This file consists of the first part of Chapter 2 of **MacAnova User's Guide** by Gary W. Oehlert and Christopher Bingham, issued as Technical Report Number 617, School of Statistics, University of Minnesota, revised August 1998, describing Version 4.07 of MacAnova.

Fonts used in this manual are Palatino, Courier, and Symbol.

For information concerning MacAnova, write University of Minnesota, Department of Applied Statistics, 352 Classroom Office Building, 1994 Buford Avenue, St. Paul, MN 55108-6042.

2. The Basics

2.1 **Getting Started** You first need to launch MacAnova. Read the appropriate Appendix (B for Macintosh, C for DOS™, D for Windows™, E for Unix™ and F for Motif) for detailed information on how to do this on your computer. The following table briefly summarizes this information.

Computer	How to Start
Macintosh	Double click on icon
IBM compatible with Windows	Double click on MacAnova for Windows icon in MacAnova program group
IBM compatible with Windows 95	Select item MacAnova for Windows on MacAnova entry under Programs on Start menu
IBM compatible DOS versions	Type `macanodj` (extended memory) or `macanobc` (limited memory) at DOS prompt
Unix Motif version	Type `macanovawx` at Unix prompt
Unix non-Motif version	Type `macanova` at Unix prompt

After MacAnova has been launched and a start-up message displayed, the *prompt*

```
Cmd>
```

appears.

You tell MacAnova what to do by typing commands (instructions) after the prompt, followed by Return or Enter. You can use the delete or backspace keys to correct mistakes. On windowed versions (Macintosh, Windows, and Motif) you can use arrow keys or the mouse to re-position the cursor to edit a command before executing. You can also use arrow keys on the Unix version and the extended memory DOS version. Until Return or Enter is pressed, a command is not executed and may be changed. On the Macintosh, Enter differs from Return in that it moves the cursor to the end of the command line and then enters a Return. In the Motif and Windows versions, Ctrl+Enter does what Enter does on a Macintosh.

During the course of a MacAnova run you can type in data or read it from a file, do standard statistical analyses such as Analysis of Variance or Regression, and create new *variables* which contain transformed data or the results of your analysis. In the windowed versions, everything you type and all results go into a command/output window which you can save in a file at any time using entries **Save Window** or **Save Window As** on the **File** menu.

Data that have been typed or read into MacAnova are contained in a *workspace* in your computer's memory, together with other variables. On all versions you can save a copy of your workspace on disk using commands `save()` or `asciisave()`.

2.2 **Quitting** You end your MacAnova run by typing `quit`, `end`, `stop`, `bye` or `exit`. On windowed versions you can also select **Quit** on the **File** menu. When you exit

MacAnova, your *workspace*, consisting of all of the data and results in MacAnova's memory, will disappear unless you have saved it to disk by commands `save()` or `asciisave()` (see Sec. 2.17). In the windowed versions, when you quit, you are automatically asked if you want to save your workspace and any command/output windows in files unless you type `quit(F)`.

2.3 **Functions and macros** You tell MacAnova to do what you want by typing commands after the `Cmd>` prompt rather than by clicking on "buttons" or selecting items on menus.

MacAnova commands can be described as "functional" – they have *arguments* (inputs) and *return* values or *results* (outputs). You enclose a function's arguments in parentheses following the function's name, separating them by commas (**Example**: `rep(1,10)` runs function `rep()` with two arguments, 1 and 10).

Some functions have "side effects" – printing or plotting information and/or creating or deleting variables – in addition to returning a value. For example, linear model commands such as `anova()` and `regress()` normally print the results of the statistical analysis and create variable `RESIDUALS` containing the residuals from the model fitted. Both the printing and the variable creation are considered to be side effects.

You may sometimes find it helpful to think of MacAnova as a statistical programming language, although you do not need to understand programming to use MacAnova effectively.

In this manual, whenever a MacAnova function is referred to by name, the name will be followed by `()`, as in `anova()`. Functions in MacAnova include statistical functions such as `anova()` (which does analysis of variance calculations) and `describe()` (which computes descriptive statistics like mean and variance); mathematical functions such as `log()`, `exp()`, and `cos()`; commands such as `print()` and `plot()` for printing and graphing results; and housekeeping functions such as `vecread()`, `vector()` and `delete()` for reading data from files, creating and combining variables, deleting variables and so forth.

When the output from one function makes sense as input to another, the functions may be composed (strung together). Thus `describe(log(x+3))` takes the data in `x`, adds 3 to it, takes the logarithm of the sum, and then computes descriptive statistics on the logged values.

We often use the word *command* to refer to functions such as `print()`, `matread()`, or `delete()` that are not primarily involved with changing or manipulating data. We sometimes call symbols such as `+` and `–` *arithmetic operators* and symbols such as `>` and `<=` *comparison operators*.

Besides functions, MacAnova has *macros* that, like functions, produce results from arguments. In fact, they are used identically, that is as a name followed by arguments enclosed in parentheses and separated by commas as in `boxcox(x,1/3)`. There are many macros such as `hist`, `resvsrankits` and `boxcox` that are pre-defined. You can read other macros from files using `macroread()` (Sec. 7.5.1) and `getmacros` (Sec. 7.5.3). Files `MacAnova.mac`, `Tser.mac` and `Design.mac` that are distributed with MacAnova contain many useful macros. Finally, you can create your own macros to extend the

range of what MacAnova can do (Sec. 8.4).

2.4 **Variables** Data are stored in ordinary or temporary *variables* which have names such as `x` or `height` of up to 12 characters. Ordinary names consist of letters (`a - z` and `A - Z`), numbers (`0 - 9`), and the underscore character _ but must start with a letter or _ (`var_2` but not `2_var`). Once created, ordinary variables exist until they are explicitly deleted or until the MacAnova session ends. The names of temporary variables start with the character @ followed by a letter or _ (`@mean_2` but not `@2_mean`). Temporary variables are deleted the next time the prompt is printed.

Variables with names starting with _ or @_ are "invisible"; see also Sec. 2.8.7.

All names are *case sensitive*, that is, for example, `residuals`, `Residuals`, and `RESIDUALS` are three different names. It is a good idea to avoid names with all capital letters, as MacAnova deletes and creates certain variables with all capital names (for example, `RESIDUALS`).

You can get an alphabetized listing of some or all active variables using commands `listbrief()` or `list()` (Sec. 2.8.9).

One way you can display the value of a variable is to type its name:

```
Cmd> PI # PI is a predefined REAL variable
(1)       3.1416
```

This also illustrates that a command line can include a descriptive comment – anything starting with the character #.

You *assign a value* to a variable using the left pointing arrow "`<-`" (less than followed by hyphen).

```
Cmd> euler <- .577215664; euler # semi-colon used as separator
(1)      0.57722
```

This also illustrates that you do more than one thing on a single command line if you separate commands and expressions with a semi-colon. First `euler <- .577215664` assigns a value to variable `euler`. Then, `euler` after the semi-colon displays the value as if it had been after the prompt in a new command line.

2.5 **Data types – REAL, LOGICAL, and CHARACTER** MacAnova variables can represent data of several types. The most common types are `REAL`, `LOGICAL`, and `CHARACTER`.

`REAL` data are just numbers like `2.4`, `-1`, and `3.14159`. They are entered just by typing their values.

`LOGICAL` data have values True or False which are typed and printed as `T` and `F`.

```
Cmd> untrue <- F; untrue
(1) F
```

`CHARACTER` data consist of letters or other characters – letters, digits, spaces, and punctuation. The basic element of `CHARACTER` data is known as a *string*, consisting of one or more characters. When typing a string on the keyboard, it must be surrounded by *double* quotes, for example, `"this is a character string"`; single quotes won't do. Similarly a string is normally printed surrounded by double quotes.

```
Cmd> greetings <- "Hello!"
Cmd> greetings # typing the variable name prints the variable
(1) "Hello!"
```

In some MacAnova output, LOGICAL and CHARACTER are abbreviated to LOGIC and CHAR.

Note: Any character you can type is allowed inside a string, even the character produced by pressing the Enter key (Return on a Macintosh). A common error is to forget to add the closing double quote to a string you are entering. When you press Enter or Return, you think you are done and expect MacAnova to respond with a prompt or other output, but it doesn't. Until you type the closing quote, MacAnova doesn't know you are finished and is waiting for you to type more characters to add to the string variable. If you don't type the closing quote nothing will ever happen.

If you want to include a double quote *in* a string, you must precede it with a backslash.

```
Cmd> print("\"Hello\"") # inner double quotes are part of string
"Hello"
```

This use of a backslash is sometimes called "escaping" the double quote. If you want to include a backslash in a string, you must "escape" *it*, that is precede it with a backslash, yielding a *double* backslash.

```
Cmd> print("\\Hello\\") #backslash is escaped.
\Hello\
```

It is possible to have a variable that has no associated data. Such a variable is a NULL variable. You can create a NULL variable a by

```
Cmd> a <- NULL
```

In additions to types REAL, LOGICAL, CHARACTER and NULL, there are several more specialized data types. These are STRUC, the data type a structure (Sec. 2.8.16), MACRO, the data type of macro (Sec. 9.3), GRAPH (Sec. 8.5.3) and LONG (Sec. 9.7.3, 9.7.4).

You can use `list()` (Sec. 2.8.9) to list active variables together with their data types and dimensions.

2.6 **Shapes of variables – scalars, vectors, matrices, arrays** REAL, LOGICAL, and CHARACTER data can be organized as *scalars* (a single number, logical value or string), *vectors* (several numbers, logical values or strings with no further structure), *matrices* (a two dimensional table of values), and more generally as *arrays* of up to 31 dimensions.

The basic command for creating a vector from several items is `vector()` (**Examples**: `vector(1,7,5)`, `vector(T,F,T)`, and `vector("A","B","C")` create REAL, LOGICAL, and CHARACTER vectors of length three). A synonym for `vector()` is `cat()`. Macros `enter` and `enterchars` are also useful for creating vectors. See Sec. 2.8.10.

You can refer to individual elements of a vector, matrix or array using *subscripts*, that is, indices in `[...]` after its name (**Examples**: `x[4]`, `y[2,3]`, `w[2,2,4]`). See Sec. 2.8.11.

There are several pre-defined variables including REAL variables E (e^1 = 2.718281828...),

PI (π = 3.14159265...), DEGPERRAD (the number of degrees per radian = 180 / π) and CHARACTER variables DATAFILE, MACROFILES and MACROPATHS.

