Paper 144-2013

Getting Started with the SAS/IML® Language

Rick Wicklin, SAS Institute Inc.

ABSTRACT

Do you need a statistic that is not computed by any SAS® procedure? Reach for the SAS/IML® language! Many statistics are naturally expressed in terms of matrices and vectors. For these, you need a matrix language.

This paper introduces the SAS/IML language to SAS programmers who are familiar with elementary linear algebra. The focus is on statements that create and manipulate matrices, read and write data sets, and control the program flow. The paper demonstrates how to write user-defined functions, interact with other SAS procedures, and recognize efficient programming techniques.

INTRODUCTION

The SAS/IML language is a high-level matrix programming language that enables you to use natural mathematical syntax to write custom algorithms and to compute statistics that are not built into any SAS procedure. The initials IML stand for “interactive matrix language.”

This paper is based on Chapters 2–4 of Wicklin (2010b).

COMPARISON WITH THE DATA STEP

The SAS/IML language shares many similarities with the SAS DATA step. Neither language is case sensitive, variable names can contain up to 32 characters, and statements must end with a semicolon. Although some DATA step syntax is not supported by SAS/IML software (such as the OR, AND, EQ, LT, and GT operators), the two languages have similar syntax for many statements. For example, you can use the same symbols to test for equality (=) and inequality (^=), and to compare quantities (<=). The SAS/IML language enables you to call the same mathematical functions that are provided in the DATA step, such as LOG, SQRT, ABS, SIN, COS, CEIL, and FLOOR, but the SAS/IML versions act on vectors and matrices.

Conceptually, there are three main differences between a DATA step and a SAS/IML program:

- A DATA step implicitly loops over observations in an input data set; a typical SAS/IML program does not.
- The fundamental unit in the DATA step is an observation; the fundamental unit in the SAS/IML language is a matrix.
- The DATA step reads and writes data sets; the SAS/IML language keeps data and results in RAM.

SAS/IML software is most often used for statistical computing rather than for data manipulation. The SAS/IML language enables you to write statistical expressions more concisely than you can in the DATA step.

The SAS/IML language offers excellent performance for computations that fit in memory and that can be vectorized. A computation is *vectorized* if it consists of a few executable statements, each of which operates on a fairly large quantity of data, usually a matrix or a vector. A program in a matrix language is more efficient when it is vectorized because most of the computations are performed in a low-level language such as C. In contrast, a program that is not vectorized requires many calls that transfer small amounts of data between the high-level program interpreter and the low-level computational code. To vectorize a program, take advantage of built-in functions and linear algebra operations. Avoid loops that access individual elements of matrices.

HOW TO RUN SAS/IML PROGRAMS

SAS/IML software has two components: the IML procedure and the SAS/IML® Studio application. PROC IML is a computational procedure that implements the SAS/IML language for matrix programming. You can run PROC IML as part of a larger SAS program that includes DATA steps, macros, and procedure calls.

SAS/IML Studio provides an environment for developing SAS/IML programs. SAS/IML Studio runs on a Windows PC and can connect to one or more SAS Workspace Servers. It provides an editor that color-codes keywords, has debugging features, and enables you to use multiple workspaces, each with its own Work library. (SAS® Enterprise Guide® is not an ideal environment for developing programs that use an interactive procedure such as PROC IML. Every time you submit a PROC IML statement from SAS Enterprise Guide, it appends a QUIT statement to your program. The QUIT statement terminates the procedure and deletes all previously computed matrices.)

GETTING STARTED: A FIRST SAS/IML PROGRAM

The formula $F = (9/5)C + 32$ converts a temperature from the Celsius scale to Fahrenheit (F). The SAS/IML language enables you to use vector quantities instead of scalar quantities to perform computations. The following SAS/IML program converts a vector of temperatures from Celsius to Fahrenheit:

```
proc iml;          /* In SAS/IML Studio, PROC IML stmt is optional   */

/* convert temperatures from Celsius to Fahrenheit scale */
Celsius = {-40, 0, 20, 37, 100};          /* vector of temperatures  */
Fahrenheit = 9/5 * Celsius + 32;          /* convert to Fahrenheit   */
print Celsius Fahrenheit;                 /* send to ODS destination */
```

The vector **Celsius** contains five elements. Figure 1 shows the result of computations that affect every element of the vector. The **Fahrenheit** vector is a linear transformation of the **Celsius** vector.

Notice that the SAS/IML syntax is identical to the mathematical expression and that the transformation does not require a loop over the elements of the vectors. Notice also that there is no RUN or QUIT statement; each statement executes immediately when it is submitted.

Figure 1 Result of Vector Computations

Celsius	Fahrenheit
-40	-40
0	32
20	68
37	98.6
100	212

Most SAS/IML statements do not create output. Consequently, the PRINT statement is used frequently in this paper in order to display results. A comma in the PRINT statement displays the subsequent matrix in a new row. If you omit the comma, the matrices are displayed side by side.

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

COLNAME=*matrix*
specifies a character matrix to be used for column headings.

ROWNAME=*matrix*
specifies a character matrix to be used for row headings.

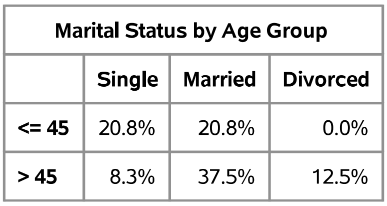
LABEL=*label*
specifies a label for the matrix.

FORMAT=*format*
specifies a valid SAS format or user-defined format to use in displaying matrix values.

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

```
proc iml;
/* print marital status of 24 people */
ageGroup = {"<= 45", " > 45"};              /* row headings        */
status = {"Single" "Married" "Divorced"};   /* column headings     */
counts = {   5         5         0,         /* data to print       */
             2         9         3     };
p = counts / sum(counts);                   /* compute proportions */
print p[colname=status
        rowname=ageGroup
        label="Marital Status by Age Group"
        format=PERCENT7.1];
```

Figure 2 Matrices Displayed by PRINT Options



Marital Status by Age Group			
	Single	**Married**	**Divorced**
<= 45	20.8%	20.8%	0.0%
> 45	8.3%	37.5%	12.5%

CREATING MATRICES AND VECTORS

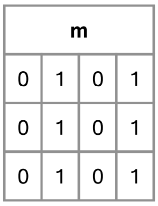
There are several ways to create matrices in the SAS/IML language. For small matrices, you can manually type the elements. Use spaces to separate columns; use commas to separate rows. The following statements define a numerical scalar matrix (**s**), a 2×3 numerical matrix (**x**), and a 1×2 character row vector (**y**):

```
proc iml;
/* manually create matrices of various types */
s = 1;                                   /* scalar                 */
x = {1 2 3, 4 5 6};                      /* 2 x 3 numeric matrix   */
y = {"male" "female"};                   /* 1 x 2 character matrix */
```

You can use the J and REPEAT functions to create vectors of constant values. The J function creates a matrix of identical values. The syntax `J(r, c, v)` returns an $r \times c$ matrix in which each element has the value **v**. The REPEAT function creates a new matrix of repeated values. The syntax **REPEAT(x, r, c)** replicates the values of the **x** matrix r times in the vertical direction and c times in the horizontal direction. The following statements demonstrate these functions:

```
z = j(2, 3, 0);              /*  2 x 3 row vector of zeros         */
m = repeat({0 1}, 3, 2);     /* repeat vector: down 3x and across 2x */
print m;
```

Figure 3 Constant Matrices

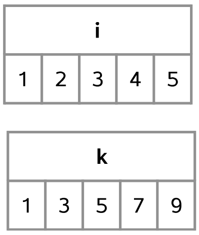


m			
0	1	0	1
0	1	0	1
0	1	0	1

Another useful construction is a vector of sequential values. The DO function enables you to create an arithmetic sequence from a to b in steps of s. The syntax is **DO(a, b, s)**. For sequences that have a unit step size, you can use the colon index operator (:), as follows:

```
i = 1:5;                    /* increment of 1                */
k = do(1, 10, 2);           /* odd numbers 1, 3, ..., 9      */
print i, k;
```

Figure 4 Sequential Vectors



i				
1	2	3	4	5

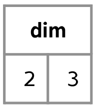
k				
1	3	5	7	9

MATRIX DIMENSIONS

A matrix has two dimensions: the number of its rows and the number of its columns. The NROW function returns the number of rows, and the NCOL function returns the number of columns. To get both of these numbers at once, use the DIMENSION call, which returns a row vector, as follows:

```
x = {1 2 3, 4 5 6};
n = nrow(x);
p = ncol(x);
dim = dimension(x);
print dim;
```

Figure 5 Dimensions of a Matrix



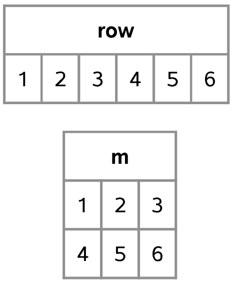
dim	
2	3

You can change the dimensions of a matrix without changing the data. This is called *reshaping* the matrix. The SHAPE function reshapes the data into another matrix that contains the same number of elements. The syntax **SHAPE(x, r, c)** reshapes **x** into an $r \times c$ matrix.

For example, matrix **x** in the previous example has six elements, so the data fit into a 1×6 row vector, a 2×3 matrix, a 3×2 matrix, or a 6×1 column vector. Matrices in the SAS/IML language are stored in row-major order. The following statements reshape the matrix **x** twice. The results are shown in Figure 6.

```
/* to save space, the 3 x 2 and 6 x 1 matrices are not computed */
row = shape(x, 1);                                  /* 1 x 6 vector */
m = shape(x, 2, 3);                                 /* 2 x 3 matrix */
print row, m;
```

Figure 6 Reshaped Matrices



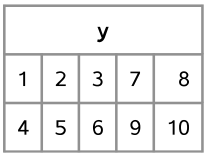
row					
1	2	3	4	5	6

m		
1	2	3
4	5	6

Another way to change the dimensions of a matrix is to increase the number of rows or columns. The following statements use the horizontal concatenation operator (||) to append a row vector onto the bottom of **x**, and the vertical concatenation operator (//) to append two columns:

```
z = x // {7 8 9};                        /* add new row at bottom */
y = x || {7 8, 9 10};                    /* add two new columns   */
print y;
```

Figure 7 The Result of Horizontal Concatenation



y				
1	2	3	7	8
4	5	6	9	10

MATRIX AND VECTOR OPERATIONS

The fundamental data structure in the SAS/IML language is a matrix. Binary operators such as addition and multiplication act on matrices. The rules of linear algebra define matrix operations and also define the dimensions of matrices for which binary operators are well defined.

Three types of SAS/IML operations enable you to combine matrices of compatible dimensions:

- Elementwise operations act on each element of a matrix. Examples include linear operations such as $sX + tY$, where s and t are scalar values and X and Y are matrices that have the same dimensions. In addition to scalar multiplication, the SAS/IML language includes the elementwise operators for addition (+), subtraction (–), multiplication (#), division (/), and exponentiation (##). For example, if $A = X \# Y$, then $A_{ij} = X_{ij} Y_{ij}$ for all values of i and j.
- Matrix multiplication, which includes the inner and outer products of two vectors, is a matrix operation. If A is an $n \times p$ matrix and B is a $p \times m$ matrix, then $A * B$ is an $n \times m$ matrix and the (i, j)th element of the product is $\Sigma_{k=1}^{p} A_{ik} B_{kj}$. The number of columns of A must equal the number of rows of B.
- A hybrid operation is an elementwise operation that acts on the rows or columns of matrices that have different dimensions. The SAS/IML language looks at the dimensions to determine whether it can make sense of an arithmetic expression. For example, suppose A is an $n \times p$ matrix and v is a $1 \times p$ vector. Because A and v both have the same number of columns, it is possible to evaluate the expression $A + v$ as an $n \times p$ matrix whose (i, j)th element is $A_{ij} + v_j$. In other words, v_j is added to the jth column of A. Similarly, if u is an $n \times 1$ nonzero vector, then A/u is computed as an $n \times p$ matrix whose (i, j)th element is A_{ij}/u_i.

The following program demonstrates the three types of matrix operations. The hybrid example standardizes a matrix by subtracting the mean of each column and then dividing each column by its standard deviation.

```
proc iml;
/* true elementwise operations */
u = {1 2};
v = {3 4};
w = 2*u - v;                  /* w = {-1 0} */

/* true matrix operations */
A = {1 2, 3 4};
b = {-1, 1};
z = A*b;                      /* z = {1, 1} */

/* hybrid elementwise operations */
x =  {-4 9,
       2 5,
       8 7};
```

```
mean= {2 7};
std = {6 2};

center = x - mean;              /* subtract mean[j] from jth column */
stdX = center / std;            /* divide jth column by std[j]      */
print stdX;
```

Figure 8 Result of Matrix Operations

stdX	
-1	1
0	-1
1	0

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

Table 1 Behavior of Elementwise Operators

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]**

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 in statistical textbooks. (Because the transpose operator can be difficult to see, some programmers prefer to use the T function to transpose matrices.)

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 a $1 \times p$ vector b such that Xb is close to y in the least squares sense. You can use the so-called *normal equations*, $(X'X)b = X'y$, to find the OLS solution. 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 SAS/IML language has many built-in functions for solving linear and nonlinear equations and for finding the optima of functions. You can use the SOLVE function to solve the normal equations for the regression parameter estimates, as follows:

```
b = solve(xpx, xpy);                /* solve for parameter estimates */
print b;
```

Figure 9 Solution of Normal Equations

b
2.1071429
2.9761905

ROWS, COLUMNS, and SUBMATRICES

Matrices consist of columns and rows. It is often useful to extract a subset of observations or columns as part of an analysis. For example, you might want to extract observations that correspond to all patients who smoke. Or you might want to extract columns of highly correlated variables.

In general, a rectangular subset of rows and columns is called a submatrix. You can specify a submatrix by using subscripts to specify the rows and columns of a matrix. Use square brackets to specify subscripts. For example, if **A** is a SAS/IML matrix, the following are submatrices:

- The expression `A[2,1]` is a scalar that is formed from the second row and the first column of **A**.
- The expression `A[2, ]` specifies the second row of **A**. The column subscript is empty, which means “use all columns.”
- The expression `A[ , {1 3}]` specifies the first and third columns of **A**. The row subscript is empty, which means “use all rows.”
- The expression `A[3:4, 1:2]` specifies a 2×2 submatrix that contains the elements that are in the intersection of the third and fourth rows of **A** and the first and second columns of **A**.

The following SAS/IML program specifies a few submatrices. Figure 10 shows the matrices **r** and **m**.

```
A = {1  2  3,
     4  5  6,
     7  8  9,
    10 11 12};
r = A[2, ];              /* second row */
m = A[3:4, 1:2];         /* intersection of specified rows and cols */
print r, m;
```

Figure 10 Submatrices

r		
4	5	6

m	
7	8
10	11

You can use subscripts not only to extract submatrices, but also to assign matrix elements. The following statements assign values to elements of **A**:

```
A[2, 1] = .;
A[3:4, 1:2] = 0;            /* assign 0 to ALL of these elements  */
A[{1 5 9}] = {-1 -2 -3};    /* assign elements in row-major order */
print A;
```

Figure 11 Assigning to Matrix Elements

A		
-1	2	3
.	-2	6
0	0	-3
0	0	12

READING AND WRITING DATA

A major reason to use SAS/IML software rather than another matrix language is the ease with which the language interacts with other parts of SAS software. For example, SAS/IML software can easily read SAS data sets into matrices and vectors, and it is easy to create a SAS data set from SAS/IML matrices.

CREATING MATRICES FROM SAS DATA SETS

You can use the USE and READ statements to read data into SAS/IML matrices and vectors. You can read variables into vectors by specifying the names of the variables that you want to read. The following statements read the first three observations from the Sashelp.Cars data set:

```
proc iml;
/* read variables from a SAS data set into vectors */
varNames = {"Make" "Model" "Mpg_City" "Mpg_Highway"};
use Sashelp.Cars(OBS=3);                /* open data for reading    */
read all var varNames;                  /* create vectors: Make,... */
close Sashelp.Cars;                     /* close data set           */
print Make Model Mpg_City Mpg_Highway;
```

Figure 12 Reading from SAS Data Set into Vectors

Make	Model	MPG_City	MPG_Highway
Acura	MDX	17	23
Acura	RSX Type S 2dr	24	31
Acura	TSX 4dr	22	29

You can also read a set of variables into a matrix (assuming that the variables are either all numeric or all character) by using the INTO clause in the READ statement. The following statements illustrate this approach. Again, only three rows of the data are read.

```
/* read variables from a SAS data set into a matrix */
varNames = {"Mpg_City" "Mpg_Highway" "Cylinders"};
use Sashelp.Cars(OBS=3);               /* open data for reading     */
read all var varNames into m;          /* create matrix with 3 cols */
print m[c=varNames];                   /* C= same as COLNAME=       */
```

Figure 13 Reading from SAS Data Set into a Matrix

m		
Mpg_City	**Mpg_Highway**	**Cylinders**
17	23	6
24	31	4
22	29	4

You can read only the numeric variable in a data set by specifying the _NUM_ keyword in the READ statement:

```
read all var _NUM_ into y[colname=NumericNames];
```

The columns of the matrix `y` contain the data for the numeric variables; the matrix `NumericNames` is filled with the names of those variables. The _CHAR_ keyword works similarly.

CREATING SAS DATA SETS FROM MATRICES

In a similar way, you can use the CREATE and APPEND statements to create a SAS data set from data in vectors or matrices. The following statements create a data set called Out in the Work library:

```
proc iml;
x = 1:5; y = 5:1; v = "v1":"v5";          /* define the data */
create Out var {"x" "y" "v"};             /* name the vars   */
append;                                   /* write the data  */
close Out;
```

The CREATE statement opens Work.Out for writing. The APPEND statement writes the values of the vectors that are listed in the VAR clause of the CREATE statement. The CLOSE statement closes the data set.

If you want to create a data set from a matrix of values, you can use the FROM clause in the CREATE and APPEND statements. If you do not explicitly specify names for the data set variables, the default names are **COL1**, **COL2**, and so on. You can explicitly specify names for the data set variables by using the COLNAME= option, as shown in the following statements:

```
/* create SAS data set from a matrix */
m = {1 2, 3 4, 5 6, 7 8};                  /* 4 x 2 matrix   */
create Out2 from m[colname={"x" "y"}];     /* name vars      */
append from m;                             /* write the data */
close Out2;
```

PROGRAMMING FUNDAMENTALS

The key to efficient programming in a matrix language is to use matrix computations as often as possible. A program that loops over rows and columns of a matrix is usually less efficient than a program that takes advantage of built-in functions and matrix computations.

AVOIDING LOOPS

However, the SAS/IML language *does* have an iterative DO statement that enables you to execute a group of statements repeatedly. It takes practice to know whether a DO loop is essential or whether it can be avoided. For example, suppose you want to compute the mean of each column of a matrix. A novice programmer might write the following double loop:

```
proc iml;
s = {1 2 3, 4 5 6, 7 8 9, 10 11 12};    /* 4 x 3 matrix      */
results = j(1, ncol(s));                /* allocate results  */
```

```
/* First attempt: Double loop (very inefficient) */
do j = 1 to ncol(s);                         /* loop over columns   */
   sum = 0;
   do i = 1 to nrow(s);                      /* loop over rows      */
      sum = sum + s[i,j];                    /* sum of column       */
   end;
   results[j] = sum / nrow(s);               /* mean of j_th column */
end;
```

Notice that there are no vector operations in this program, only a lot of scalar operations. A more experienced SAS/IML programmer might use the SUM function, which can return the sum of each column. Consequently, the following program is more efficient:

```
/* Second attempt: Single loop over columns (slightly inefficient) */
do i = j to ncol(s);
   results[j] = sum(s[ ,j]) / nrow(s);
end;
```

The program now consists of a series of vector operations, which is a step in the right direction. However, the program can be improved further if you use the MEAN function, which computes the mean values of each column of a matrix. Consequently, the program reduces to a single function call that takes a matrix as an argument:

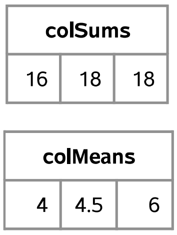
```
/* MEAN function operates on cols. No loop!  */
results = mean(s);                             /* mean of each col  */
```

SUBSCRIPT REDUCTION OPERATORS

You can also avoid writing loops by using the SAS/IML subscript reduction operators. These operators enable you to perform common statistical operations (such as sums, means, and sums of squares) on either the rows or the columns of a matrix. For example, the following statements compute the sum and mean of columns and of rows for a matrix. Figure 14 shows the results.

```
proc iml;
/* compute sum and mean of each column */
x = {1 2 3,
     4 5 6,
     7 8 9,
     4 3 .};
colSums  = x[+, ];
colMeans = x[:, ];                          /* equivalent to mean(x) */
rowSums  = x[ ,+];
rowMeans = x[ ,:];
print colSums, colMeans, rowSums rowMeans;
```

Figure 14 Sums and Means of Rows and Columns



colSums		
16	18	18

colMeans		
4	4.5	6

Figure 14 *continued*

rowSums	rowMeans
6	2
15	5
24	8
7	3.5

The expression `x[+, ]` uses the `'+'` subscript operator to “reduce” the matrix by summing the row elements for each column. (Recall that not specifying a column in the second subscript is equivalent to specifying all columns.) The expression `x[:, ]` uses the `':'` subscript operator to compute the mean for each column. The row sums and means are computed similarly. Notice that the subscript reduction operators correctly handle the missing value in the third column.

Table 2 summarizes the subscript reduction operators for matrices and shows an equivalent function call.

Table 2 Subscript Reduction Operators for Matrices

Operator	Action	Equivalent Function
+	Addition	sum(x)
#	Multiplication	prod(x)
><	Minimum	min(x)
<>	Maximum	max(x)
>:<	Index of minimum	loc(x=min(x))[1]
<:>	Index of maximum	loc(x=max(x))[1]
:	Mean	mean(x) /* mean of columns */
##	Sum of squares	ssq(x)

LOCATE OBSERVATIONS

One situation where vectorization is important is in finding observations that satisfy a given set of conditions. You can locate observations by using the LOC function, which returns the indices of elements that satisfy the condition.

For example, suppose you want to find vehicles in the Sashelp.cars data set that have fewer than six cylinders and that get more than 35 miles per gallon in city driving. An inefficient way to find these vehicles would be to loop over all observations and to use concatenation to build up a vector that contains the observation numbers. An efficient approach follows:

```
proc iml;
varNames = {"Cylinders" "Mpg_City"};
use Sashelp.Cars;
read all var varNames into X;                         /* read data  */
idx = loc(X[,1]<6 & X[,2]>35);                        /* row vector */

print (idx`)[label="Row"] (X[idx,])[c=varNames];
```

Figure 15 Observations That Satisfy Criteria

Row	Cylinders	Mpg_City
150	4	46
151	3	60
156	4	36
374	4	59
405	4	38

If no observations satisfy the criteria, the LOC function returns an empty matrix. It is a good programming practice to check for an empty matrix, as follows:

```
if ncol(idx) > 0 then do;
   /* obs found... do something with them */
end;
else do;
   print "No observations satisfy the condition.";
end;
```

You can skip the check if you are certain that at least one observation satisfies the criteria.

HANDLE MISSING VALUES

Although many built-in SAS/IML functions handle missing values automatically, you might need to find and delete missing values when you implement your own algorithms. The LOC function is useful for finding nonmissing values. For example, the following statements extract the nonmissing data values into a vector:

```
proc iml;
x = {1, ., 2, 2, 3, .};
nonMissing = loc(x ^= .);                       /* {1 3 4 5}        */
y = x[nonMissing];                              /* y = {1,2,2,3};   */
```

If you account for missing values at the beginning of a program, the rest of the program is often much easier to write and to understand.

For a data matrix, it is often useful to delete an entire row of a matrix if any element in the row is missing. The remaining rows are called *complete cases*. Many multivariate and regression analyses begin by deleting records for which any variable is missing. In the SAS/IML language, you can use the LOC and COUNTMISS functions to find and keep the complete cases, as follows:

```
/* exclude rows with missing values */
z = {1 .,
     2 2,
     . 3,
     4 4};
numMiss = countmiss(z, "row");         /* count missing in each row */
y = z[ loc(numMiss=0), ];              /* z[{2 4}, ] = {2 2, 4 4}   */
```

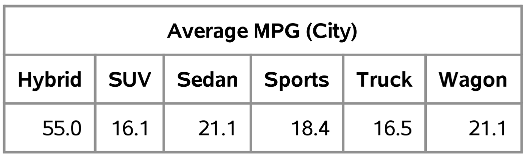
ANALYZE LEVELS OF CATEGORICAL VARIABLES

Another useful application of the LOC function is to compute statistics for each level of a categorical variable. The UNIQUE function enables you to find the unique (sorted) values of a vector. You can iterate over the unique values and use the LOC function to extract the observations for each category. You can then perform a statistical analysis on the observations for that category. This is analogous to BY-group processing in other SAS procedures.

For example, suppose you want to compute the mean fuel efficiency for vehicles in the Sashelp.cars data set, grouped by categories of the **Type** variable. The following SAS/IML program uses the UNIQUE-LOC technique to compute the means:

```
proc iml;
use Sashelp.Cars;
read all var {"Type" "Mpg_City"};
close Sashelp.Cars;

/* UNIQUE-LOC technique */
uC = unique(Type);                    /* find unique values          */
mean = j(1, ncol(uC));                /* allocate vector for results */
do i = 1 to ncol(uC);
   idx = loc(Type = uC[i] );          /* locate these obs            */
   mean[i] = mean( Mpg_City[idx] );   /* find mean of mpg            */
end;
print mean[colname=uC label="Average MPG (City)" format=4.1];
```

Figure 16 Mean Values by Categories



Average MPG (City)					
Hybrid	SUV	Sedan	Sports	Truck	Wagon
55.0	16.1	21.1	18.4	16.5	21.1

If all you want to do is count the frequencies of observations in each category, you do not need to use the UNIQUE-LOC technique. Given a vector of categories, the TABULATE routine returns two values: a matrix of the unique categories and the frequencies of each category.

USER-DEFINED FUNCTIONS

The SAS/IML language supports hundreds of built-in functions, and you can also call hundreds of Base SAS® functions. However, if you need additional statistical functionality, you can extend the SAS/IML library by defining a module. A module is a function or subroutine that is written in the SAS/IML language and that you can define, store, and call from your code as if it were a built-in function. Modules enable you to package, reuse, and maintain related SAS/IML statements in a convenient way.

A module definition begins with a START statement and ends with a FINISH statement. For a function module, a RETURN statement specifies the return matrix.

For example, suppose you want to define a function that standardizes the columns of a matrix. To standardize a column, you subtract the mean value from each column and divide the result by the standard deviation of the column. The following function uses the MEAN and STD functions to standardize each column of a matrix. (The STD function returns the standard deviation of each column of a matrix.) Each column of the resulting matrix has zero mean and unit variance.

```
proc iml;
/* standardize each column of x to have mean 0 and unit variance */
start Stdize(x);
   return( (x - mean(x)) / std(x) );
finish;

/* test it */
A = {0 2 9,
     1 4 3,
    -1 6 6};
z = Stdize(A);
print z;
```

Figure 17 Calling a User-Defined Module

z		
0	-1	1
1	0	-1
-1	1	0

You can use the STORE statement to store a module in a user-defined library so that you can use it later. To use a stored module, use the LOAD statement to load the module from the library.

LOCAL VERSUS GLOBAL MATRICES

Inside a module, all matrix names are local to the module. In particular, they do not conflict with or overwrite matrices that have the same name but are defined outside the module. The following statements define a matrix named **x** that is outside all modules. A matrix named **x** is also inside the Sqr module. When the Sqr module is run, the value of **x** inside the module is set to 4. However, the value of **x** outside the module is unchanged, as shown in Figure 18.

```
proc iml;
x = 0;                                    /* x is outside function  */
start Sqr(t);
   x = t##2;                              /* this x is local        */
   return (x);
finish;

s = Sqr(2);                               /* inside module, x is 4  */
print x[label="Outside x (unchanged)"];   /* outside, x not changed */
```

Figure 18 Matrix Outside a Module (Unchanged)

Outside x (unchanged)
0

If you want a module to be able to refer to or change a matrix that is defined outside the module, you can use the GLOBAL statement to specify the name of the matrix. In the following statements, the **x** matrix is declared to be a global matrix. When the module makes an assignment to **x**, the value of **x** outside the module is changed, as shown in Figure 19.

```
start Sqr2(t) global(x);                   /* GLOBAL matrix         */
   x = t##2;                               /* this x is global      */
finish;

run Sqr2(2);
print x[label="Outside x (changed)"];      /* outside, x is changed */
```

Figure 19 Matrix Outside a Module (Changed)

Outside x (changed)
4

PASSING ARGUMENTS BY REFERENCE

The GLOBAL clause is sometimes used when a module needs to read the values of parameters. However, it is considered a poor programming practice for a module to change the value of a global matrix. Instead, the preferred convention is to pass the matrix to a module as an argument.

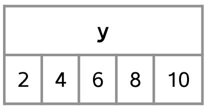
All SAS/IML modules pass parameters by reference, which means that a module can change the values of its arguments. This is a very efficient way to pass matrices because it avoids having to allocate memory and copy values every time a module is called. On the other hand, the programmer needs to be careful not to inadvertently change the value of an argument.

The following example illustrates the concept of passing a parameter by reference:

```
proc iml;
start Double(x);                                   /* arg is changed */
   x = 2*x;
finish;

y = 1:5;
run Double(y);
print y;
```

The Double module doubles the elements of its parameter and overwrites the original elements with new values. “Passing by reference” means that the matrix **y** outside the module shares the same memory as the matrix **x** inside the module. Consequently, when the values of **x** are updated inside the module, the values of **y** outside the module also change, as shown in Figure 20.

Figure 20 Passing by Reference

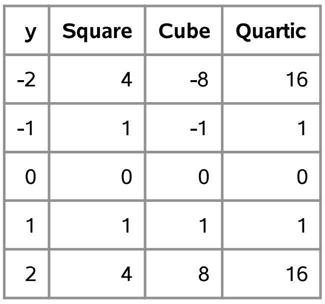


y				
2	4	6	8	10

You can use this fact to return multiple matrices from a subroutine. For example, suppose you want to compute matrices that contain the square, cube, and fourth power of elements of an input matrix. You can do this by defining a subroutine that takes four parameters. By convention, the output parameters are listed first and the input parameters are listed last, as shown in the following statements:

```
/* define subroutine with output arguments */
start Power(x2, x3, x4, x);
  x2 = x##2;
  x3 = x##3;
  x4 = x##4;
finish;

y = {-2, -1, 0, 1, 2};
run Power(Square, Cube, Quartic, y);
print y Square Cube Quartic;
```

Figure 21 Returning Multiple Values



y	Square	Cube	Quartic
-2	4	-8	16
-1	1	-1	1
0	0	0	0
1	1	1	1
2	4	8	16

This program demonstrates that the output arguments do not need to be allocated in advance if they are assigned inside the module.

CALLING SAS PROCEDURES

A very useful feature of the SAS/IML language is its ability to call SAS procedures (including DATA steps and macros) from within a SAS/IML program. This enables the SAS/IML programmer to access any statistic that can be produced in SAS!

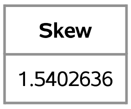
Calling a SAS procedure from within a SAS/IML program usually consists of three steps:

1. Write the data to a SAS data set.
2. Use the SUBMIT and ENDSUBMIT statements to execute SAS code.
3. Read the results into SAS/IML matrices.

As a simple example, suppose you want to compute the skewness of data that are contained in a SAS/IML vector. You could write a module that computes the skewness, but it is quicker and less prone to error to call the MEANS procedure, which reads data from a SAS data set. The following statements create a data set, call the MEANS procedure inside a SUBMIT-ENDSUBMIT block, and read the results back into a SAS/IML scalar matrix:

```
proc iml;
/* start with data in SAS/IML vector */
x = {1 1 1 1 2 2 2 3 4 4 5 6 6 8 9 11 11 15 22}`;

/* 1. Write to SAS data set */
create In var {"x"};
append;
close In;

/* 2. Call SAS procedure */
submit;
  proc means data=In noprint;
    var x;
    output out=Output Skewness=Skew;
  run;
endsubmit;

/* 3. Read results */
use Output;
read all var {"Skew"};
close Output;

print Skew;
```

Figure 22 Results of Calling a SAS Procedure

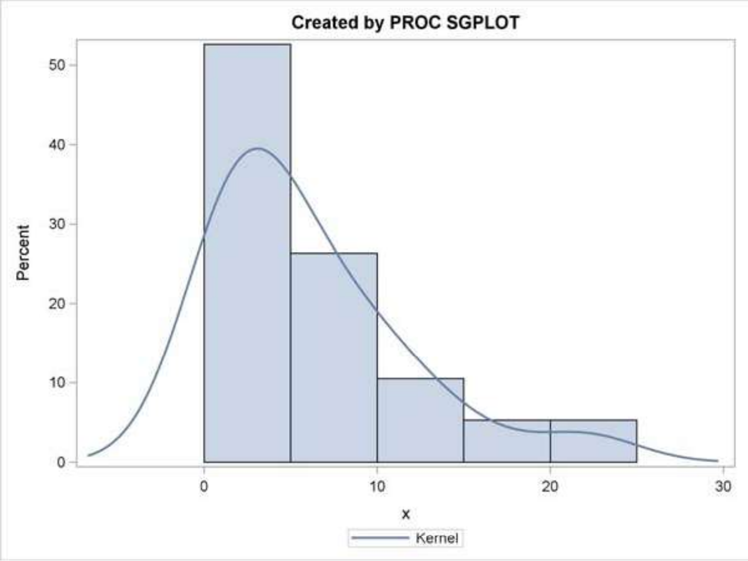


Skew
1.5402636

In a similar way, you can use SUBMIT and ENDSUBMIT statements to call graphical procedures, such as the SGPLOT procedure. For example, if you want to visualize the data in the previous example, you can use PROC SGPLOT to create a histogram:

```
submit;
  proc sgplot data=In;
    title "Created by PROC SGPLOT";
    histogram x;
    density x / type=kernel;
  run;
endsubmit;
```

Figure 23 Results of Calling a Graphical Procedure



In a similar way, you can call R (an open-source statistical language) from SAS/IML software. The *SAS/IML User's Guide* and Wicklin (2010a) provide examples.

APPLICATIONS AND STATISTICAL TASKS

You can use the SAS/IML language to compute many statistical quantities. The SAS/IML language is used by SAS testing groups to validate the results of almost every analytical procedure—from regression and multivariate analyses in SAS/STAT® to time series modeling in SAS/ETS® to data mining methods in SAS® Enterprise Miner™.

SAS customers use SAS/IML software to compute statistics that are not produced by other SAS procedures. Wicklin (2010a) describes how to use the SAS/IML language to implement techniques in modern data analysis. The remainder of this paper describes two common uses of SAS/IML software: simulation and optimization.

SIMULATING DATA

Simulating data is an essential technique in modern statistical programming. You can use data simulation for hypothesis testing, for computing standard errors, and for estimating the power of a test (Wicklin 2013). Simulation is also useful for comparing the performance of statistical techniques.

A complete description of how to use SAS/IML for efficient simulation is beyond the scope of this paper. However, the main idea is that the SAS/IML language supports the RANDGEN subroutine, which fills an entire matrix with random values by making a single call.

To illustrate simulating data, the following SAS/IML program simulates 10,000 random samples of size 10 drawn from a uniform distribution on $[0, 1]$. For each sample, the mean of the sample is computed. The collection of sample means approximates the sampling distribution of the mean.

```
proc iml;
N = 10;                          /* number of obs in each sample      */
NumSamples = 10000;              /* number of samples                 */

call randseed(123);              /* set seed for random number stream */
x = j(N, NumSamples);            /* each column is sample of size N   */
call randgen(x, "Uniform");      /* simulate data                     */
s = mean(x);                     /* compute mean for each col         */

/* summarize approximate sampling distribution */
s = T(s);
Mean = mean(s);
StdDev = std(s);
print Mean StdDev;
```

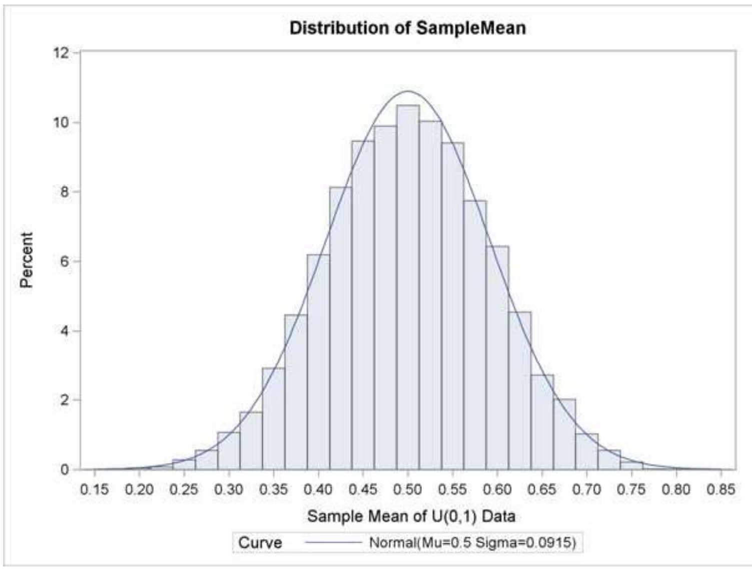
Figure 24 shows summary statistics for the sampling distribution. The central limit theorem (CLT) states that the sampling distribution of the mean is approximately normally distributed for large sample sizes. In fact, the CLT states that for a uniform population on $[0, 1]$, the mean and standard deviation of the sampling distribution will be approximately 0.5 and $1/\sqrt{12N}$ for large N. Although $N = 10$ is not "large," the values in Figure 24 are nevertheless close to 0.5 and 0.0913.

Figure 24 Descriptive Statistics for Sampling Distribution

Mean	StdDev
0.499969	0.0915287

You can write the data to a SAS data set and use the UNIVARIATE procedure to display a histogram and overlay a normal curve, as shown in Figure 25.

Figure 25 Sampling Distribution of Mean



OPTIMIZATION

Many statistical methods require optimization of nonlinear functions. A common example is maximum likelihood estimation, which requires finding parameter values that maximize the (log-) likelihood function. One situation where this arises is in fitting parametric distributions to univariate data.

For example, suppose you want to fit a normal density curve to the **SepalWidth** variable in Fisher's famous iris data set (Sashelp.Iris). To fit the data well, you have to find some way to choose the parameters μ and σ of the normal density. Maximum likelihood estimation produces one way to fit the data. For the normal distribution, the maximum likelihood estimates (MLE) can be found by calculus. The optimal values are $(\mu, \sigma) = (\bar{x}, s_n)$, where $\bar{x}$ is the sample mean and $s_n^2 = \Sigma_{i=1}^{n}(x_i - \bar{x})/n$ is the biased sample variance.

The following statements read the **SepalWidth** data from the Sashelp.Iris data set and compute the MLEs, which are shown in Figure 26:

```
proc iml;

use Sashelp.Iris;                                      /* read data */
read all var {SepalWidth} into x;
close Sashelp.Iris;

/* print the optimal parameter values */
muMLE = mean(x);
n = countn(x);
sigmaMLE = sqrt( (n-1)/n * var(x) );
print muMLE sigmaMLE;
```

Figure 26 Maximum Likelihood Estimates for Normal Parameters

muMLE	sigmaMLE
30.573333	4.3441097

Unfortunately, the MLEs do not have a convenient closed-form solution for most nonnormal distributions. For most distributions, numerical optimization is the only way to obtain the parameter estimates. To demonstrate numerical optimization, the following example computes the MLEs for the normal example and compares the numerical solution to the exact values. The same numerical optimization technique can be used for cases in which an exact solution is not available.

To use maximum likelihood estimation, you need to write a module that defines the log-likelihood function. The following statements define a module named NormLogLik that takes a single argument: the vector of parameters to be optimized. A GLOBAL statement enables the function to reference the data in the **x** matrix. The function returns the value of the log-likelihood function for any (μ, σ) value.

```
/* 1. compute the log-likelihood function for Normal distrib */
start NormLogLik(parm)  global (x);             /* param = {mu sigma} */
   mu = parm[1];  sigma2 = parm[2]##2;
   n = nrow(x);
   return( -n/2*log(sigma2) - 0.5/sigma2*sum((x-mu)##2) );
finish;
```

The SAS/IML run-time library supports several functions for nonlinear optimization. The following example uses the Newton-Raphson (NLPNRA) algorithm, which takes four input parameters:

1. The function to be optimized, which is NormLogLik. This function must take a single argument: the vector of parameters to optimize. Other relevant information (such as the data values) is specified by using the GLOBAL clause.

2. An initial value for the parameters to optimize. For this problem, the vector has two elements, which you can choose arbitrarily to be $(\mu_0, \sigma_0) = (35, 5.5)$.

3. A vector of options that control the Newton-Raphson algorithm. You can specify many options, but this example uses only two. The first option specifies that the Newton-Raphson algorithm should find a maximum, and the second option specifies that the algorithm should print information that is related to the solution.

