", "type = vector", etc. will register the Add-in for all matrix object types (including coef, rowvector, and sym objects). Sample objects do not support object-specific Add-ins so that "type = sample" is not allowed.
proc = *arg*	User--defined command/procedure name. If omitted, the Add-in will not have a command form.
menu = *arg*	Text for the Add-in menu entry. If omitted, the Add-in will not have an associated menu item. Note that you may use the "&" symbol in the entry text to indicate that the following character should be used as a menu shortcut.
desc = *arg*	Brief description of the Add-in that will be displayed in the Add-ins management dialog.
docs = *arg*	Path and filename for the Add-in documentation. Determination of the path follows the rules specified above for the `addin` filename.

Examples

```
addin(proc="myaddin", desc="This is my add-in") .\myaddin.prg
```

registers the file "Myaddin.prg" as a global Add-in, with the user-defined global command `myaddin`, no menu support, and no assigned documentation file. The description "This is my add-in" will appear in the main Add-ins management dialog. Note that the ".\" indicates the directory from which the program containing the `addin` command was run, or the EViews default directory if `addin` is run interactively.

```
addin(type="graph", menu="Add US Recession Shading",
    proc="recshade", docs=".\recession shade.txt", desc="Applies US
    recession shading to a graph object.") .\recshade.prg
```

registers the file "Recshade.prg" as a graph specific Add-in. The Add-in supports the object-command `recshade`, has an object-specific menu item "Add US Recession Shading", and has a documentation file "Recession shade.txt".

```
addin(type=equation, menu="Simple rolling regression", proc=roll,
    docs="<addins>\Roll\Roll.pdf", desc="Rolling Regression -
    simple version") "<addins>\Roll\roll.prg"
```

registers the Add-in file "Roll.prg" as an equation specific Add-in. Note that the documentation and program files are located in the "Roll" subdirectory of the default Add-ins directory.

Cross-references

See "Registering an Add-in" on page 184 in the *Command and Programming Reference* for further details.

@addinspath	Support Functions

Syntax: **@addinspath**

Return: string

Returns a string containing the EViews add-ins directory path.

Examples

If your currently add-ins path is "d:\eviews\add_ins", then

```
%y = @addinspath
```

assigns a string of the form "D:\EVIEWS\ADD_INS".

Cross-references

See also `@evpath` (p. 657) and `@temppath` (p. 669).

call	Program Statements

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 139. See also `subroutine` (p. 668), `endsub` (p. 655).

clearerrs	Support Commands

Set the current error count to 0.

May only be used in programs.

See also `@errorcount` (p. 657), `seterrcount` (p. 666), `seterr` (p. 665), and `setmaxerrs` (p. 666).

@date	Support Functions

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. 670).

else	Program Statements

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 127. See also, `if` (p. 660), `endif` (p. 655), `then` (p. 669).

endif	Program Statements

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 127. See also, `if` (p. 660), `else` (p. 654), `then` (p. 669).

endsub	Program Statements

Mark the end of a subroutine.

Syntax

subroutine name*(arguments)*

commands

endsub

Cross-references

See "Defining Subroutines," beginning on page 137. See also, `subroutine` (p. 668), `return` (p. 665).

@env	Support Functions

Syntax: **@env**(*str*)
Argument: string, *str*
Return: string

Returns a string containing the value of the Windows environment variable given by *str*. For a list of common Windows environment variables, see:

```
http://en.wikipedia.org/wiki/
   Environment_variable#Examples_from_Microsoft_Windows
```

or:

```
http://www.microsoft.com/resources/documentation/windows/xp/all/
   proddocs/en-us/ntcmds_shelloverview.mspx?mfr=true
```

Examples

```
@env("username")
```

returns the user-name of the current logged in user.

```
@env("computername")
```

returns the name of the computer.

```
@env("userpath")
```

returns the location of the current logged in user's user-folder on disk.

