

Chapter 2

Getting Started with the SAS/IML Matrix Programming Language

Contents

2.1	Overview of the SAS/IML Language	18
2.2	Creating Matrices	18
2.2.1	Printing a Matrix	19
2.2.2	The Dimensions of a Matrix	20
2.2.3	The Type of a Matrix	21
2.2.4	The Length of a Character Matrix	22
2.3	Using Functions to Create Matrices	24
2.3.1	Constant Matrices	24
2.3.2	Vectors of Sequential Values	25
2.3.3	Pseudorandom Matrices	27
2.4	Transposing a Matrix	28
2.5	Changing the Shape of Matrices	29
2.6	Extracting Data from Matrices	30
2.6.1	Extracting Rows and Columns	31
2.6.2	Matrix Diagonals	33
2.6.3	Printing a Submatrix or Expression	35
2.7	Comparison Operators	36
2.8	Control Statements	38
2.8.1	The IF-THEN/ELSE Statement	38
2.8.2	The Iterative DO Statement	39
2.9	Concatenation Operators	41
2.10	Logical Operators	43
2.11	Operations on Sets	46
2.12	Matrix Operators	47
2.12.1	Elementwise Operators	47
2.12.2	Matrix Computations	49
2.13	Managing the SAS/IML Workspace	51

2.1 Overview of the SAS/IML Language

SAS/IML is a programming language for high-level, matrix-vector computations. Matrices are rectangular arrays that usually contain numbers. A matrix that contains character data is often explicitly called a *character matrix*. In statistical programming, matrices often hold data for analysis. Each row of the matrix is an observation; each column of the matrix is a variable.

If your data are in a matrix, you can carry out many statistical operations by using matrix operations. The SAS/IML language has functions and matrix operations that enable you to manipulate matrices as a unit, regardless of the number of rows or columns in the matrix. For an example, see the section “Case Study: Standardizing the Columns of a Matrix” on page 83.

Operations on numerical matrices are also used to describe a wide variety of statistical techniques, including ordinary least squares (OLS) regression and principal component analysis.

This chapter is an introduction to the SAS/IML syntax. It includes basic information about how to define matrices, compare quantities, and call functions and subroutines. It includes a description of basic programming statements such as IF-THEN/ELSE and the iterative DO statement.

2.2 Creating Matrices

A matrix is an $n \times p$ array of numbers or character strings. The integers n and p are the dimensions of the matrix. The row dimension is n ; the column dimension is p . A vector is a special case of a matrix. An $n \times 1$ matrix is called a *column vector*, whereas a $1 \times n$ matrix is called a *row vector*. A 1×1 matrix is called a *scalar*. In general, this book refers to any SAS/IML variable as a matrix, regardless of its dimensions.

In a SAS/IML program, all variables are matrices, so you do not need to specify the type of a variable. Furthermore, matrices are dynamically reassigned as needed, so you do not need to specify the size or the type (numeric or character) of a matrix. For example, the following statements are all valid:

```
/* create matrices of various types and sizes */
x = 1;                               /* scalar */
x = {1 2 3};                          /* reassign to row vector */
y = {1 2 3, 4 5 6};                   /* 2 x 3 numeric matrix */
y = {"male" "female"};               /* reassign to 1 x 2 character matrix */
```

See the section “Running a PROC IML Program” on page 7 for instructions on how to run SAS/IML programs. The first statement creates **x** as a numerical scalar matrix. The second statement redefines **x** as a numerical row vector; spaces separate entries in different columns. The third statement defines **y** as a 2×3 numerical matrix; commas indicate a new row. The last statement redefines **y** as a 1×2 character matrix.

Programming Tip: When defining a matrix, use a comma to indicate a new row.

2.2.1 Printing a Matrix

You can use the PRINT statement to display the value of one or more matrices. The following statement displays the values of the matrices defined in the previous section:

```
print x, y;
```

Figure 2.1 Numeric and Character Matrices

		x		
		1	2	3
y				
male				
female				

Notice that a comma in the PRINT statement indicates that the second matrix should be displayed on a new row. If you omit the comma, the matrices are displayed side by side.

The PRINT statement has four useful options that affect the way a matrix is displayed:

COLNAME=*matrix*

specifies a character matrix to be used for column headings.

FORMAT=*format*

specifies a valid SAS or user-defined format to use when printing matrix values.

LABEL=*label*

specifies a label for the matrix. If this option is not specified, the name of the matrix is used as a label.

ROWNAME=*matrix*

specifies a character matrix to be used for row headings.

These options are specified by enclosing them in square brackets after the name of the matrix that you want to display, as shown in the following example:

```
/* print marital status of 24 people */
ageGroup = {"<= 45", "> 45"};          /* headings for rows */
status = {"Single" "Married" "Divorced"}; /* headings for columns */
counts = { 5      5      0,          /* data to print */
           2      9      3      };
print counts[colname=status
             rowname=ageGroup
             label="Marital Status by Age Group"];
```

```
pct = counts / 24;          /* compute proportions */
print pct[format=PERCENT7.1]; /* print as percentages */
```

Figure 2.2 Matrices Displayed with PRINT Options

Marital Status by Age Group			
	Single	Married	Divorced
<= 45	5	5	0
> 45	2	9	3
pct			
	20.8%	20.8%	0.0%
	8.3%	37.5%	12.5%

You can also use these options by specifying their first letters: **C=**, **F=**, **L=**, and **R=**.

2.2.2 The Dimensions of a Matrix

You can determine the dimensions of a matrix by using the `NROW` and `NCOL` functions, as shown in the following statements:

```
/* dimensions of a matrix */
n_x = nrow(x);
p_x = ncol(x);
print n_x p_x;
```

Figure 2.3 Dimensions of a Matrix

<code>n_x</code>	<code>p_x</code>
1	3
<code>n_u</code>	<code>p_u</code>
0	0

A matrix that contains no elements is called an *empty* matrix. There are several reasons why a matrix can be empty:

- It has not been defined.
- Its memory was freed by using the `FREE` statement.
- It is the result of a query that returned the empty set (such as the intersection of disjoint sets).

The output from the following statements (see [Figure 2.4](#)) shows that each dimension of an empty matrix is zero.

```

/* dimensions of an empty matrix */
n_u = nrow(empty_matrix);
p_u = ncol(empty_matrix);
print n_u p_u;

```

Figure 2.4 Dimensions of an Empty Matrix

	n_u	p_u
	0	0

2.2.3 The Type of a Matrix

A matrix is either numeric or character or undefined; you cannot create a matrix that contains both numbers and character strings. You can determine whether a matrix is numeric or character by using the TYPE function, as shown in the following statements:

```

/* determine the type of a matrix */
x = {1 2 3};
y = {"male" "female"};
type_x = type(x);
type_y = type(y);
print type_x type_y;

```

Figure 2.5 Types of Matrices

	type_x	type_y
	N	C

If a matrix is numeric, the TYPE function returns the character 'N'. If a matrix is character, the TYPE function returns the character 'C'. If a matrix is empty, then the TYPE function returns 'U' (for “undefined”) as shown in the following statements:

```

/* handle an empty or undefined matrix */
type_u = type(undefined_matrix);
if type_u = 'U' then
  msg = "The matrix is not defined.";
else
  msg = "The matrix is defined.";
print msg;

```

Figure 2.6 Result of Handling an Undefined Matrix

	msg
	The matrix is not defined.

Programming Tip: A matrix is either numeric or character or undefined; you cannot create a matrix that contains both numbers and character strings.

2.2.4 The Length of a Character Matrix

SAS/IML character matrices share certain similarities with character variables in the DATA step. In the DATA step, the length of a character variable is determined when the variable is initialized: either explicitly by using the LENGTH statement or implicitly by the length of the first value for the variable. Similarly, every element in a SAS/IML character matrix has the same number of characters: the length of the longest element. This length is determined *when the matrix is initialized*. Strings shorter than the longest element are padded with blanks on the right.

