Paper 2440-2015

Permit Me to Permute: A Basic Introduction to Permutation Tests with SAS/IML®

John Vickery, North Carolina State University

ABSTRACT

If your data do not meet the assumptions for a standard parametric test, you might want to consider using a permutation test. By randomly shuffling the data and recalculating a test statistic, a permutation test can calculate the probability of getting a value equal to or more extreme than an observed test statistic. With the power of matrices, vectors, functions, and user-defined modules, the SAS/IML® language is an excellent option. This paper covers two examples of permutation tests: one for paired data and another for repeated measures analysis of variance. For those new to SAS/IML® software, this paper offers a basic introduction and examples of how effective it can be.

INTRODUCTION

If your data do not meet the assumptions for a standard parametric test, you may want to consider using a permutation test. By randomly shuffling the data and recalculating a test statistic, a permutation test can calculate the probability of getting a value equal to or more extreme that an observed test statistic.

This paper shows two examples of permutation tests using SAS/IML. Both examples covered in this paper are variations of repeated measures. One example is a comparison of mean differences for paired data. The parametric equivalent would be the matched pair t-test. Also shown is an example using SAS/IML to do a permutation test equivalent to repeated measures ANOVA. As with the paired data example, this example permutes within the rows of a matrix and repeatedly calculates a test statistic (in this case, the F ratio). While there are many variations and applications of permutation/randomization tests, the examples shown here can serve as a good entry point to SAS/IML as they illustrate features of SAS/IML that are difficult to implement in BASE SAS.

This paper begins with a short overview of permutation tests. Next, using code from the complete programs in the last section, the paper covers some basic, important features of SAS/IML such as getting data into matrices and vectors. The paper then covers a few elements of SAS/IML that make permutation tests possible and efficient such as the RANPERM function and vectorization. The code in this section is also taken from the full example programs. The last section steps through the code line by line for each of the two examples.

While there is much more related to permutation tests than is covered here, this paper may provide programmers new to SAS/IML a way to get started.

DATA SET USED IN THE EXAMPLES

The data set used in the examples shown in this paper was derived from a project to compare the performance of three different online database products often used by academic libraries. The products were compared by testing a random sample of actual search query terms in each database and recording the number of relevant results returned within the first ten results. That is, each query (i.e. subject) was tested under three different conditions. The data set has 139 observations and data values for the three variables range from 0 to 10.

The following DATA step will re-create the data set used in each example:

```
data permtest;
   input prod1-prod3 @@;
datalines;
0 0 6       8 3 8         3 4 4         10 10 10    2 6 4       7 7 10
10 10 10    0 0 0         10 10 10      3 4 9       2 4 5       9 10 10
2 0 0       10 10 10      2 2 0         3 3 9       4 4 7       2 1 10
3 0 4       2 10 10       6 5 7         7 7 0       5 4 4       2 1 0
```

```
0 10 3      9 10 9      1 1 4       6 6 8       10 10 10    4 3 6
10 10 9     8 7 4       3 7 8       4 1 1       4 5 6       4 4 7
2 4 3       0 2 3       9 1 6       10 10 10    7 0 9       10 10 10
0 0 3       1 1 5       10 0 10     6 9 1       4 3 5       10 4 10
10 6 10     6 2 6       9 9 7       10 10 10    10 10 10    1 4 6
7 1 7       9 9 0       7 4 5       4 7 7       1 0 5       5 3 6
7 2 1       6 2 6       3 7 4       0 1 1       7 10 10     3 4 6
1 0 10      3 2 10      10 2 10     6 1 5       1 0 5       0 0 0
0 0 4       7 8 10      10 6 10     3 10 10     5 5 5       5 0 0
0 0 1       4 5 7       0 0 0       0 0 1       9 10 5      9 10 8
7 0 8       10 10 10    0 3 1       3 0 6       9 7 6       0 6 4
0 0 0       10 10 10    1 2 4       7 8 9       9 9 10      8 9 9
8 9 7       2 0 0       10 9 10     6 8 10      0 8 7       9 10 10
8 10 10     9 10 10     10 10 10    1 1 8       9 8 8       1 2 9
10 10 0     1 3 8       0 0 0       4 2 2       10 10 10    8 10 2
3 5 7       0 2 1       5 7 2       1 3 6       2 1 3       0 0 0
10 10 10    1 0 4       3 4 0       0 1 0       4 7 7       2 2 1
0 0 5       0 0 0       4 5 6       2 3 4       4 5 3       0 0 8
3 7 5       8 10 0      6 3 2       3 0 2       8 10 3      10 10 4
0 0 3
;
run;
```

THE BASIC IDEA BEHIND A PERMUTATION TEST

In *A Dictionary of Statistics*, Upton (2014) defines a permutation test as follows:

> A simple type of hypothesis test. Denote the value of some test statistic by *T*. The observed data values are randomly redistributed amongst the experimental units. The test statistic is calculated for each such redistribution. Depending on the number of data values, either all possible permutations are made, or a random selection (of say 1000 permutations) is made. For each permutation the value of the test statistic is considered. The significance of the value *T* is determined by the proportion of permutations that lead to values greater than, or equal to, *T*.

The concept of permutation tests was discussed by Fisher and others as early as the 1930's (Anderson 2001, Good 2005). As stated in the definition above, the basic outline of the permutation test is that you can exchange the labels of the data and repeatedly calculate a test statistic. By randomly shuffling the data and recalculating a test statistic, a permutation test can calculate the probability of getting a value equal to or more extreme than an observed test statistic. The permutation test allows you to build a distribution of the test statistic under the null hypotheses.

The practicality of the method, however, was limited by computing power until relatively recently. With even relatively small data sets, the number of possible permutations becomes very large. To illustrate, consider the data we will use for the two examples. The dataset has 139 observations. In the matched pair example, each query is tested under two products resulting in 2^{139} total possible permutations of the labels within pairs. That is, the first observation has two possible assignments which could each be paired with two possible assignments for the second observation and so on. For the repeated measures ANOVA example, three products are compared resulting in $(K!)^n$ possible permutations of the labels where K is the number of products or conditions under which the search queries were tested. For 139 observations with three products $(K!)^n$ results in $(3!)^{139} = 6^{139}$ possible permutations. In practice, you do not calculate the test statistic for every possible permutation but rather on the order of 10,000.

The examples covered in this paper are both versions of repeated measures. In sections 6.1 through 6.10 of *Randomization Tests*, Edgington and Onghena provide a detailed overview of using randomization tests to determine P-values in repeated measures (Edgington & Onghena 2007). Of course, permutation tests are not limited to repeated measures designs.

MATRICES, VECTORS, FUNCTIONS AND MODULES

This section covers some important, basic features of SAS/IML. We first look at matrices, vectors and how to get data into SAS/IML. Next, we look at a few select functions that are used in the two examples in this paper. Finally, we show how it is to define and call a user-defined module in SAS/IML.

SAS/IML differs from the DATA step in that the fundamental units are matrices and vectors as opposed to observations or rows. Also, the DATA step implicitly loops over all observations. SAS/IML usually does not.

Matrices are two dimensional arrays of either numeric or character values. Numeric matrices have double-precision values for the elements. Character matrices have equal length character string elements.

Matrix dimensions are defined by the number of rows and columns. You will often see a matrix dimensions expressed as *n x p*. This indicates that the matrix has *np* elements arranged with *n* rows and *p* columns. A 1x1 matrix is called a *scalar*. A 1 x *p* matrix is a *row vector*. And an *n* x 1 matrix is a *column vector*.

The examples in this paper use PROC IML as you may implement it from within the SAS Display Manager. However, as Wicklin points out, PROC IML does not work well with SAS Enterprise Guide (Wicklin 2013). Enterprise Guide automatically appends a QUIT statement each time a PROC IML statement is submitted. The procedure is terminated and any computed matrices are deleted. Readers are also encouraged to investigate the additional capabilities of IMLPlus and SAS/IML Studio. If you have a license to SAS/IML, you also have access to IMLPlus and SAS/IML Studio.

PROC IML is an interactive procedure similar to PROC SQL or PROC REG. That is, each statement is executed as soon as it is submitted. To exit PROC IML, use the QUIT statement. Note that in SAS/IML, the RUN statement is used to execute a subroutine or module so do not end IML programs with a RUN statement.

GETTING DATA INTO A MATRIX OR VECTOR

The first step in using SAS/IML is to get your data into a matrix or vector. SAS/IML has several ways in which to read data from a SAS dataset into a matrix. Data set variables can be read into column vectors. You can also create a matrix whose columns correspond to data set variables using all the observations in a data set or a subset of them. The SAS/IML User's Guide chapter, "Working with SAS Data Sets" offers more detail.

The USE and READ statements allow you to read data from SAS datasets into a matrix or vector. To read dataset variables into individual vectors, you can first create a character matrix containing the names of the variables you need.

Here two variables from the SAS data set *PERMTEST* described above are read into two vectors:

```
proc iml;                              /* initiate SAS/IML */
❶  varnames = {'prod1' 'prod2'};       /* matrix of names of variables to read */
❷  use permtest;                       /* open SAS dataset for reading */
❸  read all var varnames;              /* read all obs from variables */
❹  close permtest;                     /* closes the dataset */
… more code below …
```

In this snippet:

1. VARNAMES is a character vector that contains the variable names to be read in from the SAS dataset
2. The USE statement opens the *PERMTEST* data set
3. The READ statement with the ALL keyword will read in all observations from the PROD1 and PROD2 variables

4. The CLOSE statement closes the SAS dataset. While it is not required, it is good programming practice to close datasets when you are done.

As is almost always the case, there are many ways to accomplish a task in SAS. For example, an alternative to using a character vector to hold the names of the variables to read would be to use a literal matrix of names in the READ statement:

```
read all var {prod1 prod2};
```

Likewise, instead of reading variables into two vectors, you can use the INTO clause to read the variables into columns of a matrix:

```
read all var {prod1 prod2} into x;
```

Or even:

```
read all var into x;
```

SUBSCRIPT REDUCTION OPERATORS

Reduction operators perform statistical operations such as sum, mean and sum of squares that return a smaller dimension matrix. A feature of SAS/IML is that by using these subscript reduction operators you can avoid writing unnecessary loops. The examples in this paper use the MEAN and ADDITION reduction operators. Subscript reduction operators are enclosed in brackets. The MEAN operator is written as a colon [:]. The ADDITION operator is written as a plus symbol [+]. Table 1 below lists each of the subscript reduction operators.

Table 1.

Operator	Description
+	Addition
#	Multiplication
<>	Maximum
><	Minimum
<:>	Index of maximum
>:<	Index of minimum
:	Mean
##	Sum of squares

Table 1. Subscript Reduction Operators

The code snippet above read two variables from the *PERMTEST* data set into two vectors. As part the matched pairs example, we need to compute the observed mean of the differences between the two products. The lines of code below create a vector of the differences and then use the MEAN subscript operator to calculate the mean of the observed differences:

```
Prod12 = prod1-prod2;      /* vector of differences */
ObsDiff = Prod12[:];       /* observed mean of differences */
```

Note that ObsDiff is a scalar quantity. That is, the MEAN operator [:] has *reduced* the vector Prod12.

J, LOC AND RANPERM FUNCTIONS

SAS/IML contains over 300 built in functions and subroutines. You also have access to hundreds of BASE SAS functions that can be called from SA/IML programs. Three functions in particular are important to the example programs in this paper: the J, LOC and RANPERM functions. Other functions are used in the example programs but these deserve a closer look.

The J function is considered a *Matrix Reshaping Function*. The J function creates a matrix of identical values. The form of the function is:

```
J(nrow <, ncol> <, value> );
```

The J function creates a matrix with *nrow* rows and *ncol* columns with all elements equal to value. An important use of the J function is to allocate a matrix of a desired size prior to initiating a DO loop. Statements within the loop fill the new matrix. This is opposed to concatenating results within the loop itself. It is much more efficient and helps to vectorize your program.

The LOC function is considered a *Matrix Inquiry Function*. This type of function provides information about the data in a matrix. The LOC function returns indices of a matrix that satisfy a given condition. The form of the LOC function is simply:

```
LOC(matrix);
```

Instead of looping over observations to find those that satisfy a condition, the LOC function locates the nonzero elements in a matrix or vector. The LOC function returns a row vector containing the indices that satisfy the given condition. With these indices, you can then subset the data.

Permuting values within the rows of a matrix with the LOC function

The following snippet from the matched pair example program shows the J function being used prior to a DO loop as well as the LOC function being used to locate rows to permute. Wicklin explained this method in his 2010 SAS Global Forum paper (Wicklin 2010). The RANPERM function can also be used to permute values within the rows of a matrix. However, the LOC function is particularly useful for many applications and deserves the attention of new SAS/IML programmers.

```
… more code above …
❶  u = j(nrow(y),1);              /* allocate vector to hold random numbers */
   B = 10000;                     /* number of bootstrap resamples */
❷  s = j(B,1);                    /* allocate vector to hold bootstrap dist */
 do i = 1 to B;
  ❸  call randgen(u, "Uniform");
     z = y;                          /* copy original data */
  ❹  rows = loc(u > 0.5);            /* locate rows for which u[i] > 0.5 */
  ❺  z[rows,] = z[rows,{2 1}];       /* swap values of these rows */
     x = z[,1] - z[,2];              /* difference of permuted data */
     s[i] = x[:];                    /* mean of difference */
 end;
… more code below …
```

In this snippet:

1. The vector *u* is allocated using the NROW function to determine the number of rows in the y matrix. The second argument of 1 specifies a single column. Note that it is allocated *prior* to the DO loop.
2. The vector *s* is allocated in the same manner as the vector u above.
3. The RANDGEN call fills the vector *u* with random values from the uniform distribution.
4. The LOC function finds the rows within the vector *u* that are greater than 0.5. These rows are stored in the vector ROWS and used as indices for which elements to exchange.
5. This is where the values are exchanged. The vector ROWS used as the first index to matrix Z specifies those rows identified in #4. The `{2 1}` as the second index specifies the value in the second column is exchanged with the value in the first column.

Permuting values within the rows of a matrix with the RANPERM function

Note that in the method used above to swap the values within each row could be also be accomplished by using the RANPERM function. In the example snippet that follows, the RANPERM function is used to permute the elements within the rows of the three column matrix from the repeated measures ANOVA example.

The RANPERM function returns random permutations of a set of *n* elements. The form of the function is:

```
RANPERM (set, <, numperm> ) ;
```

The first argument can be a scalar or vector. A scalar argument will return indices in the range of 1-*n*. For a vector argument, the number of elements in the vector determines *n.*

The default is for the RANPERM function to return a single random combination. If the optional second <numperm> argument is used, then the function will return a matrix with *numperm* rows and *n* columns where each row of the returned matrix is a single permutation.

This code block from the repeated measures ANOVA program, shows the RANPERM function being used to permute the values within the rows of a matrix:

```
… more code above …
  call randseed(1234);
❶ x = j(nrow(xobt),ncol(xobt));     /* allocate matrix for permuted data */
  n = nrow(x);                      /* number of rows to permute */
  B = 10000;                        /* number or reps */
❷ fdist = j(B,1);                   /* vector to hold bootstrap dist. of F */

❸ do j = 1 to B;                    /* bootstrap loop */
❹    p = xobt;
     /* permute each row */
❺    do i = 1 to n;
❻       q = ranperm(p[i,]);         /* random permutation of row i */
❼       x[i,] = q;
     end;
     F = fmod(x);                   /* calculate F on permuted data */
     fdist[j,] = F;                 /* store calculated F in fdist vector */
  end;
… more code below …
```

In this snippet:

1. Matrix *x* is allocated prior to the bootstrap loop. The NROW and NCOL functions to specify the same dimensions as the *xobt* matrix.
2. The vector *fdist* is also allocated prior to the bootstrap loop.
3. The outer loop is the bootstrap loop
4. The original matrix *xobt* is copied to the matrix *p*.
5. The inner loop loops over each row of the matrix. Note that this loop can be eliminated by permuting the column indices rather than the elements of the matrix. This is covered in the following section on vectorization.
6. The row vector *q* is assigned a random permutation of row *i*.
7. These values are then added to row *i* of the pre-allocated matrix *x*.

DEFINING AND CALLING A MODULE

Another benefit of SAS/IML is the ability to define your own function or subroutine module. These modules can be called from within the same program that it was defined. Or they can be stored and called from other programs. Modules that return a value are *function modules*. A module that does not return a value is a *subroutine module.*

Modules begin with the START statement and end with the FINISH statement. To invoke a function, its name is used in an assignment statement. A subroutine is executed by issuing a RUN or CALL statement.

The snippet below shows the basic form of a function module. This module calculates and returns the F statistic value for the repeated measures ANOVA example:

```
… more code above …
❶ start fmod(x);
      … more code …
      F = MSt/MSerror;
❷    return(F);
❸ finish;
… more code below …
```

In this snippet:

1. The START statement begins the module definition. Here, the module is named *fmod* and the argument to the module is listed in parentheses as (x). The argument to this module is an input variable. Arguments can also be output or returned variables.
2. The RETURN statement here includes an operand in parentheses to return a value. The operand can be a variable name or expression. Here, it is a variable name *F*.
3. The FINISH statement signals the end of the module definition. You can optionally include the name of the defined module in the FINISH statement such as FINISH *fmod.*

VECTORIZE BY CONVERTING MATRIX SUBSCRIPTS TO INDICES

Vectorization is an important concept to grasp for new SAS/IML programmers. If you have been programming in BASE SAS, you are aware that the basic unit is the observation and that the DATA step is an implied loop. In SAS/IML, the basic unit is a matrix or vector and there is no implied loop. The idea behind vectorization is to reduce the complexity of a program to a minimal number of executable statements. By operating on the matrix and vector level, you can perform operations in SAS/IML that would typically require looping over observations in BASE SAS. To vectorize a program, think about avoiding loops wherever possible.

In the example snippets shown above, you have already seen simple examples of vectorization in the form of subscript reduction operators. For example, rather than looping over observations to calculate the grand mean for of a matrix (as the implicit loop in a DATA STEP does), you can use the colon [:] subscript reduction operator to calculate the overall (grand) mean of a matrix X as follows:

```
grandmean = x[:];
```

The code shown above in the section about the RANPERM function can be vectorized to permute the column indices rather than permuting the elements of the matrix. You can then convert the column indices into matrix indices. In order to convert the subscripts to indices, we use the following formula.

For a matrix with *p* number of columns, the index of the element that is located in row *i* and column *j* is:

j + p*(i-1).

The terminology and calculations are explained in detail by Wicklin in a Feb. 2011 post on The DO Loop blog, “Converting matrix subscripts to indices” (Wicklin 2011).

In order to vectorize the within rows permutation of *k* groups, Wicklin proposed the following solution as a function named *PermuteWithinRows* (Wicklin 2014). This function can then be called from within the bootstrap loop of the program. Rather than looping over the rows of the matrix and then permuting the elements, this function operates on the matrix as a whole. That is, the operation is *vectorized.* As a new SAS/IML programmer, this is possibly one of the most important concepts to practice. Using the PRINT statement to print the column and matrix indices step by step can help visualize how vectorization works.

By using this function, it is possible to eliminate the inner loop used above in the section “Permuting values within the rows of a matrix with the RANPERM function”:

```
… more code above …
❶ start PermuteWithinRows(m);
❷   colIdx = ranperm(1:ncol(m), nrow(m));    /* permute column indices */
❸   f = (row(m) - 1) * ncol(m);
❹    matIdx = f + colIdx;                    /* generate matrix indices */
❺ return( shape(m[matIdx], nrow(m)) );
finish;
… more code below …
```

In this snippet:

1. The module is named *PermuteWithinRows* and the argument to the module is listed in parenthesis.
2. *colIdx* is a matrix of random permutations of the column indices {1 2 3}. The second argument to the RANPERM function specifies that the number of permutations to return is equal to the number of rows in the input matrix *m*. *colIdx* is the equivalent of *j* from the formula above.
3. *f* is the equivalent of the *p*(i-1)* from the formula above. The ROW function function returns a matrix that has the same dimensions as the x matrix and whose *i*th row has the value *i*.
4. *matIdx* is the matrix of matrix indices. It can be helpful to use the PRINT statement to see these operations step by step.
5. The SHAPE function reshapes the input matrix *m* according to the matrix indices and its number of rows and the RETURN statement returns this permuted matrix.

LINE-BY-LINE THROUGH THE CODE

This section puts together the pieces discussed above and looks at the code line by line. Shown first is the code for a permutation test to compare the means of paired data. Two permutation methods are shown in the matched-pairs example. The first method uses the LOC function. The second method uses the PERMITWITHINROWS module.

The second example is a program for a permutation test for repeated measures analysis of variance. This program also uses the *PERMITWIITHINROWS* function module to avoid looping over rows of the matrix and permuting matrix elements.

EXAMPLE 1: MATCHED PAIRS

```
1. proc iml;
2.    varnames = {'prod1' 'prod2'};
3.    use permtest;
4.    read all var varnames;
5.    close permtest;

6.    prod12 = prod1-prod2;

7.    ObsDiff = prod12[:];
8.    call symputx('obsdiff',ObsDiff);

9.    call randseed(1234);          /* initialize seed for random numbers */
10.   y = prod1 || prod2;           /* concatenate values into n x 2 matrix */

/* METHOD #1 - USING LOC FUNCTION */
11.   u = j(nrow(y),1);             /* allocate vector to hold random numbers */
12.   B = 10000;                    /* number of bootstrap resamples */
13.   s = j(B,1);                   /* allocate vector to hold bootstrap dist */
14.   do i = 1 to B;
15.      call randgen(u, "Uniform");
```

```
16.       z = y;                          /* copy original data */
17.       rows = loc(u > 0.5);            /* locate rows for which u[i] > 0.5 */
18.       z[rows,] = z[rows,{2 1}];       /* swap values of these rows */
19.       x = z[,1] - z[,2];              /* difference of permuted data */
20.       s[i] = x[:];                    /* mean of difference */
21.    end;

/* compute empirical two-sided p-value */
22.    pval = sum(s > abs(ObsDiff)) / B + sum(s < -abs(ObsDiff)) / B;
23.    print pval[label='PROD1-PROD2 P Value - USING LOC FUNCTION'];

/* METHOD #2 - USING RANPERM FUNCTION AND PERMUTEWITHINROWS MODULE */
/* module to independently permute elements of each row of a matrix */
24.    start PermuteWithinRows(m);
25.       colIdx = ranperm(1:ncol(m), nrow(m));
26.       f = (row(m) - 1) * ncol(m);
27.       matIdx = f + colIdx;
28.       return( shape(m[matIdx], nrow(m)) );
29.    finish;

30.    B = 10000;
31.    mpdist = j(B,1);
32.    do i = 1 to B;
33.       x = PermuteWithinRows(y);
34.       x = x[,1] - x[,2];
35.       mpdist[i,] = x[:];
36.    end;

37.    create bootmp var {mpdist};    /* create dataset from mp vector*/
38.    append;
39.    close bootmp;

/* compute empirical two-sided p-value */
40.    pval = sum(mpdist>abs(ObsDiff)) / B + sum(mpdist<-abs(ObsDiff)) / B;
41.    print pval[label='PROD1-PROD2 P Value - USING RANPERM AND MODULE'];
42.    call symputx('prod12p',pval);

43. quit;
```

1. Start the IML procedure
2. *VARNAMES* is a character matrix of the variables you want to read from a SAS data set
3. The USE statement opens the SAS data set *PERMTEST*
4. The READ statement reads observations from the data set. The keyword ALL specifies that all observations are read. The VAR clause specifies which variables to read. Here, the variables to read are specified by the *VARNAMES* matrix assigned in line #2. *PROD1* and *PROD2* are read in as column vectors.
5. The CLOSE statement closes the SAS data set. While this is not required, it is good practice.
6. *PROD12* is a column vector of the differences between *PROD1* and *PROD2*.
7. The scalar value *OBSDIFF* is the mean of the PROD12 vector. It is created by using the mean subscript reduction operator `[:]` with *PROD12*.

8. The CALL SYMPUTX routine assigns *OBSDIFF* to a macro variable. Note that you can call most Base SAS functions from PROC IML. While not shown here, you could use this as part of a SGPLOT procedure to graph the null distribution with a reference line for the observed mean of the difference.
9. Initialize the random number seed
10. Concatenate the *PROD1* and *PROD2* vectors into a matrix, *Y*, using the `||` concatenation operator. The matrix *Y* will have two columns and the same number of rows as the *PROD1* and *PROD2* vectors.
11. Allocate the *U* vector to hold random numbers by calling the J function. The first argument specifies the number of rows. Here, NROW function is called to return the number of rows in the matrix, *Y.* The second argument specifies a single column. Note that you should allocate vectors and matrices prior to filling them in a loop.
12. *B* is a scalar value for the number of bootstrap resamples.
13. Similar to line #11, allocate the vector *S* to hold the results of the bootstrap loop.
14. Start the bootstrap loop.
15. CALL RANDGEN fills the matrix, *U* (allocated in line #11), with random numbers from the uniform distribution.
16. The matrix, *Z*, is a copy of the original data from line #10.
17. The LOC function finds the rows within the vector *U* that are greater than 0.5. These rows are stored in the vector ROWS and used as indices for which elements to exchange.
18. This is where the values are exchanged. The vector ROWS used as the first index to matrix Z specifies those rows identified in #4. The `{2 1}` as the second index specifies the value in the second column is exchanged with the value in the first column.
19. *X* is a column vector of the permuted data. Note the use of subscripts: `z[,1]` means all rows and the first column of *Z*. `z[,2]` means all rows and the *second* column of *Z*.
20. Row *i* of the vector *S* (allocated in line #13) is filled with the mean of *X.*
21. End of the bootstrap loop.
22. Compute the empirical P-value. The `>` and `<` comparison operators return a matrix of ones and zeros. If element comparisons resolve to true, then ones are returned. Otherwise, zeros are returned. Here, the scalar value *OBSDIFF* is compared to each element of the matrix *S* and the sum function totals the ones.
23. The PRINT statement prints *PVAL* using the LABEL option.
24. The START statement begins the user-defined *PERMITWITHINROWS* module with matrix *M* as input.
25. *COLIDX* is a matrix of random permutations of the column indices {1 2 3}. The second argument to the RANPERM function specifies that the number of permutations to return is equal to the number of rows in the input matrix *M*.
26. *F* is the equivalent of the *p*(i-1)* from the formula in the previous section. The ROW function function returns a matrix that has the same dimensions as the x matrix and whose *i*th row has the value *i*.
27. *MATIDX* is the matrix of matrix indices.
28. The SHAPE function reshapes the input matrix *M* according to the matrix indices and its number of rows and the RETURN statement returns this permuted matrix.
29. The FINISH statement ends the *PERMUTEWITHINROWS* definition.
30. *B* is a scalar value for the number of bootstrap resamples.

31. Allocate the vector *MPDIST* to hold the results of the bootstrap loop.
32. Start the bootstrap loop.
33. The matrix *X* is assigned as the returned result from the *PERMUTEWITHINROWS* module. Note that the matrix *Y* (from line #10) is the input for the module.
34. Same as line #19
35. Same as line #20 except that the mean differences are stored in the *MPDIST* vector.
36. End of the bootstrap loop.
37. The CREATE statement with the VAR clause creates the SAS data set *BOOTMP* from the *MPDIST* matrix. This data set can then be used in a subsequent PROC or DATA step. For example, a PROC SGPLOT program could use this data set to plot a histogram of the null distribution. You could also create a data set from the vector *S* from line #20.
38. The APPEND statement adds observations to the output data set. It is usually used without any arguments.
39. CLOSE statement closes the *BOOTMP* data set.
40. Same as line #22.
41. Same as lines #23.
42. As in line #8, assign the value of *PVAL* to a macro variable for use in graphs, etc.
43. All done! Note that PROC IML is an *interactive procedure* so each statement is executed when it is submitted. Since the RUN statement in PROC IML is used to execute a module or subroutine, don't use the RUN statement as the last statement. QUIT is all you need.

EXAMPLE 2: REPEATED MEASURES ANOVA

```
1. proc iml;
2.    use permtest;
3.    read all var{prod1 prod2 prod3} into xobt;
4.    close permtest;
/* module to calculate F ratio */
5.    start fmod(x);
6.       gmean = x[:];                          /* grand mean scalar */
7.       n = nrow(x);                           /* number of rows/subjects */
8.       k = ncol(x);                           /* number of cols. */
9.       tmeans = x[:,];                        /* between means */
10.      submeans = x[,:];                      /* subject means */
         /* SS between */
11.      SSb = (tmeans-gmean)##2;
12.      SSb = n*(SSb[+]);
         /* SS within */
13.      SSw = (tmeans-x)##2;
14.      SSw = SSw[+,];
15.      SSw = SSw[+];
         /* SS subjects */
16.      SSsub = k*((submeans-gmean)##2);
17.      SSsub = SSsub[+];
18.      SSerror = SSw-SSsub;                    /* SSerror */
19.      MSt = SSb/(k-1);                        /* MSt */
20.      MSerror = SSerror/((n-1)*(k-1));        /* MSerror */
21.      F = MSt/MSerror;                        /* F statistic */
22.      return(F);
```

```
23.    finish;

/* module to independently permute elements of each row of a matrix */
24.    start PermuteWithinRows(m);
25.       colIdx = ranperm(1:ncol(m), nrow(m));   /* permute col. indices */
26.       f = (row(m) - 1) * ncol(m);             /* constant for row */
27.       matIdx = f + colIdx;                    /* matrix indices */
28.       return( shape(m[matIdx], nrow(m)) );
29.    finish;

30.    fobt = fmod(xobt);           /* calculate F for observed data */
31.    print fobt;
32.    call symputx('fobt',fobt);

33.    call randseed(12345);
34.    B = 10000;             /* number or reps */
35.    fdist = j(B,1);        /* allocate vector to hold bootstrap dist of F */
36.    do j = 1 to B;                    /* bootstrap loop */
37.       x = PermuteWithinRows(xobt);   /* call PermuteWithinRows module */
38.       F = fmod(x);                   /* calculate F on permuted data */
39.       fdist[j,] = F;                 /* store calc'd F in fdist vector */
40.    end;

/* compute p-value */
41.    pval = sum(fdist > abs(fobt)) / B;
42.    print pval[label='P-value'];
43.    call symputx('p',pval);

/* create dataset */
44.    create bootf var {fdist};
45.    append;
46.    close bootf;
47.  quit;
```

1. Start the IML procedure
2. The USE statement opens the SAS data set *PERMTEST*
3. The READ statement reads observations from the data set. The keyword ALL specifies that all observations are read. The VAR clause specifies which variables to read. Here, the variables to read are specified by a literal matrix of names. The INTO clause is used to read the variables into columns of a matrix (named *XOBT* here).
4. The CLOSE statement closes the SAS data set. While this is not required, it is good practice.
5. The START statement begins the module definition. Here, the module is named *fmod* and the argument to the module is listed in parentheses as (x). The argument to this module is an input variable. This function is used to calculate the F ratio for repeated measures ANOVA.
6. *GMEAN* is the overall mean of the input matrix. Note the mean subscript reduction operator `[:]`
7. The NROW function returns the number of rows in the matrix argument.
8. The NCOL function returns the number of columns in the matrix argument.
9. *TMEANS* is a 1 x p row vector (where p is the number of columns) of the means of each column in the input matrix. Omitting the second subscript specifies that the operator apply to all columns.
10. *SUBMEANS* is an n x 1 (where n is the number of rows) column vector of the means of each row in the input matrix. Omitting the first subscript specifies that the operator apply to all rows.

11. The elementwise power operator ## is used to square the difference of *TMEANS* and the overall mean.
12. The sum of *SSB* (note the addition subscript reduction operator `[+]`) is multiplied by *N* from line #7.
13. The elementwise power operator is used to square the difference of X and TMEANS.
14. Same as line #9 but the addition subscript operator is used rather than the mean.
15. Overall sum from line #14.
16. Essentially the same as lines #11 and #13.
17. Same as line #15.
18. The subtraction matrix operator is used to subtract SSsub from SSw.
19. The mean sum of squares for the products/conditions.
20. Mean sum of squares for error.
21. The F ratio!
22. The RETURN statement here includes an operand in parentheses to return a value. The operand can be a variable name or expression. Here, it is the F ratio from line #21 named *F*.
23. The FINISH statement ends the *FMOD* definition.
24. The START statement begins the user-defined *PERMITWITHINROWS* module with matrix *M* as input.
25. *COLIDX* is a matrix of random permutations of the column indices {1 2 3}. The second argument to the RANPERM function specifies that the number of permutations to return is equal to the number of rows in the input matrix *M*.
26. *F* is the equivalent of the *p*(i-1)* from the formula from the previous section. The ROW function function returns a matrix that has the same dimensions as the x matrix and whose *i*th row has the value *i*.
27. *MATIDX* is matrix of indices for the input matrix where each row is a random permutation.
28. The SHAPE function reshapes the input matrix *M* according to the matrix indices and its number of rows and the RETURN statement returns this permuted matrix.
29. The FINISH statement ends the *PERMUTEWITHINROWS* definition.
30. Calculate the F statistic for the observed data by passing the *XOBT* matrix to the *FMOD* function defined above.
31. PRINT the observed F statistic.
32. Assign the value of *FOBT* to a macro variable using CALL SYMPUTX for use in a subsequent PROC or DATA step.
33. Initialize the random number seed
34. *B* is a scalar value for the number of bootstrap resamples.
35. Allocate the vector *FDIST* to hold the results of the bootstrap loop. Remember, allocate prior to a DO loop.
36. Start the bootstrap loop.
37. The matrix *X* is assigned as the returned result from the *PERMUTEWITHINROWS* module. Note that the matrix *XOBT* (from line #3) is the input for the module.
38. Calculate the F statistic for the permuted data by passing the *X* matrix to the *FMOD* function defined above.

39. Store the calculated F in row *j* of the *FDIST* vector.
40. End of the bootstrap loop.
41. Compute the empirical P-value. The > comparison operator returns a matrix of ones and zeros. If element comparisons resolve to true, then ones are returned. Otherwise, zeros are returned. Here, the scalar value *FOBT* is compared to each element of the matrix *FDIST* and the sum function totals the ones.
42. The PRINT statement prints *PVAL* using the LABEL option.
43. The CALL SYMPUTX routine assigns *PVAL* to a macro variable. Note that you can call most Base SAS functions from PROC IML
44. The CREATE statement with the VAR clause creates the SAS data set *BOOTF* from the *fdist* matrix.
45. The APPEND statement adds observations to the output data set. It is usually used without any arguments.
46. CLOSE statement closes the *BOOTMP* data set.
47. All done! Note that PROC IML is an *interactive procedure* so each statement is executed when it is submitted. Since the RUN statement in PROC IML is used to execute a module or subroutine, don't use the RUN statement as the last statement. QUIT is all you need.

CONCLUSION

SAS/IML enables you to easily perform permutation tests that may be cumbersome to perform using BASE SAS. This paper, geared towards programmers new to SAS/IML, explored two examples of permutation tests using SAS/IML. One example showed a matched-pair permutation. The second example used SAS/IML to perform a permutation test equivalent to repeated measures ANOVA. If your data do not meet the assumptions for a standard parametric test a permutation test may be an option. By adding SAS/IML to your toolkit you can efficiently perform permutation tests.

REFERENCES

Anderson, Marti J. 2001. "Permutation tests for univariate or multivariate analysis of variance." Canadian Journal of Fisheries and Aquatic Sciences, 58: 626-639. DOI: 10.1139/cjfas-58-3-626

Edgington, E. S., & Onghena, P. 2007. *Randomization Tests*. Boca Raton: Chapman & Hall / CRC.

Good, P. 2005. *Permutation, Paramentric, and Bootstrap Tests of Hypothesis.* New York: Springer.

Upton, G., & Cook, I. 2014. permutation test. In A Dictionary of Statistics. : Oxford University Press. Accessed 2 Mar. 2015, http://www.oxfordreference.com/view/10.1093/acref/9780199679188.001.0001/acref-9780199679188-e-2108.

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

Wicklin, R. 2010. "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. http://support.sas.com/resources/papers/proceedings10/329-2010.pdf

Wicklin, R. 2011. "Converting matrix subscripts to indices." *The DO Loop Blog*. Feb. 16, 2011. Accessed 2 Mar. 2015, http://blogs.sas.com/content/iml/2011/02/16/converting-matrix-subscripts-to-indices/

Wicklin, R. 2013. "Getting Started with the SAS/IML® Language," in Proceedings of the SAS Global Forum 2013 Conference, Cary, NC: SAS Institute Inc. http://support.sas.com/resources/papers/proceedings13/144-2013.pdf

Wicklin, R. 2014. "Permute elements within each row of a matrix." *The DO Loop Blog*. May 29, 2014. Accessed 2 Mar. 2015, http://blogs.sas.com/content/iml/2014/05/29/permute-elements-within-each-row-of-a-matrix/

ACKNOWLEDGMENTS

Thanks to Joy Smith at North Carolina State University for excellent SAS classes.

RECOMMENDED READING

- *The DO Loop blog - http://blogs.sas.com/content/iml/*
- *SAS/IML® Users Guide*
- Wicklin, Rick. 2010. *Statistical Programming with SAS/IML® Software. Cary, NC: SAS Institute Inc.*

CONTACT INFORMATION

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

Name:	John Vickery
Organization:	NCSU Libraries, North Carolina State University
Address:	Box 7111
City, State ZIP:	Raleigh, NC 27695
Phone:	(919) 513-0344
Email:	john_vickery@ncsu.edu