@equaloption	Support Functions

Syntax: **@equaloption**(*"option_keyword"*)
Syntax: **@equaloption**(*string*)
Return: string

returns the text to the right of the "*option_keyword* = " option provided in the `exec` (p. 328) or `run` (p. 427) command. If the option keyword is not found, the function will return an empty string. For example, if you specify the option "kernel = tri" in your `exec` command,

```
string opt = @equaloption("kernel")
```

will return the string "TRI".

Cross-references

See `exec` (p. 328) and `run` (p. 665).

@errorcount	Support Functions

Syntax: **@errorcount**
Argument: none
Return: integer

Number of errors encountered in a program. Returns a scalar containing the number of errors encountered during program execution.

Cross-references

See also `@maxerrcount` (p. 663), `setmaxerrs` (p. 666), `clearerrs` (p. 654), `seterr` (p. 665), and `seterrcount` (p. 666).

@evpath	Support Functions

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` (p. 285), `@addinspath` (p. 653), and `@temppath` (p. 669).

exec	Support Commands

Execute a program. The `exec` command executes a program. The program may be located in memory or stored in a program file on disk.

See `exec` (p. 328).

exitloop	Program Statements

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 129. See also, `stop` (p. 668), `return` (p. 665), `for` (p. 659), `next` (p. 663), `step` (p. 667).

@fileexist	Support Functions

Syntax: **@fileexist**(*str*)
Argument: string, *str*
Return: integer

Check for an file's existence. *str* should contain the full path and file name of the file you want to check for. Returns a "1" if the file exists, and a "0" if it does not exist.

Cross-references

See also `@wdir` (p. 678) and `@folderexist` (p. 658)

@folderexist	Support Functions

Syntax: **@folderexist**(*str*)
Argument: string, *str*
Return: integer

Check for a folder's existence. *str* should contain the full path of the folder you want to check for. Returns a "1" if the file exists, and a "0" if it does not exist.

Cross-references

See also `@wdir` (p. 678) and `@fileexist` (p. 658).

for	Program Statements

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 129. See also, `exitloop` (p. 658), `next` (p. 663), `step` (p. 667).

@getnextname	Support Functions

Syntax: **@getnextname**(*str*)
Argument: string, *str*
Return: string, *name*

Returns a string containing the next available variable name in the workfile, starting with *str* (*i.e.* entering "result" will return "RESULT01" unless there is already a RESULT01, in which case it will return "RESULT02").

Cross-references

See also `@isobject` (p. 661) and `@makevalidname` (p. 662).

@getthistype	Support Functions

Syntax: **@getthistype**
Return: string

Returns the object type of the `_this` object.

If no workfile is open, or if no object has yet been opened in a workfile, the function will return the string "NONE". Note this latter behavior is in contrast to using the data member syntax "`_this.@type`", which will error on those cases.

Cross-references

See also the `@type` data member of each object in Chapter 1. "Object View and Procedure Reference," beginning on page 2. See "The Active Object Keyword" on page 191 in the *Command and Programming Reference* for a discussion of the `_this` object.

@hasoption	Support Functions

Syntax: **@hasoption***("option")*
Syntax: **@hasoption***(string)*
Return: 0 or 1

returns 1 or 0 depending on whether the specified option is or is not in the program options provided in the `exec` (p. 328) or `run` (p. 427) command.

Cross-references

See `exec` (p. 328) and `run` (p. 665).

if	Program Statements

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 127. See also `else` (p. 654), `endif` (p. 655), `then` (p. 669).

include	Program Statements

Include another file in a program.

The `include` statement is used to include the contents of another file in a program file.

If an include file is specified without an absolute path, the base location will be taken from the location of the program file, not from the default directory.

Syntax

include *filename*

Cross-references

See "Multiple Program Files" on page 136. See also `call` (p. 654).

@isobject	Support Functions

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.

Cross-references

See also `@getnextname` (p. 659).

@isvalidname	Support Functions

Syntax: **@isvalidname**(*str*)
Argument: string, *str*
Return: integer

Check for whether a string represents a valid name for an EViews object. Returns a "1" if the name is valid, and a "0" if it is not.

Cross-references

See also `@makevalidname` (p. 662).

@lasterrnum	Support Functions

Syntax: **@lasterrnum**
Return: integer

Returns a integer containing the error number for the previously issued command. If there the previous program line did not generate an error, the result will be a 0.

May only be used in a program.

Cross-references

See also `@lasterrstr` (p. 662).

@lasterrstr	Support Functions

Syntax: **@lasterrstr**
Return: string

Returns a string containing the error message for the previously issued command. If there the previous program line did not generate an error, the result will be an empty string.

May only be used in a program.

Cross-references

See also `@lasterrnum` (p. 662).

@makevalidname	Support Functions

Syntax: **@makevalidname**(*str*)
Argument: string, *str*
Return: string, *name*

Returns a string containing an uppercased valid EViews name based on *str*. If *str* is a valid name, then the original string *str* is returned. If *str* is not valid, invalid characters will be replaced in the new string with "_" prior to the return (*i.e.* the string "re!sult%" will return "RE_SULT_").

Cross-references

See also `@isvalidname` (p. 661) and `@getnextname` (p. 659).

@maxerrcount	Support Functions

Syntax: **@maxerrcount**
Return: integer

Returns an integer containing the current value of the maximum number of errors that a program may encounter before execution is halted.

May only be used in a program.

Cross-references

See also `@errorcount` (p. 657), `setmaxerrs` (p. 666), `clearerrs` (p. 654), `seterr` (p. 665), and `seterrcount` (p. 666).

next	Program Statements

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 129. See also, `exitloop` (p. 658), `for` (p. 659), `step` (p. 667).

open	Support Commands

Open a file. Opens a workfile, database, program file, or ASCII text file.

See `open` (p. 378).

@option	Support Functions

Syntax: **@option***(i), integer*
Return: string

returns the i-th option provided in the `exec` (p. 328) or `run` (p. 427) command.

Cross-references

See `exec` (p. 328) and `run` (p. 665).

output	Support Commands

Redirects printer output or display estimation output.

See `output` (p. 387).

poff	Program Statements

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 779 of the *User's Guide I* for a discussion of printer control.

See also `pon` (p. 664).

pon	Program Statements

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. 387)). 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 779 of the *User's Guide I* for a discussion of printer control.

See also `poff` (p. 664).

program	Support Commands

Create a program.

See `program` (p. 415).

return	Program Statements

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 137. See also `exitloop` (p. 658), `stop` (p. 668).

run	Support Commands

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. 427). See also `exec` (p. 328).

seterr	Support Commands

Set a user-specified execution error.

Syntax

seterr *string*

sets an execution error using the specified string. May only be used in programs.

See also `@errorcount` (p. 657), `@maxerrcount` (p. 663), `clearerrs` (p. 654), and `setmaxerrs` (p. 666).

seterrcount	Support Commands

Set the current program execution error count.

Syntax

seterrcount *integer*

sets the current error count to the specified integer value. May only be used in programs.

See also `@errorcount` (p. 657), `@maxerrcount` (p. 663), `clearerrs` (p. 654), `seterr` (p. 665), and `setmaxerrs` (p. 666).

setmaxerrs	Support Commands

Set the maximum number of errors that a program may encounter before execution is halted.

Syntax

setmaxerrs *integer*

sets the maximum number of errors to the specified integer value.

See also `@errorcount` (p. 657), `@maxerrcount` (p. 663), `clearerrs` (p. 654), and `seterrcount` (p. 666).

statusline	Support Commands

Send a text message to the EViews status line.

Syntax

statusline *string*

Example

```
for !i = 1 to 10
    statusline Iteration !i
next
```

sleep	Program Statements

Pause program execution.

Syntax

sleep *n*

pauses a program by *n* milliseconds (default is 5000).

Example

```
sleep 1000
```

step	Program Statements

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, for 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 129. See also exitloop (p. 658), for (p. 659), next (p. 663).

stop	Program Statements

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. 658), return (p. 665).

subroutine	Program Statements

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 137. See also endsub (p. 655).