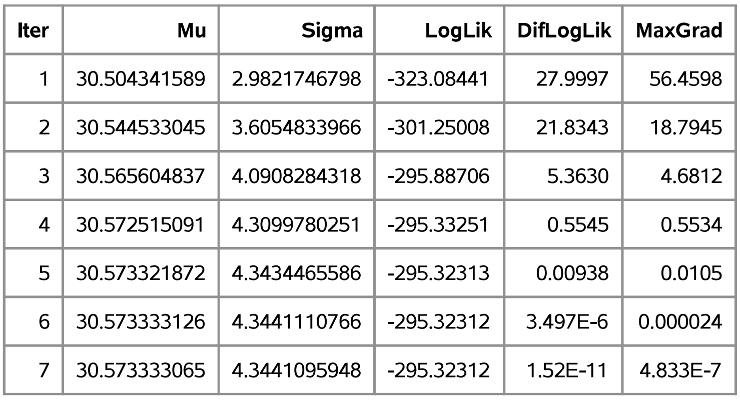
4. A constraint matrix for the parameters. For this problem, the μ parameter is unconstrained, but the standard deviation σ must be a positive quantity.

The following statements set up the initial conditions, options, and constraints, and they call the optimization routine:

```
parm = {35 5.5};     /* 2. initial guess for solution (mu, sigma)   */
optn = {1,           /* 3. find max of function, and                */
        4};          /*    print moderate amount of output          */
con  = {.   0,       /* 4. lower bound: -infty < mu; 0 < sigma, and */
        .   .};      /*    upper bound:  mu < infty; sigma < infty  */

/* 5. Provide initial guess and call NLP function */
call nlpnra(rc, result, "NormLogLik", parm, optn, con);
```

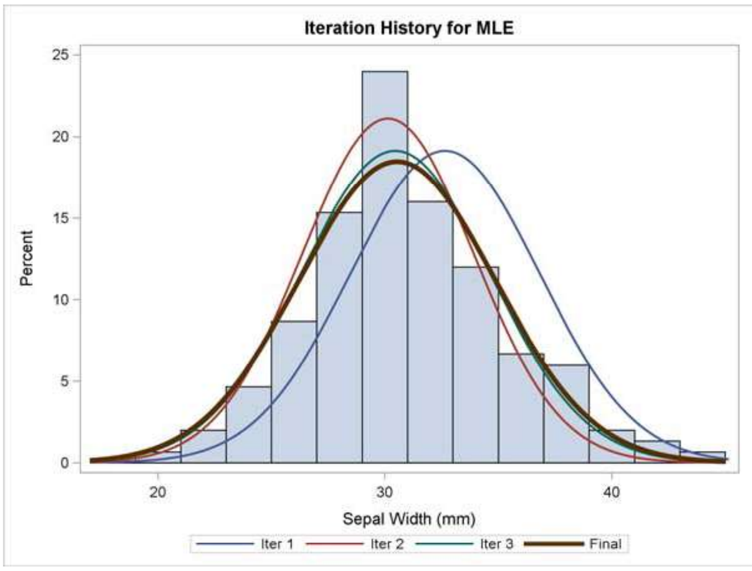
Figure 27 displays the iteration history for the Newton-Raphson algorithm. The table shows that the algorithm requires six iterations to progress from the initial guess to the final parameter estimates. For each step, the “Objective Function” (the NormLogLik function) increases until it reaches a maximum at iteration 7. You can visualize the process by using the SGPLOT procedure to overlay several normal density curves on a histogram of the data, as shown in Figure 28. The final parameter estimates differ from the theoretical values by less than 10^{-6}.

Figure 27 Iteration History for Newton-Raphson Optimization



Iter	Mu	Sigma	LogLik	DifLogLik	MaxGrad
1	30.504341589	2.9821746798	-323.08441	27.9997	56.4598
2	30.544533045	3.6054833966	-301.25008	21.8343	18.7945
3	30.565604837	4.0908284318	-295.88706	5.3630	4.6812
4	30.572515091	4.3099780251	-295.33251	0.5545	0.5534
5	30.573321872	4.3434465586	-295.32313	0.00938	0.0105
6	30.573333126	4.3441110766	-295.32312	3.497E-6	0.000024
7	30.573333065	4.3441095948	-295.32312	1.52E-11	4.833E-7

Figure 28 Sampling Distribution of Mean



CONCLUSION

This paper introduces the SAS/IML language to SAS programmers. It also focuses on features such as creating matrices, operating with matrices, and reading and writing SAS data sets. The paper also describes how to write efficient programs, how to create user-defined functions, and how to call SAS procedures from within a SAS/IML program. Finally, it demonstrates two popular uses of the SAS/IML language: simulating data and optimizing functions.

The SAS/IML language is a powerful addition to the toolbox of a SAS statistical programmer. But like any powerful tool, it requires practice and experience to use effectively. You can use the resources described in the following list to learn more about the SAS/IML language:

- Much of this paper is taken from material in Chapters 2–4 of Wicklin (2010b), which contains many additional examples and practical programming techniques.
- *The DO Loop* blog often demonstrates statistical programming in the SAS/IML language. You can browse or subscribe to this blog at `blogs.sas.com/content/iml`.
- The SAS/IML Support Community at `communities.sas.com/community/support-communities` is an online forum where you can ask questions about SAS/IML programming.
- The first six chapters of the *SAS/IML User's Guide* are intended for beginners. The "Language Reference" chapter contains complete cut-and-paste examples for every function in the SAS/IML language.
- *SAS/IML Studio for SAS/STAT Users* provides a short introduction to programming dynamically linked graphics in SAS/IML Studio.

REFERENCES

Wicklin, R. (2010a), "Rediscovering SAS/IML Software: Modern Data Analysis for the Practicing Statistician," in *Proceedings of the SAS Global Forum 2010 Conference*, Cary, NC: SAS Institute Inc.
URL `http://support.sas.com/resources/papers/proceedings10/329-2010.pdf`

Wicklin, R. (2010b), *Statistical Programming with SAS/IML Software*, Cary, NC: SAS Institute Inc.

Wicklin, R. (2013), *Simulating Data with SAS*, Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

The author is grateful to Maribeth Johnson for the invitation to present this material as a Hands-On Workshop.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author:

Rick Wicklin
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513