From Chapter 10 - Programming with matrices and vectors - IML

10.1: Basic matrix definition + subscripting
10.2: Diagonal matrices and stacking matrices
10.3: Repeating, Element-wise operations and Matrix Multiplication
10.4 Importing SAS data sets into IML and exporting matrices from IML to data set
10.4.1: Creating matrices from SAS data sets and vice versa
10.5: CASE STUDY 1: Using IML to implement Monte Carlo integration to estimate \( \pi \)
10.6: CASE STUDY 2: IML to implement a bisection root finder
10.7: CASE STUDY 3: Randomization test using matrices imported from PLAN
10.8: CASE STUDY 4: IML module to implement Monte Carlo integration to estimate \( \pi \)

Summary

References

Exercises

SAS data sets are rectangular objects that can be manipulated using various PROCs.

IML is short for Interactive Matrix Language and represents a tool for manipulating matrices.

What is a matrix? A matrix can be thought of as a special case of a data set.

A matrix is a rectangular object where all elements are of the same data type (e.g. all numeric, all character). Thus,

\[
A = \begin{bmatrix}
17 & 2.1 & 11 \\
15 & 3.7 & 15 \\
18 & 4.4 & 17 \\
\end{bmatrix}
\]

is a matrix while

\[
B = \begin{bmatrix}
17 & \text{George} & 11 \\
15 & \text{Jungle} & 15 \\
18 & \text{Jane} & 17 \\
\end{bmatrix}
\]

is not.

Matrices, such as \( A \), have properties including dimension corresponding to the number of rows and the number of columns. In this simple example, \( A \) has 3 rows and 3 columns or has dimension 3x3. Elements of a matrix are referenced by their row and column position, for example, “11” is the (1,3) element of \( A \), or \( A[1,3]=11 \). Matrices with one dimension equal to 1
are called **vectors**. For example, \( X = \begin{bmatrix} 17 \\ 15 \\ 18 \end{bmatrix} \), a 3x1 matrix, is called a **column vector** while \( Y = \begin{bmatrix} 17 & 15 & 18 \end{bmatrix} \), a 1x3 matrix is called a **row vector**. Finally a 1x1 matrix \( Z = [12] \) is called a **scalar**. Names of matrices in **SAS IML** are valid SAS names, 1-32 characters in length and can begin of a letter or underscore and can continue with letters, numbers or underscores. In the next example, we see how matrices can be defined in **PROC IML** and perform some basic manipulations (e.g. extracting a row or column, summing over rows or columns).

10.1: Basic matrix definition + subscripting

**Display 10.1: Basic matrix manipulations in IML**

```sas
PROC IML;
* makes a 2x3 matrix;
C = {1 2 3, 4 5 6};
print '2x3 example matrix C = {1 2 3, 4 5 6} = ' C;

* select 2nd row;
C_r2 = C[2,];
print '2nd row of C = C[2,] = ' C_r2;

* select 3rd column;
C_c3 = C[,3];
print '3rd column of C = C[,3] = ' C_c3;

* select last two columns;
Col23 = C[,2:3];
print 'Columns 2 and 3 of C = ' Col23;

* select the (2,3) element of C;
C23 = C[2,3];
print '(2,3) element of C = C[2,3] = ' C23;

* makes a 1x3 matrix by summing over rows in each column;
C_csum = C[+,];
print '1x3 column sums of C = C[+,] = ' C_csum;

* makes a 2x1 matrix by summing over columns in each rows;
C_rsum = C[,+];
print '2x1 row sums of C = C[,+] = ' C_rsum;
run;
```

**Display 10.2: Printed output from basic matrix manipulations in IML**

<table>
<thead>
<tr>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

\[ C_{\text{R2}} \]
square matrix has the same number of rows and columns.

diagonal matrix is a square matrix with non-zero elements on the diagonal and all off-diagonal elements equal to zero.

identity matrix is a diagonal matrix with all diagonal elements equal to 1.

The IML function DIAG can be used to construct a diagonal matrix from a vector while the IML function VECDIAG can be used to extract the diagonal elements from a matrix into a vector. In addition to defining special matrices, you may want to combine matrices. You might want to “stack” matrices vertically producing a matrix with the same number of columns but with additional rows (analogous to SET-ing data sets) or to combine matrices horizontally producing a matrix with the same number of rows but with additional columns (analogous to MERGE-ing data sets). Note that these manipulations assume that the matrices conform – e.g. you can only stack matrices vertically if they have the same number of columns. The IML special symbols “//” and “||” produce the vertical and horizontal stacking, respectively.