@tablenames	Support Functions

Syntax: **@tablenames**("*str*")
Argument: string, *str*
Return: string, *names*

Returns a space delimited string containing the names of the table names of a foreign file. The name (and path) of the foreign file should be specified in *str*, enclosed in quotes. For an Excel file, the names of the Excel sheets will be returned.

@temppath	Support Functions

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 `@evpath` (p. 657) and `@addinspath` (p. 653).

then	Program Statements

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 127. See also, else (p. 654), endif (p. 655), if (p. 660).

@time	Support Functions

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. 654).

to	Expression || Program Statements

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 129.

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 129. See also, exitloop (p. 658), for (p. 659), next (p. 663).

@toc	Support Functions

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. 451) and toc (p. 452).

@uidialog	Dialog Display Functions

Syntax: **@uidialog**(*control1_info[, control2_info, control3_info, ...]*)
Arguments: string or scalar parameters, based on control type
Return: integer

Displays a dialog with any number of controls that you define, including edit fields, list-boxes, radio buttons, checkboxes, text, and captions. You may specify an combination of controls, and they will be arranged in the order they are entered. EViews attempts to lay the controls out in a visually pleasing manner, and will do its best to size and arrange the dialog accordingly.

The dialog shows an **OK** and **Cancel** button, and will return an integer representing the button clicked: Cancel (-1), OK (0).

Each *control_info* item is a specification of the control type in quotes ("Edit"), followed by the appropriate parameters. You may specify any combination of controls, where each should follow one of the following forms:

Edit field	"Edit", *IO_String, prompt_string[, max_edit_length]*
Listbox	"List", *IO_StringOrScalar, prompt_string, list_string*
Radio buttons	"Radio", *IO_Scalar, prompt_string, list_string*
Checkbox	"Check", *IO_Scalar, prompt_string*
Text	"Text", *text_string*
Caption	"Caption", *caption_string*
Column break	"Colbreak"

In the above table, the parameters can be described as:

IO_String	string used to initialize an edit field and hold the final edit field text.
IO_Scalar	scalar used to initialize a radio or checkbox selection, and hold the final selection.
IO_StringOrScalar	string or scalar used to initialize a listbox selection, and hold the final selection.
prompt_string	string used as the label for the control, or the groupbox label for radio buttons.
max_edit_length	scalar for the maximum characters allowed in an edit field.
list_string	space delimited list of entries for a listbox or radio buttons.
text_string	text to be used in a text control.
caption_string	text to be used as the caption of the dialog in its titlebar.

Examples

```
scalar dinner = 2
string dinnerPrompt = "Choose dinner"
string menu = """Chicken Marsala"" ""Beef Stew"" Hamburger Salad"
string name
string namePrompt = "Enter your name"
scalar result = @uidialog("Caption", "Dinner Menu", "Edit", name,
    namePrompt, 64, "Radio", dinner, dinnerPrompt, menu)
```

These commands display a dialog with an edit field and four radio buttons labeled “Chicken Marsala”, “Beef Stew”, “Hamburger”, and “Salad”. The title “Enter your name” appears above the edit field, while the groupbox surrounding the radio buttons is labeled “Choose dinner”. The words “Dinner Menu” appear in the caption area, or titlebar, of the dialog. The edit field is initially blank, since we did not assign any text to the string NAME. The dinner selection radio buttons are initialized with the value 2, so the “Beef Stew” radio will be selected. We limit the name length to 64 characters.

```
string introString = "We have many pieces to suit your style."
string clothesList = "Shirt Pants Shoes Hat Tie"
string listTitle = "Please make a selection"
scalar selection = 3
@uidialog("text", introString, "list", selection, listTitle,
    clothesList)
```

This creates a dialog with a text string and a listbox. The text “We have many pieces to suit your style” appears at the top. A listbox follows, with the label “Please make a selection” and the options: “Shirt”, “Pants”, “Shoes”, “Hat”, and “Tie”. The listbox is initialized to the third item, “Shoes”. If the user then selects the first item, “Shirt”, from the listbox and presses OK, the scalar SELECTION will hold the value 1.

Note that the text control can be used to help achieve the layout you desire. You can use it to insert empty text strings to add space between adjacent controls. For example,

```
scalar a = 1
scalar b = 0
@uidialog("check", a, "Option 1", "text", "", "check", b, "Option
    2")
```

will leave a space between the two checkboxes. In more complicated dialogs, this may also push the latter controls to a second column. You may have to experiment with the appearance of more complex dialogs.

Cross-references

See “User-Defined Dialogs” on page 147 for discussion.

For a detailed description of each control type, see `@uiedit` (p. 674), `@uilist` (p. 675), `@uiprompt` (p. 676), `@uiradio` (p. 677).

@uiedit	Dialog Display Functions

Syntax: **@uiedit**(*edit_string, prompt_string[, max_edit_len]*)
Argument 1: string, *edit_string*
Argument 2: string, *prompt_string*
Argument 3: integer, *max_edit_len*
Return: integer

Displays a dialog with an edit field and prompt string. *edit_string* is used to initialize the edit field and will return the value after editing. Specify an *edit_string* and *prompt_string* and optionally the maximum character length of the edit field. The default maximum length is 32 characters.

The dialog shows an OK and Cancel button, and will return an integer representing the button clicked: Cancel (-1), OK (0).

Examples

```
string name = "Joseph"
@uiedit(name, "Please enter your First Name:")
```

These commands display a dialog with the text “Please enter your First Name:” followed by an edit field, initialized with the string “Joseph”. If the user edits the string to read “Joe” and presses the OK button, the dialog returns a value of 0 and NAME will now contain the string: Joe.

Similarly,

```
@uiedit("", "Please enter your age:", 2)
```

brings up a dialog with the prompt “Please enter your age” and an empty edit field with a maximum length of two characters. The user will not be able to enter more than two characters into the edit field.

Cross-references

See “User-Defined Dialogs” on page 147 for discussion.

See also `@uidialog` (p. 671), `@uilist` (p. 675), `@uiprompt` (p. 676), `@uiradio` (p. 677).

@uilist	Dialog Display Functions

Syntax: **@uilist***(select_item, prompt_string, listbox_items)*
Argument 1: string or scalar *select_item*
Argument 2: string, *prompt_string*
Argument 3: string, *listbox_items*
Return: integer

Displays a dialog with a listbox and prompt string. Fill the listbox by specifying *listbox_items*, a space delimited list of items. *prompt_string* specifies the text to be used as a label for the listbox. Initialize the listbox selection with *select_item,* which may be either a string or scalar, and which will return the selection value on exit of the dialog.

The dialog shows an OK and Cancel button, and will return an integer representing the button clicked: Cancel (-1), OK (0).

Examples

```
string selection = "Item2"
@uilist(selection, "Please select an item:", "Item1 Item2 Item3")
```

These commands display a dialog with a listbox containing the items “Item1”, “Item2”, and “Item3”. The title “Please select an item:” appears above the listbox, and the second item is initially selected. If the user selects the third item and presses OK, the string SELECTION will contain “Item3”, and the dialog will return the value 0.

Similarly,

```
scalar sel = 3
@uilist(sel, "Please select an item:", "Item1 Item2 Item3")
```

brings up the same dialog with the third item selected. The scalar SEL will contain the user’s selection if OK is pressed.

Cross-references

See “User-Defined Dialogs” on page 147 for discussion.

See also `@uidialog` (p. 671), `@uiedit` (p. 674), `@uiprompt` (p. 676), `@uiradio` (p. 677).

@uiprompt	Dialog Display Functions

Syntax: **@uiprompt***(msg_string[, button_type])*
Argument 1: string, *msg_string*
Argument 2: string, *button_type*
Return: integer

Displays a prompt dialog with OK, Cancel, or Yes and No buttons. You must specify a message string for the prompt dialog, and optionally the type of icon and buttons the dialog should display.

The dialog will return an integer representing the button clicked: Cancel (-1), OK (0), Yes (1), or No (2).

button_type can be one of the following keywords:

"O"	Exclamation icon with OK button *(default)*.
"OC"	Exclamation icon with OK and Cancel buttons.
"YN"	Question icon with Yes and No buttons.
"YNC"	Question icon with Yes, No, and Cancel buttons.

Examples

```
@uiprompt("Error: Too many observations")
```

Displays a dialog with the message "Error: Too many observations" and the OK button.

```
@uiprompt("Exceeded count. Would you like to continue?", "YNC")
```

Displays a dialog with the error message "Exceeded count. Would you like to continue?" and the YES, NO, and CANCEL buttons. If the user presses the YES button, the dialog will return the value 1.

Cross-references

See "User-Defined Dialogs" on page 147 for discussion.

See also `@uidialog` (p. 671), `@uilist` (p. 675), `@uiedit` (p. 674), `@uiradio` (p. 677).

@uiradio	Dialog Display Functions

Syntax: **@uiradio***(select_item, groupbox_label, radio_items)*
Argument 1: scalar, *select_item*
Argument 2: string, *groupbox_label*
Argument 3: string, *radio_items*
Return: integer

Displays a dialog with a series of radio buttons. Create the radio buttons by specifying *radio_items*, a space delimited list of items. You should initialize the selection and retrieve the result with *select_item*. The *groupbox_label* specifies the label for the groupbox that will appear around the radio buttons.

The dialog shows an OK and Cancel button, and will return an integer representing the button clicked: Cancel (-1), OK (0).

Examples

```
scalar selection = 2
@uiradio(selection, "Please select an item:", "Item1 Item2 Item3")
```

These commands display a dialog with three radio buttons, labeled “Item1”, “Item2”, and “Item3”. The title “Please select an item:” appears above the radio buttons, and the second item is initially selected. If the user selects the third item and presses OK, the scalar SELECTION will contain the value 3, and the dialog will return the value 0.

Cross-references

See “User-Defined Dialogs” on page 147 for discussion.

See also `@uidialog` (p. 671), `@uilist` (p. 675), `@uiprompt` (p. 676), `@uiedit` (p. 674).

@vernum	Support Functions

Syntax: **@vernum**
Return: scalar

returns a scalar containing the EViews version number.

@verstr	Support Functions

Syntax: **@verstr**
Return: string

returns a string containing the EViews product name string (e.g. "EViews Enterprise Edition").

@wdir	Support Functions

Syntax: @wdir(*directory_str*)
Argument: string list, *directory_str*
Return: string list

Returns a string list of all files in the directory *directory_str*. Note that this does not include other directories nested within *directory_str*.

Example:

```
@wdir("C:\Documents and Settings")
```

returns a string list containing the names of all files in the "C:\Documents and Settings" directory.

Cross-references

See also `@fileexist` (p. 658) and `@folderexist` (p. 658).

wend	Program Statements

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 133. See also `while` (p. 679).

while	Program Statements

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 133. See also `wend` (p. 678).

@wlookup	Support Functions

Syntax:	@wlookup(*pattern_list[*, *object_type_list]*)
Argument 1:	pattern string list, *pattern_list*
Argument 2:	object type string list, *object_type_list*
Return:	string list

Returns a string list of all objects in the workfile or database that satisfy the *pattern_list* and, optionally, the *object_type_list*. The *pattern_list* may be made up of any number of "?" (indicates any single character) or "*" (indicates any number of characters). The pattern must exactly match the object name in the workfile or database.

Example:

If a workfile contains three graph objects named "GR1", "GR2", and "GR01" and two series objects named "SR1" and "SER2", then

```
@wlookup("GR*")
```

returns the string list "GR1 GR2 GR01". All objects beginning with "GR" and followed by any number of characters are included. Alternately,

```
@wlookup("?R*")
```

returns the string list "GR1 GR2 GR01 SR1". SER2 is not included because "?" specifies a single character preceding an "R".

```
@wlookup("?R?","series")
```

returns the string “SR1”. The object type “series” drops all graph objects from the list, and “?R?” filters out SER2 because “?” specifies a single character.

@wquery	Support Functions

Syntax: @wquery(*database, search_expression, [attribute_list]*)
Argument 1: database name, *pattern_list*
Argument 2: database query expression, *search_expression*
Argument 3: object attribute list, *attribute_list*
Return: string list

Returns a string list of object attributes for all objects in the database that satisfy the query specified by *search_expression*. The *search_expression* should be a standard database query.

By default @wquery will return the names of the objects in the database that match the search criteria. However you may specify other object attributes to be returned, by listing them (comma delimited) in *attribute_list*.

Example

If a database, MYDB, contains a series object named “GDP_1” and a series called “GDP_2”, with the first being annual frequency and the second being quarterly frequency, then

```
@wquery("mydb", "name matches GDP* and freq=Q", "name")
```

returns the string “GDP_2”. Alternately,

```
@wquery("mydb", "name matches GDP* and freq=A", "name, freq")
```

returns the string list “GDP_1 A”.

```
@wquery("basics.edb", "freq = q and start>1970 and not
    scale=millions")
```

returns a string list of the names of all objects in the database BASICS.EDB with quarterly frequency and a start date after 1970 which do not have the custom attribute ‘scale’ set to ‘millions’.

```
@wunique(@wquery("mydb", "start>2000", "freq"))
```

returns a string list of all the frequencies of objects in the database whose start date is post 2000. Note that the `@wunique` function is used to remove duplicate frequencies.

@wread	Support Functions

Syntax: @wread(“*file*”)
Argument 1: file name of a text file on disk.
Return: string

Returns a string containing the contents of the specified text file on disk. Note that any line breaks in the text file will be removed.

Example

```
@wread(“c:\temp\myfile.txt”)
```

Returns a string containing the contents of the “c:\temp\myfile.txt”.

Appendix A. Wildcards

EViews supports the use of wildcard characters in a variety of situations where you may enter a list of objects or a list of series. For example, among other things, you can use wildcards to:

- fetch, store, copy, rename or delete a list of objects
- specify a group object
- query a database by name or filter the workfile display

The following discussion describes some of the issues involved in the use of wildcard characters and expressions.

Wildcard Expressions

There are two wildcard characters: "*" and "?". The wildcard character "*" matches zero or more characters in a name, and the wildcard "?" matches any single character in a name.

For example, you can use the wildcard expression "GD*" to refer to all objects whose names begin with the characters "GD". The series GD, GDP, GD_F will be included in this list GGD, GPD will not. If you use the expression GD?, EViews will interpret this as a list of all objects with three character names beginning with the string "GD": GDP and GD2 will be included, but GD, GD_2 will not.

You can instruct EViews to match a fixed number of characters by using as many "?" wildcard characters as necessary. For example, EViews will interpret "??GDP" as matching all objects with names that begin with any two characters followed by the string "GDP". USGDP and F_GDP will be included but GDP, GGDP, GDPUS will not.

You can also mix the different wildcard characters in an expression. For example, you can use the expression "*GDP?" to refer to any object that ends with the string "GDP" and an arbitrary character. Both GDP_1, USGDP_F will be included.

Using Wildcard Expressions

Wildcard expressions may be used in filtering the workfile display (see "Filtering the Workfile Display" on page 73 of the *User's Guide I*), in selected EViews commands, and in creating a group object.

For example, the following commands support the use of wildcards: `show` (p. 433), `store` (p. 444), `fetch` (p. 332), `copy` (p. 306), `rename` (p. 421) and `delete` (p. 326).

To create a group using wildcards, simply select **Object/New Object.../Group**, and enter the expression, EViews will first expand the expression, and then attempt to create a group using the corresponding list of series. For example, entering the list,

```
y x*
```

will create a group comprised of Y and all series beginning with the letter X. Alternatively, you can enter the command:

```
group g1 x* y?? c
```

defines a group G1, consisting of all of the series matching "X*", and all series beginning with the letter "Y" followed by two arbitrary characters.

When making a group, EViews will only select series objects which match the given name pattern and will place these objects in the group.

Once created, these groups may be used anywhere that EViews takes a group as input. For example, if you have a series of dummy variables, DUM1, DUM2, DUM3, …, DUM9, that you wish to enter in a regression, you can create a group containing the dummy variables, and then enter the group in the regression:

```
group gdum dum?
equation eq1.ls y x z gdum
```

will run the appropriate regression. Note that we are assuming that the dummy variables are the only series objects which match the wildcard expression DUM?.

Source and Destination Patterns

When wildcards are used during copy and rename operations, a pattern must be provided for both the source and the destination. 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 patterns,

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

which 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 new 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. This allows you to both add and remove characters from the source name during the copy or rename process. Some examples should make this 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 will result in characters being removed from the source name when forming 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::
```

Resolving Ambiguities

Note that an ambiguity can arise with wildcard characters since both "*" and "?" have multiple uses. The "*" character may be interpreted as either a multiplication operator or a wildcard character. The "?" character serves as both the single character wildcard and the pool cross section identifier.

Wildcard versus Multiplication

There is a potential for ambiguity in the use of the wildcard character "*".

Suppose you have a workfile with the series X, X2, Y, XYA, XY2. There are then two interpretations of the wildcard expression "X*2". The expression may be interpreted as an auto-

series representing X multiplied by 2. Alternatively, the expression may be used as a wildcard expression, referring to the series X2 and XY2.

Note that there is only an ambiguity when the character is used in the middle of an expression, not when the wildcard character "*" is used at the beginning or end of an expression. EViews uses the following rules to determine the interpretation of ambiguous expressions:

- EViews first tests to see whether the expression represents a valid series expression. If so, the expression is treated as an auto-series. If it is not a valid series expression, then EViews will treat the "*" as a wildcard character. For example,

  ```
  y*x
  ```

  ```
  2*x
  ```

  are interpreted as auto-series, while,

  ```
  *x
  ```

  ```
  x*a
  ```

  are interpreted as wildcard expressions.

- You can force EViews to treat "*" as a wildcard by preceding the character with another "*". Thus, expressions containing "**" will always be treated as wildcard expressions. For example, the expression:

  ```
  x**2
  ```

  unambiguously refers to all objects with names beginning with "X" and ending with "2". Note that the use of "**" does *not* conflict with the EViews exponentiation operator "^".

- You can instruct EViews to treat "*" as a series expression operator by enclosing the expression (or any subexpression) in parentheses. For example:

  ```
  (y*x)
  ```

  always refers to X times Y.

We *strongly* encourage you to resolve the ambiguity by using parentheses to denote series expressions, and double asterisks to denote wildcards (in the middle of expressions), whenever you create a group. This is especially true when group creation occurs in a program; otherwise the behavior of the program will be difficult to predict since it will change as the names of other objects in the workfile change.

Wildcard versus Pool Identifier

The "?" wildcard character is used both to match any single character in a pattern and as a place-holder for the cross-section identifier in pool objects.

EViews resolves this ambiguity by not allowing the wildcard interpretation of "?" in any expression involving a pool object or entered into a pool dialog. "?" is used exclusively as a cross-section identifier. For example, suppose that you have the pool object POOL1. Then, the expression,

```
pool1.est y? x? c
```

is a regression of the pool variable Y? on the pool variable X?, and,

```
pool1.delete x?
```

deletes all of the series in the pool series X?. There is no ambiguity in the interpretation of these expressions since they both involve POOL1.

Similarly, when used apart from a pool object, the "?" is interpreted as a wildcard character. Thus,

```
delete x?
```

unambiguously deletes all of the series matching "X?".

Index

C

D

E

F

G

H

L

M

N

P

S

T

U