2.7 **Missing values** REAL and LOGICAL data may have "missing values". When typed from the keyboard, the code for a REAL missing value is a question mark "?". When printed, a missing value normally prints as MISSING. Although you cannot enter missing LOGICAL data directly, missing LOGICAL values may be created by a comparison involving missing data, for example

```
Cmd> 10 > ? # comparison of a REAL with a MISSING
WARNING: arithmetic comparison with missing values(s)
(1) MISSING
```

You can use ismissing(x) to find which values in a REAL or LOGICAL data set x are MISSING. It returns a LOGICAL variable with the same size and shape (dimensions) as x whose values are True where x is MISSING, and False where x is not missing. If x is a CHARACTER variable, ismissing(x) treats empty strings ("") as MISSING.

```
Cmd> ismissing(vector(1,?,3)) # 1 MISSING REAL
(1) F       T       F

Cmd> ismissing(vector("Hello","","World")) # "" treated as missing
(1) F       T       F
```

Function anymissing() returns True or False depending on whether its argument has any MISSING values or empty strings as elements.

```
Cmd> anymissing(vector(1,?,5)) # one or more missing values
(1) T

Cmd> anymissing(vector(1,3,5)) # no missing values
(1) F

Cmd> anymissing(vector("Hello","","World?")) # "" treated as missing
(1) T
```

2.8 **Introduction to MacAnova syntax** As a statistical "language," MacAnova has a well defined syntax based on operators such as +, * and > and functions. Operators and functions are alike in that both transform input into output. For an operator, the inputs are known as *operands*. For example, in a + b, a and b are operands. For a function, the inputs are *arguments*. For example, in cos(x/DEGPERRAD), x/DEGPERRAD is an argument. The output is of course just the result of the computation.

The syntax involving operators such as +, - or / is borrowed from algebra. You can make free use of ordinary parentheses (and) and braces { and } to clarify what is wanted. For example, (3 * x + 5)/7 is a legal MacAnova expression. You cannot substitute square brackets [...] for (...) or {...}. See Sec. 2.8.11 and 2.8.16 for the use of square brackets and Sec. 9.2.1 - 9.2.3 for the use of braces to delimit compound commands.

As mentioned earlier, you invoke a function or macro by typing its name, followed by a list of arguments, separated by commas and enclosed in parentheses. Arguments may be variables, as in hconcat(x1,x2,x3) (the *name* is hconcat, the *arguments* are x1,

x2 and x3), or may be expressions, possibly involving other functions such as sqrt() as in boxcox(x+sqrt(3),1/3) (the *name* is boxcox, the *arguments* are x+sqrt(3) and 1/3).

You can have several commands or functions in a single line if they are separated by semicolons:

```
Cmd> (3 + 4)*sqrt(2); 3+4*sqrt(2) # 2 commands separated by ';'
(1)       9.8995   Output from (3 + 4)*sqrt(2)
(1)       8.6569   Output from 3+4*sqrt(2)
```

When the line you are entering becomes too long to fit on the screen, you can just keep typing. In all versions except those for Windows or Motif or extended memory DOS, the line you are typing will "wrap around" to the next line. In the Windows and Motif versions, the contents of the window scrolls to the left while in the extended memory DOS version, the line shifts left. Alternatively, outside of a quoted string, you can type a backslash (\) followed by Return or Enter and continue typing on the following line. This is illustrated in several examples below. In the limited memory DOS version, it is sometimes essential to use a backslash and start a new line since no single DOS input line can contain more than 128 characters.

This section covers only the most basic aspects of MacAnova syntax. See Chapter 7 for discussion of more advanced features, including the use of for(...){} and while(...){...} loops (Sec. 9.2.3) and if(...){...}elseif(...){...}else{...} (Sec. 9.2.2).

2.8.1 **Spaces and comments** Spaces are generally ignored outside of quoted strings, except you cannot embed a space in a name or number. For example:

```
Cmd>  x <- 23 456.7  # embedded space in number is error.
ERROR: problem with input near  x <- 23 456.7

Cmd> des cribe(x) # embedded space in command name is error.
ERROR: problem with input near des cribe
```

You *must* have a space after the special assignment operators <-+ and <-- (a <-+3 without a space is interpreted as a <- +3), see Sec. 2.8.8.

Anything typed after # on a line is ignored unless it forms part of a quoted string. This allows you to add descriptive or explanatory comments to a line. We have already used this feature and will continue to do so.

2.8.2 **Keywords** The arguments to some functions and macros may be *keyword phrases* of the form name:value. For example, keywords may be used to label axes in a graph (**Example**: plot(Time:x,Distance:y)), to label printed output (**Example**: print(average:sum(x)/n)), or to specify options (**Example**: sort(x,down:T)). The keyword name may have no more than 10 characters, and the value may be a variable or expression. Many keywords have values T or F, with T meaning "yes" or "enable", and F meaning "no" or "disable".

2.8.3 **Arithmetic expressions** It is simple to do arithmetic with REAL data. You may add, subtract, multiply, divide, and raise to a power in natural expressions by using the

operators +, -, *, /, and ^, respectively. Instead of ^, you can use ** to raise to a power.

```
Cmd> vector(3 + 4, -7/8, 2*PI, 3^.5)
(1)            7       -0.875       6.2832       1.7321
```

An additional operator is %% which computes the modular quotient; that is, x %% y is the remainder when x is divided by y.

```
Cmd> vector(19.3 %% 5,-19.3 %% 5)
(1)          4.3          0.7
```

MacAnova operators have specific levels of *precedence* determining, in part, the order in which operations are carried out when there are several operators in an expression. The higher the precedence, the earlier the operation is performed. Addition and subtraction have lower *precedence* than multiplication, division, and modular division, which in turn have lower precedence than raising to a power. This means, for example, that 3 + 4 * 5 - 6 is interpreted as 3 + (4*5) - 6 rather than (3+4) * (5-6) or ((3 + 4) * 5) - 6, and 17 - 3^-2*4 is interpreted as 17 - (3^(-2))*4.

Arithmetic operators can be used with LOGICAL data as well, with True and False interpreted as 1 and 0, respectively. The result is always REAL. Thus F*T is 0 and 3+T is 4.

As in ordinary algebra, you can use parentheses to specify the order in which operations should be performed:

```
Cmd> vector((1+2)*3, 1+2*3)
(1)            9            7
Cmd> vector(2^3*2, 2^(3*2))
(1)           16           64
Cmd> vector((3 + 4)*sqrt(2), 3 + 4*sqrt(2))
(1)       9.8995       8.6569
```

If both sides of an operator have the same size and shape, the operation is applied to each element. For example:

```
Cmd> vector(1,3,-2)^vector(1,2,3)#same as vector(1^1,3^2,(-2)^3)
(1)            1            9           -8
```

If one side of an operator is a scalar (single number) it is combined with all the elements of the other side. For example:

```
Cmd> vector(1,3,5)/2 # same as vector(1/2,3/2,5/2)
  (1)          0.5          1.5          2.5
```

Thus MacAnova can be used as a powerful interactive calculator. See Sec. 2.10.2 for more details.

2.8.4 **Comparison operators and logical operators** You can compare REAL, LOGICAL and CHARACTER data using the *comparison operators* == (equals), != (not equals), < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to). On the Macintosh, you can use ≤, ≥, and ≠ in place of <=, >=, and !=. LOGICAL values T and F are treated as 1 and 0, respectively. The result of a comparison is a LOGICAL value.

```
Cmd> vector(2 == 1, 3 != 3, 2 >= 5, 3 < 1, 2 == 2, 3 != 2)
(1) F       F       F       F       T       T

Cmd> vector(T == T, F < T, F != F)
(1) T       T       F

Cmd> vector("A" < "B", "Ab" <= "Ac", "Bb" > "ab")
(1) T       T       F
```

The third element in the last example ("Bb" > "ab" which evaluates to False) illustrates that comparison of CHARACTER data uses the ASCII collating sequence, with all upper case letters coming before ("less than") lower case letters. See Sec. 2.12.3 for details.

You can combine or change LOGICAL values using the *logical operators* && (logical and) , || (logical or), and ! (logical not).

```
Cmd> vector(T || T, T || F, F || T, F || F, 3 < 5 && 5 < 10)
(1) T       T       T       F       T

Cmd> vector(!T, !F, !(3 < 2))
(1) F       T       T
```

Both operands of && and || are always evaluated, even when it may not be necessary to do so. For example, in log10(5) < 1 || sqrt(5) > 2, *both* log10(x) and sqrt(x) are computed even though the value of the expression (True) can be determined once log10(5) has been computed. Pre-defined macros alltrue and anytrue, both of which accept any number of LOGICAL scalar arguments, provide the same results as && and || except that no more arguments are evaluated than is necessary.

```
Cmd> anytrue(log10(5) < 1 || sqrt(5) > 2) # sqrt(5) not computed
(1) T

Cmd> alltrue(log10(5) > 1 , sqrt(5) > 2) # sqrt(5) not computed
(1) F
```

Comparison and logical operators work with vectors similarly to arithmetic operators.

```
Cmd> 7 <= vector(4, 5, 6, 7, 8, 9)# compare 7 with several numbers
(1) F       F       F       T       T       T

Cmd> vector(T, F, F) || vector(F, F, T)
(1) T       F       T
```

2.8.5 **Bit-wise operations on integers** Any integer can be represented in the binary system with binary "digits" or "bits" either 0 or 1. For instance, $37 = 1\times2^5 + 0\times2^4 + 0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$ has binary representation 100101b. MacAnova has four operations and a function that work with the representation of integers between 0 and 2147483647 = $2^{32}-1$, considered as sets of 32 bits.

Bit Operation	Meaning
a %| b	Bitwise Or (OR)
a %^ b	Bitwise Exclusive Or (XOR)
a %& b	Bitwise And (AND)
%!a	Bitwise Complement (COMPL)

For %&, a bit of the result is 1 if and only if the corresponding bits in the operands are both 1.

```
Cmd> 25 %& 19 # 11001b AND 10011b
(1)           17       10001b
```

For %|, a bit of the result is 1 if and only if at least 1 of the corresponding bits in the operands is 1.

```
Cmd> 25 %| 19 # 11001b OR 10011b
(1)           27       11011b
```

For %^, a bit of the result is 1 if and only if exactly 1 of the corresponding bits in the operands are 1, that is, if the corresponding bits differ.

```
Cmd> 25 %^ 19 # 11001b XOR 10011b
(1)           10       01010b
```

Operator %! operates on the immediately following variable considered as a collection of 32 bits, changing 1's to 0's and 0's to 1's.

```
Cmd> print(nsig:10, %!25) #COMPL(00000000000000000000000011001b)
NUMBER:
(1)        4294967270      11111111111111111111111111100110b
Cmd> print(nsig:10, %! 0) #COMPL(00000000000000000000000000000000b)
NUMBER:
(1)        4294967295      11111111111111111111111111111111b
```

See Sec. 11.2.5 for a table summarizing these operators. See Sec. 7.4 for the use of keyword nsig.

If an operand is LOGICAL, it is treated as having value 0 (False) or 1 (True). The result is always REAL, and if any operand is MISSING, so is the result.

nbits(x) computes the number of bits in each element of REAL variable x. The result is MISSING for any element of x that is not an integer between 0 and 4294967295 or is MISSING.

```
Cmd> nbits(vector(0,25,27)) # all values appropriate integers
(1)            0            3            4
Cmd> nbits(vector(4294967270,4294967295,?)) # 1 MISSING value
WARNING: missing values in argument(s) to nbits()
(1)           29           32      MISSING

Cmd> nbits(vector(1/3,-1,5000000000)) # all improper values
WARNING: nbits of illegal value set to MISSING
(1)      MISSING      MISSING      MISSING
```

2.8.6 **Mathematical functions and transformations** MacAnova provides the following mathematical functions or transformations.

Function	Computes what	Function	Computes what
abs()	Absolute value	acos()	Arc cosine
sqrt()	Square root	asin()	Arc sine
log()	Natural (base e) logarithm	atan()†	Arc tangent
log10()	Common (base 10) logarithm	cosh()	Hyperbolic cosine
exp()	Exponential e^x	sinh()	Hyperbolic sine
round()†	Round to nearest integer	tanh()	Hyperbolic tangent
floor()	Round down to next integer	acosh()	Inverse hyperbolic cosine
ceiling()	Round up to next integer	asinh()	Inverse hyperbolic sine
hypot()‡	$\sqrt{x^2 + y^2}$	atanh()	Inverse hyperbolic tangent
cos()	Cosine	lgamma()	Natural log of gamma function
sin()	Sine	rational()	Rational function§
tan()	Tangent		
† allows two argument	‡ requires two arguments	§ two or three arguments	

```
Cmd> abs(3-5)+log10(10)+sqrt(25) # 2+1+5
(1)            8

Cmd> vector(round(-3.2), floor(-3.2), ceiling(-3.2))
(1)           -3           -4           -3
```

See Sec. 2.10.1 for what happens when the argument to one of these functions is not a scalar.

Transformations atan() and round() both accept two arguments. atan(x,y), computes the angle θ such that $\tan(\theta) = x/y$ while choosing the quadrant so that $x = r\sin(\theta)$, $y = r\cos(\theta)$, where $r = \sqrt{x^2 + y^2}$. round(x,n) rounds x to n decimal places, where n in an integer, either positive or negative.

hypot(x,y) is mathematically equivalent to sqrt(x^2 + y^2) but gives answers for a wider range of arguments. sqrt(x^2 + y^2) fails but hypot(x,y) may not when x^2 + y^2 is (i) too big to be represented in the computer or (ii) evaluates to 0 because it is smaller than the smallest non-zero value representable in the computer.

```
Cmd> DEGPERRAD*vector(atan(1,sqrt(3)),atan(-1,-sqrt(3)))#degrees
(1)           30         -150

Cmd> vector(round(PI,2), round(10000*PI,-2))
(1)         3.14        31400

Cmd> x <- y <- 1e154; vector(hypot(x,y),sqrt(x^2+y^2))
WARNING: result of arithmetic too large, set to MISSING;operation is +
WARNING: missing values in argument(s) to sqrt()
(1)  1.4142e+154      MISSING  Should be sqrt(2)*1e154
```

```
Cmd> x <- y <- 1e-165; vector(hypot(x,y),sqrt(x^2+y^2))
(1)  1.4142e-165            0  Should be sqrt(2)*1e-165
```

Function `rational()` is another special case. It computes a rational function of the form $r(x;a,b) \equiv \frac{a_0 + a_1 x + \ \ + a_{m-1}x^{m-1}}{b_0 + b_1 x + \ \ + b_{n-1}x^{n-1}}$ for specified constant coefficients $\{a_k\}$ and $\{b_k\}$. Rational functions often appear in formulas used to approximate mathematical functions. When `a` and `b` are `REAL` vectors of lengths m and n respectively, and `x` is a `REAL` scalar, then `rational(x,a,b)` computes $r(x;a,b)$, where a_0 = `a[1]`, a_1 = `a[2]`, ..., a_{m-1} = `a[m]` and b_0 = `b[1]`, b_1 = `b[2]`, ..., b_{n-1} = `b[n]`.

`rational(x,a)` is equivalent to `rational(x,a,1)` and evaluates the polynomial $a_0 + a_1 x + \ \ + a_{m-1}x^{m-1}$. `rational(x,,b)` is equivalent to `rational(x,1,b)` and evaluates the recprocal polynomial $\frac{1}{b_0 + b_1 x + \ \ + b_{n-1}x^{n-1}}$.

Here is a relatively simple example of the use of `rational()` to compute an approximate standard normal quantile z_p that satisfies $P(Z \ge z_p) = p$, where Z is normal with mean 0 and standard deviation 1. For $0 < p \le 0.5$, z_p can be approximated with a maximum error of 0.00279 by $t - \frac{2.30753 + .27061t}{1 + .99229t + .04481t^2}$, where $t = \sqrt{-2\log p}$ (Hastings 1955, Abramowitz and Stegun 1964, eq. 26.6.22).

```
Cmd> a <- vector(2.30753, .27061); b <- vector(1,.99229, .04481)
Cmd> tt <-  sqrt(-2*log(.05));tt - rational(tt,a,b) # upper 5%
(1)       1.6445       Actual value is 1.6449
Cmd> tt <-  sqrt(-2*log(.01));tt - rational(tt,a,b) # upper 1%
(1)       2.3277       Actual value is 2.3263
```

An important use of `rational()` is in writing macros (see Sec. 9.3) that compute mathematical functions that are not directly available in MacAnova but which can be approximated using rational functions. See for example, macros `i0` and `i1` in file MacAnova.mac distributed with MacAnova. These compute modified Bessel functions of the first kind.

All the listed functions except `rational()` also accept `CHARACTER` variables as arguments. This is best illustrated by example.

```
Cmd> log(vector("height","weight"))
(1) "log(height)"
(2) "log(weight)"
Cmd> atan("height","distance")
(1) "atan(height,distance)"
Cmd> round("score",2) # or round("score","2")
(1) "round(score,2)"
```

There is an exception. When an element of a `CHARACTER` argument is `""` or starts with "@", "[", "(", "{", "<", "/" or "\" it is not transformed; for `atan()`, the corresponding

argument of the second argument is ignored.

This feature is useful in creating labels for transformed variables. See Sec. 8.3.2.

You can use boxcox(x, p) where x is REAL vector or matrix with positive elements and p is a REAL scalar, to compute the Box-Cox power transformation of x. Under the Box-Cox transformation with power p of the data set $x_1, x_2, ..., x_n$, x_i is transformed to $\frac{(x_i^{\,p} - 1)}{p(\prod_{j=1}^{n} x_i)^{(p-1)/n}}$ when $p \neq 0$ and to $(\prod_{j=1}^{n} x_i)^{1/n} \log(x_i)$ when $p = 0$.

boxcox operates separately on each column of x when it is a matrix. Computing a power transformation using boxcox is sometimes preferred to x^p because the values for different values of p are comparable, and because boxcox(x,0) is proportional to log(x). See Sec. 10.7 for an example of the use of boxcox.

2.8.7 **Assignment of values to variables** To create variables you must assign values to them using the *assignment operator*, the left pointing arrow <- (the symbols "less than" and "hyphen"). The value on its right side is assigned to the variable on its left side:

```
Cmd> x <- 6; x # Assign 6 to variable x and display the value
(1)            6
```

makes x a REAL scalar whose value 6. Any subsequent use of x will have the same effect as using the number 6; for example x+2 is 8.

You can assign CHARACTER and LOGICAL data in the same way:

```
Cmd> department <- "Applied Statistics"; true <- T
```

A sequence of the form LeftSide <- RightSide is an *assignment expression*. Besides making the assignment, an assignment expression itself has a value equal to the value of the assigned variable. This lets you save intermediate values on the fly:

```
Cmd> y <- (x <- 6) + 8; u <- w <- x + 10; vector(x,y,u,w)
(1)            6           14           16           16
```

The value of an expression or command that is not an assignment expression is normally printed. This is probably the most common way to get output from computations.

```
Cmd> 3*5
(1)           15
```

If an assignment is made nothing is printed.

```
Cmd> x <- 3*5 # no output because assignment

Cmd> x
(1)           15
```

The first statement does not print anything but the second does.

An exception is made in the case of an "invisible" variable whose name starts with _ or @_. You can use such a variable in an expression or assign it to another variable, but if

you just type the name, nothing is printed.

```
Cmd> _pi <- PI; twopi <- 2*_pi

Cmd> _pi # nothing printed because "invisible"

Cmd> twopi # not "invisible"
(1)       6.2832
```

Warning A frequent mistake made by those familiar with programming languages such as Basic, Fortran or C is to use "=" instead of "<-" for assignment. In MacAnova, `a = 3` is actually interpreted as `a == 3` and has value True or False depending on the value of `a`.

2.8.8 **Arithmetic assignment operators** Sometimes you just want to modify a variable by adding a constant or another variable or multiplying by a constant or another variable. For instance, if you are doing something repetitively and want to keep count of how many times it's done, you might use `count <- count + 1` every time you do it, initializing `count` by `count <- 0` before you start.

A short cut method for doing this is to use the *arithmetic assignment operator* `<-+`: `count <-+ 1`. In general, `a <-+ b` is equivalent to `a <- a + b`.

The other arithmetic assignment operators are `<--`, `<-*`, `<-/`, `<-^`, `<-**`, and `<-%%`. Each combines the expression on the right hand side with the left hand side and then modifies the left hand side. Thus, for example, `x <-/ n` is equivalent to `x <- x/n`. Both `<-+` and `<--` must be followed by at least one space.

```
Cmd> a <- 17; b <- 5; c <- 3

Cmd> a <-/ 2; b <-+ 1; c <-^ 2; vector(a,b,c) # 17/2, b+1, c^2
(1)         8.5           6           9
```

2.8.9 **print(), write(), list(), listbrief() and delete()** You can print one or more variables using command `print()`.

```
Cmd> x <- 1/6; y <- 1/14; print(x,y,x+y,"Hello")
x:
(1)      0.16667
y:
(1)     0.071429
NUMBER:
(1)       0.2381
STRING:
(1) "Hello"
```

As you can see, each argument printed is preceded by a name. In the case of unnamed items such as quoted strings or the results of an expressions like `x+y`, `print()` uses descriptive names such as `STRING`, `NUMBER`, `VECTOR`, or `MATRIX`.

The parenthesized numbers to the left of the output will be explained below (2.8.13).

The case of a *single* `CHARACTER` argument is special. No name is printed and the enclosing quotes are omitted.

```
Cmd> print("This string is the only argument to print")
This string is the only argument to print
```

`write()` works just like `print()` except more significant digits are printed for REAL variables.

```
Cmd> write(x,y) # default is 9 significant digits
x:
(1)      0.166666667
y:
(1)     0.0714285714
```

It's easy to forget the names of variables you create or even which ones you have created. Typing `listbrief()` prints an alphabetized list of all currently defined variables, excluding temporary and invisible variables and typing `list()` does the same, including each variable's type and dimensions.

If you want information only about some of the variables, use the variables as arguments to `list()`. For example, `list(x,y,z)` gives information about only variables `x`, `y` and `z`.

If you are not sure of the exact name of a variable, use a single CHARACTER string that contains one or more "wild card" symbols "`*`" and "`?`". A "`?`" will match any single character in a name, regardless of what it is, so `list("a??")` will list all variables whose names start with "a" and are exactly three letters long. A "`*`" will match *any* 0 or more consecutive characters in a name, so `list("ab*")` lists all variables whose name starts with "ab", `list("*yz")` lists all variables whose name ends with "yz", and `list("*w*?")` lists all variables whose name contains "w" and has at least one character after "w".

```
Cmd> x <- 5; y <- 6; z <- vector("A","B")

Cmd> list(x,y,z) # full information
x               REAL   1
y               REAL   1
z               CHAR   2

Cmd> listbrief(x, y, yy) # NOTE: yy has not been previously defined
WARNING: yy is not defined
x           y

Cmd> listbrief("y*") # list all variables whose names start with 'y'
y           yhat

Cmd> listbrief("*x") # list all variables whose names end with 'x'
boxcox      resvsindex   x

Cmd> listbrief("????plot")#list all 8 letters names ending with plot
fcolplot    frowplot     vboxplot  3 pre-defined macros
```

If you want information only about some types of variables, use any or all of the keyword phrases `real:T`, `logic:T`, `char:T`, `factor:T`, `struct:T` or `graph:T` as arguments to `listbrief()` or `list()`. Alternatively, `list(all:T,macros:F)`, for example, gives information about all variables except macros.

```
Cmd> listbrief(real:T)
DEGPERRAD    E            x            z
DELTAT       PI           y
```

If you want a list consisting only of variables with specific sizes, you can use one or more of the keyword phrases ncols:nc, nrows:nr, and ndims:nd, where nc, nr and nd are positive integers. You can use them together and with char:T, real:T, or logic:T to be even more specific. For instance,

```
Cmd> list(nrows:2,char:T) # all CHARACTER variables with 2 rows
z               CHAR   2
```

list() and listbrief() normally do not list "invisible" variables (see Sec. 2.4, 2.8.7). You can force them to be listed with keyword phrase invisible:T, which also enables listing of temporary variables (names starting with "@").

```
Cmd> az <- 5; _z <- 10 # _z is "invisible"

Cmd> @z <- 5; listbrief("*z") # only az and z are listed
az          z

Cmd> @z<-5; listbrief("*z",invisible:T)
@z          _z          az          z
```

When used as an argument to list() and listbrief(), keyword phrase keep:T allows you to save the names of the variables found. Nothing is printed, but both functions return a CHARACTER scalar or vector (see Sec. 2.8.10) whose elements are the names of the variables that would otherwise be listed.

```
Cmd> zvars <- listbrief("*z",keep:T) # or list ("*z",keep:T)

Cmd> zvars
(1) "az"
(2) "z"
```

You use delete() to remove one or more variables. Thus

```
Cmd> delete(E, PI)
```

will remove the pre-defined variables E and PI from computer memory. The variable or variables deleted can be REAL, CHARACTER or LOGICAL, or a more complex type like structure (Sec. 2.8.16).

Occasionally you may want to delete whole classes of variables. You can delete all CHARACTER variables by delete(char:T). Other keywords you can use instead of char are real, char, logical, structure, null, macro, graph and all. delete(all:T, real:F,macro:F) deletes everything except REAL variables and macros.

See Sec. 9.3.6 for a special use of delete() in a macro.

2.8.10 **Vectors – vector(), enter and enterchars** *Vectors* are 1 dimensional collections of REAL, LOGICAL, or CHARACTER values. The most common way to create a vector is to use function vector(). A synonym is cat() (short for con**cat**enate). The expression vector(1,2,3,5.5), for example, has a value which is a REAL vector of length 4 with elements 1, 2, 3, and 5.5; z <- vector(1,7,4,9,6) makes z a REAL vector of length 5 with elements 1, 7, 4, 9, and 6; and then z2 <- vector(z,-z) creates a REAL vector of length 10 containing a copy of z followed by the negatives of the elements in z.

If any argument to vector() is a matrix or array (Sec. 2.8.13, 2.8.15), its elements are

"unravelled" into a vector, the first subscript changing fastest. Thus `vector(x)` may be used to turn a matrix or array x into a vector.

```
Cmd> print(a,vector(a)) # a is 2 by 5; vector(a) is a 10-vector
a:
(1,1)          1           3           5           7           9
(2,1)          2           4           6           8          10
VECTOR:       vector(a)
(1)           1           2           3           4           5
(6)           6           7           8           9          10
```

Similarly, if an argument to `vector()` is a structure (see Sec. 2.8.16), all of whose components are the same type, all the components are unravelled and concatenated.

You can use keyword `labels` to add labels to each element.

```
Cmd> data <- vector(146,140,152,112,\
    labels:vector("Tom","Dick","Harry","Elizabeth")); data
      Tom         Dick        Harry    Elizabeth
      146          140          152          112
```

See Sec. 8.4.1 for more details on labels. See Sec. 8.9 for information on using keyword `notes` to attach descriptive notes to a vector.

The name `vector()` was introduced with version 4.00. Earlier versions used `cat()` instead of `vector()`.

You can determine the number of elements in a vector using `length()` or `nrows()`.

```
Cmd> z <- vector(1,7,4,9,6); length(z)
(1)           5

Cmd> nrows(z)
(1)           5
```

Predefined macros `enter` and `enterchars` are convenient alternatives to `vector()` for creating short REAL or CHARACTER vectors, respectively. `enter` is used the same way as `vector()` except that the commas between values are optional.

```
Cmd> enter(1 7 4 -1 6)
(1)           1           7           4          -1           6
```

Macro `enterchars` is useful for creating short CHARACTER vectors when each element consists of a single "word", with no embedded spaces, commas, parentheses or brackets. No separating commas are needed and the "words" *must not* be in quotes. In fact, if they are, the quotation marks will be taken to be part of the word.

```
Cmd> labels <- enterchars(gender height weight "strength"); labels
(1) "gender"
(2) "height"
(3) "weight"
(4) "\"strength\""     Note the "escaped" quotes
```

2.8.11 **Using subscripts with vectors** You can select or extract the elements of a vector using *subscripts* enclosed in square brackets `[...]`. There are three types of subscripts.

(i) A *positive* *subscript* is single positive integer or a vector of positive integers.

Thus, if z is vector(1,7,4,9,6), z[4] is the fourth element of z and has the value 9, and z[vector(1,1,2,2,4,5,3)] is the same as vector(1,1,7,7,9,6,4). It is an error for any element of the subscript vector to be larger than the length of the vector being subscripted (you can't extract something that isn't there!).

(ii) A *logical subscript* is LOGICAL vector of *the same length* as the vector being subscripted. The value is a vector consisting of the elements in the source vector corresponding to all the True elements in the LOGICAL vector. For z given above, z > 4 is a logical vector equivalent to vector(F,T,F,T,T) and therefore z[z>4] is the vector consisting of the elements of z with values greater than 4, that is vector(7,9,6). If no elements are selected (all elements of the subscript vector are False), the result is NULL.

(iii) A *negative subscript* is a vector containing only negative integers with no duplicates. In this case, a negative integer -5, say, specifies that element 5 of the source vector is *not* to be selected. Thus z[vector(-1,-5)] (or z[-vector(1,5)]) is the vector vector(7,4,9), that is vector z excluding elements 1 and 5. If no elements are selected (for example, z[-run(5)]), the result is NULL.

Note: You cannot mix positive and negative values in a subscript vector. Thus z[vector(1,-2)] is illegal.

An empty subscript selects the entire vector, that is z[] is equivalent to z.

A NULL subscript selects nothing; the result is NULL.

You can also change an element or elements of a vector by assigning to a subscript. Thus you can change element 4 of z to 11 by z[4] <- 11, or change elements 1 and 3 of z to 17 and 19 by z[vector(1,3)] <- vector(17,19). To change all the elements except the first to 13, use z[-1] <- 13. If the subscript is NULL or selects no elements, then the right hand side must be a scalar (single number) or NULL and the variable is not changed.

Let's repeat these examples and see what the MacAnova output looks like.

```
Cmd> z <- vector(1,7,4,9,6); z
(1)           1           7           4           9           6

Cmd> z[] # Note empty set of subscripts
(1)           1           7           4           9           6

Cmd> vector(z,-z)#Note second line of output starts with subscript 6
(1)           1           7           4           9           6
(6)          -1          -7          -4          -9          -6

Cmd> z[4]
(1)           9

Cmd> z[vector(1,1,2,2,4,5,3)] # positive subscripts can be repeated
(1)           1           1           7           7           9
(6)           6           4

Cmd> z > 4
(1) F       T       F       T       T

Cmd> z[z > 4] # use of logical subscript
(1)           7           9           6
```

```
Cmd> z[vector(-1,-5)]# negative subscripts
(1)           7           4           9

Cmd> a <- z[NULL] ; list(a) # NULL subscript returns NULL
a             NULL

Cmd> z[4] <- 11 ; z # change element 4 of z
(1)           1           7           4          11           6

Cmd> z[vector(1,3)] <- vector(17,19); z # change elements 1 and 3
(1)          17           7          19          11           6

Cmd> z[-1] <- 13; z # change all except z[1] to 13
(1)          17          13          13          13          13

Cmd> z[10] # error, 10 is too big
ERROR: subscript out of range near z[10]

Cmd> z[vector(-1,-2,-1)] # error
ERROR: duplicate negative subscripts near z[vector(-1,-2,-1)]

Cmd> nullvar <- z[vector(F,F,F,F,F)]; list(nullvar) # none selected
nullvar       NULL

Cmd> nullvar <- z[-run(5)]; list(nullvar) # none selected
nullvar       NULL

Cmd> nullvar <- z[z > 20] <- 20; z # no change is made
(1)          17          13          13          13          13

Cmd> list(nullvar) # result of z[z > 20] <- 20 is NULL
nullvar       NULL

Cmd> z[z > 20] <- run(2) # right hand side is not scalar or NULL
ERROR: wrong number of replacement items near z[z > 20] <- run(2)
```

The number in parentheses on the left directly under the Cmd> prompt is the subscript or index of the first element printed in that row.

On a standard size window, REAL data are normally printed five values per row and LOGICAL data are printed nine values per row, although this can be changed by command setoptions() (see Sec. 7.1) or by changing the size of the output window in a windowed version (Macintosh, Windows or Motif).

2.8.12 **rep() and run()** Two useful vectors are a vector consisting of several repetitions of a single value (for example vector(3,3,3,3,3)) and a vector that consists of the integers 1, 2, ..., n, where n is a positive integer (for example vector(1,2,3,4)). Such vectors can be created using functions rep() and run().

```
Cmd> rep(3,5) # 5 copies of 3
(1)           3           3           3           3           3

Cmd> run(4) # numbers 1 through 4
(1)           1           2           3           4
```

More generally, run() can have two arguments as in run(start,end)), or three arguments as in run(start,end,d).

The three argument form `run(start,end,d)` computes `vector(start,start+d, start+2*d,...)`. When `end` ≥ `start`, `d` must be positive and the last element computed is the largest value of the form `start+k*d` that does not exceed `end`. When `end` < `start`, `d` must be negative and the last element computed is the smallest value of the form `start+k*d` that is not less than end. If `(end - start)/d` is an integer, that is, `d` exactly divides `end–start`, it is guaranteed that the last element of `run(start,end,d)` will be `end`.

The two argument form `run(start,end)` assumes a step of 1 (when `start` ≤ `end`) or -1 (when `start` > `end`).

```
Cmd> run(3,5)
(1)             3             4             5

Cmd> run(3,4.6,.5) # can replace 4.6 by any x, 4.5 <= x < 5
(1)             3           3.5             4           4.5

Cmd> run(1,-1/3,-1/3)
(1)             1       0.66667       0.33333             0      -0.33333
```

See Sec. 2.14 for a more complicated way to use `rep()`.

2.8.13 **Matrices and matrix()** A *matrix* is a two dimensional array of data elements. In MacAnova, a matrix can be `REAL`, `LOGICAL`, or `CHARACTER`, although the last is rare. You can create a matrix directly from data using function `matrix()`. Its usage is `matrix(data,n)`, where `data` is a vector of elements to go into the matrix and `n` is the number of rows in the matrix. Obviously `n` must exactly divide the number of elements in `data`. The matrix created is built up, column by column, from the elements of `data`.

```
Cmd> x <- matrix(run(6),3); x # 3 divides 6 = length(run(6))
(1,1)           1             4
(2,1)           2             5
(3,1)           3             6

Cmd> matrix(run(7),3) # 3 does not divide 7 = length(run(7))
ERROR: number of rows must divide length of data

Cmd> list(x) # x has 3 rows and 2 columns
x               REAL   3     2
```

Matrices are printed row by row with (usually) up to five numbers printed per line. The numbers in parentheses at the beginning of each output line give the matrix subscripts of the first element of the line. Thus the first element in the row starting `(2,1)` is $x_{2,1}$.

Two useful functions are `nrows()` and `ncols()` which compute the number of rows and columns of the arguments, respectively. `length()` computes the total number of elements in a matrix, that is the product of the dimensions.

```
Cmd> vector(nrows(x),ncols(x),length(x)) # x as above
(1)             3             2             6
```

You can use keyword `labels` to add row and column labels.

```
Cmd> data <- matrix(vector(17,19,23, 146,112,140),3,\
    labels:structure(vector("Tom","Betty","Dick"),\
    vector("Age","Weight"))); data
              Age       Weight
Tom            17          146
Betty          19          112
Dick           23          140
```

See Sec. 2.8.16 for information on `structure()` and Sec. 8.4.1 for more details on labels. See Sec. 8.9 for information on using keyword `notes` to attach descriptive notes to a matrix.

2.8.14 **Using subscripts with matrices** To access elements of a matrix you must specify *two* subscripts, separated by a comma, identifying both the rows and columns to be selected. For example, `x[3,1]` is the element in row 3 of column 1 of `x`, more mathematically notated $x_{3,1}$, and `x[run(2),2]` is the 2 by 1 matrix consisting of rows 1 and 2 of column 2 of `x`. You can use positive subscripts, logical subscripts, and negative subscripts just as with vectors (see Sec. 2.8.11)

An empty subscript selects all rows or columns. For example, `x[3,]` selects row 3 of `x` (row 3 and all columns) as a 1 by 2 matrix and `x[,]` selects all rows and columns, that is, the entire matrix.

If any subscript is `NULL` or selects no elements (all False or a complete set of negative subscripts), the result is `NULL`.

Elements extracted from matrices are still matrices, that is, are of dimension two. If you want, you can turn the result into an ordinary vector (variable with one dimension) by function `vector()`. For example, `vector(x[3,])` is a vector of length 2. However, for most purposes, there is no need to convert. Generally, a matrix with a single column, that is a column vector, can be used as argument to a function which expects an ordinary vector. Conversely, a plain vector of length n is equivalent for most purposes to a n by 1 matrix. However, a *row vector*, that is a matrix with a single row, is not considered equivalent to a vector.

You can change elements of a matrix by assigning to subscripts. **Examples**: `x[,1] <- vector(7,8,9)` replaces column 1 of `x`; `x[3,] <- 7` sets all elements of row 3 to 7.

```
Cmd> x[3,1]   # Note the two subscripts (1,1) at the left
(1,1)            3

Cmd> x[,2] # Select column 2 of x, getting a column vector
(1,1)            4
(2,1)            5
(3,1)            6

Cmd> x[run(2),2] # Select rows 1 and 2 and column 2 of x
(1,1)            4
(2,1)            5

Cmd> x[-run(2),]  # Note 2 subscripts in output for this row vector
(1,1)            3           6   Row 3 of x

Cmd> vector(x[-run(2),])# Note this ordinary vector has 1 subscript
(1)              3           6    Row 3 of x as a vector
```

Notice the difference in the subscripting above when `vector()` is used.

```
Cmd> x[,1] <- -vector(7,8,9); x # change column 1 of x
(1,1)           -7           4
(2,1)           -8           5
(3,1)           -9           6
```

If any subscript is `NULL` or selects no elements (all False or a complete set of negative subscripts), the variable being assigned must be a scalar or `NULL` and the value of the assignment is `NULL`.

```
Cmd> nullvar <- x[-run(3),] <- 6 ; x # no change, NULL result
(1,1)            1           4
(2,1)            2           5
(3,1)            3           6

Cmd> list(nullvar)
nullvar         NULL

Cmd> x[,run(2) < 0] <- run(2)# non-scalar assigned, nothing selected
ERROR: wrong number of replacement items near
x[,run(2) < 0] <- run(2)
```

2.8.15 **Arrays – array()** An array is a set of elements arranged in a multidimensional form. Vectors and matrices are arrays with 1 and 2 dimensions, respectively. You create an array using function `array(data,n1, n2,...)` where `data` is a `REAL`, `LOGICAL`, or `CHARACTER` vector of values, and `n1`, `n2`, ... are positive integers whose product equals the number of elements in `data`. This creates an array with *dimensions* `n1`, `n2`, ... with the elements of `data` entered into the array with the first subscript varying fastest. You can also group the dimensions into a vector as in `array(data, vector(n1,n2,...))`. If `data` has length `m*n`, `array(data,m,n)` is equivalent to `matrix(data,m)` (Sec. 2.8.13).

You can subscript arrays just like matrices, except that you must provide a subscript, possibly empty, for *each* dimension, not just two. If there are k dimensions, the subscripts must have $k - 1$ commas. If any subscript is `LOGICAL` and is all False or is a negative subscript and omits all elements, the result is `NULL`.

The first argument to `array()` can itself be a matrix or array. In that case, its elements are "unravelled" with the first subscript changing fastest, the second next fastest, and so on. That is `array(data,n1,n2,...)` is equivalent to `array(vector(data), n1,n2,...)`.

```
Cmd> a <- array(run(8),vector(2,2,2)); a # create a 2 by 2 by 2
array
(1,1,1)           1           5
(1,2,1)           3           7
(2,1,1)           2           6
(2,2,1)           4           8
```

Note there there are now three subscripts labelling each line. These identify the first element in the line. All the elements in a row have the same first two subscripts with the third changing. For example, the third row consists of `a[2,1,1]` and `a[2,1,2]` (mathematical notation a_{211} and a_{212}). Study this example carefully to understand the

order in which elements are entered into an array and the order in which they are printed. In *building* the array, the first subscript changes fastest, the second is second fastest, and so on, while in *printing* the last subscript changes fastest, the next to last is second fastest, and so on.

It is not necessarily an error to use too many subscripts. Any trailing extra empty subscripts are ignored (Example: when a is a matrix, a[1,2,,,] is equivalent to a[1,2]). In addition, extra subscripts with value 1 are allowed, although the dimension of the result will be affected. Thus if a is a 2 by 3 matrix, a[2,,1] is a 1 by 3 by 1 array whose elements correspond to row 2 of a, and a[2,,vector(1,1)] is a 1 by 3 by 2 array containing two copies of row 2 of a.

```
Cmd> matrix(run(6),3)[3,,,] # extra 2 empty subscripts are ignored
(1,1)            3            6

Cmd> matrix(run(6),3)[3,,1,] # extra subscript 1 and empty subscript
(1,1,1)          3   3 dimensional array
(1,2,1)          6

Cmd> matrix(run(6),3)[3,,,1]
(1,1,1,1)          3   4 dimensional array
(1,2,1,1)          6
```

This example also illustrates that subscripts may be used directly with the value of functions that computes a vector, matrix or array.

As with vectors and matrices, you can modify arrays by assigning to subscripted elements. See Sec. 2.8.11, 2.8.14 for details. Here is an example using array a created above:

```
Cmd> a[1,,1] <- 10; a # change a[1,1,1] and a[1,2,1] to 10
(1,1,1)         10           5
(1,2,1)         10           7
(2,1,1)          2           6
(2,2,1)          4           8

Cmd> a[1,-run(2),1] <- 12; a # no change made
(1,1,1)         10           5
(1,2,1)         10           7
(2,1,1)          2           6
(2,2,1)          4           8
```

You can use keyword labels to add labels to each coordinate of an array. See Sec. 2.8.13 for an example with a matrix (two dimensional array) and Sec. 8.4.1 for details on labels. See Sec. 8.9 for information on using keyword notes to attach descriptive notes to an array.

If x is an array, length(x) computes the number of elements in x, that is the product of the dimensions.

Function dim() returns the list of dimensions of a vector, matrix, or array as a vector.

Function ndims() returns the number of dimensions of a vector, matrix, or array.

```
Cmd> length(a)
(1)          8
```

```
Cmd> print(dim(run(5)),dim(matrix(run(10),2)),\
dim(array(run(24),vector(2,3,4))))
NUMBER:
(1)          5                          Vector of length 5
VECTOR:
(1)          2          5               2 by 5 matrix
VECTOR:
(1)          2          3          4    2 by 3 by 4 array
Cmd> vector(ndims(run(5)),ndims(matrix(run(10),2)),\
ndims(array(run(24),vector(2,3,4))))
(1)          1          2          3
```

2.8.16 **Structures – structure()** Variables may also be *structures*, labelled STRUC in output from `list()`. A *structure* consists of one or more named components, each of which may be of any type – REAL, CHARACTER, LOGICAL, GRAPH, NULL or even a structure, and of any shape – vector, scalar, matrix, or array. Typically the components are related in some way. One example is the output of function `describe()` (see Sec. 2.12.1), a structure with 8 components, `n`, `min`, `q1` (lower quartile), `median`, `q3` (upper quartile), `max`, `mean`, and `var`, all referring to the same data.

Structures are useful with "ragged" data sets, which cannot be neatly put into rows and columns form. For instance if you have 10 measurements made on Saturday, 5 made on Sunday, and 15 made on Monday, you can create a structure with three components, each of which is a vector, but of differing lengths. Here is what it might look like.

```
Cmd> temperatures
component: Saturday
(1)         65         71         75         86         91
(6)         93         89         78         69         59
component: Sunday
(1)         61         73         85         83         81
component: Monday
 (1)         51         65         71         78         83
 (6)         84         85         84         81         75
(11)         69         64         59         49
```

Structures are also useful when you want to keep your data in some self-describing form. A data set, let's call it `trees`, could be stored as a structure with three components called `info`, `varnames`, and `data`. The `info` component would be a CHARACTER variable giving a description of the data and its source; the `data` component would be a REAL matrix containing the actual `data`, one row per tree; and the `names` component would contain the names of the variables recorded for each tree (columns in the `data` component matrix).

```
Cmd> trees
component: info
(1) "Made up data on 6 trees"
component: varnames
(1) "Species"
(2) "DBH"
component: data
(1,1)            1         5.6
(2,1)            1         4.5
(3,1)            1         8.9
(4,1)            2         7.3
(5,1)            2         9.9
(6,1)            2        11.3
```

See Sec. 8.9 for another method of having self-describing data.

You can extract a component of a structure by *name* using the dollar sign $ or by *number*, using square brackets ([. . .]) in the manner of subscripts.

```
Cmd> temperatures$Sunday # or temperatures[2]
(1)          61          73          85          83          81

Cmd> trees[3][run(2),] # or trees$data[run(2),]
(1,1)            1         5.6
(2,1)            1         4.5
```

In this example, *temperatures$Sunday* refers to the named component Sunday of structure *temperatures*, and trees[3] refers to the third component (data) of trees, from which, using subscripts, we extract the first 2 rows.

You can extract several components using positive, logical or negative subscript vectors.

```
Cmd> trees[run(2)] # or trees[-3], extract first two components
component: info
(1) "Made up data on 6 trees"
component: varnames
(1) "Species"
(2) "DBH"
```

Just as when used with vectors, matrices or arrays, if the subscript selects no components (is NULL or all False or a complete set of negative subscripts), the result is NULL.

```
Cmd> nullvar <- trees[-run(3)]; list(nullvar) #or trees[rep(F,3)]
a               NULL
```

As mentioned in Sec. 2.8.10, when a structure is an argument to vector() all the elements of all the components are combined into a single vector.

```
Cmd> vector(temperatures)
 (1)          65          71          75          86          91
 (6)          93          89          78          69          59
(11)          61          73          85          83          81
(16)          51          65          71          78          83
(21)          84          85          84          81          75
(26)          69          64          59          49
```

Several MacAnova functions, including `describe()`, `split()`, `coefs()`, `secoefs()`, `tabs()`, and `regpred()` compute structures as their values. Functions `structure()`, `strconcat()`, `changestr()` and `split()` allow you to create or modify structures directly. For example, the two structures above were created by

```
Cmd> temperatures <- structure(Saturday:vector(65,71,75,86,91,93,\
89,78,69,59),Sunday:vector(61,73,85,83,81),\
Monday:vector(51,65,71,78,83,84,85,84,81,75,69,64,59,49))

Cmd> trees <- structure(info:"Made up data on 6 trees",\
  varnames:vector("Species","DBH"),\
  data:matrix(vector(1,1,1,2,2,2,5.6,4.5,8.9,7.3,9.9,11.3),6))
```

See Sec. 9.1 – 9.1.3 for more information about structures.

2.8.17 **Matrix subscripts** Sometimes using ordinary subscripts to extract several elements from a matrix or array is difficult and takes lots of steps. You may be able to use a single "subscript" which is itself a matrix. This is best explained starting with an example. Suppose you wish to select elements `a[1,1,1]`, `a[2,2,2]` of 3-dimensional array `a` (Sec. 2.8.15). Then, if `i` is the 2 by 3 matrix $\begin{bmatrix}1 & 1 & 1\\ 2 & 2 & 2\end{bmatrix}$, `a[i]` extracts exactly these elements as a vector of length 2 = `nrows(i)`. Each row of `i` must consist of a complete set of positive integer subscripts specifying an element of `a`, and one element is extracted for each row of `i`. More generally, when `a` is a *k*-dimensional array and `i` is a *m* by *k* matrix with each row a set of legal subscripts for `a`, then `a[i]` is the following vector of length *m*:

```
vector(a[i[1,1],i[1,2],...,i[1,k]],a[i[2,1],i[2,2],...,i[2,k]],
    ...,a[i[m,1],i[m,2],...,i[m,k]]).
```

```
Cmd> a <- matrix(vector(11,21,31,12,22,32,13,23,33,14,24,34),3);a
(1,1)          11          12          13          14
(2,1)          21          22          23          24
(3,1)          31          32          33          34

Cmd> i <- matrix(vector(1,2,3, 1,2,3),3);i
(1,1)           1           1
(2,1)           2           2
(3,1)           3           3

Cmd> a[i] # same as diag(a); see Sec. 2.10.6
(1)          11          22          33

Cmd> i <- matrix(vector(1,2,3, 4,3,2),3); i
(1,1)           1           4
(2,1)           2           3
(3,1)           3           2

Cmd> a[i] # anti-diagonal elements of a
(1)          14          23          32

Cmd> vector(a[1,4],a[2,3],a[3,2]) # same thing
(1)          14          23          32
```

2.9 **Getting help – help() and usage()** Extensive online help is provided through commands `help()` and `usage()`. The former provides extensive (some have said *too* extensive) help with all aspects of MacAnova while the latter usually provides a one or two line outline of usage. Simply typing `help()` gets you brief summary of how to get more help.

```
Cmd> help()
Type 'help(foo)' for help on topic foo
Type 'usage(foo)' for very brief information on topic foo
Type 'help("*")' for a list of all topics
Type 'help(key:"?")' for a list of keys to topics
Type 'help(help)' for more information about help().
Type 'help(usage)' for more information about usage().
Some general topics are
 arithmetic   glm          macanova     number
 assignment   graphs       macros       options
 clipboard    graph_files  macro_files  quitting
 comments     graph_keys   macro_syntax structures
 complex      graph_ticks  matrices     subscripts
 customize    keywords     memory       syntax
 data_files   labels       models       time_series
 design       launching    notes        variables
 files        logic        NULL         vectors
```

On versions using windows (Macintosh, Windows or Motif), selecting **Help** from the or **Help** menu is equivalent to typing `help()`.

`usage(topic)` gives a very abbreviated summary of how to use a function such as `anova()`, while `help(topic)` gives complete help.

```
Cmd> usage(makecols)
makecols(x,var1,var2, ...), where x is a REAL matrix, var1, var2, ...
  unquoted variable names
makecols(x,vector("var1","var2", ... ))
```

```
Cmd> help(makecols)
makecols(x,name1,...,namek), where x is a REAL matrix and name1, ...,
namek are unquoted names, creates new REAL vectors name1, name2,
... from the columns of x.  Thus makecols is a sort of inverse to
hconcat().

makecols(x,vector("name1","name2",...,"namek")) is an alternative

*****  Interrupt  *****    Interrupt key hit to stop output
```

To get information on several topics, include them all in the argument list as in `usage(anova,coefs)` or `help(regress,resid)`.

You must quote names of the special syntax words `for`, `while`, `break`, `breakall`, `if`, `else`, and `elseif` (`help("while","for")` (see Sec. 9.2). Similarly, you must quote any topic name longer than 12 characters (`help("transformations")`).

One defect of `help()` in windowed versions (Macintosh, Windows, Motif) is that when help for a topic is long, all of it is printed at once so that you have to scroll back to see the start. If you include keyword phrase `scrollback:T` as an argument to `help()` the

scrolling back is done automatically. See also Sec. 8.1.3 for information on option `scrollback` which enables such scrolling back for any command producing long output.

On the Macintosh, when a function or topic name has been selected in the command window by double clicking on it, **Help** from the **File** menu or **⌘H** will obtain help on that topic.

2.9.1 **Using the help() wild card characters – "*" and "?"** When you can't remember the full name of a command but know or are able to guess part of it, you can use a *pattern* in the form of a quoted string containing one or more of the "wild card" characters "*" and "?". All topics whose names are matched by the pattern are listed. A "*" matches *any* set of 0 or more consecutive letters and a "?" matches *any* single letter. For example, `help("res*")` lists all topic names beginning with `res`, `help("*line*")` lists all topic names containing `line`, and `help("*i*e*t")` lists all topic names containing `i` and `e`, in that order, and ending with `t` (`assignment` and `lineplot`, for example). `help("*pl?t")` lists all topic names whose last 4 characters or `p`, `l`, any character, and `t` respectively (`plot`, `lineplot` and `split`, for example).

```
Cmd> help("res*") # find all topics starting with "res"
resid        restore      resvsindex   resvsrankits resvsyhat
For help on topic foo, enter help(foo) or help("foo")

Cmd> help("*plot*") # find all topics containing "plot"
boxplot   colplot   fchplot   flineplot frowplot  plot      showplot
chplot    fboxplot  fcolplot  fplot     lineplot  rowplot   vboxplot
For help on topic foo, enter help(foo) or help("foo")

Cmd> help("*pl?t") # like *plot but also lists split
boxplot   fboxplot  flineplot lineplot  showplot
chplot    fchplot   fplot     plot      split
colplot   fcolplot  frowplot  rowplot   vboxplot
For help on topic foo, enter help(foo) or help("foo")

Cmd> help("c*o*e")
console   convolve  customize

Cmd> help("???????????") # all topics with 11 letter names
adddatapath arginfo_fun dos_windows graph_ticks time_series
appendnotes attachnotes graph_files macro_files
For help on topic foo, enter help(foo) or help("foo")
```

If only one topic matches the pattern, the full help is listed.

`help("*")` produces a list of all the more than 300 help topics. These include all commands and functions, and many more general topics.

2.9.2 **Using help() index keys** If you haven't a clue as to topics you might be interested in, you can get lists of topics associated with any one of up to 32 index "keys". To get a list of topics associated with a key, say `Regression`, type `help(key:"regression")`. `help(key:"reg")` would do just as well since you need only use as many letters as are necessary to identify the full key, and upper and lower case letters are considered the same. To get a list of all the index keys, type `help(key:"?")`.

```
Cmd> help(key:"regression") # or help(key:"reg")
The following help topics concern Regression
coefs      glmfit      poisson     regpred     resvsindex   screen
design     glmpred     power2      regress     resvsrankits secoefs
glm        logistic    probit      regs        resvsyhat    wtregress
glm_keys    models     regcoefs    resid       robust       yhat
For help on topic foo, enter help(foo) or help("foo")

Cmd> help(key:"?") # get list of all keys
Type 'help(key:"heading")', where heading is in following list:
ANOVA                   General                 Plotting
Categorical Data        Input                   Probabilities
CHARACTER Variables     LOGICAL Variables       Random Numbers
Combining Variables     NULL Variables          Regression
Comparisons             Macros                  Residuals
Complex Arithmetic      Matrix Algebra          Structures
Confidence Intervals    Missing Values          Syntax
Control                 Multivariate Analysis   Time Series
Descriptive Statistics  Operations              Transformations
Files                   Ordering                Variables
GLM                     Output
```

The help information is normally read from a file, `MacAnova.hlp`, although you can read from a alternative file using keyword `file` (see Sec. 8.6).

2.9.3 **Getting help on macros – macrousage()** You can use `help()` and `usage()` to get information on all the standard pre-defined macros such as `boxcox` and `colplot`. Other macros, possibly ones you have written yourself (see Sec. 9.3) or ones you have read from a file (Sec. 7.5) will usually not have help information. However, a thoughtful programmer will always include one or more comment lines (lines that start with "#") in a macro that describe how it is used. If `mymacro` is the name of a macro, `macrousage(mymacro)` will print all such lines, if any. You can include several macros in the argument list.

```
Cmd> macrousage(boxcox,readcols)
boxcox:
 # usage: boxcox(x,power), x a vector or matrix, power a scalar
readcols:
 # readcols(filename,name1,...,namek [,echo:T or F]), only filename
    quoted
```

An alternative usage is, for example, `macrousage(vector("boxcox", "readcols"))`.

2.10 **Operations with vectors, matrices, arrays and structures** You can use all the arithmetic and logical operations with arbitrarily shaped variables, and indeed, with structures, subject to certain limitations. In addition, MacAnova implements many standard matrix operations.

2.10.1 **Transformations** If the argument to any of the one argument mathematical functions listed in Sec. 2.8.6 or the first argument to `round(x,p)` or `rational(x,a, b)` is a vector, matrix, or array the result is a variable of the same size and shape, with

the function applied to each element of the argument.

```
Cmd> sqrt(vector(2,4,7))
(1)       1.4142            2       2.6458

Cmd> log10(matrix(run(6),2))
(1,1)            0      0.47712      0.69897
(2,1)      0.30103      0.60206      0.77815
```

If the argument is a structure with REAL components, the result is also a structure with the same pattern of components.

```
Cmd> sqrt(temperatures)
component: Saturday
(1)       8.0623       8.4261       8.6603       9.2736       9.5394
(6)       9.6437        9.434       8.8318       8.3066       7.6811
component: Sunday
(1)       7.8102        8.544       9.2195       9.1104            9
component: Monday
 (1)       7.1414       8.0623       8.4261       8.8318       9.1104
 (6)       9.1652       9.2195       9.1652            9       8.6603
(11)       8.3066            8       7.6811            7
```

For the two argument functions, atan(x,y) and hypot(x,y), x and y must be the same size and shape except one may have extra dimensions with length 1. If x and y are structures, their matching components must be the same size and shape except for extra dimensions of length 1.

2.10.2 **Arithmetic with vectors, matrices and arrays** Vectors, matrices, and arrays can be used in arithmetic expressions, just like individual numbers. Generally, the sizes and shapes of the variables on each side of an operator must match but there are some important exceptions.

Two REAL or LOGICAL variables x and y may be used in an expression x *op* y, where *op* is one of +, -, *, /, ^ , or %%, as long as one of the following conditions is true:

(i) Both x and y have exactly the same dimensions. Each element of the result is computed by combining the corresponding elements of x and y using *op*.

(ii) One of x or y is a *scalar*, that is, has length 1. The scalar is combined with each element of the other argument.

(iii) One of x or y is a *m* by n matrix and the other is a vector of length *m* or a *m* by 1 matrix, that is a column vector of length *m* . The vector is combined with each *column* of the matrix resulting in new matrix of the same shape.

(iv) One of x or y is a *m* by *n* matrix and the other is a 1 by *n* matrix, that is, a row vector of length *n*. The row vector is combined with each *row* of the matrix resulting in new matrix of the same shape.

(v) One of x or y is a vector or column vector of length *m* and the other is a row vector of length *n*. Each element of the column vector is combined with each element of the row vector to form a *m* by *n* matrix.

As usual, if an operand is LOGICAL, True is interpreted as 1 and False as 0.

Rule (i) with operators + and – represents the usual mathematical definition of matrix addition and subtraction; rule (ii) with operator * represents scalar multiplication. Except for these cases the operations are not completely standard mathematical matrix operations but are still extremely useful. Note that * represents *elementwise* multiplication (sometimes called the Hadamard product in case (i)), not matrix multiplication (see Sec. 2.10.4).

Rules (i) – (v) also apply to comparison operators ==, !=, <, >, <=, and >=, logical operators || and &&, and bitwise operators %&, %| and %^ (Sec. 2.8.4 and 2.8.5).

```
Cmd> x <- matrix(run(6),3); x
(1,1)            1            4
(2,1)            2            5
(3,1)            3            6

Cmd> x*5  # Rule (ii), op = '+' ; 5*x would yield the same
(1,1)            5           20   matrix multiplied by scalar
(2,1)           10           25
(3,1)           15           30

Cmd> 2^x  #  Rule (ii), op = '^'
(1,1)            2           16
(2,1)            4           32
(3,1)            8           64

Cmd> x + vector(6,5,4,3,2,1) #length of vector ≠ number of rows of x
ERROR: dimension mismatch for + near
x + vector(6,5,4,3,2,1) #length of vector ≠ number of rows of x

Cmd> vector(x) + vector(1,1,2,2,3,3)# Rule (i), op = '+'
(1)            2            3            5            6            8
(6)            9

Cmd> x <= 4 # Rule (ii), op = '<=' (Sec. 2.8.4)
(1,1) T       T
(2,1) T       F
(3,1) T       F

Cmd> x + vector(1,3,5) # Rule (iii), op = '+'
(1,1)            2            5
(2,1)            5            8
(3,1)            8           11

Cmd> x == matrix(vector(2,4),1)#Rule (iv), op = '==' (Sec. 2.8.4)
(1,1) F       T
(2,1) T       F
(3,1) F       F

Cmd> run(3)  %& vector(1, 2)'#Rule (v), op = '%&' (Sec. 2.8.5)
(1,1)            1            0
(2,1)            0            2
(3,1)            1            2
```

2.10.3 **Matrix transposition** If x is an *m* by *n* matrix with elements x[i,j], x' computes the transpose of x, that is the matrix *y* with elements y[i,j] = x[j,i].

```
Cmd> print(x, transposex:x')
x:
(1,1)            1            4
(2,1)            2            5
(3,1)            3            6
transposex:
(1,1)            1            2            3
(2,1)            4            5            6
```

When x is a vector of length *m* , say run(m), then x' is a 1 by *m* matrix, a row vector. If x is a row vector of length *m* , that is a 1 by *m* matrix, then x' is a *m* by 1 matrix, a column vector. Thus run(5)'' is a 5 by 1 matrix.

If x is an array, x' is an array with the same number of dimensions obtained by reversing the order of the dimensions.

For compatibility with earlier versions of MacAnova, t(x) can be used instead of x' to compute a transpose.

2.10.4 **Matrix multiplication** If x and y are REAL matrices with dimensions that conform to each other, that is ncols(x) = nrows(y), then you can compute their matrix product by x %*% y. It is an error if either x or y contain any MISSING values.

There are two other matrix multiplication operators – %c% and %C%. x %c% y is equivalent to x' %*% y and x %C% y is equivalent to x %*% y'. For %c%, nrows(x) = nrows(y) is required and for %C%, ncols(x) = ncols(y) is required.

```
Cmd> y <- matrix(vector(1,3,2,4),2);x <- matrix(vector(4,1,2,3),2)

Cmd> print(x,y)
x:
(1,1)            4            2
(2,1)            1            3
y:
(1,1)            1            2
(2,1)            3            4

Cmd> print(x %*% y, x %c% y, x %C% y) # all three products
MATRIX:
(1,1)           10           16
(2,1)           10           14
MATRIX:
(1,1)            7           12
(2,1)           11           16
MATRIX:
(1,1)            8           20
(2,1)            7           15

Cmd> matrix(run(4),2) %*% rep(10,5)
ERROR: Dimension mismatch: 2 by 2 %*% 5 by 1 near
matrix(run(4),2) %*% rep(10,5)
```

2.10.5 **Matrix inversion and linear equation solving** If A is a square matrix and B is the same size and satisfies $A\ B = I$, where I is the identity matrix (1's down the diagonal and 0's elsewhere), then B is the *inverse* of A and is usually notated A^{-1}. You can compute A^{-1} by `ainv <- solve(a)`. If the inverse does not exist (A is singular or non-invertible), an error message is printed.

```
Cmd> a <- matrix(run(4),2); ainv <- solve(a)

Cmd> print(a, ainv)
a:
(1,1)            1            3
(2,1)            2            4
ainv:
(1,1)           -2          1.5
(2,1)            1         -0.5

Cmd> a %*% ainv # should be the identity matrix
(1,1)            1            0
(2,1)            0            1

Cmd> solve(matrix(vector(1,2, 2,4),2))
ERROR: argument to solve() is singular

Cmd> solve(matrix(vector(27.7,7.4,2.6,23.5,23.8,23.4),3))#3 by 2 arg
ERROR: argument to solve() not square matrix
```

Because computer arithmetic is not always exact, it is not guaranteed that `solve()` will actually detect that a matrix is singular.

If `a` is an invertible n by n matrix, and `b` is a n by k matrix, `a %\% b` computes the solution `x` to the set of linear equations summarized by `a %*% x = b`. That is, `a %*% (a %\% b)` should be the same as `b` except for rounding error. This is mathematically, but not computationally equivalent to `solve(a) %*% b`. The expression `a %\% b` might be interpreted as "dividing" `b` by `a` on the left.

```
Cmd> b <- matrix(vector(5,-7,1, 2),2); b
(1,1)            5            1
(2,1)           -7            2

Cmd> x <- a %\% b

Cmd> print(x, aTimesx:a %*% x)
x:
(1,1)        -20.5            1
(2,1)          8.5            0
aTimesx:  a %*% x = b
(1,1)            5            1
(2,1)           -7            2
```

`solve(a,b)` does exactly the same computation as `a %\% b`.

```
Cmd> solve(a, b)
(1,1)        -20.5            1
(2,1)          8.5            0
```

Similarly, if `a` is an invertible n by n matrix, and `c` is a k by n matrix, `c %/% a` computes the solution `x` to the set of linear equations summarized by `x %*% a = b`.

That is, `(c %/% a) %*% a` should be the same as `c` except for rounding error. `c %/% a` is mathematically but not computationally equivalent to `c %*% solve(a)`. The expression `c %/% a` might be interpreted as "dividing" `c` by `a` on the right.

```
Cmd> c <- vector(1, 1)'; x <- c %/% a

Cmd> print(x, xTimesa:x %*% a)
x:
(1,1)          -1           1    Solution to x %*% a = c
xTimesa:
(1,1)           1           1    x %*% a is indeed c
```

`rsolve(a,c)` does exactly the same computation as `c %/% a`.

```
Cmd> rsolve(a,c)
(1,1)          -1           1
```

2.10.6 **Other matrix functions – det(), trace(), hconcat(), vconcat(), diag(), dmat(), nrows(), ncols(), select(), reverse()** You can use `det()` and `trace()` to compute the determinant and trace (sum of diagonal elements) of a square matrix.

```
Cmd> b <- matrix(vector(2891,1851,1302,2139),2); b
(1,1)        2891        1302
(2,1)        1851        2139

Cmd> det(b) # b[1,1]*b[2,2] - b[1,2]*b[2,1]
(1)   3.7738e+06

Cmd> det(b,mantexp:T) # base 10 mantissa and exponent form
(1)      3.7738           6     Vector of length 2

Cmd> trace(b) # b[1,1] + b[2,2]
(1)         5030
```

Especially in multivariate analysis you may want to combine two or more matrices into a larger matrix. You can do this in MacAnova using functions `hconcat()` (**h**orizontal concatenation) and `vconcat()` (**v**ertical concatenation).

```
Cmd> a1<-matrix(run(6),3);a2<-vector(1,0,-1);a3<-matrix(rep(1,6),3)

Cmd> print(a1,a2,a3)
a1:
(1,1)           1           4
(2,1)           2           5
(3,1)           3           6
a2:
(1)          1           0          -1
a3:
(1,1)           1           1
(2,1)           1           1
(3,1)           1           1

Cmd> hconcat(a1,a2,a3)
(1,1)           1           4           1           1           1
(2,1)           2           5           0           1           1
(3,1)           3           6          -1           1           1
```

```
Cmd> vconcat(a1,a3)
(1,1)           1           4
(2,1)           2           5
(3,1)           3           6
(4,1)           1           1
(5,1)           1           1
(6,1)           1           1
```

You can extract the diagonal elements of a matrix using function `diag()`. When `a` is a matrix, `diag(a)` returns a vector containing the diagonal elements of `a`, that is `vector(a[1,1], a[2,2], …)`. Argument `a` can be a `REAL`, `LOGICAL` or `CHARACTER` matrix and need not be square. If `a` is not square, the length of `diag(a)` is `min(nrows(a),ncols(a))`.

```
Cmd> diag(a1) # same as vector(a1[1,1],a1[2,2])
(1)             1           5
```

If you want just one element from each row of a matrix, you can use `select()`. When `x` is a matrix with m rows and n columns, and `k` is a vector with m of positive integers $\le n$, then `select(k,x)` returns `vector(x[1,k[1]], x[2,k[2]],..., x[m,k[m]])`. When m_1 = `nrows(k)` < m, only elements from the first m_1 of `x` are selected.

```
Cmd> select(vector(1,3),a1') # or vector(a1'[vector(1,3),])
(1)             1           6
```

When `x` is a square matrix, `select(run(nrows(x),x)` selects the diagonal elements and is equivalent to `diag(x)`. `select(run(nrows(x),1),x)` selects the cross diagonal `vector(x[1,n],x[2,n-1], ..., x[n,1])`.

Argument `k` can also be a `LOGICAL` vector, with True and False interpreted as 2 and 1, respectively. If `x` has two columns, `select()` can be used to select column 1 or column 2 of `x` depending on whether `k[i]` is False or True.

```
Cmd> select(vector(T,F,F),a1) # same as select(vector(2,1,1),a1)
(1)             4           2           3.
```

You can create a diagonal matrix, that is with all off-diagonal elements 0, using function `dmat()`. This has two usages, `dmat(vec)` where `vec` is a vector, and `dmat(n,val)` where `n` is a positive integer and `val` is a scalar (single data element). Arguments `vec` or `val` can be `REAL`, `LOGICAL` or `CHARACTER`. Both usages return a square matrix of the same type as `vec` or `val`, all of whose off diagonal elements are 0 , False or "" depending on whether `vec` or `val` is `REAL`, `LOGICAL` or `CHARACTER`. `dmat(vec)` is `n` by `n` where `n` = `length(vec)`, and has diagonal elements taken from `vec`. `dmat(n,val)` is equivalent to `dmat(rep(val,n))`, that is, the result is `n` by `n`, with the diagonal consisting of `n` copies of `val`. In particular, `dmat(n,1)` computes the n by n identity matrix.

```
Cmd> b <- matrix(run(9)^2,3); b
(1,1)           1          16          49
(2,1)           4          25          64
(3,1)           9          36          81
```

```
Cmd> diag(b)
(1)             1            25            81

Cmd> dmat(diag(b))
(1,1)           1             0             0
(2,1)           0            25             0
(3,1)           0             0            81

Cmd> dmat(2,7)
(1,1)           7             0
(2,1)           0             7
```

You can reverse the order of the rows of a matrix or vector using `reverse()`.

```
Cmd> reverse(matrix(run(12),3)) # reverses order of rows
(1,1)           3             6             9            12
(2,1)           2             5             8            11
(3,1)           1             4             7            10
```

2.11 **Reading data from a file** Although you can type small data sets directly into MacAnova using `vector()` (Sec. 2.8.10) and `matrix()` (Sec. 2.8.13), it is often more convenient to read data from a file on your hard or floppy disk. The file might have been created in a word processor such as Microsoft Word, a text editor such as Edit in DOS, or by a spread sheet or data base program.

MacAnova can read data only from plain *text* files (type `TEXT` on the Macintosh). A plain text file (sometimes called an ASCII file) is one with no additional information such as font or point size. Hence, if you use a word processor to create or edit data files to be read by MacAnova, it is *essential* that they be saved as Text or ASCII files. In some programs such as Microsoft Word, you may have to click on a **File Format** button to display choices; in others the options may be displayed in the dialog box brought up by **Save** or **Save As...**. If you do not choose a Text format, MacAnova will probably not be able to read a file created in a word processor.

The MacAnova functions for reading data from a plain text file are `vecread()` and `matread()`. In addition there are two pre-defined macros, `readcols` and `getdata`.

Commands which read a file must have the file name as their first argument. This must be a quoted string or `CHARACTER` variable giving the name of the data file. In the windowed versions (Macintosh, Windows and Motif), when the file name is `""`, that is, it consists of two adjacent quotation marks, you are presented with a dialog box with a scrolling list of files from which you can select the file.

All the commands which read from a text file, including `vecread()`, `matread()` and `macroread()` can also "read" from MacAnova `CHARACTER` variables using keyword phrase `string:CharVar` as first argument instead of the file name. See Sec. 8.3.2 for details.