10.2: Diagonal matrices and stacking matrices

Display 10.3: Summarizing, extracting, shaping and manipulation vectors and matrices in IML

```sas
PROC IML;
* makes a 2x3 matrix;
  C = {1 2 3, 4 5 6};
  print '2x3 example matrix C = {1 2 3, 4 5 6} = ' C;
* makes a 1x3 matrix by summing over rows in each column
  C_C3 = C[,3] =
         3
         6
(2,3) element of C = C[2,3] = 6

COLUMNS 2 AND 3 OF C =
  2   3
  5   6

1x3 COLUMN SUMS OF C = C[+,] =
  5   7   9

2x1 ROW SUMS OF C = C[+1] =
  6
  15
```
and a 2x1 matrix by summing over columns in each row;
   C1=C[+,];
   C2=C[+,];

* makes a matrix (col. vector) out of second column of C;
   F = C[,2];
   print 'extract 2nd column of C into new vector (F) = C[,2] = ' F;

*puts second column of c on diagonal;
   D = DIAG( C[,2] );
   print 'put 2nd column of C into a diagonal matrix (D) = DIAG(C[,2]) = ' D;

*makes a vector out of the diagonal;
   CC= VECDIAG(D);
   print 'convert diagonal (of D) into vector (CC) = VECDIAG(D) = ' CC;

*puts C next to itself - column binds C with itself;
   E = C || C;
   print 'Column bind C with itself yielding E = C||C = ' E;

*puts a row of 2's below C - row bind ;
   F = C // SHAPE(2,1,3);
   print "Row bind C with vector of 2's (F) = C // SHAPE(2,1,3) =" F;

*creates a 6x6 matrix [C // C // C] || [C // C // C];
   K = REPEAT(C,3,2);
   print '6x6 matrix = ' K;
quit;

Display 10.4: Output from summarizing, extracting, shaping and manipulation vectors and matrices in IML

<table>
<thead>
<tr>
<th></th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x3 example matrix C = {1 2 3,4 5 6} =</td>
<td>1 2 3</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 5 6</td>
</tr>
<tr>
<td>C1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>9</td>
</tr>
<tr>
<td>1x3 column sums of C = C[+,] =</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>9</td>
</tr>
<tr>
<td>C2</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>15</td>
</tr>
<tr>
<td>2x1 row sums of C = C[+,] =</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>extract 2nd column of C into new vector (F) = C[,2] =</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>put 2nd column of C into a diagonal matrix (D) = DIAG(C[,2]) =</td>
<td>CC</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>convert diagonal (of D) into vector (CC) = VECDIAG(D) =</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
10.3: Repeating, Element-wise operations and Matrix Multiplication

Display 10.5: Matrix multiplication and element-wise operations in IML

```
PROC IML;
   C = {1 2 3, 4 5 6};   * a 2x3 matrix;
   D = {1, 1, 1};       * a 3x1 column vector;

   * matrix multiplication – C post-multiplied by D;
   row_sum = C*D;
   print "row_sum = " row_sum;

   *raises each entry of columns 2 & 3 of C to the third power then
   multiples by 3 and adds 3;
   G = 3+3*(C[,2:3]##3);
   print '3 + 3*(col2&3)^3 (G) = ' G;

   *raises each entry of C to itself;
   H = C ## C;
   print 'raise C elements to itself (H) = C##C = ' H;

   *multiplies each entry of C by itself;
   J = C # C;
   print 'element-wise multiplication of C with itself (J) = C#C = ' J;
quit;
```

Display 10.6: Output from IML Matrix multiplication and element-wise operations

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Row bind C with vector of 2's (F) = C // SHAPE(2,1,3) =

F

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

K

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

ROW_SUM

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>row_sum =</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

3 + 3*(col2&3)^3 (G) =

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>84</td>
</tr>
<tr>
<td>378</td>
<td>651</td>
</tr>
</tbody>
</table>
```
10.4 Importing SAS data sets into IML and exporting matrices from IML to data set

* IML functions `use` and `read` can be used to construct a matrix from a SAS data set

* IML functions `create` and `append` can be used to construct a SAS data set from an IML matrix.

10.4.1: Creating matrices from SAS data sets and vice versa

Display 10.7: Reading SAS data set containing the nitrofen study results into IML