For example, the following statements define a character matrix with length 4:

```
c = {"Low" "Med" "High"};          /* maximum length is 4 characters */
```

The matrix `c` is initialized to have length 4, the length of its longest character string. Shorter strings such as “Low” are padded on the right so that the first element of `c` is stored as `Low□` where `□` indicates a blank character.

Programming Tip: The length of a character matrix is determined by the length of its longest element at the time it is created.

You can find the length of a character matrix by using the `NLENG` function, which returns a single number. You can find the number of characters for each element of a character matrix by using the `LENGTH` function, which returns a number for each element. These functions are shown in the following statements:

```
/* find the length of a character matrix and of each element */
nlen = nlen(c);          /* length of matrix          */
len = length(c);        /* number of characters in each element */
print nlen len;
```

Figure 2.7 Lengths of Elements in a Character Matrix

	nlen	len		
	4	3	3	4

Notice that the `LENGTH` function returns a numerical matrix that is the same dimension as its input argument. For example, the ij th element of `len` is the number of characters in the ij th element of `c`. (Strictly speaking, both functions return numbers that represent bytes. Because each English character is one byte long, the numbers also give the number of characters for matrices that contain English strings.)

When you set the value of an element of an existing matrix, the value is truncated if it is too long, as shown in the following statements:

```
/* assign a long string to a matrix with a shorter length */
c[2] = "Medium";          /* value is truncated! */
print c;
```

Figure 2.8 A Truncated Character String

```

              c
          Low Medi High
```

The output from these statements is shown in Figure 2.8. The matrix `c` was initialized to have length 4, so when you assign a longer string to an element, only the first four characters fit into the matrix element.

You cannot dynamically change the length of a character matrix, but you can copy its contents to a vector that has a longer (or shorter) character length. The following statements use the PUTC function in Base SAS software to copy a vector of values:

```
/* copy character strings to a matrix with a longer length */
c = {"Low" "Med" "High"}; /* maximum length is 4 characters */
d = putc(c, "$6.");       /* copy into vector with length 6 */
d[2] = "Medium";        /* value fits into d without truncation */
nlen = nlength(d);
print nlen, d;
```

Figure 2.9 Result of Changing the Length of a Character Vector

```

              nlen
                  6
              d
          Low  Medium High
```

In the previous statements, the PUTC function applies the \$6. format to every element of `c`. The PUTC function returns a matrix with the same dimensions as `c`. This matrix is stored into `d`.

Notice that `c` contains a vector of character strings, but that the PUTC function acted on each element. This is generally true: you can pass a matrix of values to Base SAS functions and expect them to act on each element.

Programming Tip: You can call functions in Base SAS software from SAS/IML programs. In most cases, these functions act on each element of a matrix.

Strings that are smaller than the maximum length of a character matrix are padded with blanks on

the right. You can see this in [Figure 2.9](#) by noticing that there is more space between the words “Low” and “Medium” than between the words “Medium” and “High.” The value stored in the first element of `d` is “Low□□□” where □ indicates a blank character.

The padding of character strings with blanks on the right can cause a problem when you use the `CONCAT` function or the string concatenation operator (+) to concatenate strings. The solution to this problem is to use the `TRIM` function in Base SAS software to remove trailing blanks, as shown in the following statements:

```
/* use the '+' operator to concatenate strings */
msg1 = "I like the " + d[1] + " value.";          /* blanks! */
msg2 = "I like the " + trim(d[1]) + " value.";    /* no blanks */
print msg1, msg2;
```

Figure 2.10 Result of Concatenating Strings and Removing Trailing Blanks

msg1
I like the Low value.
msg2
I like the Low value.

There are times when a character string also has leading blanks. You can use the `STRIP` function in Base SAS software to remove both leading and trailing blanks. In most situations that involve string concatenation, you will want to remove leading and trailing blanks.

Programming Tip: When concatenating character strings, use the `STRIP` function to remove leading and trailing blanks from elements of a character matrix.

2.3 Using Functions to Create Matrices

The SAS/IML matrices in the previous section were created by typing in the elements. More typically, matrices are obtained by using SAS/IML functions to generate data or by reading data from a SAS data set. This section describes creating matrices by using SAS/IML functions; creating matrices by reading a SAS data set is covered in Chapter 3, “Programming Techniques for Data Analysis.”

2.3.1 Constant Matrices

The simplest matrix is a constant matrix. In SAS/IML, the `J` function creates a constant matrix. The syntax is `J(nrow, ncol, value)`, although you can omit the third argument, which defaults to the

value 1. For example, the following statements create several constant matrices:

```

/* create constant matrices */
c = j(10, 1, 3.14);          /* 10 x 1 column vector      */
r = j( 1, 5);                /* 1 x 5 row vector of 1's   */
m = j(10, 5, 0);            /* 10 x 5 matrix of zeros    */
miss = j(3, 2, .);          /* 3 x 2 matrix of missing values */

```

The first statement creates a column vector with ten rows; each element has the value 3.14. The second statement creates a row vector with five columns; each element is a 1 because the *value* argument is omitted. The third statement creates a 10 × 5 matrix of zeros. The last statement creates a 3 × 2 matrix in which every element is a missing value.

Programming Tip: Do not confuse an empty matrix with a matrix that contains missing values or with a zero matrix. An empty matrix has no rows and no columns. A matrix that contains missing values has at least one row and column, as does a matrix that contains zeros.

You can also use the REPEAT function to create a constant matrix or, more generally, a matrix with a repeating pattern of values. The REPEAT function creates a new matrix by repeating a given matrix a specified number of times in each dimension, as shown in the following example:

```

/* create a matrix by repeating values from another matrix */
g = repeat({0 1}, 3, 2);      /* repeat the pattern 3 times down */
print g;                     /* and 2 times across              */

```

Figure 2.11 A Repeated Pattern of Values

g				
	0	1	0	1
	0	1	0	1
	0	1	0	1

The REPEAT function takes three arguments: a matrix of values, the number of times that matrix should be repeated in the vertical dimension, and the number of times that matrix should be repeated in the horizontal dimension. In the example, the vector {0 1} is repeated three times down the rows and twice across the columns.

You can also use the J function and the REPEAT function to create character matrices.

2.3.2 Vectors of Sequential Values

The next simplest matrix to construct is a matrix in which entries follow an arithmetic progression. The DO function (not to be confused with the iterative DO statement) creates a vector with elements that follow an arithmetic series. The syntax is DO(*start*, *stop*, *increment*). Similar to the DO function is the “colon operator,” which can create an arithmetic series with an increment of 1 or −1.

(The colon operator is also known as the *index creation operator* since it is often used to create an index *start:stop*.) The following statements demonstrate these functions:

```
/* create a vector of sequential values */
i = 1:5;                /* increment of 1          */
j = 5:1;                /* increment of -1        */
k = do(1, 10, 2);      /* odd numbers 1, 3, ..., 9 */
print i, j, k;
```

Figure 2.12 Vectors with Sequential Values

			i		
	1	2	3	4	5
			j		
	5	4	3	2	1
			k		
	1	3	5	7	9

The index creation operator (:) has lower precedence than arithmetic operators, as shown by the following statements and by [Figure 2.13](#):

```
/* the index creation operator has low precedence */
n1 = 1;
n2 = 10;
h = n1+2:n2-3;          /* equivalent to 3:7 */
print h;
```

Figure 2.13 Precedence of Index Creation Operator

			h		
	3	4	5	6	7

Programming Tip: The index creation operator (:) has lower precedence than arithmetic operators. Therefore, $a+b*c:d+e*f$ is equivalent to $(a+b*c):(d+e*f)$.

The DO function requires that its arguments are numerical. But there is one situation in which the index creation operator can be used with character values: the creation of variable names with a common prefix and a numerical suffix. For example, if you have 10 variables and want to name them x_1, x_2, \dots, x_{10} , the index creation operator can create these variable names, as shown in the following statements:

```
/* create variable names with sequential values */
varNames = "x1":"x10";
print varNames;
```

Figure 2.14 Variable Names with Sequential Values

varNames									
x1	x2	x3	x4	x5	x6	x7	x8	x9	x10

Programming Tip: Use the index creation operator to create a vector of names with a common prefix and a numerical suffix (for example, "x1": "x10").

2.3.3 Pseudorandom Matrices

Probability theory and statistics describe events that contain aspects of randomness. A *random number algorithm* is an algorithm that generates a sequence of numbers whose statistical properties are such that the sequence is indistinguishable from a truly random sequence. Of course, it is technically impossible to *generate* a random sequence, since, by definition, a random process is not deterministic. Therefore, some people prefer use the term *pseudorandom numbers* to describe a sequence of numbers that are generated by a computer and that behave similarly to random variates from a specified distribution.

In SAS software a pseudorandom sequence of numbers is initialized with a *seed value* that determines the sequence. If you use a different seed value, you get a different sequence of numbers.

There are several SAS/IML functions and subroutines that generate pseudorandom variates. The UNIFORM and NORMAL functions generate random variates from the uniform distribution on [0, 1] and from the standard normal distribution, respectively. For example, the following statements generate two variables that are linearly related to each other.

```
/* create pseudorandom vectors */
seed = j(10, 1, 1);          /* set seed (=1) and dimensions */
x = uniform(seed);          /* 10 x 1 pseudorandom uniform vector */
y = 3*x + 2 + normal(seed); /* linear response plus normal error */
```

The UNIFORM and NORMAL functions are convenient for simple simulations and for quickly generating test data. In the previous statements, the seed value is 1. The size and shape of the **seed** matrix determines the shape of the output from the UNIFORM and NORMAL functions. Since **seed** is a 10×1 vector, the column vector **x** contains 10 pseudorandom numbers in the interval [0, 1]. Similarly, the NORMAL function returns a normally distributed “error vector,” so that the vector **y** is a linear function of **x** plus an error term.

Programming Tip: Use the UNIFORM and NORMAL functions when you need to quickly generate a small random sample for an example or for testing purposes.

The statistical properties of the pseudorandom numbers generated by the UNIFORM and NORMAL functions are not as good as those generated by the newer Mersenne-Twister random number

generator that is implemented in the RANDGEN subroutine. Consequently, you should use the RANDGEN subroutine when you intend to generate millions of pseudorandom numbers.

The RANDGEN subroutine is used extensively in Chapter 13, “Sampling and Simulation.” When you use RANDGEN, you need to allocate a matrix that will hold the random numbers, perhaps by using the J function. For the sake of completeness, the following statements use the RANDGEN subroutine to generate two variables that are linearly related to each other:

```

/* create pseudorandom vectors (better statistical properties) */
call randseed(12345);          /* set seed for RANDGEN          */
x = j(10, 1);                 /* allocate 10 x 1 vector       */
e = x;                        /* allocate 10 x 1 vector       */
call randgen(x, "Uniform");    /* fill x; values from uniform dist */
call randgen(e, "Normal");    /* fill e; values from normal dist */
y = 3*x + 2 + e;              /* linear response plus normal error */

```

Programming Tip: Use the RANDGEN subroutine when the statistical properties of the pseudorandom numbers are important or when you intend to generate millions of variates.

For more information about the numerical properties of SAS routines that generate pseudorandom numbers, see the section “Using Random-Number Functions and CALL Routines” in the *SAS Language Reference: Dictionary*.

2.4 Transposing a Matrix

The vectors created by the DO function and the index creation operators are row vectors. To get a column vector, you can use the T function, which transposes a matrix. The syntax is $T(\text{matrix})$.

The transpose of a row vector is a column vector, and the transpose of a column vector is a row vector. The transpose of a two-dimensional matrix is obtained by flipping the matrix about its main diagonal. Specifically, if a_{ij} is the element in the i th row and j th column of a matrix A , then a_{ji} is the element in the i th row and j th column of the transpose of A . The following statements demonstrate the transpose function:

```

/* transpose a matrix */
s = {1 2 3, 4 5 6, 7 8 9, 10 11 12};          /* 4 x 3 matrix */
transpose = t(s);                            /* 3 x 4 matrix */
print transpose;

```

Figure 2.15 A Transposed Matrix

transpose			
1	4	7	10
2	5	8	11
3	6	9	12

A mathematical notation for the transpose of a matrix A is A' . The SAS/IML language also supports this syntax:

```
sPrime = s` ;           /* alternative notation to transpose a matrix */
```

The transpose operator is described in [Section 2.12](#).

2.5 Changing the Shape of Matrices

Sometimes it is convenient to reshape the data in a matrix. Suppose you have a 1×12 matrix. This same data can fit into many other matrices: for example, a 2×6 matrix, a 3×4 matrix, a 4×3 matrix, and so on. The SHAPE function enables you to specify the number of rows and columns for a new matrix. The values for the new matrix come from an existing matrix, as shown in the following statements:

```
/* reshape a matrix */
x = 1:12;           /* 1 x 12 matrix */
s = shape(x, 4, 3); /* reshape data into 4 x 3 matrix */
```

The matrix s is identical to the one specified manually in the example in the preceding section.

The data in SAS/IML matrices are stored in row-major order and this ordering of the elements is used in reshaping the data. The SHAPE function does not change the order of the data elements in memory; it merely changes how those data are interpreted as a matrix.

Programming Tip: Data in SAS/IML matrices are stored in row-major order.

The previous example specifies both the number of rows and the number of columns to the SHAPE function. You can also specify only the number of rows or only the number of columns. The dimension that is not specified is determined automatically by dividing the number of elements in the matrix by the number of specified rows or columns. To specify only the number of rows, omit the third argument or use 0 for the number of columns. To specify only the number of columns, specify 0 for the number of rows, as shown in the following statements:

```
/* omit dimension; automatically determined */
s1 = shape(x, 4);           /* 4 rows ==> 3 columns */
s2 = shape(x, 0, 3);       /* 3 columns ==> 4 rows */
```

You can also reshape data into dimensions that are not congruent to the original data by specifying a value to use when the original data “runs out,” as shown in the following statements:

```
/* pad data with a specified value (missing) */
s = shape(1:10, 4, 3, .); /* 4 x 3, pad with missing value */
print s;
```

Figure 2.16 Result of the SHAPE Function

s		
1	2	3
4	5	6
7	8	9
10	.	.

2.6 Extracting Data from Matrices

This section describes various techniques for extracting or modifying the subsets of a matrix. The subsets include, rows, columns, submatrices, and diagonal elements.

You can specify elements of a matrix with one index or with two. For example, $\mathbf{x}[3]$ is the third element of the matrix \mathbf{x} , specified in row-major order. Similarly, $\mathbf{y}[1, 2]$ specifies the element in the first row and the second column of the matrix \mathbf{y} . You can use indices on either side of an assignment statement, as the following statements indicate:

```

/* use indices on either side of assignment statements */
x = {1 2 3, 4 5 6};
y = x[2, 3];          /* value of 2nd row, 3rd column          */
x[2, 3] = 7;         /* changes the value of x[2,3]; y is unchanged */
print x, y;

```

Figure 2.17 Result of Assignment Statements

x		
1	2	3
4	5	7

y		
6		

If \mathbf{m} is an $n \times p$ matrix, it is an error to refer to $\mathbf{m}[i, j]$ when $i > n$ or $j > p$. It is also an error to use nonpositive indices.

It is an error to refer to a subscript of an undefined matrix. For example, the following statements show that you cannot assign elements of a vector if the vector has not been created:

```

/* ERROR: subscripting a matrix that has not been created */
z[1] = 0;          /* z is undefined */

```

The error message for this mistake is displayed in the SAS log, which is shown in [Figure 2.18](#).

Figure 2.18 Error Message for Undefined Matrix

```

ERROR: (execution) Matrix has not been set to a value.

operation : [ at line 1520 column 2
operands  : z, *LIT1001, *LIT1002

z          0 row          0 col      (type ?, size 0)

*LIT1001   1 row          1 col      (numeric)
           1

*LIT1002   1 row          1 col      (numeric)
           0

statement : ASSIGN at line 1520 column 1

```

Each line of the error message gives information about the error:

- The error is that you are trying to access an element of a matrix that has not been assigned a value.
- The error occurred in the left-bracket (subset) operation.
- The operation involved three operands: the matrix **z** and two unnamed literals.
 - The matrix **z** has zero rows and zero columns. It is an undefined matrix.
 - The first unnamed literal has the value 1.
 - The second has the value 0.
- The error occurred during the assignment statement.

Programming Tip: It is an error to refer to a subscript of an undefined matrix.

2.6.1 Extracting Rows and Columns

You can specify submatrices of a matrix by specifying vectors of indices. For example, the following statements assign the matrix **w** to the values in the odd rows and even columns of the matrix **z**:

```

/* extract submatrix */
z = {1 2 3 4, 5 6 7 8, 9 10 11 12};
rows = {1 3};
cols = {2 4};
w = z[rows, cols];
print w;

```

Figure 2.19 A Submatrix

	w	
	2	4
	10	12

You can specify all rows of a matrix by omitting the row index. Columns are handled similarly. The following statements use the previous definitions of `rows` and `columns` to show this syntax:

```
/* extract only rows or columns */
oddRows = z[rows, ];           /* specified rows; all columns */
evenCols = z[ , cols];        /* all rows; specified columns */
```

Programming Tip: You can specify all rows of a matrix by omitting the row index. Columns are handled similarly.

When you extract a subset of a matrix by specifying a single set of indices, the resulting matrix is always a column vector. At times, this is a surprising result, as shown in the following statements.

```
/* different ways to specify elements of a matrix */
z = {1 3 5 7 9 11 13};        /* row vector */
w = z[{2 4 6}];              /* column vector with values {3,7,11} */
t = z[ , {2 4 6}];          /* row vector with values {3 7 11} */
print w t;
```

Figure 2.20 Results of Two Indexing Schemes

	w	t		
	3	3	7	11
	7			
	11			

Note that `w` is a column vector even though `z` is a row vector! To get a row vector, you must explicitly omit the row index, as for the vector `t` (or use the `SHAPE` function).

Programming Tip: If you extract a subset of a matrix by specifying a single set of indices, the resulting vector is a column vector.

You can also extract all rows *except* those that you enumerate. The `SETDIF` function (described in [Section 2.11](#)) is useful for this. The following statements extract all rows except those contained in the vector `v`:

```

/* extract all rows except those specified in a vector v */
q = {1 2, . 3, 4 5, 6 7, 8 .};
v = {2 5};                               /* specify rows to exclude */
idx = setdif(1:nrow(q), v);             /* start with 1:5, exclude values in v */
r = q[idx, ];                           /* extract submatrix */
print idx, r;

```

Figure 2.21 Rows That Are Not Excluded

idx	
1	3 4
r	
1	2
4	5
6	7

In the previous statements, the second and fifth rows of the q matrix contain missing values. Suppose you want to exclude these rows from a computation. You can define v to be the vector that contains the rows to exclude and use the SETDIF function to remove those rows from the vector of all row numbers in q . The vector idx contains the rows that are retained, as shown in Figure 2.21.

2.6.2 Matrix Diagonals

An important subset of a matrix is the *diagonal* of the matrix. For an $n \times p$ matrix A , the diagonal is the set of elements A_{ii} for $i = 1, \dots, \min(n, p)$. You can extract the diagonal of a matrix by using the VECDIAG function, as shown in the following example:

```

/* extract matrix diagonal */
m = {1 2 3,
     2 5 6,
     3 6 10};
d = vecdiag(m);
print d;

```

Figure 2.22 The Diagonal of a Matrix

d
1
5
10

The VECDIAG function returns a vector of diagonal elements from a matrix, whereas the DIAG function returns a diagonal matrix created from a specified vector of values, as shown in the following example:

```

/* create a diagonal matrix from a vector */
vals = {5, 2, -1};
s = diag(vals);
print s;

```

Figure 2.23 A Diagonal Matrix

s		
5	0	0
0	2	0
0	0	-1

The I function returns a square matrix that contains ones on the diagonal. This is called the *identity matrix*. You need to specify the dimension of the matrix, as shown in the following example:

```

/* create an identity matrix */
ident = I(3);
print ident;
/* 3 x 3 identity matrix */

```

Figure 2.24 An Identity Matrix

ident		
1	0	0
0	1	0
0	0	1

If you want to extract or modify the diagonal values of a general $n \times p$ matrix, you can directly index the elements of the diagonal by using the following statements:

```

/* index the diagonal elements of a matrix */
n = nrow(m);
p = ncol(m);
diagIdx = do(1, n*p, p+1);
print diagIdx;
d2 = m[diagIdx];
print d2;
/* indices of the diagonal */
/* extract the diagonal */

```

Figure 2.25 Indices and Values of a Matrix Diagonal

diagIdx		
1	5	9

d2		
1		
5		
10		

In the example, the DO function constructs the indices of the diagonal for a general $n \times p$ matrix. These are the correct indices because SAS/IML matrices are stored in row-major order. The indices are stored in the `diagIdx` vector.

You can use the indices in `diagIdx` to extract the diagonal, as shown in the previous example, or to assign to the diagonal elements. For example, a common matrix operation is to subtract a constant from the diagonal of a square matrix, as shown in the following statements:

```

/* compute B = (m - lambda*I) for lambda=1 */
/* First approach: use the formula */
B1 = m - I(n);          /* I(n) gives n x n identity matrix */

/* Second approach: Do not physically create the identity matrix */
B = m;
B[diagIdx] = m[diagIdx] - 1; /* modify the diagonal */
print m B;
    
```

Figure 2.26 Matrix with Modified Diagonal

m			B		
1	2	3	0	2	3
2	5	6	2	4	6
3	6	10	3	6	9

The example creates the matrix $m - I$ in two different ways. The first way is to use the I function to explicitly create an identity matrix and then to subtract that matrix from m to form the matrix B . This works, but is not as efficient as the alternative approach. The alternative approach initializes the matrix B with the values of m and then subtracts 1 from each diagonal element.

2.6.3 Printing a Submatrix or Expression

There are times when it is convenient to print a submatrix of a matrix, or, in general, a temporary result of some matrix expression. You can print submatrices and expressions by enclosing the term that you want to print in parentheses.

For example, suppose that a data matrix contains hundreds of rows, but you want to print only the first few rows. You can index the rows that you want to print, and enclose the expression in parentheses, as shown in the following statements:

```

x = shape(1:1000, 0, 4);          /* 4 columns, hundreds of rows */
print (x[1:3,]);                /* print first three rows */
    
```

Figure 2.27 First Three Rows of a Matrix

1	2	3	4
5	6	7	8
9	10	11	12

The output is shown in Figure 2.27. Notice that the output does not contain any matrix names. This is in contrast to printed output in previous examples in which the name of the matrix is printed in the output. The explanation for this is that when SAS/IML encounters an expression enclosed in parentheses, it creates a temporary matrix to hold the result and the PRINT statement does not output a name for temporary matrices. However, as described in section “Printing a Matrix” on page 19, you can use the LABEL= and COLNAME= options in the PRINT statement to make the printed output easier to understand, as shown in the following statements:

```
varNames = 'Col1':'Col4';
print (x[1:3,]) [label="My Data" colname=varNames];
```

Figure 2.28 Adding a Label and Column Names to Printed Output

	My Data			
	Col1	Col2	Col3	Col4
	1	2	3	4
	5	6	7	8
	9	10	11	12

In the same way, you can print matrix expressions by enclosing the expression in parentheses. For example, the following statements print a linear transformation of a row vector:

```
x = 1:4;
print (3*x + 1); /* print expression */
```

Figure 2.29 Result of Printing an Expression

4	7	10	13
---	---	----	----

Programming Tip: To print a submatrix, you must enclose the submatrix in parentheses:

```
print (x[1:3]);
```

This is a peculiarity of the PRINT statement. You also need to enclose matrix expressions in parentheses:

```
print (3*x + 1);
```

2.7 Comparison Operators

In the SAS/IML language, the equal sign (=) plays two roles. The equal sign can be an assignment operator or a comparison operator. The other comparison operators are less than (<), less than or equals (<=), greater than (>), greater than or equals (>=), and the not equals operator (^=). Usually you will compare a matrix with a scalar value or with another matrix of the same dimensions, although it is also possible to compare a matrix with a vector. The result of the comparison is a

matrix of zeros and ones. The result matrix has the value one for locations where the comparison is true and the value zero for locations where the comparison is false. This is shown in the following statements:

```
/* comparison operators */
x = {1 2 3, 2 1 1};
s1 = (x=2);
print s1;

z = {1 2 3, 3 2 1};
s2 = (x<z);
print s2;
```

Figure 2.30 Results of Comparison Operators

	s1		
	0	1	0
	1	0	0
	s2		
	0	0	0
	1	1	0

The matrix **s1** has the same dimensions as the matrix **x**. The element **s1[i, j]** has the value 1 when **x[i, j]** equals 2. Similarly, the matrix **s2** has the value 1 in locations where **x[i, j]** is less than **z[i, j]**, and has the value 0 otherwise.

The parentheses in the previous example are not necessary, but might help to remind you that the first equal sign is an assignment, whereas the second is a comparison operator. The SAS/IML language does not support the “multiple assignment” syntax found in some other languages. The expression **x=y=0** does *not* assign the value zero to the matrices **x** and **y**; instead, it compares the matrix **y** with zero, and assigns the result to **x**.

Programming Tip: The comparison operators return a matrix of zeros and ones.

Programming Tip: Unlike the DATA step, the SAS/IML language does not support the mnemonic keywords EQ, NE, GT, LT, GE, or LE as a replacement for the symbols =, ^=, >, <, >=, or <=.

The comparison operators treat a missing value as a value that is less than any valid nonmissing value, as shown in the following example:

```
/* missing values compare as less than any nonmissing value */
m = .;
n = 0;
r = (m<n);
print r;
```

Figure 2.31 Result of Comparing a Missing Value

r
1

2.8 Control Statements

This section describes SAS/IML statements that control the flow of a program. This includes the IF-THEN/ELSE statements, the iterative DO statement, the DO/WHILE statement, and the DO/UNTIL statement.

2.8.1 The IF-THEN/ELSE Statement

The comparison operators are most often used in the conditional expression of an IF-THEN/ELSE statement, as shown in the following statements:

```

/* an IF-THEN/ELSE statement */
x = {1 2 3, 2 1 1};
z = {1 2 3, 3 2 1};
if x <= z then
  msg = "all(x <= z)";
else
  msg = "some element of x is greater than the corresponding element of z";
print msg;

```

Figure 2.32 Result of an IF-THEN/ELSE Statement

msg
all(x <= z)

The syntax is identical to that used in the DATA step, except that the conditional expression in the SAS/IML language can be a matrix. Recall that the expression $\mathbf{x} \leq \mathbf{z}$ resolves to a matrix of zeros and ones. If every element of $\mathbf{x} \leq \mathbf{z}$ is nonzero (that is, the expression is true for all elements), then the statement following the THEN keyword is executed. Otherwise, the statement following the ELSE keyword is executed. The ELSE statement is optional.

Programming Tip: A matrix expression in an IF-THEN statement is true provided that all elements of the matrix expression are nonzero.

You can use the DO and END statements to group multiple statements that should be executed, as shown in the following statements:

```

/* a DO-END block of statements */
if x <= z then do;
  msg = "all(x <= z)";
  /* more statements ... */
end;
else do;
  msg = "some element of x is greater than the corresponding element of z";
  /* more statements ... */
end;

```

You can use the ALL function to emphasize the condition you are testing:

```

/* the ALL statement */
if all(x <= z) then do;
  msg = "all(x <= z)";
  /* more statements ... */
end;

```

The ALL function returns the value 1 if every element of its argument is nonzero; otherwise, it returns the value 0. If you want to test whether *any* element in a matrix is nonzero, you can use the ANY function, as shown in the following statements:

```

/* the ANY statement */
if any(x < z) then
  msg = "some element of x is less than the corresponding element of z";
print msg;

```

Figure 2.33 Result of the ANY Function

<pre> msg some element of x is less than the corresponding element of z </pre>
--

The ANY function returns 1 if any element of its argument is nonzero; otherwise, it returns 0.

2.8.2 The Iterative DO Statement

The iterative DO statement enables you to repeat a group of statements several times. For example, you can loop over elements in a matrix or loop over variables in a data set. The syntax is identical to that used in the DATA step, as shown in the following statements:

```

/* the iterative DO statement */
x = 1:5;
sum = 0;
do i = 1 to ncol(x);
  sum = sum + x[i];
end;
print sum;
/* inefficient way to sum elements */

```

40 Chapter 2: Getting Started with the SAS/IML Matrix Programming Language

The statements loop over the columns of \mathbf{x} and add up each entry. The partial sum at each step in the iteration is accumulated in the variable `sum`. When the loop ends, `sum` contains the sum of the elements in \mathbf{x} , as shown in [Figure 2.34](#).

Figure 2.34 Result of the Iterative DO Statement

<code>sum</code>
15

In general, you should try to avoid looping over every element in a vector. The SAS/IML language has many functions and statements that enable you to avoid explicit loops. For example, you can rewrite the previous example by using the `SUM` function to eliminate the loop:

```
x = 1:5;
sum = sum(x);          /* efficient; eliminate DO loop */
```

The `SUM` function and other functions that act on matrices are described in [Appendix C](#). The section “[Writing Efficient SAS/IML Programs](#)” on page 79 discusses other ways to avoid loops.

The iterative `DO` statement supports an optional `WHILE` or `UNTIL` clause. You can use the `WHILE` or `UNTIL` clauses when you want to exit the `DO` loop after a certain criterion is satisfied, as shown in the following statements:

```
/* an UNTIL clause */
x = 1:5;
sum = 0;
do i = 1 to ncol(x) until(sum > 8);
    sum = sum + x[i];
end;
print sum;
```

Figure 2.35 Result of the UNTIL Clause

<code>sum</code>
10

The `WHILE` clause is evaluated at the top of the loop, whereas the `UNTIL` clause is evaluated at the bottom of the loop. Consequently, you can use the `WHILE` clause when you want to block the execution of the body of the loop. For example, the following statements find the partial sum of a vector until a missing value is encountered:

```
/* a WHILE clause */
x = {1 2 . 4 5};
sum = 0;
do i = 1 to ncol(x) while(x[i]^=.);
    sum = sum + x[i];
end;
print sum;
```

Figure 2.36 Result of the WHILE Clause

sum 3

The statements work correctly because the WHILE clause forces the loop to exit as soon as `i` is incremented to 3. If you try to use an UNTIL clause (such as `until(x[i]=.)`), the body of the loop executes and results in the missing value being added to the `sum` variable.

You can also use the WHILE and UNTIL clauses in conjunction with a noniterative DO statement. In this case, you need to control the iteration yourself, and you need to avoid conditions that might lead to infinite loops. For example, the following statements use a DO-UNTIL statement to approximate the first local maximum of the sine function for $x > 0$:

```
/* a DO-UNTIL statement */
x = 0;
dx = 0.01;
do until(sin(x) > sin(x+dx));
  x = x + dx;
end;
print x;
```

Figure 2.37 Result of the DO-UNTIL Statement

x 1.57

The true value of the local maximum is $\pi/2 \approx 1.57$. The loop adds a small increment to `x` at each iteration. The condition in the UNTIL clause tests whether the sine function is increasing at the current value of `x`. The loop continues until `x` attains a value at which the sine function is no longer increasing.

2.9 Concatenation Operators

A previous section described ways to extract a subset of a matrix. This section describes how you can concatenate matrices together to form a bigger matrix from smaller ones.

The SAS/IML language provides two concatenation operators. The horizontal concatenation operator (`||`) appends new columns to a matrix, or combines two matrices that have the same number of rows. A typical usage is to create a data matrix consisting of columns from several vectors, as shown in the following statements:

```

/* horizontal concatenation */
a = {1,2,3,4,5};                /* 5 x 1 column vectors */
b = {3,5,4,1,3};
c = {0,1,0,0,1};
x = a || b || c;                /* 5 x 3 matrix */
print x;

```

Figure 2.38 Result of Horizontal Concatenation

x		
1	3	0
2	5	1
3	4	0
4	1	0
5	3	1

The vertical concatenation operator (//) appends new rows or combines matrices that have the same number of columns. A typical usage is that you want to combine several subsets of data, where each subset is contained in its own matrix. For example, the following statements combine a subset of the data (say, for males) with another subset (say, for females):

```

/* vertical concatenation */
males = {1 3 0, 2 5 1, 3 4 0};    /* 3 x 3 matrix */
females = {4 1 0, 5 3 1};        /* 2 x 3 matrix */
x = males // females;            /* 5 x 3 matrix */

```

Because the concatenation operators allocate a new matrix and copy data into it, you should try to avoid using concatenation operators inside of a loop. For example, suppose you want to construct a vector of even integers {2 4 6 8 10}. The following statements show an inefficient way to construct this vector:

```

/* construct vector of even integers */
free x;                          /* make sure x is empty */
do i = 1 to 5;
  x = x || 2*i;                   /* inefficient! 5 allocations */
end;

```

The program first uses the FREE statement to ensure that **x** is an empty matrix. Inside the DO loop, the program allocates a new matrix for each iteration of the loop. The first iteration concatenates the empty matrix with the scalar that contains 2. At the end of the first iteration, the matrix **x** is 1×1 . At the end of the second iteration, the matrix **x** is 1×2 , and so on.

It is not merely the allocation of memory that is inefficient. The algorithm is also inefficient because of the way it copies elements multiple times. During the five iterations, the value 2 is copied into a new matrix five times. The value 4 is copied four times, and so on. In total, the five values are copied 15 times before the loop ends. If the same algorithm is used to create a vector of length n , the values are copied a total of $n(n-1)/2$ times!

If the purpose of the loop is to compute values for a matrix with a given number of rows and columns, it is usually more efficient to allocate space for the matrix prior to the loop. You can assign the values to the preallocated matrix inside the loop, as shown in the following statements:

```

/* Improved algorithm: allocate the matrix to hold the
   even integers. Assign values into the vector. */
x = j(1, 5, .); /* allocate; fill with missing */
do i = 1 to ncol(x);
  x[i] = 2*i; /* assign value */
end;

```

This algorithm requires a single allocation of a vector, and no values are copied unnecessarily. Of course, for this simple example, you can avoid the DO loop altogether:

```
x = 2*(1:5);
```

Programming Tip: Whenever possible, avoid using concatenation operators to construct a matrix row-by-row (or column-by-column) inside a loop.

You cannot always avoid concatenation within a DO loop because there are situations in which you do not know in advance how many iterations are required. Nevertheless, even in those situations you can often obtain an upper bound on the number of iterations and preallocate the results matrix. An example is a root-finding algorithm that iterates until convergence. You do not know in advance how many iterations are required, but you can decide to limit the algorithm to no more than 50 iterations before stopping the algorithm. In this case, you can preallocate space for up to 50 results.

The concatenation operators enable you to append one matrix to the end of another. If you need to insert rows or columns into the middle of a matrix, you can use the INSERT function. If you need to remove rows or columns from the middle of a matrix, you can use the REMOVE function. For example, the following statements remove the first, third, and sixth elements of a vector, which has the effect of removing all missing values from the vector:

```

y = {., 1, ., 2, 3, .};
v = remove(y, {1 3 6});

```

See the *SAS/IML User's Guide* for details of the INSERT and REMOVE statements.

2.10 Logical Operators

The IF-THEN/ELSE, DO-UNTIL, and DO-WHILE statements all test whether a condition is true. When the condition involves a matrix, the condition is considered “true” provided that it is true for each element of the matrix.

You can also test whether multiple conditions are true and combine two or more conditions into a single logical expression. The three logical operators are the AND operator (&), the OR operator (|), and the NOT operator (^). The operators perform logical operations on the elements of matrices, as shown in the following statements:

```

/* logical operators */
r = {0 0 1 1};
s = {0 1 0 1};
and = (r & s);
or  = (r | s);
not = ^r;
print and, or, not;

```

Figure 2.39 Results of Logical Operators

		and		
	0	0	0	1
		or		
	0	1	1	1
		not		
	1	1	0	0

As the example indicates, the matrix **and** contains a nonzero value only where the corresponding elements of **r** and **s** are nonzero. Similarly, the matrix **or** contains a nonzero value if either of the corresponding elements of **r** and **s** are both nonzero. Lastly, the matrix **not** is nonzero only where the matrix **r** is zero.

Programming Tip: Unlike the DATA step, the SAS/IML language does not support the mnemonic keywords AND, OR, or NOT as a replacement for the symbols **&**, **|**, and **^**.

You can use the logical operators to combine multiple criteria into a single criterion. For example, the following statements use the logical AND operator to test whether the values of a vector are all in the domain of a function:

```

/* logical combination of criteria */
x = do(-1, 1, 0.5);
if (x >= -1) & (x <= 1) then
  y = sqrt(1-x##2); /* the ## operator squares each element of x */
else
  y = "Unable to compute the function.";
print y;

```

Figure 2.40 Result of the Logical AND (&) Operator

		y		
	0	0.8660254	1	0.8660254
	0			

There are two conditions in the IF-THEN statement. The first tests whether all of the elements of \mathbf{x} are greater than or equal to -1 . The second tests whether all of the elements of \mathbf{x} are less than or equal to 1 . The logical AND operator ensures that the SQRT function is called only if both of the conditions are satisfied. The parentheses are not necessary, but are included for clarity. For this example, both conditions are satisfied, so the SQRT function is called. The argument to the SQRT function uses the elementwise power operator (##) to square each element of \mathbf{x} . Consequently, the i th element of \mathbf{y} is equal to $\sqrt{1 - x_i^2}$, where x_i is the i th element of \mathbf{x} .

You can often write a logical condition in multiple ways. For example, the following IF-THEN statements are logically equivalent to the previous IF-THEN statement:

```
/* different ways to compute the same logical condition */
if (x< -1) | (x>1) then
  y = "Unable to compute the function.";
else
  y = sqrt(1-x##2);

if ^(x< -1) & ^(x>1) then
  y = sqrt(1-x##2);
else
  y = "Unable to compute the function.";
```

Some programming languages support a feature known as *minimal evaluation* or *short-circuit evaluation*. In these languages, the second argument of a logical AND or OR expression is evaluated only if the first evaluation does not already determine the truth of the expression. For example, in the expression $\mathbf{r} \ \& \ \mathbf{s}$, if \mathbf{r} contains a zero, then the entire expression is false, regardless of the value of \mathbf{s} . Similarly, in the expression $\mathbf{r} \ | \ \mathbf{s}$, if \mathbf{r} contains all nonzero values, then the entire expression is true. The SAS/IML language does *not* support short-circuit evaluation.

Programming Tip: The SAS/IML language does not support short-circuit evaluation.

Why does this matter? Some programmers write logical expressions that assume short-circuiting will occur. For example, in some languages you can write the following logical expression:

```
/* incorrect: second condition evaluated even if first is false */
if (x>0) & (log(x)>1) then do;          /* ERROR if x<=0 */
  /* more statements */
end;
```

This is a valid expression in a language that short-circuits operations because the expression that contains $\log(\mathbf{x})$ is executed only if \mathbf{x} is positive, as ensured by the first condition. However, in the SAS/IML language, both expressions are evaluated, and then the AND operator combines the results. This means that the LOG function is always called and that the program will stop with an error if some element of \mathbf{x} is nonpositive.

Instead, you need to write the previous statements by using nested IF-THEN statements, as shown in the following example:

```

/* correct: second condition not evaluated if first is false */
if x>0 then
  if log(x)>1 then do;
    /* more statements */
  end;

```

2.11 Operations on Sets

There are three SAS/IML functions that perform operations on sets. The UNION function returns the union of the values in one or more matrices. The XSECT function returns the intersection of the values in two or more matrices. (The intersection is the set of values common to all matrices.) The SETDIF function accepts two arguments, say **A** and **B**, and returns the values in matrix **A** that are not found in matrix **B**.

These set functions are shown in the following statements:

```

/* union, intersection, and difference of sets */
A = 1:4;
B = do(2.5, 4.5, 0.5);
u = union(A, B);           /* union                */
inter = xsect(A, B);      /* intersection        */
dif = setdif(A, B);       /* difference between sets */
print u, inter, dif;

```

As shown in Figure 2.41, each of these functions returns a row vector. The vector **u** contains the union of the values in **A** and **B**, the vector **inter** contains the intersection of the values, and **dif** contains the difference between the sets determined by **A** and **B**.

Figure 2.41 Results of Operations on Sets

u						
1	2	2.5	3	3.5	4	4.5
inter						
		3	4			
dif						
	1		2			

In this example, the intersection and the difference between sets were both nonempty, but this is not always the case. The XSECT function returns an empty matrix if there are no elements in the intersection of the matrices. Similarly, the SETDIF function returns an empty matrix if the elements of **A** are a proper subset of the elements of **B**. Consequently, you need to check that the matrix returned by XSECT or SETDIF is nonempty before you print it or use it in further computations.

The following statements show an example in which the intersection is empty:

```

/* check whether intersection and set difference are empty */
A = 1:4;
B = do(5, 8, 0.5);
inter = xssect(A, B);
if ncol(inter)>0 then
  print inter;

dif = setdif(A, B);
if ncol(dif)>0 then
  print dif;

```

Figure 2.42 A Nonempty Set Difference

dif				
	1	2	3	4

The matrix `inter` is created by the XSECT function, but it has zero rows and zero columns. You can therefore use the NCOL function to detect the empty intersection. (You could also check to see if `type(inter)` has the value 'U'.) The program avoids run-time errors by only executing the PRINT statement for nonempty matrices.

Programming Tip: The intersection of sets can be empty, so always check the matrix returned by XSECT and handle the possibility that the matrix is empty. Do the same for set differences found by using the SETDIF function.

2.12 Matrix Operators

The fundamental unit in the SAS/IML language is the matrix. This section describes how to perform arithmetic operations that involve scalars, vectors, and matrices.

2.12.1 Elementwise Operators

The elementwise operators in the SAS/IML language consist of one unary operator and several binary operators. The unary operator is the negation operator (`-`), which is equivalent to multiplication by `-1`. The elementwise binary operators are the addition operator (`+`), the subtraction operator (`-`), the elementwise multiplication operator (`#`), the division operator (`/`), and the power or exponentiation operator (`##`).

In most cases, the SAS/IML language enables you to write concise, high-level expressions that combine scalars, vectors, and matrices. The SAS/IML software determines if it can make sense

of an arithmetic expression in which the matrices have different dimensions. For example, the following statements show how you can write an arithmetic expression that contains a matrix and also a scalar or a vector:

```

/* elementwise matrix operations with scalar or vector */
x = {7 7, 8 9, 7 9, 5 7, 8 8};
grandmean = 7.5;          /* mean of all elements      */
y = x - grandmean;        /* subtract 7.5 from each element */
mean = {7 8};             /* mean of each column      */
xc = x - mean;            /* subtract (7 8) from each row */
print y xc;

```

The assignment statement for **y** contains an expression that subtracts a scalar value from a 5×2 matrix, **x**. The SAS/IML language assumes that this is shorthand notation for subtracting the scalar 7.5 from every element of **x**. On the assignment statement for **xc**, the program subtracts a 1×2 vector from **x**. Since SAS 9.2, the SAS/IML language has assumed that this is shorthand for subtracting the vector {7 8} from every row of **x**. The resulting matrices are shown in Figure 2.43.

Figure 2.43 Results of Elementwise Operations

y		xc	
-0.5	-0.5	0	-1
0.5	1.5	1	1
-0.5	1.5	0	1
-2.5	-0.5	-2	-1
0.5	0.5	1	0

In general, if **m** is an $n \times p$ matrix, you can perform elementwise operations with a second matrix **v** provided that **v** has one of the following dimensions: 1×1 , $n \times 1$, $1 \times p$, or $n \times p$. The result of the elementwise operation is shown in Table 2.1, which describes the behavior of elementwise operations for the **+**, **-**, **#**, **/**, and **##** operators.

Table 2.1 Behavior of Elementwise Operators (SAS/IML 9.2)

Size of v	Result of m op v
1×1	v applied to each element of m
$n \times 1$	v [i] applied to row m [i ,]
$1 \times p$	v [j] applied to column m [, j]
$n \times p$	v [i , j] applied to element m [i , j]

Programming Tip: If you follow the rules presented in Table 2.1, you can perform elementwise operations on rows or columns of a matrix by specifying an appropriate column vector or row vector.

The following statements (which continue from the previous example) give concrete examples for the remaining rules presented in Table 2.1:

```

/* elementwise matrix operations with vector or matrix */
/* r is row vector */
r = {1.225 1};                               /* std dev of each col      */
std_x = xc / r;                               /* divide each column (normalize) */
/* c is column vector */
c = x[,1];                                   /* first column of x        */
y = x - c;                                   /* subtract from each column */
/* m is matrix */
m = {6.5 7.5, 7.9 9.1, 7.5 8.5, 5.6 6.4, 7.5 8.5};
deviations = x - m;                          /* difference between matrices */
print std_x y deviations;

```

The program shows that you can multiply or divide every row (or column) of a matrix by a scalar or a vector in a natural way. For example, the statement that assigns the matrix `std_x` divides each column of `x` by its standard deviation, thus normalizing the scale of each column. The results of this and the other computations are shown in Figure 2.44.

Figure 2.44 More Results of Elementwise Operations

std_x		y		deviations	
0	-1	0	0	0.5	-0.5
0.8163265	1	0	1	0.1	-0.1
0	1	0	2	-0.5	0.5
-1.632653	-1	0	2	-0.6	0.6
0.8163265	0	0	0	0.5	-0.5

2.12.2 Matrix Computations

The SAS/IML language supports operators for specifying high-level matrix operations such as matrix multiplication. The matrix multiplication operator is `*`. If \mathbf{A} is an $n \times p$ matrix and \mathbf{B} is a $p \times m$ matrix, the product $\mathbf{A} * \mathbf{B}$ is an $n \times m$ matrix. The ij th entry of the product is the sum $\sum_{k=1}^p A_{ik} B_{kj}$. If \mathbf{B} is a column vector, then set $j = 1$ in the previous formula.

For example, the following statements multiply a matrix and a vector:

```

/* matrix multiplication */
A = {7 7, 8 9, 7 9, 5 7, 8 8};             /* 5 x 2 matrix          */
v = {1, -1};                               /* 2 x 1 vector          */
y = A*v;                                    /* result is 5 x 1 vector */
print y;

```

Figure 2.45 Result of Matrix Multiplication

y
0
-1
-2
-2
0

As a convenience, you can also use the `*` operator to multiply a matrix by a scalar quantity, such as `3*A`. This performs elementwise multiplication and is equivalent to `3#A`.

Another matrix operator is the transpose operator (```). This operator is typed by using the GRAVE ACCENT key. (The GRAVE ACCENT key is located in the upper left corner on US and UK QWERTY keyboards.) The operator transposes the matrix that follows it. This notation mimics the notation often seen in statistical textbooks that discuss matrix equations.

For example, given an $n \times p$ data matrix X and a vector of n observed responses, y , a goal of ordinary least squares (OLS) regression is to find the linear combination of the columns of X that best approximates y . The so-called *normal equations* provide a computational solution to this problem. The normal equations are written as $(X'X)b = X'y$. Solving this equation means solving for the unknown $p \times 1$ parameter vector b .

The following statements use the matrix transpose operator to compute the $X'X$ matrix and the $X'y$ vector for example data:

```
/* Set up the normal equations (X`X)b = X`y */
x = (1:8)`; /* X data: 8 x 1 vector */
y = {5 9 10 15 16 20 22 27}`; /* corresponding Y data */

/* Step 1: Compute X`X and X`y */
x = j(nrow(x), 1, 1) || x; /* add intercept column */
xpx = x` * x; /* cross-products */
xpy = x` * y;
```

The vector \mathbf{x} is initially defined as the column vector that results from transposing the row vector `1:8`. The vector \mathbf{y} is also defined by transposing a row vector. (You can equivalently define $\mathbf{y}=\{5, 9, \dots, 27\}$ by typing commas between each pair of values, but it is easier to type one grave accent than to type seven commas.) The statements next append a vector of ones to the \mathbf{x} data; this column is used to compute an intercept term in an OLS regression model. The matrix \mathbf{xpx} and the vector \mathbf{xpy} are computed by using a natural notation that mimics the mathematics of the problem.

Because the transpose operator can be difficult to see, some programmers prefer to use the `T` function to indicate transposition. However, the SAS/IML language optimizes certain computations such as the computation of \mathbf{xpx} . For that reason, using the transpose operator can sometimes result in better performance than using the `T` function.

Programming Tip: The SAS/IML language optimizes certain computations that involve the transpose operator (```). Consequently, you should use the transpose operator instead of the `T` function when you are concerned about the speed of matrix computations.

The SAS/IML language also provides functions that enable you to numerically solve the normal equations for the unknown parameter b . Textbooks often write the solution of the normal equations as $b = (X'X)^{-1}X'y$, where A^{-1} indicates the inverse of the matrix A . The SAS/IML language provides the `INV` function for computing the inverse of a function, so many SAS/IML programmers use the following statements to numerically solve the normal equations:

```

/* solve linear system */
/* Solution 1: compute inverse with INV (inefficient) */
xpxi = inv(xpx);           /* form inverse crossproducts */
b = xpxi * xpy;           /* solve for parameter estimates */

```

A better technique is to use the SOLVE function to numerically solve the normal equations:

```

/* Solution 2: compute solution with SOLVE. More efficient */
b = solve(xpx, xpy);      /* solve for parameter estimates */

```

Explicitly solving for the inverse matrix is inefficient if you are only interested in solving a linear equation. Not only does the INV function need to allocate memory, but the INV function (which solves for a *general* solution) is often less accurate than the SOLVE function (which solves for a particular solution). There might occasionally be situations in which you need to compute the inverse matrix, but you should avoid it when you have the option.

Programming Tip: If you need to solve the linear equation $Ab = z$, use the SOLVE function (`b=solve(A, z)`) unless there is a compelling reason to explicitly compute the inverse matrix A^{-1} .

2.13 Managing the SAS/IML Workspace

The SAS/IML language enables you to view and manage the matrices that are kept in memory. At any point in your program, you can see the matrices that are defined, free the memory that is associated with matrices, and store matrices to a SAS catalog.

You can use the SHOW statement to display the names, dimensions, and type of matrices that are defined and are in scope. You can specify a list of matrices or you can use the SHOW NAMES statement to display information about all matrices. Both of these techniques are shown in the following statements:

```

/* display the names, dimensions, and type of matrices */
proc iml;
x = 1:3;           /* define some matrices */
y = j(1e5, 100);  /* large matrix: 10 million elements */
animals = {"Cat" "Dog" "Mouse",
           "Cow" "Pig" "Horse"};
show names;      /* show information about all matrices */
show y animals;  /* show information about specified matrices */

```

Figure 2.46 Information about Defined Matrices

SYMBOL	ROWS	COLS	TYPE	SIZE
animals	2	3	char	5
x	1	3	num	8
y	100000	100	num	8
Number of symbols = 3 (includes those without values)				
		y 100000 rows	100 cols	(numeric)
	animals	2 rows	3 cols	(character, size 5)

If you no longer need certain matrices, you can use the FREE statement to free the associated memory. For example, the following statements free the memory that is associated with a large matrix:

```
/* delete one or more matrices by listing their names */
free y;
show names;
```

Figure 2.47 Deleting a Matrix

SYMBOL	ROWS	COLS	TYPE	SIZE
animals	2	3	char	5
x	1	3	num	8
Number of symbols = 3 (includes those without values)				

You can free all matrices by placing a slash (/) on the FREE statement, as shown in the following example. Figure 2.48 shows that three matrix names (symbols) have been used in this session, but none of the matrices have values after the FREE statement is executed.

```
/* delete all matrices */
free /;
show names;
```

Figure 2.48 Deleting All Matrices

SYMBOL	ROWS	COLS	TYPE	SIZE
Number of symbols = 3 (includes those without values)				

If you need to shut down your computer or exit the SAS System, but you want to save the state of your program so that you can resume work at a later time, you can save some or all of the matrices that are currently defined. This is especially important if some matrices are the result of a lengthy computation. You can use the STORE statement to save matrices. By default, the STORE statement saves the matrices to a SAS catalog in the Work library. However, the Work library vanishes at the

end of a SAS session, so it is best to use the RESET STORAGE statement to store the matrices to a permanent storage location, as shown in the following example. Figure 2.49 shows that the `animals` and `x` matrices are saved in the MyStorage catalog.

```

/* store all matrices to a permanent location */
x = 1:3;
animals = {"Cat" "Dog" "Mouse", "Cow" "Pig" "Horse"};

libname MyLib "C:\My Data";           /* set directory for storage */
reset storage=MyLib.MyStorage;       /* set the storage catalog */
store _all_;                          /* store all matrices */
show storage;                          /* display catalog contents */

```

Figure 2.49 Stored Matrices

```

Contents of storage library = MYLIB.MYSTORAGE

Matrices:
ANIMALS                X

Modules:

```

When you are ready to resume your work, you can start a new SAS/IML session and use the LOAD statement to restore the matrices, as shown in the following statements:

```

/* load matrices from a storage catalog */
proc iml;
reset storage=MyLib.MyStorage;
load _all_;
show names;

```

Figure 2.50 Loading Matrices from Storage

```

SYMBOL    ROWS    COLS TYPE    SIZE
-----
ANIMALS    2      3 char     5
X          1      3 num     8
Number of symbols = 3 (includes those without values)

```

When you no longer need the matrices that are stored in the SAS catalog, you can remove them by using the REMOVE statement, as shown in the following statements:

```

/* remove all contents from a storage catalog */
remove _all_;
show storage;                          /* show the contents of the catalog */

```

Figure 2.51 Removing Matrices from Storage

```
Contents of storage library = MYLIB.MYSTORAGE  
  
Matrices:  
  
Modules:
```

As an alternative to using the `_ALL_` keyword, you can specify a list of names on the `STORE` and `LOAD` statements to indicate which matrices you want to store and load. Similarly, you can remove a specified list of matrices from storage.