```sas
libname mydat 'folder-containing-nitrofen-data';
proc iml;
/* read SAS data in IML */
use mydat.nitrofen;
read all var { total conc } into nitro;
/* alternative coding */
use mydat.nitrofen var{ total conc }; read all into nitro2;

nitro = nitro || nitro[,2]**2;  * adding column with conc^2;  
nitro2 = nitro2 || (nitro2[,2]- nitro2[+,2]/nrow(nitro2)) ; * add column with centered concentration;
nitro2 = nitro2 || nitro2[,3]**2;  * adding column with scaled conc^2;
show names;  * show matrices constructed in IML;
*print nitro;
*print nitro2;
create n2 from nitro2;  * creates SAS data set n2 from matrix nitro;
append from nitro2;
/* a little graphing in IML */
call pgraf(nitro[,2:1],'*','Nitrofen concentration', 'Number of young', 'Plot of #young vs. conc');
```
quit;

proc print data=n2;
  title 'print of data constructed in IML';
run;
Display 10.8: IML plot of the total young vs. nitrofen concentration data

Plot of #young vs. conc

print of data constructed in IML
Display 10.9: Data set produced by IML output of total young, nitrofen concentration, centered concentration and squared centered concentration

<table>
<thead>
<tr>
<th>Obs</th>
<th>COL1</th>
<th>COL2</th>
<th>COL3</th>
<th>COL4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>27</td>
<td>0</td>
<td>-157</td>
<td>24649</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>0</td>
<td>-157</td>
<td>24649</td>
</tr>
<tr>
<td>3</td>
<td>34</td>
<td>0</td>
<td>-157</td>
<td>24649</td>
</tr>
<tr>
<td>4</td>
<td>33</td>
<td>0</td>
<td>-157</td>
<td>24649</td>
</tr>
<tr>
<td>5</td>
<td>36</td>
<td>0</td>
<td>-157</td>
<td>24649</td>
</tr>
</tbody>
</table>

... (rows deleted) ...

<table>
<thead>
<tr>
<th>Obs</th>
<th>COL1</th>
<th>COL2</th>
<th>COL3</th>
<th>COL4</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>0</td>
<td>310</td>
<td>153</td>
<td>23409</td>
</tr>
<tr>
<td>45</td>
<td>15</td>
<td>310</td>
<td>153</td>
<td>23409</td>
</tr>
<tr>
<td>46</td>
<td>5</td>
<td>310</td>
<td>153</td>
<td>23409</td>
</tr>
<tr>
<td>47</td>
<td>6</td>
<td>310</td>
<td>153</td>
<td>23409</td>
</tr>
<tr>
<td>48</td>
<td>4</td>
<td>310</td>
<td>153</td>
<td>23409</td>
</tr>
<tr>
<td>49</td>
<td>6</td>
<td>310</td>
<td>153</td>
<td>23409</td>
</tr>
<tr>
<td>50</td>
<td>5</td>
<td>310</td>
<td>153</td>
<td>23409</td>
</tr>
</tbody>
</table>

10.5 CASE STUDY 1: Using IML to implement Monte Carlo integration to estimate $\pi$

Display 10.10: Using IML to estimate $\pi$ using Monte Carlo integration

```sas
proc iml;
nsim = 4000;
temp_mat = J(nsim,2,0);
/* Alternate method 1: Generate (X,Y) with single call using "randgen"
*/
call randgen(temp_mat,'uniform');

/* Alternate method 2: Not as efficient */
/* do isim = 1 to nsim;
temp_mat[isim,1] = uniform(0); * generate X ~ unif(0,1);
temp_mat[isim,2] = uniform(0); * generate Y ~ unif(0,1);
end;
*/
temp_mat = temp_mat ||
   (temp_mat[,2]<=
    sqrt(J(nsim,1,1)-temp_mat[,1]#temp_mat[,1]));
pi_over4 = temp_mat[+,3]/nsim;
pi_est = 4*pi_over4;
se_est = 4*sqrt(pi_over4*(1-pi_over4)/nsim);
pi_LCL = pi_est - 2*se_est;
pi_UCL = pi_est + 2*se_est;
```
* Estimating PI using MC simulation methods with nsim data points;
print 'Estimating PI using MC simulation methods with nsim data points';
print 'PI-estimate = pi_est se_est pi_LCL pi_UCL';
quit;

Executing the program above generates the following output.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PI_EST</td>
<td>SE_EST</td>
<td>PI_LCL</td>
<td>PI_UCL</td>
</tr>
<tr>
<td>3.19</td>
<td>0.025416</td>
<td>3.1391679</td>
<td>3.2408321</td>
</tr>
</tbody>
</table>

10.6 CASE STUDY 2: IML to implement a bisection root finder [OMIT]

The code executes until the interval is sufficiently narrow, here a convergence criterion of “hi-lo” < 10^-7 is set. The IML code in Display 10.11 generates a data set “process” containing the (lo, hi) pairs, i.e. the iteration history, which is then printed and plotted. We often encounter examples when we need to solve for roots for an equation, e.g. x such that f(x) = 0 when x is in some interval, say [a,b], where f(a) < 0 < f(b) or f(a) > 0 > f(b). The following example solves for the \( \sqrt{3} \) by finding the root of f(x) = x-\( \sqrt{3} \). (This example was used by my colleague Bob Noble to illustrate IML coding in his class, and I have included it here.) The lower limit “lo” and upper limit “hi” are specified to be 0 and 3, respectively. The bisection method finds the midpoint of the current interval, mid = (lo + hi)/2, and checks to see if mid^2 > 3, or equivalently, “mid^2” > 3. If it is, then our “guess” for the solution, mid, is too high and thus, we set “hi” to “mid” and the process begins again. If not, then “lo” is set to “mid” and the process begins again. (I know this is a lot of work to find a value that is one stroke on a calculator keypad; however, this gives you the flavor of the example and demonstrates the flexibility of IML coding.) This code also saves a matrix with the history of (lo, hi) intervals checked.

Display 10.11: Using the method of bisection to estimate \( \sqrt{3} \)
The solution to $x - \sqrt{3} = 0$ is $x = 1.732\ldots$, i.e. $\sqrt{3} = 1.732$, as we see in Display 10.12. This display also includes the iteration history of this bisection method. We see that the convergence criterion of $10^{-7}$ is achieved in iteration number 26.

Display 10.12: Iteration history from using the method of bisection to estimate $\sqrt{3}$

<table>
<thead>
<tr>
<th>Obs</th>
<th>ITERATION</th>
<th>LOW</th>
<th>HIGH</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0.00000</td>
<td>3.00000</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1.50000</td>
<td>3.00000</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1.50000</td>
<td>2.25000</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1.50000</td>
<td>1.87500</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>1.68750</td>
<td>1.87500</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>1.68750</td>
<td>1.78125</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>1.68750</td>
<td>1.73438</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>1.71094</td>
<td>1.73438</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>1.72266</td>
<td>1.73438</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>1.72852</td>
<td>1.73438</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>1.73145</td>
<td>1.73438</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>1.73145</td>
<td>1.73291</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>1.73145</td>
<td>1.73218</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
<td>1.73181</td>
<td>1.73218</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>1.73199</td>
<td>1.73218</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>1.73199</td>
<td>1.73209</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
<td>1.73204</td>
<td>1.73209</td>
</tr>
<tr>
<td>18</td>
<td>17</td>
<td>1.73204</td>
<td>1.73206</td>
</tr>
<tr>
<td>19</td>
<td>18</td>
<td>1.73204</td>
<td>1.73205</td>
</tr>
<tr>
<td>20</td>
<td>19</td>
<td>1.73205</td>
<td>1.73205</td>
</tr>
<tr>
<td>21</td>
<td>20</td>
<td>1.73205</td>
<td>1.73205</td>
</tr>
<tr>
<td>22</td>
<td>21</td>
<td>1.73205</td>
<td>1.73205</td>
</tr>
<tr>
<td>23</td>
<td>22</td>
<td>1.73205</td>
<td>1.73205</td>
</tr>
<tr>
<td>24</td>
<td>23</td>
<td>1.73205</td>
<td>1.73205</td>
</tr>
<tr>
<td>25</td>
<td>24</td>
<td>1.73205</td>
<td>1.73205</td>
</tr>
<tr>
<td>26</td>
<td>25</td>
<td>1.73205</td>
<td>1.73205</td>
</tr>
</tbody>
</table>
Display 10.13: Plot of the iteration history from using the method of bisection to estimate $\sqrt{3}$

10.7 CASE STUDY 3: Randomization test using matrices imported from PLAN [OMIT]

We revisit the two-group randomization test example from Chapter 8 in the following code. (You may want to look back at the Chapter 8 illustration to remind yourself about randomization test ideas and SAS code for generating permutations.) In this example, PROC PLAN is used to generate permutations of the indices 1:20 (i.e. the labels “1” to “n1+n2”) which are then imported into IML for calculation of the permutation test statistic. The actual data are imported into IML as the matrix nitro while the indices are contained in the matrix perm_index. The matrix perm.resp is the data in “nitro” rearranged according to the rows of indices in perm_index. The permuted value of the test statistic, the difference in sample means which is equivalent to the difference in sample sums since the two groups have equal sample sizes ($n_1=n_2=10$), and an indicator of whether this is larger than the observed value of the test statistic are stored in the two columns of the results matrix, PERM_RESULTS. The proportion of 1s in the second column of PERM_RESULTS defines the P-value here.

Display 10.14: IML code to implement randomization test

```
options nocenter nodate;
```
libname class 'folder-containing-nitrofen-data';

title "Randomization test in IML - Nitrofen conc 0 vs. 160 compared";
data test; set class.nitrofen;
  if conc=0 | conc=160;

proc plan;
  factors test=4000 ordered in=20;
  output out=d_permut;
run;

proc transpose data=d_permut prefix=in out=out_permut(keep=in1-in20); by test;
run;

proc iml;
/* read SAS data in IML */
use class.nitrofen;
read all var { total conc } where (conc=0|conc=160) into nitro;

/* read the indices for generating the permutations into IML */
use out_permut;
read all into perm_index;

obs_vec = nitro[,1];
obs_diff = sum(obs_vec[1:10]) - sum(obs_vec[11:20]);  * test statistic;

PERM_RESULTS = J(nrow(perm_index),2,0);  * initialize results matrix;

do iperm = 1 to nrow(perm_index);
  ind = perm_index[iperm,];                * extract permutation index;
  perm_resp = obs_vec[ind];                * select corresponding obs;
  perm_diff = sum(perm_resp[1:10]) - sum(perm_resp[11:20]);
  PERM_RESULTS[iperm,1] = perm_diff;       * store perm TS value/indicator;
  PERM_RESULTS[iperm,2] = abs(perm_diff) >= abs(obs_diff);
end;

perm_Pvalue = PERM_RESULTS[+,2]/nrow(PERM_RESULTS);
print 'Permutation P-value = ' perm_Pvalue;

Executing the code in Display 10.14 produces a Permutation P-value of 0.03575.

10.8 CASE STUDY 4: IML module to implement Monte Carlo integration to estimate $\pi$

Display 10.15: IML module to estimate $\pi$ using Monte Carlo integration

options nocenter nodate;
proc iml;
/* MODULE TO ESTIMATE PI
   - Monte Carlo simulation used
   - Strategy:
Generate $X \sim \text{Unif}(0,1)$ and $Y \sim \text{Unif}(0,1)$
Determine if $Y \leq \sqrt{1-X \times X}$

- INPUT
  - nsim
- OUTPUT
  - estimate of PI along with SE and CI

```
start MC_PI(nsim);
  temp_mat = J(nsim,2,0);
  call randgen(temp_mat,'uniform');
  temp_mat = temp_mat ||
              (temp_mat[,2]<=
               sqrt(J(nsim,1,1)-temp_mat[,1]*temp_mat[,1]));
  pi_over4 = temp_mat[+,3]/nsim;
  pi_est = 4*pi_over4;
  se_est = 4*sqrt(pi_over4*(1-pi_over4)/nsim);
  pi_LCL = pi_est - 2*se_est;
  pi_UCL = pi_est + 2*se_est;
  *-----------------------------------------------------------;
  print 'Estimating PI using MC simulation methods with' nsim 'data points';
  print pi_est se_est pi_LCL pi_UCL;
finish MC_PI;
```

```
runcat MC_PI(400);
runcat MC_PI(1600);
runcat MC_PI(4000);
quit;
```
For these three cases, the CI for $\pi$ does contain the true value of $\pi$ ($3.14159\ldots$). This will not always be the case. Given that we are constructing 95% confidence intervals, we would expect that if we ran MC_PI(4000) a large number of times, 5% of these runs would result in intervals that did not contain $\pi$.

Summary

In this chapter, basic matrix definitions and manipulations are introduced. Importing SAS data sets for processing in IML is introduced. Finally, IML solutions to a number of familiar problems are reviewed. This chapter is not intended to serve as a comprehensive introduction of matrix manipulation or IML. If you are interested in learning more about the functions that are built into IML, then you should dig into the SAS/IML documentation. In addition, there will be a number of cool SUGI contributions that you might be able to dig up in SUGI Proceedings.

References:
