Permutation Tests (and Sampling Without Replacement) Orders of Magnitude Faster Using SAS®

John Douglas ("J.D.") Opdyke,* DataMineIt

Abstract

Six permutation test algorithms coded in SAS® are compared. The fastest ("OPDN"), which uses no modules beyond Base SAS®, achieves speed increases orders of magnitude faster than the relevant "built-in" SAS® procedures (over 215x faster than Proc SurveySelect, over 350x faster than NPAR1WAY (which crashes on datasets less than a tenth the size OPDN can handle), and over 720x faster than Proc Multtest). OPDN combines SAS®'s fast sequential access, its most efficient array manipulation and memory usage, and a simple draw-by-draw procedure to very quickly and efficiently perform simple random sampling without replacement (SRSWOR). The particular draw-by-draw method used allows for the repeated creation of many SRSWOR permutation samples without requiring any additional storage or memory space. Based on these results, there appear to be no faster or more scalable permutation test or SRSWOR algorithms in SAS®.

Keywords: Permutation, SAS, Scalable, Without Replacement, Sampling, SRSWOR
JEL Classifications: C12, C14, C15, C63, C88 Mathematics Subject Classification: 62G09, 62G10

Introduction

Permutation tests are as old as modern statistics (see Fisher, 1935), and their statistical properties are well understood and well documented in the literature (see Mielke & Berry, 2001, and Pesarin, 2001, for comprehensive treatments and extensive bibliographies). As nonparametric hypothesis tests that do not make restrictive, and often unrealistic, parametric (distributional) assumptions about the data being tested,[1] they also are very widely used and remain the most appropriate statistic of choice in a very wide range of scientific and industry settings. Their only potential drawback lies in the fact that, as a data-intensive resampling method, they can be runtime prohibitive, especially when applied to large or even medium-sized data samples drawn from large datasets. The data explosion over the past few decades has made this a common occurrence, and it highlights the increasing need for faster, and more efficient and more scalable, permutation test algorithms.

* J.D. Opdyke is Managing Director of Quantitative Strategies at DataMineIt, a consultancy specializing in applied statistical, econometric, and algorithmic solutions for the financial and consulting sectors. Clients include multiple Fortune 50 banks and credit card companies, big 4 and economic consulting firms, venture capital firms, and large marketing and advertising firms. J.D. has been a SAS® user for over 20 years and routinely writes SAS® code faster (often orders of magnitude faster) than SAS® Procs (including but not limited to Proc Logistic, Proc MultTest, Proc Summary, Proc NPAR1WAY, Proc Plan, and Proc SurveySelect). He earned his undergraduate degree from Yale University, his graduate degree from Harvard University where he was a Kennedy Fellow, and has completed additional post-graduate work as an ASP Fellow in the graduate mathematics department at MIT. Additional of his peer reviewed publications spanning number theory/combinatorics, statistical finance, statistical computation, applied econometrics, and hypothesis testing for statistical quality control can be accessed at www.DataMineIt.com.

[1] The only assumption made by a permutation test is that the subscripts of the data samples are exchangeable under the null hypothesis.

During this same time period, SAS® has grown to become one of the most widely used statistical computing platforms globally.[2] The footprint of researchers and analysts using SAS® to perform permutation tests, therefore, is considerable. This, and its reputation for speed, make SAS® a very good choice as a platform for developing fast, scalable permutation test algorithms, and probing and expanding the limits on the speed with which they can be executed on large datasets. Key to this effort, of course, is developing scalable, fast, simple random sampling without replacement (SRSWOR), which serves as the core of the permutation test approach to hypothesis testing.

This paper compares six permutation test algorithms coded in SAS®, three of which (OPDN, OPDN-Alt, and Bebb-Sim) require only the Base SAS® module, and three of which (PROCSS, PROCMT, and PROCNPAR) rely on SAS® Procedures available as part of the SAS/STAT® module. All implement the conventional sampling-without-replacement permutation test, where the null hypothesis is the equality of the two population distributions represented by the two samples, and the permuted statistic is the sample sum.[3] However, four of the algorithms (OPDN, OPDN-Alt, Bebb-Sim, and PROCSS) easily can be modified to permute any statistic, and the remaining two (PROCNPAR, which uses Proc NPAR1WAY, and PROCMT, which uses Proc Multtest) have a finite number of choices for permutation test statistics.

Background:
When is O(N) better than O(n) (even when $N>>n$)? Exploiting the Fast Sequential Access of SAS®

Key to this paper is the fact that SAS® is not a "matrix" language (like MATLAB or Gauss) or a "vector" language (like S-Plus or R), but rather, with a few exceptions, it processes data sequentially, record-by-record. Since the late 1970's SAS® has become extremely fast and efficient at such sequential record processing,[4] and naturally, this strongly shapes the algorithms presented herein. It also leads to a bit of a paradox when comparing the time complexity of these algorithms: the real runtime ranking of these algorithms, when implemented in SAS®, can deviate notably from a ranking based on their theoretical time complexity. In other words, due to SAS®'s fast sequential access, O(N) algorithms are often "better" in SAS® than O(n) algorithms, even when $N>>n$. While this is important to note, and an example is explained in the paper, the focus of this paper is real runtimes, and the speed with which SAS® users can obtain actual results. Resource constraints such as I/O speeds, storage space, and memory are discussed below, and CPU runtimes are presented alongside real runtimes in the Results section, but the goal here is not to develop or compare algorithms based on their theoretical time complexity (although the time complexity of OPDN is presented, based on its empirical runtimes).

---

[2] See www.SAS.com. With over 45,000 registered user sites, SAS® is arguably the most widely used statistical platform globally, even without including users of the many SAS® "clones" that exist as well (see WPS (from World Programming – see http://teamwpc.co.uk/home), Dap (see http://www.gnu.org/software/dap/ and http://en.wikipedia.org/wiki/DAP_(software)), and arguably Carolina (see DullesOpen.com, a unit of Dulles Research LLC), and formerly, PRODAS (from Conceptual Software) and BASS (from Bass Institute)).

[3] Proc Multtest, as used in PROCMT, uses pooled-variance t-statistics, which provide a permutation p-value mathematically identical to that provided by sample sums.

[4] To quote the author of a previous SAS® bootstrap paper that extols the virtues of PROC SurveySelect, "Tools like bootstrapping and simulation are very useful, and will run very quickly in SAS .. if we just write them in an efficient manner." (Cassell, 2007).

The Six Algorithms

The six algorithms include: One Pass, Duplicates-No ("OPDN"), One Pass, Duplicates-No - Alternate ("OPDN-Alt"), PROCSS (Proc SurveySelect), PROCMT (Proc Multtest), PROCNPAR (Proc NPAR1WAY), and Bebb-Sim (Simultaneous Bebbington). SAS® v.9.2 code for all the algorithms is presented in Appendix A, along with code that generates the datasets on which the algorithms are run to produce the runtimes shown in the Results section. Each algorithm is a macro that is completely modular and takes input values for six macro variables: the name of the input dataset, the name of the output dataset, the name of the variable to be permuted, the "by variables" defining the strata, the name of the variable designating the "Control" and "Treatment" samples, the number of permutation samples to be used in the permutation test, and a value (optional) for a random number generation seed (one of the macros, PROCMT, takes a value for a seventh macro variable: upper or lower p-values). Aside from general SAS® programming language rules (e.g. file naming conventions), assumptions made by the code are minimal (e.g. the variable designating the "Control" and "Treatment" samples contains corresponding values of "C" and "T", "by variables" are assumed to be character variables, and input datasets are presumed to be sorted by the specified "by variables"). Each is discussed below.

OPDN:

This is a completely new and original memory-intensive algorithm, which is one of the reasons it is so fast.[5] It uses no storage space, other than the original input dataset and a summary dataset of the record counts for each stratum. OPDN makes one pass through the dataset, record by record, stratum by stratum, and efficiently builds a large array of data values, essentially converting a column of data into a row of data (for one stratum at a time).[6] The fast sequential access of SAS® ensures that the *N*-celled array (where *N* is the size of the current stratum) is loaded with data values automatically, and very quickly, simply by using a SET statement and a _TEMPORARY_ array (which automatically retains values across records as each record is being read). Then SRSWOR is performed, in memory, using this array. The approach for SRSWOR is essentially that of Goodman & Hedetniemi (1977) which, as defined by Tille (2006), is a draw-by-draw algorithm.[7] While Tille (2006) describes the standard draw-by-draw approach for SRSWOR as having the major drawback of being "quite complex" (p.47), the draw-by-draw SRSWOR algorithm used by OPDN unarguably is not: as shown below in pseudo code, it is straightforward and easily understood.

OPDN implementation #1 of Goodman & Hedetniemi (1982) for Permutation Tests:

*** temp[] is the array filled with all the data values, for current stratum, of the variable being permuted
*** psums[] is the array containing the permutation sample statistic values for every permutation sample

```
do m = 1 to #permutation tests
   x ← 0
   tot_FREQ_hold ← # records in current stratum
   tot_FREQ ← tot_FREQ_hold
```

[5] The only algorithm at all similar is OPDY (see Opdyke, 2010), a SAS® algorithm which is used to conduct very fast bootstraps orders of magnitude faster than the relevant SAS® Procedure (Proc SurveySelect).

[6] The "One Pass" referenced in the name of this algorithm refers to SAS® making one sequential pass through the dataset when reading it. The sampling without replacement that is done using the *N*-celled array, after the entire stratum of *N* data values has been read into the array, obviously is not sequential or "one-pass."

[7] "Definition 37. A sampling design of fixed sample size n is said to be draw by draw if, at each one of the n steps of the procedure, a unit is definitively selected in the sample" (p. 35).


```
   do n = 1 to # records in smaller of Control and Treatment samples
      cell ← uniform random variate on 1 to tot_FREQ
      x    ← temp[cell] + x
      hold ← temp[cell]
      temp[cell] ← temp[tot_FREQ]
      temp[tot_FREQ] ← hold
      tot_FREQ ← tot_FREQ -1
   end;
   psums[m] ← x
end;
```

An explanation of the SRSWOR algorithm is as follows: random draws are made from the array using a uniform random variate, cell, drawn initially from 1 to *N* (where *N* is the size of the current stratum). The value from temp[cell] is used in the calculation of the sample statistic, and then swapped with that of temp[*N*]. This is repeated in a loop that iterates *n* times, where *n* is the size of the permutation sample (for efficiency, *n* is always the smaller of either the Control or Treatment sample, by design), but in each loop *N* is decremented by 1, so selected values cannot be selected more than once as they are placed at the end of the array which is never again touched by the random number generator (see Appendix C for a simple proof of this algorithm as a valid SRSWOR algorithm).

An alternate presentation of the SRSWOR algorithm is shown below, and it utilizes the fact that even if statistical calculations are not being performed as the sample is being selected (as the sample sum is cumulatively calculated above), the entire without-replacement-sample ends up in the last *n* cells of the array. So if applying a function to this collective set of cells is faster or more efficient than cumulatively calculating the sample statistic, it would be the preferable approach.

OPDN implementation #2 of Goodman & Hedetniemi (1982) for Permutation Tests:

*** temp[] is the array filled with all the data values, for current stratum, of the variable being permuted
*** psums[] is the array containing the permutation sample statistic values for every permutation sample

```
do m = 1 to #permutation tests
   tot_FREQ_hold ← # records in current stratum
   tot_FREQ ← tot_FREQ_hold
   do n = 1 to # records in smaller of Control and Treatment samples
      cell ← uniform random variate on 1 to tot_FREQ
      hold ← temp[cell]
      temp[cell] ← temp[tot_FREQ]
      temp[tot_FREQ] ← hold
      tot_FREQ ← tot_FREQ -1
   end;
   psums[m] ← sum(temp[tot_FREQ] to temp[tot_FREQ_hold])
end;
```

While use of this general algorithm appears to be fairly widespread (for example, see Pesarin, 2001, p.81), it does not appear to be common in some areas of statistics where it would be of benefit (for example, Tille (2006), which is an authoritative statistical sampling resource, does not appear to present it), nor does it appear to be in common usage within the SAS® community.

In addition to its efficiency for drawing a single without-replacement sample, another crucially important advantage of using this particular SRSWOR algorithm for performing permutation tests, or any statistical procedure requiring many without-replacement samples, is that the same array of data values can be used repeatedly, for generating all the permutation (SRSWOR) samples, even though it has been reordered by the selection of the previous sample. Fortunately, the order of the data values in the array does not matter for SRSWOR: the selection of each item is random, so sampling on the array can be performed regardless of the initial order of the data values in its cells. This means that no additional storage space is required when generating all the permutation samples – for each new without-replacement sample, the array simply can be used as it was left from the previous sample. This makes the repeated implementation of SRSWOR, as required by permutation tests, extremely fast and efficient: its runtime complexity is $O(n)$, and its storage (memory) space complexity is $O(N)$.

Although Goodman & Hedetniemi's (1982) approach has been modified and improved to require even less than $O(N)$ storage space (see Ernvall & Nevalainen, 1982), SAS®'s very fast built-in sequential access automatically fills the entire N-celled (_TEMPORARY_) array much faster, under almost all conditions, than SAS® could selectively fill an even much smaller array by choosing specific records using, say, a point= option on a SET statement (and thus, $O(N)<O(n)$ even when $N>>n$). With reasonable amounts of memory (e.g. 16GB of RAM), only if the stratum size approaches one billion records will the N-cells be too many for a _TEMPORARY_ array and cause the program to crash (by comparison, Proc NPAR1WAY crashes on strata less than a tenth the size of those that OPDN can handle). So the OPDN code relying on Goodman & Hedetniemi's (1982) approach using the full N-celled (_TEMPORARY_) array is not only faster than alternatives with smaller theoretical time and space complexity, but also, in SAS®, more robust than those alternatives, and more than adequate for handling sampling from all but the very largest of datasets.

A final point regarding OPDN: because it is memory intensive, OPDN uses code to optimize SAS®'s efficient use of memory, and to that end, the algorithm uses a data _null_ and saves the calculated permutation test results in macro variables cumulated by strata, rather than in a dataset specified in the data step (which reserves a nontrivial amount of memory and decreases the size of the strata OPDN could otherwise handle). This makes OPDN more robust, and for those familiar with SAS® macro code, understanding it is not onerous. The precision loss due to saving numerical results in macro variables (which are character strings) is, for most applications, trivial – to the tenth decimal place rather than the fourteenth. If that additional precision is needed, the user probably should be using a symbolic programming language, such as Mathematica®.

As described below, OPDN-Alt handles the data-step memory conservation issue a bit differently and with slightly less code, but it typically is slightly slower as a result.

OPDN-Alt:

This algorithm (OPDN "Alternative") is essentially the same algorithm as OPDN but with one coding difference: instead of saving the permutation p-values in macro variables cumulated by strata, rather than in a dataset specified on the data step to conserve memory, the OPDN-Alt code creates an empty dataset with missing values for all the necessary variables, and then uses a modify statement to update those values.[8] Otherwise, the algorithm is identical. The only arguable advantage to this approach is that it uses slightly

[8] The idea of using a modify statement instead of macro variables cumulated by strata came from a SAS® user with the email iebupdte@gmail.com who was commenting on the OPDY bootstrap algorithm presented in Opdyke (2010). When I asked for his/her name, he/she would not provide it, so I can only give credit to the email for the suggestion to try this approach when implementing the OPDN algorithm presented herein. I also modified the OPDY bootstrap algorithm presented in Opdyke (2010) and implemented it as OPDY-Alt, with very similar results: OPDY is slightly faster than OPDY-Alt.

less code. The unarguable disadvantage, however, is that it typically is slightly slower. That it is almost as fast is not surprising, since it is essentially the same algorithm, with a slightly different (and less efficient) approach to handling memory constraints in SAS®.

PROCSS:
PROCSS uses the built-in SAS® procedure, Proc SurveySelect, to create a sampling variable in the original dataset indicating which records in the stratum are selected into the without-replacement random sample, and multiple without-replacement samples can be created automatically for multiple strata. After this sampling is performed by Proc SurveySelect, the (large) output sampling dataset is summarized according to the permutation statistic being used with a Proc Summary, and the subsequent code calculating the permutation test p-values is identical to that of OPDN and OPDN-Alt. According to the SAS® Online Documentation, for simple random sampling without replacement, if there is enough memory Proc SurveySelect uses Floyd's ordered hash table algorithm (see Bentley and Floyd (1987) and Bentley and Knuth (1986) for details). If there is not enough memory available for Floyd's algorithm, Proc SurveySelect switches to the sequential algorithm of Fan, Muller, and Rezucha (1962), which requires less memory but might require more time to select the sample.[9]

Aside from speed, which is discussed in the Results section, the major disadvantage of relying on Proc SurveySelect instead of OPDN, OPDN-Alt, or Bebb-Sim is that it requires disk space for the large output sampling dataset that it generates, while these other algorithms have no such resource requirement.

PROCMT:
PROCMT uses Proc Multtest to calculate permutation test p-values, and prior to the relatively recent advent of Proc SurveySelect, Proc Multtest had been widely used for this purpose. Although Proc Multtest does not explicitly allow the user to specify sample sums as the permutation test statistic, using t-scores based on the pooled-variance provides mathematically identical permutation test p-values, so the pooled-variance option is specified in PROCMT. This option is easily changed if the assumption is not warranted by the dataset in use.

One limitation of Proc Multtest is that for each time the Proc is used, it only provides one p-value (either the left, right, or two-tailed p-value). So to obtain all three p-values the user must run the Proc three times. For comparison purposes in this paper, PROCMT runs Proc Multtest twice, obtaining one one-tailed p-value (specified by the user in a macro variable) and the two-tailed p-value. Since Proc NPAR1WAY provides the smaller of the two one-tailed p-values and the two-tailed p-value, and OPDN, OPDN-Alt, Bebb-Sim, and PROCSS provide all three p-values, this appeared to be a reasonable compromise for purposes of comparison. Obviously, if the user knows in advance that he/she only needs one of the three p-values, then the runtimes presented in the Results section for PROCMT will be slightly less than twice as large as they need to be.

Aside from speed, which is discussed in the Results section, the major disadvantage of relying on Proc Multtest instead of OPDN, OPDN-Alt, Bebb-Sim, or PROCSS is that the range of permutation test statistics available to the user is limited, whereas any of the latter can use any permutation test statistic.

[9] Note that the SASFILE statement used with Proc SurveySelect (see Cassell, 2007) is useless when the datasets to be permuted are too large for the extant memory – and that is the only time that fast permutation tests really are needed.

PROCNPAR:
PROCNPAR uses Proc NPAR1WAY to generate the smaller of the two one-tailed permutation test p-values and the two-tailed permutation test p-value. The options specified execute Pitman's permutation test, which uses the data values themselves as scores, and the sum as the sample statistic, consistent with the five other algorithms presented herein. The monte carlo option is used so that the number of permutation samples can be specified by the user.

Aside from speed, which is discussed in the Results section, there are two major disadvantages to relying on Proc NPAR1WAY instead of OPDN. First, for a fixed amount of memory, Proc NPAR1WAY crashes on strata less than a tenth the size of those that OPDN can handle with no problem. Secondly, like Proc Multtest, the range of permutation test statistics available to the user is limited, whereas OPDN, OPDN-Alt, Bebb-Sim, and PROCSS can use any permutation test statistic.

Bebb-Sim:
"Simultaneous Bebbington" refers to Bebbington (1975), one of the first and most straightforward sequential-sampling-without-replacement algorithms. Like the SRSWOR used in OPDN, Bebbington requires that N is known ahead of time, and it makes exactly n selections from N items, each with equal probability. But unlike OPDN, Bebbington is sequential: it makes a sample selection decision for each data record as it is encountered, sequentially (in order), in the dataset. Bebbington (1975) is presented in pseudo-code below for the reader's convenience.

```
1. Initialize: Let i ← 0, N′ ← N + 1, n′ ← n
2. i ← i + 1
3. If n′ = 0, STOP Algorithm
4. Visit Data Record i
5. N′ ← N′ – 1
6. Generate Uniform Random Variate u ~ Uniform[0, 1]
7. If u > (n′ / N′), Go To 2.
8. Otherwise, Output Record i into Sample
9.            n′ ← n′ – 1
10.           Go To 2.
```

Bebbington's (1975) algorithm above guarantees that 1) all possible samples of size n drawn from N are equally likely; 2) exactly n items will be selected; 3) no items in the sample will be repeated; and 4) items in the sample will appear in the same order that they appear in the population dataset.

The "Simultaneous" refers to the fact that two arrays of size m (where m is the number of permutation samples) are created and used in the dataset to simultaneously execute Bebbington m times as the dataset is being (sequentially) read, record by record. One array contains the cumulated test statistics for that particular sample, and the other array contains the corresponding counters for each sample, counting the number of items already selected into that sample (n' in the algorithm shown above). The counters are necessary to apply the correct probability of selection to each item in order to make the algorithm a valid SRSWOR procedure.

Aside from speed, which is discussed in the Results section, an advantage of relying on Bebb-Sim rather than Proc SurveySelect is that it is memory intensive, like OPDN, and does not require any disk space beyond the input dataset and a small dataset of counts for each stratum.

Other Possibilities:

Proc Plan:
Proc Plan can be used to conduct permutation tests, but since it does not have "by statement" functionality, it must be used once for each set of by-variable values in the original dataset. These results can be SET together and merged with the original dataset (after an observation counter that counts within each stratum is created) to obtain without-replacement samples. However, this is a much slower method, and possibly part of the impetus for the more recent creation of Proc SurveySelect.

DA:
The widely used and aging DA (Direct Access) method, which uses a SET statement with a point=x option, where x is a random variate, to randomly select an observation in a dataset, was shown by Opdyke (2010) to be far less efficient and far more than an order of magnitude slower than alternatives such as the OPDY algorithm for calculating bootstraps (see Opdyke, 2010). In the permutation test setting, DA is now actually irrelevant, since the only way (known to this author) to implement it is using Bebbington (1975) in a nested loop, where an inner loop iterates *n* times to obtain a single sample without replacement, and an outer loop iterates *m* times to obtain *m* without-replacement permutation samples (using this approach with only the inner loop to obtain a single SRSWOR sample is common to many SAS® programs). While technically possible to implement, a nested loop is extremely slow for this approach, and so it is not viable as a scalable algorithm for executing fast permutation tests in SAS®.

Results

The real and CPU runtimes of each of the algorithms, relative to those of OPDN, are shown in Table 1 below for different #strata, N = strata size, and m = size of the permutation samples (n = permutation sample size = 1,000 for all runs; for the absolute runtimes, see Table B1 in Appendix B). The code was run on a PC with only 2 GB of RAM and a 2 GHz Pentium chip.

OPDN dominates in all cases, except that typically it is only slightly faster than OPDN-Alt which, as mentioned above, is essentially the same algorithm, but with a slightly different and less efficient (and slightly less speedy) approach to memory conservation.

The second fastest algorithm is Bebb-Sim, followed by PROCNPAR for the smaller datasets but PROCSS for the larger datasets, and then PROCMT is much slower for all but the smallest datasets. Generally, the larger the dataset, the larger the runtime premium OPDN has over the other algorithms, achieving real runtime speeds 218x faster than PROCSS, 353x faster than PROCNPAR, and 723x faster than PROCMT. If it was runtime feasible to run PROCSS and PROCMT on even larger datasets, all indications are that the runtime premium of OPDN would only continue to increase. This would not be possible for PROCNPAR, however, because Proc NPAR1WAY crashes on datasets less than a tenth the size of those OPDN can handle with no problem (with only 2 GB of RAM, NPAR1WAY crashed on less than 10 million observations in the largest stratum, while OPDN handled over 110 million observations in the largest stratum).

Note that another advantage OPDN has over PROCSS (Proc SurveySelect) is that it needs virtually no storage (disk) space beyond the input dataset, while PROCSS needs disk space to output a potentially very large sampling dataset that subsequently must be summarized at the sample level, according to the permutation test statistic being used, to compare the permuted sample statistics to the original sample statistic.

Table 1: Real and CPU Runtimes of the Algorithms Relative to OPDN for Various *N*, #strata, and *m*

(**EX** = excessive, **CR** = crashed, sample size *n* = 1,000 for all)

			REAL					CPU				
N ***(per stratum)***	***#*** ***strata***	***m***	**OPDN -Alt**	**PROC SS**	**PROC MT**	**PROCN PAR**	**Bebb -Sim**	**OPDN -Alt**	**PROC SS**	**PROC MT**	**PROC NPAR**	**Bebb -Sim**
10,000	2	500	1.2	11.1	4.7	4.2	2.9	1.1	9.1	6.0	4.1	4.2
100,000	2	500	1.7	19.8	66.9	15.9	15.4	1.1	21.7	96.7	22.3	22.3
1,000,000	2	500	1.1	33.7	177.3	47.2	29.4	1.1	46.7	321.6	85.3	53.4
10,000,000	2	500	1.0	44.6	255.5	CR	36.4	1.0	57.4	422.1	CR	60.2
10,000	6	500	1.3	11.6	6.2	4.9	4.2	1.0	9.1	6.5	4.0	4.4
100,000	6	500	1.2	25.5	85.7	20.2	19.7	1.0	21.0	95.1	22.1	21.8
1,000,000	6	500	1.1	30.6	160.5	43.5	26.7	1.1	43.7	307.8	82.6	51.3
10,000,000	6	500	1.0	45.1	EX	CR	40.3	1.1	49.5	EX	CR	59.1
10,000	12	500	1.5	12.2	5.9	4.0	4.2	1.0	9.2	6.5	4.1	4.7
100,000	12	500	1.2	28.4	88.0	21.7	21.1	1.0	23.1	94.7	23.0	22.8
1,000,000	12	500	1.1	41.6	222.1	61.8	37.4	1.1	41.4	293.6	81.3	49.4
10,000,000	12	500	1.1	53.3	EX	CR	45.1	1.1	54.1	EX	CR	59.2
10,000	2	1000	1.1	15.2	6.5	4.0	4.4	1.0	10.3	6.8	4.1	4.7
100,000	2	1000	1.1	33.6	104.9	23.7	24.3	1.0	29.2	115.7	26.0	26.6
1,000,000	2	1000	1.3	96.6	468.9	125.0	79.1	1.0	82.3	508.0	134.7	85.8
10,000,000	2	1000	1.0	80.6	504.3	CR	73.3	1.0	96.6	794.0	CR	115.5
10,000	6	1000	1.0	12.6	6.5	4.0	4.6	1.0	9.5	6.8	4.0	4.8
100,000	6	1000	1.1	34.6	113.3	25.8	26.5	1.0	27.6	120.3	27.1	28.1
1,000,000	6	1000	1.0	55.7	291.8	77.6	50.1	1.1	74.9	520.4	138.0	89.2
10,000,000	6	1000	1.0	85.3	EX	CR	79.2	1.1	94.3	EX	CR	115.4
10,000	12	1000	1.0	13.3	6.6	4.3	4.7	1.0	9.6	6.7	4.2	4.8
100,000	12	1000	1.0	33.4	115.0	27.4	26.5	1.0	25.8	117.6	27.7	27.1
1,000,000	12	1000	1.0	78.4	412.4	111.6	70.1	1.1	74.1	516.1	139.3	87.8
10,000,000	12	1000	1.0	99.8	EX	CR	88.7	1.1	91.3	EX	CR	111.5
10,000	2	2000	1.3	10.9	5.4	3.5	3.9	1.0	9.3	6.5	3.9	4.7
100,000	2	2000	1.0	36.2	122.5	27.8	28.3	1.0	29.6	132.6	29.7	30.6
1,000,000	2	2000	1.0	135.9	723.1	191.2	123.1	1.0	116.1	798.5	211.2	136.1
10,000,000	2	2000	1.0	172.7	EX	CR	143.6	1.0	215.8	EX	CR	227.9
10,000	6	2000	1.0	13.6	6.5	4.0	4.6	1.0	10.3	6.7	4.0	4.7
100,000	6	2000	1.0	38.8	126.1	28.6	29.5	1.0	31.0	131.5	29.7	30.7
1,000,000	6	2000	1.0	97.7	497.9	136.4	85.5	1.0	122.8	810.3	214.8	138.7
10,000,000	6	2000	1.1	218.2	EX	CR	159.0	1.1	248.3	EX	CR	222.9
10,000	12	2000	1.0	15.0	6.5	4.2	4.7	1.0	11.5	6.7	4.2	4.8
100,000	12	2000	1.0	44.4	131.9	30.7	30.4	1.0	35.5	135.4	31.5	31.3
1,000,000	12	2000	1.0	147.0	685.1	185.7	117.2	1.1	136.4	798.6	216.4	136.5
10,000,000	12	2000	1.1	EX	EX	CR	EX	1.1	EX	EX	CR	EX
7,500,000	2	2000	1.1	243.6	EX	353.0	201.0	1.1	214.0	EX	396.3	227.5
7,500,000	6	2000	1.2	242.0	EX	334.8	193.9	1.1	218.2	EX	382.5	222.9
25,000,000	12	500	1.2	EX	EX	CR	EX	1.1	EX	EX	CR	EX
50,000,000	12	500	1.2	EX	EX	CR	EX	1.1	EX	EX	CR	EX
100,000,000	12	500	1.0	EX	EX	CR	EX	1.0	EX	EX	CR	EX

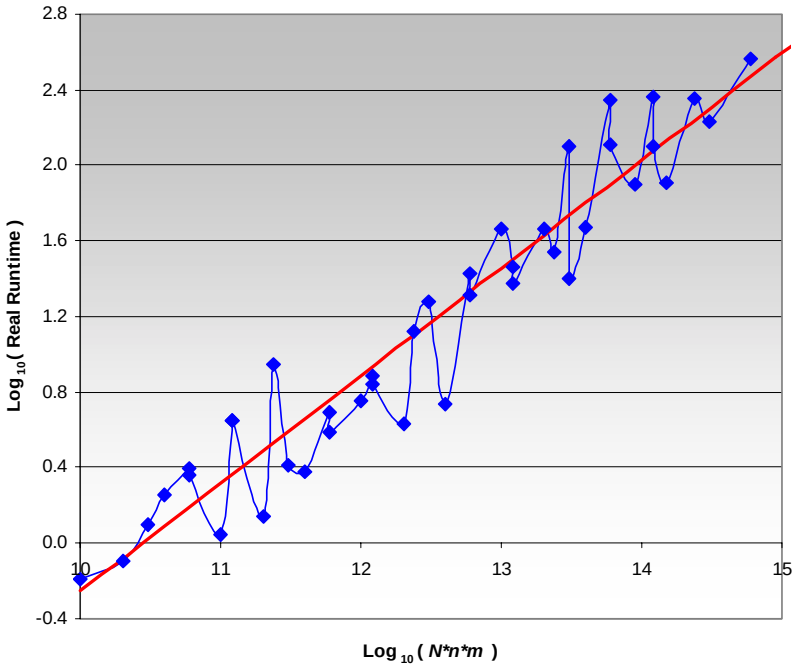
And finally, as mentioned above, the size constraint on OPDN is the size of the largest stratum in the dataset, *not* the size of the entire dataset, which appears to be what drives the runtimes of Proc SurveySelect and Proc Multtest (this does not appear to be the case for Proc NPAR1WAY, but then, Proc NPAR1WAY crashes on much smaller strata). So OPDN maintains scalability on datasets potentially many, many orders of magnitude larger than those the three Proc's can handle.

In a nutshell, OPDN is not only much, much faster than all the other methods, but also far more robust. All of these results combine to make OPDY the only truly scalable permutation test (and SRSWOR) algorithm in SAS®.

Focusing on the real runtime of OPDN and its relationship to #strata, *N*, *n*, and *m* (see Graph 1), the empirical runtimes in Table B1 yield a reasonably accurate approximation in (1), so OPDN appears to be O(*N***n***m*), which is sensible.

$\text{Log}_{10}(\text{Real Runtime}) = -5.95291 + 0.57001 * \text{Log}_{10}(N*n*m)$ (where N = all N across strata) (1)

Graph 1: OPDN Real Runtime by *Nn*m* (*N* = all strata)**



Conclusions

The goal of this paper was to develop a non-resource intensive permutation test algorithm faster than its competitors on arguably the most widely used statistical software platform globally. The OPDN algorithm accomplishes this objective. It uses negligible storage space, relatively little memory space via very efficient SRSWOR, and on the SAS® platform, it is much faster than any other alternative, including the built-in SAS®/STAT Procedures designed to perform permutation tests (Proc SurveySelect, Proc Multtest, and Proc NPAR1WAY). OPDN's relative (and absolute) speed premium increases with dataset size, and with reasonable time complexity (O(*N*n*m*)) it maintains scalability on large datasets, handling datasets (technically, the largest stratum in the dataset) more than ten times larger than those that crash Proc NPAR1WAY. It is not only the fastest permutation test and SRSWOR algorithm in SAS®, but also the most robust. Given SAS®'s reputation for speed compared to other statistical computing platforms, OPDN is likely to be a serious contender for the fastest algorithm for these purposes among all major statistical computing packages.

That said, it should be noted that the algorithm runtimes presented herein, and in any empirical algorithm research for that matter, obviously can very much depend on the hardware configurations on which the algorithms are run. Very fast I/O speeds on grid platforms, for example, may well allow Proc SurveySelect to close the speed gap on OPDN. However, it is also likely that such platforms also will have comparably souped-up memory resources, and thus, OPDN may retain or even increase its already large speed lead: all else equal, as a rule, memory processing always will be faster than I/O processing, and OPDN does not need to create and write to disk the sampling dataset that Proc SurveySelect needs to generate.

The code provided herein allows for testing and comparisons on different platforms, and this author welcomes feedback from readers regarding the relative performance of the algorithms across an array of hardware configurations.

Acknowledgments

I sincerely thank Toyo Johnson and Nicole Ann Johnson Opdyke for their support and belief that SAS® could produce a better permutation test and better sampling without replacement.

Appendix A

SAS® v.9.2 code for the OPDN, OPDN-Alt, PROCSS, PROCMT, PROCNPAR, and Bebb-Sim algorithms, and the SAS® code that generates the datasets used to test them in this paper, is presented below.

```
************************************************************************
************************************************************************
PROGRAM:  MFPTUS.SAS

DATE:     12/30/10

CODER:    J.D. Opdyke
          Managing Director, Quantitative Strategies
          DataMineIt

PURPOSE:  Run and compare 6 different SAS Permutation Tests algorithms
          including OPDN, OPDN_alt, PROCSS, PROCMT, PROCNPAR, and BEBB_SIM.
          See "Permutation Tests (and Sampling with Replacement) Orders of
          Magnitude Faster Using SAS" by J.D. Opdyke for detailed explanations
          of the different algorithms.

INPUTS:   Each macro is completely modular and accepts 6 macro parameter
          values (PROCMT accepts 7):
                indata        = the input dataset (including libname)
                outdata       = the input dataset (including libname)
                byvars        = the "by variables" defining the strata (these are
                                character variables)
                samp2var      = the (character) variable defining the two samples in each
                                stratum: TEST ("T") and CONTROL ("C"), with those values
                permvar       = the variable to be tested with permutation tests
                num_psmps     = number of Permutation Test samples
                seed          = and optional random number seed (must be an integer or
                                blank)
                left_or_right = "left" or "right" depending on whether the user wants
                                lower or upper one-tailed p-values, respectively, in
                                addition to the two-tailed p-value (only for PROCMT)

OUTPUTS:  A SAS dataset, named by the user via the macro variable outdata,
          which contains the following variables:
                1) the "by variables" defining the strata
                2) the name of the permuted variable
                3) the size (# of records) of the permutation samples (this should
                   always be the smaller of the two samples (TEST v. CONTROL))
                4) the number of permutation samples
                5) the left, right, and two-tailed p-values corresponding to the
                   following hypotheses:
                      p_left  = Pr(x>X | Ho: Test>=Control)
                      p_right = Pr(x<X | Ho: Test<=Control)
                      p_both  = Pr(x=X | Ho: Test=Control)
                   where x is the sample permutation statistic from the Control
                   sample (or the additive inverse of the Test sample if the Test
                   sample is smaller, which is atypical) and X is the distribution of
                   permutation statistic values.

                   Following standard convention, the two-tailed p-value is
                   calculated based on the reflection method.
************************************************************************
************************************************************************
***;
```



```
options
label          symbolgen fullstimer yearcutoff=1950 nocenter         ls = 256   ps = 51
msymtabmax=max mprint    mlogic     minoperator     mindelimiter=' ' cleanup;

libname MFPTUS "c:\";

%MACRO makedata(strata_size=, testproportion=, numsegs=, numgeogs=);

*** Since generating all the datasets below once takes only a couple of minutes,
    time was not wasted trying to optimize runtimes for the creation of mere
    test data.
***;

%let numstrata = %eval(&numsegs.*&numgeogs.);

*** The variable "cntrl_test" identifies the test and control samples with
    values of "T" and "C" respectively.
***;

data MFPTUS.pricing_data_&numstrata.strata_&strata_size.(keep=geography segment cntrl_test
price sortedby=geography segment);
  format segment geography $8. cntrl_test $1.;
  array seg{3} $ _TEMPORARY_ ('segment1' 'segment2' 'segment3');
  array geog{4} $ _TEMPORARY_ ('geog1' 'geog2' 'geog3' 'geog4');
  strata_size = 1* &strata_size.;
  do x=1 to &numgeogs.;
     geography=geog{x};
     do j=1 to &numsegs.;
        segment=seg{j};
        if j=1 then do i=1 to strata_size;
           if mod(i,&testproportion.)=0 then do;
              cntrl_test="T";
              price=(rand('UNIFORM')+0.175/(&testproportion./(&testproportion./10)));
           end;
           else do;
              cntrl_test="C";
              price=rand('UNIFORM');
           end;
           output;
        end;
        else if j=2 then do i=1 to strata_size;
           if mod(i,&testproportion.)=0 then do;
              cntrl_test="T";
              price=rand('POISSON',1-0.75/(&testproportion./(&testproportion./10)));
           end;
           else do;
              cntrl_test="C";
              price=rand('POISSON',1.0);
           end;
           output;
        end;
        else if j=3 then do i=1 to strata_size;
*** Make Control smaller to test code in algorithms.;
           if mod(i,&testproportion.)=0 then cntrl_test="C";
           else cntrl_test="T";
           price=rand('NORMAL');
           output;
        end;
     end;
  end;
  run;
```



```
%MEND makedata;

%makedata(strata_size=10000, testproportion=10, numsegs=2, numgeogs=1);
%makedata(strata_size=10000, testproportion=10, numsegs=2, numgeogs=3);
%makedata(strata_size=10000, testproportion=10, numsegs=3, numgeogs=4);

%makedata(strata_size=100000, testproportion=100, numsegs=2, numgeogs=1);
%makedata(strata_size=100000, testproportion=100, numsegs=2, numgeogs=3);
%makedata(strata_size=100000, testproportion=100, numsegs=3, numgeogs=4);

%makedata(strata_size=1000000, testproportion=1000, numsegs=2, numgeogs=1);
%makedata(strata_size=1000000, testproportion=1000, numsegs=2, numgeogs=3);
%makedata(strata_size=1000000, testproportion=1000, numsegs=3, numgeogs=4);

%makedata(strata_size=10000000, testproportion=10000, numsegs=2, numgeogs=1);
%makedata(strata_size=10000000, testproportion=10000, numsegs=2, numgeogs=3);
%makedata(strata_size=10000000, testproportion=10000, numsegs=3, numgeogs=4);


%makedata(strata_size=7500000, testproportion=7500, numsegs=2, numgeogs=1);
%makedata(strata_size=7500000, testproportion=7500, numsegs=2, numgeogs=3);


%makedata(strata_size=25000000, testproportion=25000, numsegs=2, numgeogs=1);
%makedata(strata_size=50000000, testproportion=50000, numsegs=2, numgeogs=1);
%makedata(strata_size=100000000, testproportion=100000, numsegs=2, numgeogs=1);



*** OPDN_alt ***;
*** OPDN_alt ***;
*** OPDN_alt ***;

%MACRO OPDN_alt(num_psmps=,
                indata=,
                outdata=,
                byvars=,
                samp2var=,
                permvar=,
                seed=
                );

*** If user does not pass a value to the optional macro variable "seed," use -1
    based on the time of day.
***;
%if %sysevalf(%superq(seed)=,boolean) %then %let seed=-1;

*** To minimize intermediate memory requirements, initialize output data set
    with missing variable values.
***;

data &outdata.(sortedby=&byvars.);
   stop;
   set &indata(keep=&byvars.);
   length permvar $32 n_psamp num_psmps p_left p_right p_both
          distance_right distance_left lastbinnum lastbinsize
          pmed pmean tot_FREQ_incr x 8;
   call missing(of _all_);
   run;

*** Obtain counts and cumulated counts for each strata.;
```



```
proc summary data=&indata. nway;
  class &byvars. &samp2var.;
  var &permvar.;
  output out=byvar_sum(keep = _FREQ_ &byvars. &samp2var. sumpvar)
                        sum = sumpvar
                       ;
  run;

*** Identify and keep the smaller of the two samples for more efficient
    sampling.  Below, invert the empirical permutation distribution if CONTROL
    sample is smaller that TEST sample (which is not typical).  That will
    remain consistent with output variables corresponding to:
         p_left  = Pr(x>X | Ho: Test>=Control)
         p_right = Pr(x<X | Ho: Test<=Control)
         p_both  = Pr(x=X | Ho: Test=Control)
    where x is the sample permutation statistic and X is the distribution of
    permutation statistic values.
***;

%let last_byvar = %scan(&byvars.,-1);

data byvar_sum_min(keep=tot_FREQ _FREQ_ &byvars. &samp2var. sumpvar sortedby=&byvars.);
  set byvar_sum;
  format tot_FREQ _FREQ_ 16.;
  by &byvars.;
  retain lag_FREQ lag_sum lag_samp2;
  if first.&last_byvar. then do;
     lag_FREQ = _FREQ_;
     lag_sum  = sumpvar;
     lag_samp2  = &samp2var.;
  end;
  else do;
     tot_FREQ = sum(lag_FREQ,_FREQ_);
     if _FREQ_<=lag_FREQ then output;
     else do;
        _FREQ_  = lag_FREQ;
        sumpvar = lag_sum;
        &samp2var. = lag_samp2;
        output;
     end;
  end;
  run;


*** Get size of largest stratum for efficient (re)use of _TEMPORARY_ array.;

proc sql noprint;
   select max(tot_FREQ) into :max_tot_freq from byvar_sum_min;
   quit;

*** In case number is large, avoid scientific notation.;
%let max_tot_freq = %sysfunc(putn(&max_tot_freq.,16.));

data &outdata.;
   if 0 then modify &outdata.;

*** To view permutation distributions for each strata in .log file.,
    comment out first line below this comment, uncomment the line below it, and
    uncomment "put _ALL_" 35 lines below it.  Note that this will slow program
    execution and often create large .log files.
```



```
***;
   array psums[&num_psmps.] _TEMPORARY_;
*  array psums{&num_psmps.} psmp1-psmp&num_psmps.;
   array temp[&max_tot_freq.]   _TEMPORARY_;
   retain num_psmps &num_psmps.;
   do _n_ = 1 by 1 until(last.&last_byvar.);
      merge &indata.(keep=&byvars. &permvar. &samp2var.)
            byvar_sum_min
            ;
      by &byvars.;
      temp[_n_]=&permvar.;
   end;
   seed = 1*&seed.;
   do m=1 to num_psmps;
      x=0;
      tot_FREQ_incr = tot_FREQ;
      do n=1 to _FREQ_;
         cell = floor(ranuni(seed)*tot_FREQ_incr) + 1;
         x = temp[cell] + x;
         hold = temp[cell];
         temp[cell]=temp[tot_FREQ_incr];
         temp[tot_FREQ_incr] = hold;
         tot_FREQ_incr+(-1);
      end;
      psums[m] = x;
   end;

   n_psamp = _FREQ_;

   p_right = 0;
   p_left  = 0;
   p_both  = 0;
   call sortn(of psums[*]);
   pmed = median(of psums[*]);
   pmean = mean(of psums[*]);

* put _ALL_;

*** Efficiently handle extreme test sample values.;

   IF sumpvar<psums[1] THEN DO;
      p_left=0;
      p_right=num_psmps;
      p_both=0;
   END;
   ELSE IF sumpvar>psums[num_psmps] THEN DO;
      p_left=num_psmps;
      p_right=0;
      p_both=0;
   END;
   ELSE DO;

*** For non-extreme cases, start with shorter tail for less looping.;

   if pmed>=sumpvar then do;
      do z=1 to num_psmps;
         if sumpvar>=psums[z] then p_left+1;
         else do;
            lastbinnum = z-1;
            distance_left = pmean - psums[z-1];
            leave;
```



```
         end;
      end;

*** Avoid loop for other (larger) p-value.
    If sumpvar equals last bin, p_right = 1 - p_left + lastbinsize.
    Otherwise, p_right = 1 - p_left.
***;
      if sumpvar = psums[lastbinnum] then do;
         lastbinsize=1;
         do k=lastbinnum to 1 by -1;
            if psums[k]=psums[k-1] then lastbinsize+1;
            leave;
         end;
         p_right = num_psmps - p_left + lastbinsize;
      end;
      else p_right = num_psmps - p_left;
   end;
   else do;
      do z=num_psmps to 1 by -1;
         if sumpvar<=psums[z] then p_right+1;
         else do;
            lastbinnum = z+1;
            distance_right = psums[z+1] - pmean;
            leave;
         end;
      end;

*** Avoid loop for other (larger) p-value.
    If psum equals last bin, p_left = 1 - p_right + lastbinsize.
    Otherwise, p_left = 1 - p_right.
***;
      if sumpvar = psums[lastbinnum] then do;
         lastbinsize=1;
         do k=lastbinnum to num_psmps;
            if psums[k]=psums[k+1] then lastbinsize+1;
            else leave;
         end;
         p_left = num_psmps - p_right + lastbinsize;
      end;
      else p_left = num_psmps - p_right;
   end;

*** Base 2-sided p-value on distance from mean of last (i.e. least extreme) bin
    of smaller p-value.  This is common practice.
***;

   if p_left<p_right then do;
      p_both = p_left;
      do z=num_psmps to 1 by -1;
         if (psums[z] - pmean) >= distance_left then p_both+1;
         else leave;
      end;
   end;
   else if p_left>p_right then do;
      p_both = p_right;
      do z=1 to num_psmps;
         if (pmean - psums[z]) >= distance_right then p_both+1;
         else leave;
      end;
   end;
   else p_both=num_psmps;
```



```
*** Account for possibility, due to psum=a particular bin value, that
    p_both>num_psmps.
***;
   p_both = min(p_both,num_psmps);

   END;

*** If CONTROL sample is smaller than TEST (which is atypical), reverse
*** p-values, as empirical distribution is mirror of itself.;

   if &samp2var.="C" then do;
      hold = p_left;
      p_left = p_right;
      p_right = hold;
   end;

   p_left  = p_left  / num_psmps;
   p_right = p_right / num_psmps;
   p_both  = p_both  / num_psmps;

   length permvar $32;
   retain permvar "&permvar";
   output;
   run;

data &outdata.;
   set &outdata.(keep=&byvars. permvar num_psmps n_psamp p_left p_right p_both);
   label permvar   = "Permuted Variable"
         n_psamp   = "Size of Permutation Samples"
         num_psmps = "# of Permutation Samples"
         p_left    = "Left p-value"
         p_right   = "Right p-value"
         p_both    = "Two-Tailed p-value"
         ;
   run;


*** Optional.;

proc datasets lib=work memtype=data kill nodetails;
   run;

%MEND OPDN_alt;

%OPDN_alt(num_psmps = 1000,
          indata    = MFPTUS.pricing_data_2strata_100000,
          outdata   = MFPTUS.OPDN_alt_100000_2strata,
          byvars    = geography segment,
          samp2var  = cntrl_test,
          permvar   = price,
          seed      =
         );



*** OPDN ***;
*** OPDN ***;
*** OPDN ***;

%MACRO OPDN(num_psmps=,
```



```
            indata=,
            outdata=,
            byvars=,
            samp2var=,
            permvar=,
            seed=
            );


*** The only assumption made within this macro is that the byvars are all
    character variables and the input dataset is sorted by the "by variables"
    that define the strata.  Also, the "TEST" and "CONTROL" samples are defined
    by a variable "cntrl_test" formatted as $1. containing "T" and "C"
    character strings, respectively, as values.
***;

*** Obtain the last byvar, count the byvars, and assign each byvar into numbered
    macro variables for easy access/processing.
***;

%let last_byvar = %scan(&byvars.,-1);
%let num_byvars = %sysfunc(countw(&byvars.));
%do i=1 %to &num_byvars.;
   %let byvar&i. = %scan(&byvars.,&i.);
%end;

*** If user does not pass a value to the optional macro variable "seed," use -1
    based on the time of day.
***;
%if %sysevalf(%superq(seed)=,boolean) %then %let seed=-1;

*** Obtain counts and cumulated counts for each strata.;

proc summary data=&indata. nway;
  class &byvars. &samp2var.;
  var &permvar.;
  output out=byvar_sum(keep = _FREQ_ &byvars. &samp2var. sumpvar)
                        sum = sumpvar
                       ;
  run;

*** Identify and keep the smaller of the two samples for more efficient
    sampling.  Below, invert the empirical permutation distribution if CONTROL
    sample is smaller that TEST sample (which is not typical).  That will
    remain consistent with output variables corresponding to:
         p_left  = Pr(x>X | Ho: Test>=Control)
         p_right = Pr(x<X | Ho: Test<=Control)
         p_both  = Pr(x=X | Ho: Test=Control)
    where x is the sample permutation statistic and X is the distribution of
    permutation statistic values.
***;

data byvar_sum_min(keep=tot_FREQ _FREQ_ &byvars. &samp2var. sumpvar sortedby=&byvars.);
  set byvar_sum;
  format tot_FREQ _FREQ_ 16.;
  by &byvars.;
  retain lag_FREQ lag_sum lag_samp2;
  if first.&last_byvar. then do;
     lag_FREQ = _FREQ_;
     lag_sum  = sumpvar;
     lag_samp2  = &samp2var.;
```



```
  end;
  else do;
     tot_FREQ = sum(lag_FREQ,_FREQ_);
     if _FREQ_<=lag_FREQ then output;
     else do;
        _FREQ_  = lag_FREQ;
        sumpvar = lag_sum;
        &samp2var. = lag_samp2;
        output;
     end;
  end;
  run;


*** Obtain number of strata, and use this number to count and separately permute
    each strata below.
***;

%let dsid = %sysfunc(open(byvar_sum_min));
%let n_byvals = %sysfunc(ifc(&dsid.
                                  ,%nrstr(%sysfunc(attrn(&dsid.,nobs))
                                          %let rc = %sysfunc(close(&dsid.));
                                          )
                                  ,%nrstr(0)
                             )
                         );
*** In case number is large, avoid scientific notation.;
%let n_byvals = %sysfunc(putn(&n_byvals.,16.));


*** Cumulate counts of strata for efficient (re)use of _TEMPORARY_ array.;

data byvar_sum_min(drop=prev_freq);
  set byvar_sum_min;
  format cum_prev_freq _FREQ_ 16.;
  retain cum_prev_freq 0;
  prev_freq = lag(tot_FREQ);
  if _n_=1 then prev_freq = 0;
  cum_prev_freq = sum(cum_prev_freq, prev_freq);
  run;


*** For access in data step below (that uses data _null_ for memory
    conservation), put into macro strings a) smaller-of-two-sample counts,
    b) total counts for each strata, c) cumulated total counts, d) summed
    permutation variable, d) which of two samples is smaller, and e) byvar
    values.
***;

proc sql noprint;
  select _freq_ into :freqs separated by ' ' from byvar_sum_min;
  quit;
proc sql noprint;
  select tot_FREQ into :tot_FREQs separated by ' ' from byvar_sum_min;
  quit;
proc sql noprint;
  select cum_prev_freq into :cum_prev_freqs separated by ' ' from byvar_sum_min;
  quit;
proc sql noprint;
  select sumpvar into :sumpvar separated by ' ' from byvar_sum_min;
  quit;
```



```
proc sql noprint;
  select &samp2var. into :samp2var separated by ' ' from byvar_sum_min;
  quit;

%do i=1 %to &num_byvars.;
   proc sql noprint;
     select &&byvar&i. into :byvals&i. separated by ' ' from byvar_sum_min;
     quit;
%end;


*** Get size of largest stratum for efficient (re)use of _TEMPORARY_ array.;

proc sql noprint;
  select max(tot_FREQ) into :max_tot_freq separated by ' ' from byvar_sum_min;
  quit;

*** In case number is large, avoid scientific notation.;
%let max_tot_freq = %sysfunc(putn(&max_tot_freq.,16.));


*** Save each stratums results in cumulated macro variables instead of
    outputting to a dataset on the data step to lessen intermediate memory
    requirements.
*** ;

*** Initialize macro variables used below.;
%let p_left =;
%let p_right =;
%let p_both =;
%let samp_small_size =;

data _null_;
  set &indata.;
  by &byvars.;
  format which_sample $1.;
*** To view permutation distributions for each strata in .log file.,
    comment out first line below this comment, uncomment the two lines below
    it, and uncomment "put _ALL_" 37 lines below it.  Note that this will slow
    program execution and often create large .log files.
***;
  array psums{&num_psmps.} _TEMPORARY_;
*  array psums{&num_psmps.} psmp1-psmp&num_psmps.;
*  retain byval_counter 0 cum_prev_freq 0 psmp1-psmp&num_psmps.;
  array temp{&max_tot_freq.} _TEMPORARY_;
  retain byval_counter 0 cum_prev_freq 0;

  temp[_n_-cum_prev_freq]=&permvar.;

  if last.&last_byvar. then do;
     byval_counter+1;
     num_psmps = &num_psmps.*1;
     psmp_size = 1 * scan("&freqs.", byval_counter,' ');
     which_sample = COMPRESS(UPCASE(scan("&samp2var.", byval_counter,' ')),' ');
     tot_FREQ_hold = 1 * scan("&tot_FREQs.", byval_counter,' ');
     seed = 1*&seed.;

     do m=1 to num_psmps;
        x=0;
        tot_FREQ = tot_FREQ_hold;
        do n=1 to psmp_size;
```



```
            cell = floor(ranuni(seed)*tot_FREQ) + 1;
            x = temp[cell] + x;
            hold = temp[cell];
            temp[cell]=temp[tot_FREQ];
            temp[tot_FREQ] = hold;
            tot_FREQ+(-1);
         end;
         psums[m] = x;
      end;
      psum = 1*scan("&sumpvar.", byval_counter,' ');
      p_right = 0;
      p_left  = 0;
      p_both  = 0;
      call sortn(of psums[*]);
      pmed = median(of psums[*]);
      pmean = mean(of psums[*]);

* put _ALL_;

*** Efficiently handle extreme test sample values.;

      IF psum<psums[1] THEN DO;
         p_left=0;
         p_right=num_psmps;
         p_both=0;
      END;
      ELSE IF psum>psums[num_psmps] THEN DO;
         p_left=num_psmps;
         p_right=0;
         p_both=0;
      END;
      ELSE DO;

*** For non-extreme cases, start with shorter tail for less looping.;

      if pmed>=psum then do;
         do z=1 to num_psmps;
            if psum>=psums[z] then p_left+1;
            else do;
               lastbinnum = z-1;
               distance_left = pmean - psums[z-1];
               leave;
            end;
         end;

*** Avoid loop for other (larger) p-value.
    If psum equals last bin, p_right = 1 - p_left + lastbinsize.
    Otherwise, p_right = 1 - p_left.
***;
         if psum = psums[lastbinnum] then do;
            lastbinsize=1;
            do k=lastbinnum to 1 by -1;
               if psums[k]=psums[k-1] then lastbinsize+1;
               leave;
            end;
            p_right = num_psmps - p_left + lastbinsize;
         end;
         else p_right = num_psmps - p_left;
      end;

      else do;
```



```
        do z=num_psmps to 1 by -1;
           if psum<=psums[z] then p_right+1;
           else do;
              lastbinnum = z+1;
              distance_right = psums[z+1] - pmean;
              leave;
           end;
        end;

*** Avoid loop for other (larger) p-value.
    If psum equals last bin, p_left = 1 - p_right + lastbinsize.
    Otherwise, p_left = 1 - p_right.
***;
         if psum = psums[lastbinnum] then do;
            lastbinsize=1;
            do k=lastbinnum to num_psmps;
               if psums[k]=psums[k+1] then lastbinsize+1;
               else leave;
            end;
            p_left = num_psmps - p_right + lastbinsize;
         end;
         else p_left = num_psmps - p_right;
      end;

*** Base 2-sided p-value on distance from mean of last (i.e. least extreme) bin
    of smaller p-value.  This is common practice.
***;

      if p_left<p_right then do;
         p_both = p_left;
         do z=num_psmps to 1 by -1;
            if (psums[z] - pmean) >= distance_left then p_both+1;
            else leave;
         end;
      end;
      else if p_left>p_right then do;
         p_both = p_right;
         do z=1 to num_psmps;
            if (pmean - psums[z]) >= distance_right then p_both+1;
            else leave;
         end;
      end;
      else p_both=num_psmps;

*** Account for possibility, due to psum=a particular bin value, that
    p_both>num_psmps.
***;
      p_both = min(p_both,num_psmps);

     END;

     p_left  = p_left  / num_psmps;
     p_right = p_right / num_psmps;
     p_both  = p_both  / num_psmps;

*** If CONTROL sample is smaller than TEST (which is atypical), reverse
*** p-values, as empirical distribution is mirror of itself.;

     if "C"=COMPRESS(UPCASE(scan("&samp2var.", byval_counter,' ')),' ') then do;
        hold = p_left;
        p_left = p_right;
```



```
        p_right = hold;
     end;

*** Cumulate key macro variables to save results.;

     call symput('p_left',symget('p_left')||" "||compress(p_left));
     call symput('p_right',symget('p_right')||" "||compress(p_right));
     call symput('p_both',symget('p_both')||" "||compress(p_both));
     cum_prev_freq = 1*scan("&cum_prev_freqs.",byval_counter+1,' ');
  end;
  run;

*** Obtain and assign the format of each byvar, all of which are assumed to be
    character variables.
***;

data lens(keep=lens);
  set &indata.(keep=&byvars. firstobs=1 obs=1);
  do i=1 to &num_byvars.;
     lens = vlengthx(scan("&byvars.",i));
     output;
  end;
  run;
proc sql noprint;
  select lens into :alllens separated by ' ' from lens;
  quit;
%macro assign_formats;
   %do i=1 %to &num_byvars.;
      &&byvar&i. $%scan(&alllens.,&i.).
   %end;
%mend assign_formats;


*** Assign each byvar value for each stratum.;

%macro assign_byvar_vals(which_strata=);
   %do j=1 %to &num_byvars.;
      &&byvar&j. = scan("&&byvals&j.",&which_strata.,' ');
   %end;
%mend assign_byvar_vals;


*** Unwind and assign all the cumulated macro variables.;

data &outdata.(sortedby=&byvars. drop=n_byvals i);
  n_byvals = 1*&n_byvals.;
  format %assign_formats;
  do i=1 to n_byvals;
     length permvar $32;
     permvar = "&permvar.";
     n_psamp = 1 * scan("&freqs.", i,' ');
     num_psmps = &num_psmps.;
     p_left = 1*scan("&p_left.",i,' ');
     p_right = 1*scan("&p_right.",i,' ');
     p_both = 1*scan("&p_both.",i,' ');
     %assign_byvar_vals(which_strata = i)
     label permvar   = "Permuted Variable"
           n_psamp   = "Size of Permutation Samples"
           num_psmps = "# of Permutation Samples"
           p_left    = "Left p-value"
           p_right   = "Right p-value"
```



```
           p_both    = "Two-Tailed p-value"
           ;
     output;
  end;
  run;


*** Optional.;

proc datasets lib=work memtype=data kill nodetails;
  run;

%MEND OPDN;

%OPDN(num_psmps = 1000,
      indata    = MFPTUS.pricing_data_2strata_100000,
      outdata   = MFPTUS.OPDN_100000_2strata,
      byvars    = geography segment,
      samp2var  = cntrl_test,
      permvar   = price,
      seed      =
     );



*** PROC_SS ***;
*** PROC_SS ***;
*** PROC_SS ***;

%MACRO PROCSS(num_psmps=,
              indata=,
              outdata=,
              byvars=,
              samp2var=,
              permvar=,
              seed=
             );


*** If user does not pass a value to the optional macro variable "seed," use -1
    based on the time of day.
***;
%if %sysevalf(%superq(seed)=,boolean) %then %let seed=-1;

*** Obtain counts and cumulated counts for each strata.;

proc summary data=&indata. nway;
  class &byvars. &samp2var.;
  var &permvar.;
  output out=byvar_sum(keep = _FREQ_ &byvars. &samp2var. sumpvar)
                        sum = sumpvar
                       ;
  run;

*** Identify and keep the smaller of the two samples for more efficient
    sampling.  Below, invert the empirical permutation distribution if CONTROL
    sample is smaller that TEST sample (which is not typical).  That will
    remain consistent with output variables corresponding to:
         p_left  = Pr(x>X | Ho: Test>=Control)
         p_right = Pr(x<X | Ho: Test<=Control)
         p_both  = Pr(x=X | Ho: Test=Control)
```



```
    where x is the sample permutation statistic and X is the distribution of
    permutation statistic values.
***;

%let last_byvar = %scan(&byvars.,-1);
data byvar_sum(keep=&byvars. _NSIZE_ sumpvar &samp2var. sortedby=&byvars.);
  set byvar_sum(rename=(_FREQ_=_NSIZE_));
  by &byvars.;
  retain lag_NSIZE lag_sum lag_samp2;
  if first.&last_byvar. then do;
     lag_NSIZE = _NSIZE_;
     lag_sum  = sumpvar;
     lag_samp2  = &samp2var.;
  end;
  else do;
     if _NSIZE_<=lag_NSIZE then output;
     else do;
           _NSIZE_  = lag_NSIZE;
           sumpvar = lag_sum;
           &samp2var. = lag_samp2;
           output;
     end;
  end;
  run;

*** From SAS Online Documentation:
    For simple random sampling without replacement, if there is enough memory
    for it PROC SURVEYSELECT uses Floyds ordered hash table algorithm (see
    Bentley and Floyd (1987) and Bentley and Knuth (1986) for details).  If
    there is not enough memory available for Floyds algorithm, PROC
    SURVEYSELECT switches to the sequential algorithm of Fan, Muller, and
    Rezucha (1962), which requires less memory but might require more time to
    select the sample.
***;

proc surveyselect data     = &indata.(drop=&samp2var.)
                  method   = srs
                  sampsize = byvar_sum(keep=&byvars. _NSIZE_)
                  rep      = &num_psmps.
                  seed     = &seed.
                  out      = PSS_perm_Samps(drop=SamplingWeight SelectionProb)
                  noprint;
  strata &byvars.;
  run;

proc summary data=PSS_perm_Samps nway;
  class &byvars. replicate;
  var &permvar.;
output out=PSS_perm_sums(sortedby=&byvars. replicate keep=&byvars. replicate &permvar.) sum=;
  run;

proc transpose data=PSS_perm_sums out=PSS_perm_sums_t(rename=(_NAME_=permvar)) prefix=psmp;
  var &permvar.;
  by &byvars.;
  id replicate;
  run;

data &outdata.(keep=&byvars. permvar n_psamp num_psmps p_left p_right p_both)
     error
     ;
  merge PSS_perm_sums_t(in=insamps)
```



```
        byvar_sum(in=insummary)
        ;
  by &byvars.;
  if insamps & insummary then do;
     array psums[&num_psmps.] psmp1-psmp&num_psmps.;
     n_psamp = _NSIZE_;
     num_psmps = 1*&num_psmps.;
     p_left  = 0;
     p_right = 0;
     p_both  = 0;
     call sortn(of psums[*]);
     pmed = median(of psums[*]);
     pmean = mean(of psums[*]);

*** Efficiently handle extreme test sample values.;

     IF sumpvar<psums[1] THEN DO;
        p_left=0;
        p_right=num_psmps;
        p_both=0;
     END;
     ELSE IF sumpvar>psums[num_psmps] THEN DO;
        p_left=num_psmps;
        p_right=0;
        p_both=0;
     END;
     ELSE DO;

*** For non-extreme cases, start with shorter tail for less looping.;

        if pmed>=sumpvar then do;
           do z=1 to num_psmps;
              if sumpvar>=psums[z] then p_left+1;
              else do;
                 lastbinnum = z-1;
                 distance_left = pmean - psums[z-1];
                 leave;
              end;
           end;

*** Avoid loop for other (larger) p-value.
    If sumpvar equals last bin, p_right = 1 - p_left + lastbinsize.
    Otherwise, p_right = 1 - p_left.
***;
           if sumpvar = psums[lastbinnum] then do;
              lastbinsize=1;
              do k=lastbinnum to 1 by -1;
                 if psums[k]=psums[k-1] then lastbinsize+1;
                 leave;
              end;
              p_right = num_psmps - p_left + lastbinsize;
           end;
           else p_right = num_psmps - p_left;
        end;

        else do;
           do z=num_psmps to 1 by -1;
              if sumpvar<=psums[z] then p_right+1;
              else do;
                 lastbinnum = z+1;
                 distance_right = psums[z+1] - pmean;
```



```
               leave;
            end;
         end;

*** Avoid loop for other (larger) p-value.
    If psum equals last bin, p_left = 1 - p_right + lastbinsize.
    Otherwise, p_left = 1 - p_right.
***;
         if sumpvar = psums[lastbinnum] then do;
            lastbinsize=1;
            do k=lastbinnum to num_psmps;
               if psums[k]=psums[k+1] then lastbinsize+1;
               else leave;
            end;
            p_left = num_psmps - p_right + lastbinsize;
         end;
         else p_left = num_psmps - p_right;
       end;

*** Base 2-sided p-value on distance from mean of last (i.e. least extreme) bin
    of smaller p-value.  This is common practice.
***;

       if p_left<p_right then do;
          p_both = p_left;
          do z=num_psmps to 1 by -1;
             if (psums[z] - pmean) >= distance_left then p_both+1;
             else leave;
          end;
       end;
       else if p_left>p_right then do;
          p_both = p_right;
          do z=1 to num_psmps;
             if (pmean - psums[z]) >= distance_right then p_both+1;
             else leave;
          end;
       end;
       else p_both=num_psmps;

*** Account for possibility, due to psum=a particular bin value, that
    p_both>num_psmps.
***;
       p_both = min(p_both,num_psmps);

     END;

     p_left  = p_left  / num_psmps;
     p_right = p_right / num_psmps;
     p_both  = p_both  / num_psmps;

*** If CONTROL sample is smaller than TEST (which is atypical), reverse
*** p-values, as empirical distribution is mirror of itself.;

     if &samp2var.="C" then do;
        hold = p_left;
        p_left = p_right;
        p_right = hold;
     end;

     label permvar   = "Permuted Variable"
           n_psamp   = "Size of Permutation Samples"
```



```
           num_psmps = "# of Permutation Samples"
           p_left    = "Left p-value"
           p_right   = "Right p-value"
           p_both    = "Two-Tailed p-value"
           ;
     output &outdata.;
  end;
  else output error;
  run;

proc datasets lib=work memtype=data kill nodetails;
  run;

%MEND PROCSS;

%PROCSS(num_psmps = 1000,
        indata    = MFPTUS.pricing_data_2strata_100000,
        outdata   = MFPTUS.PROCSS_100000_2strata,
        byvars    = geography segment,
        samp2var  = cntrl_test,
        permvar   = price,
        seed      =
        );



*** PROC_MT ***;
*** PROC_MT ***;
*** PROC_MT ***;

%MACRO PROCMT(num_psmps=,
              indata=,
              outdata=,
              byvars=,
              samp2var=,
              permvar=,
              seed=,
              left_or_right=
              );


*** If user does not pass a value to the optional macro variable "seed,"
    generate a random seed and use it for all three proc multtests below
    (although SAS OnlineDoc says seed=-1 should work, it does not).
***;
%if %sysevalf(%superq(seed)=,boolean)
%then %let seed=%sysfunc(ceil(%sysevalf(1000000000*%sysfunc(ranuni(-1)))));

*** Un/comment test statements below to perform permutation tests based
    on different assumptions about the variance structure of the samples.;
***;

proc multtest    data = &indata.
              nsample = &num_psmps.
                 seed = &seed.
                  out = mt_output_results_2t(keep=&byvars. perm_p rename=(perm_p=xp_both))
                 permutation
                   noprint;
  by &byvars.;
  class &samp2var.;
*  test mean (&permvar. / DDFM=SATTERTHWAITE);
```



```
  test mean (&permvar.);
  run;


*** To make runtime results comparable to PROC NPAR1WAY, which provides only
    two-tailed p-value and the smaller of the right or left p-values, run
    two of the three PROC MULTTESTs and calculate the second tail as one
    minus the given tail, which will usually be very close to the actual
    value unless the data is highly discretized.
***;

proc multtest    data = &indata.
              nsample = &num_psmps.
                 seed = &seed.
%if %UPCASE(%sysfunc(compress(&left_or_right.)))=RIGHT %then %do;
                  out = mt_output_results_up(keep=&byvars. perm_p rename=(perm_p=xp_right))
%end;
%if %UPCASE(%sysfunc(compress(&left_or_right.)))=LEFT %then %do;
                  out = mt_output_results_low(keep=&byvars. perm_p rename=(perm_p=xp_left))
%end;
                 permutation
                   noprint;
  by &byvars.;
  class &samp2var.;
%if %UPCASE(%sysfunc(compress(&left_or_right.)))=RIGHT %then %do;
*  test mean (&permvar. / upper DDFM=SATTERTHWAITE);
  test mean (&permvar. / upper);
%end;
%if %UPCASE(%sysfunc(compress(&left_or_right.)))=LEFT %then %do;
*  test mean (&permvar. / lower DDFM=SATTERTHWAITE);
  test mean (&permvar. / lower);
%end;
  run;

proc summary data=&indata. nway;
  class &byvars. &samp2var.;
  var &permvar.;
  output out=byvar_frq(keep = _FREQ_ &byvars.)
                          n = toss
                          ;
  run;

%let last_byvar = %scan(&byvars.,-1);
data byvar_frq(keep=&byvars. xn_psamp sortedby=&byvars.);
  set byvar_frq(rename=(_FREQ_=xn_psamp));
  by &byvars.;
  retain lag_FREQ;
  if first.&last_byvar. then lag_FREQ = xn_psamp;
  else do;
     if xn_psamp<=lag_FREQ then output;
     else do;
        xn_psamp  = lag_FREQ;
        output;
     end;
  end;
  run;


data &outdata.(drop=xn_psamp xp_left xp_both)
     error
     ;
```



```
%if %UPCASE(%sysfunc(compress(&left_or_right.)))=LEFT %then %do;
  merge mt_output_results_low(in=inlow)
           mt_output_results_2t(in=in2t)
        byvar_frq(in=infrq)
        ;
  if in2t & inlow & infrq then do;
     format permvar $32.;
     permvar = "&permvar.";
     n_psamp = xn_psamp;
     num_psmps = 1*&num_psmps.;
     p_left = xp_left;
     p_right = .;
%end;
%if %UPCASE(%sysfunc(compress(&left_or_right.)))=RIGHT %then %do;
  merge mt_output_results_up(in=inup)
           mt_output_results_2t(in=in2t)
        byvar_frq(in=infrq)
        ;
  if in2t & inup & infrq then do;
     format permvar $32.;
     permvar = "&permvar.";
     n_psamp = xn_psamp;
     num_psmps = 1*&num_psmps.;
     p_right = xp_right;
     p_left = .;
%end;
     p_both = xp_both;
     label permvar   = "Permuted Variable"
           n_psamp   = "Size of Permutation Samples"
           num_psmps = "# of Permutation Samples"
           p_left    = "Left p-value"
           p_right   = "Right p-value"
           p_both    = "Two-Tailed p-value"
           ;
     output &outdata.;
  end;
  else output error;
  run;


proc datasets lib=work memtype=data kill nodetails;
  run;

%MEND PROCMT;

%PROCMT(num_psmps     = 1000,
        indata        = MFPTUS.pricing_data_2strata_100000,
        outdata       = MFPTUS.PROCMT_100000_2strata,
        byvars        = geography segment,
        samp2var      = cntrl_test,
        permvar       = price,
        seed          = ,
        left_or_right = left
        );



*** PROCNPAR ***;
*** PROCNPAR ***;
*** PROCNPAR ***;
```



```
%MACRO PROCNPAR(num_psmps=,
                indata=,
                outdata=,
                byvars=,
                samp2var=,
                permvar=,
                seed=
                );

*** If user does not pass a value to the optional macro variable "seed," use -1
    based on the time of day.
***;
%if %sysevalf(%superq(seed)=,boolean) %then %let seed=-1;

ods listing close;
proc npar1way    data = &indata.
               scores = data
                        ;
  var &permvar.;
  by &byvars.;
  class &samp2var.;
  exact scores=data / n=&num_psmps. seed=&seed.;
  ods output DataScoresMC = hold(keep = &byvars. Name1 Label1 nValue1
                                 where = (Label1 = "Estimate")
                                 );
  run;
ods listing;

proc transpose data = hold(drop=Label1)
                out = &outdata.(drop = _NAME_);
  by &byvars.;
  id Name1;
  var nValue1;
  run;

proc summary data=&indata. nway;
  class &byvars. &samp2var.;
  var &permvar.;
  output out=byvar_frq(keep = _FREQ_ &byvars. &samp2var.)
                       n = toss
                       ;
  run;

%let last_byvar = %scan(&byvars.,-1);
data byvar_frq(keep=&byvars. permvar n_psamp num_psmps &samp2var. sortedby=&byvars.);
  set byvar_frq;
  by &byvars.;
  format permvar $32.;
  retain permvar "&permvar." lag_FREQ lag_samp2 ;
  n_psamp = _FREQ_;
  num_psmps = 1*&num_psmps.;
  if first.&last_byvar. then do;
     lag_FREQ = n_psamp;
     lag_samp2  = &samp2var.;
  end;
  else do;
     if n_psamp<=lag_FREQ then output;
     else do;
        n_psamp  = lag_FREQ;
        &samp2var. = lag_samp2;
        output;
```



```
     end;
  end;
  run;

data &outdata.(drop=hold mcpl_data mcpr_data mcp2_data &samp2var.)
     error
     ;
  merge byvar_frq(in=infrq)
        &outdata.(in=inresults)
        ;
  by &byvars.;
  if inresults & infrq then do;
  p_left  = mcpl_data;
  p_right = mcpr_data;
  p_both  = mcp2_data;
  label permvar   = "Permuted Variable"
        n_psamp   = "Size of Permutation Samples"
        num_psmps = "# of Permutation Samples"
        p_left    = "Left p-value"
        p_right   = "Right p-value"
        p_both    = "Two-Tailed p-value"
        ;

*** If CONTROL sample is smaller than TEST (which is atypical), reverse
*** p-values, as empirical distribution is mirror of itself.;

   if &samp2var.="C" then do;
      hold = p_left;
      p_left = p_right;
      p_right = hold;
   end;
   output &outdata.;
   end;
   else output error;
  run;

%MEND PROCNPAR;

%PROCNPAR(num_psmps = 1000,
          indata    = MFPTUS.pricing_data_2strata_100000,
          outdata   = MFPTUS.PROCNPAR_100000_2strata,
          byvars    = geography segment,
          samp2var  = cntrl_test,
          permvar   = price,
          seed      =
          );




*** BEBB_SIM ***;
*** BEBB_SIM ***;
*** BEBB_SIM ***;


%MACRO BEBB_SIM(num_psmps=,
                indata=,
                outdata=,
                byvars=,
                samp2var=,
                permvar=,
```



```
                seed=
                );

*** If user does not pass a value to the optional macro variable "seed," use -1
    based on the time of day.
***;
%if %sysevalf(%superq(seed)=,boolean) %then %let seed=-1;


*** Obtain counts for each strata.;

proc summary data=&indata. nway;
  class &byvars. &samp2var.;
  var &permvar.;
  output out=byvar_sum(keep = _FREQ_ &byvars. &samp2var. sumpvar)
                        sum = sumpvar
                       ;
  run;

%let last_byvar = %scan(&byvars.,-1);
data byvar_sum_min(keep=tot_FREQ _FREQ_ &byvars. &samp2var. sumpvar sortedby=&byvars.);
  set byvar_sum;
  by &byvars.;
  retain lag_FREQ lag_sum lag_samp2;
  if first.&last_byvar. then do;
     lag_FREQ = _FREQ_;
     lag_sum  = sumpvar;
     lag_samp2  = &samp2var.;
  end;
  else do;
     tot_FREQ = sum(lag_FREQ,_FREQ_);
     if _FREQ_<=lag_FREQ then output;
     else do;
           _FREQ_  = lag_FREQ;
           sumpvar = lag_sum;
           &samp2var. = lag_samp2;
           output;
     end;
  end;
  run;

*** Slightly faster to use macro variable value strings and scan() than to merge
    _FREQ_ etc. onto the "indata" dataset, especially for large "indata"
    datasets.
*** ;
proc sql noprint;
  select _freq_ into :freqs separated by ' ' from byvar_sum_min;
  quit;
proc sql noprint;
  select tot_FREQ into :tot_FREQs separated by ' ' from byvar_sum_min;
  quit;
proc sql noprint;
  select sumpvar into :sumpvar separated by ' ' from byvar_sum_min;
  quit;
proc sql noprint;
  select &samp2var. into :samp2var separated by ' ' from byvar_sum_min;
  quit;


*** simultaneously create all permstrap samples and summarize results of each
    as the end of stratum is reached
```



```
***;

%let last_byvar = %scan(&byvars.,-1);

data &outdata.(keep=&byvars. permvar n_psamp num_psmps p_left p_right p_both);
  set &indata.(keep=&byvars. &permvar.) end=lastrec;
  by &byvars.;
  retain permvar "&permvar" n_psamp 0 num_psmps &num_psmps.;

  array smalln_counter{&num_psmps.} _TEMPORARY_;
  array psums{&num_psmps.} _TEMPORARY_;

  if first.&last_byvar. then do;

     byval_counter+1;

     _freq_ = 1*scan("&freqs.",byval_counter,' ');
     n_psamp = _FREQ_;
     tot_FREQ = 1*scan("&tot_FREQs.",byval_counter,' ');

     bigN_counter = tot_FREQ+1;
     do i=1 to num_psmps;
        smalln_counter[i] = _FREQ_;
        psums[i] = 0;
     end;
  end;
  bigN_counter+(-1);

  min_mac_res_inloop = &permvar.;

  seed=1*&seed.;
  if last.&last_byvar.~=1 then do i=1 to num_psmps;
     if ranuni(seed) <= smalln_counter[i]/bigN_counter then do;
        psums[i] = min_mac_res_inloop + psums[i];
        smalln_counter[i]+(-1);
     end;
  end;
  else do i=1 to num_psmps;
     if smalln_counter[i]>0 then psums[i] = min_mac_res_inloop + psums[i];
  end;
  if last.&last_byvar. then DO;


*** If CONTROL sample is smaller than TEST (which is atypical), reverse
*** order of empirical distribution.;

     sumpvar = 1*scan("&sumpvar.",byval_counter,' ');

     p_left  = 0;
     p_right = 0;
     p_both  = 0;
     call sortn(of psums[*]);
     pmed = median(of psums[*]);
     pmean = mean(of psums[*]);

*** Efficiently handle extreme test sample values.;

     IF sumpvar<psums[1] THEN DO;
        p_left=0;
        p_right=num_psmps;
        p_both=0;
```



```
     END;
     ELSE IF sumpvar>psums[num_psmps] THEN DO;
        p_left=num_psmps;
        p_right=0;
        p_both=0;
     END;
     ELSE DO;

*** For non-extreme cases, start with shorter tail for less looping.;

     if pmed>=sumpvar then do;
        do z=1 to num_psmps;
           if sumpvar>=psums[z] then p_left+1;
           else do;
              lastbinnum = z-1;
              distance_left = pmean - psums[z-1];
              leave;
           end;
        end;

*** Avoid loop for other (larger) p-value.
    If sumpvar equals last bin, p_right = 1 - p_left + lastbinsize.
    Otherwise, p_right = 1 - p_left.
***;
        if sumpvar = psums[lastbinnum] then do;
           lastbinsize=1;
           do k=lastbinnum to 1 by -1;
              if psums[k]=psums[k-1] then lastbinsize+1;
              leave;
           end;
           p_right = num_psmps - p_left + lastbinsize;
        end;
        else p_right = num_psmps - p_left;
     end;

     else do;
        do z=num_psmps to 1 by -1;
           if sumpvar<=psums[z] then p_right+1;
           else do;
              lastbinnum = z+1;
              distance_right = psums[z+1] - pmean;
              leave;
           end;
        end;

*** Avoid loop for other (larger) p-value.
    If psum equals last bin, p_left = 1 - p_right + lastbinsize.
    Otherwise, p_left = 1 - p_right.
***;
        if sumpvar = psums[lastbinnum] then do;
           lastbinsize=1;
           do k=lastbinnum to num_psmps;
              if psums[k]=psums[k+1] then lastbinsize+1;
              else leave;
           end;
           p_left = num_psmps - p_right + lastbinsize;
        end;
        else p_left = num_psmps - p_right;
     end;

*** Base 2-sided p-value on distance from mean of last (i.e. least extreme) bin
```



```
    of smaller p-value.  This is common practice.
***;

     if p_left<p_right then do;
        p_both = p_left;
        do z=num_psmps to 1 by -1;
           if (psums[z] - pmean) >= distance_left then p_both+1;
           else leave;
        end;
     end;
     else if p_left>p_right then do;
        p_both = p_right;
        do z=1 to num_psmps;
           if (pmean - psums[z]) >= distance_right then p_both+1;
           else leave;
        end;
     end;
     else p_both=num_psmps;

*** Account for possibility, due to psum=a particular bin value, that
    p_both>num_psmps.
***;
     p_both = min(p_both,num_psmps);
     END;

     p_left  = p_left  / num_psmps;
     p_right = p_right / num_psmps;
     p_both  = p_both  / num_psmps;

*** If CONTROL sample is smaller than TEST (which is atypical), reverse
*** p-values, as empirical distribution is mirror of itself.;

     if "C"=COMPRESS(UPCASE(scan("&samp2var.", byval_counter,' ')),' ') then do;
        hold = p_left;
        p_left = p_right;
        p_right = hold;
     end;

     label permvar   = "Permuted Variable"
           n_psamp   = "Size of Permutation Samples"
           num_psmps = "# of Permutation Samples"
           p_left    = "Left p-value"
           p_right   = "Right p-value"
           p_both    = "Two-Tailed p-value"
           ;
      output &outdata.;
  END;
  run;

proc datasets lib=work memtype=data kill nodetails;
  run;

%MEND BEBB_SIM;

%BEBB_SIM(num_psmps = 1000,
          indata    = MFPTUS.pricing_data_2strata_100000,
          outdata   = MFPTUS.BEBB_SIM_100000_2strata,
          byvars    = geography segment, samp2var=cntrl_test,
          permvar   = price,
          seed      =
           );
```


Appendix B

Table B1: Real and CPU Runtimes (minutes) of the Algorithms for Various *N*, #strata, and *m*

(**EX** = excessive, **CR** = crashed, sample size n = 1,000 for all)

			REAL						CPU					
N *(per stratum)*	# *strata*	*m*	OPDN	OPDN -Alt	PROCSS	PROC MT	PROC NPAR	Bebb -Sim	OPDN	OPDN -Alt	PROC SS	PROC MT	PROC NPAR	Bebb -Sim
10,000	2	500	0.01	0.01	0.12	0.05	0.05	0.03	0.01	0.01	0.06	0.04	0.03	0.03
100,000	2	500	0.02	0.03	0.37	1.24	0.29	0.29	0.01	0.01	0.27	1.23	0.28	0.28
1,000,000	2	500	0.10	0.11	3.21	16.87	4.49	2.80	0.05	0.06	2.43	16.78	4.45	2.78
10,000,000	2	500	0.77	0.75	34.23	196.31	CR	27.98	0.46	0.48	26.54	195.30	CR	27.85
10,000	6	500	0.02	0.03	0.24	0.13	0.10	0.09	0.02	0.02	0.18	0.12	0.08	0.09
100,000	6	500	0.04	0.05	1.09	3.67	0.87	0.84	0.04	0.04	0.81	3.65	0.85	0.84
1,000,000	6	500	0.31	0.35	9.60	50.36	13.65	8.39	0.16	0.17	7.11	50.13	13.44	8.35
10,000,000	6	500	2.08	2.18	94.02	EX	CR	83.95	1.41	1.51	69.76	EX	CR	83.25
10,000	12	500	0.04	0.06	0.51	0.25	0.17	0.18	0.04	0.04	0.35	0.24	0.16	0.18
100,000	12	500	0.08	0.10	2.34	7.23	1.78	1.74	0.08	0.08	1.75	7.18	1.74	1.73
1,000,000	12	500	0.45	0.47	18.61	99.33	27.61	16.73	0.34	0.36	13.93	98.85	27.39	16.65
10,000,000	12	500	3.73	3.97	198.46	EX	CR	168.05	2.82	3.14	152.42	EX	CR	166.61
10,000	2	1000	0.01	0.01	0.21	0.09	0.05	0.06	0.01	0.01	0.12	0.08	0.05	0.06
100,000	2	1000	0.02	0.02	0.78	2.45	0.55	0.57	0.02	0.02	0.61	2.43	0.55	0.56
1,000,000	2	1000	0.07	0.09	6.86	33.29	8.88	5.62	0.07	0.07	5.36	33.10	8.78	5.59
10,000,000	2	1000	0.77	0.78	61.95	387.61	CR	56.31	0.49	0.51	46.91	385.75	CR	56.10
10,000	6	1000	0.04	0.04	0.48	0.25	0.15	0.18	0.04	0.04	0.34	0.24	0.15	0.17
100,000	6	1000	0.06	0.07	2.24	7.35	1.67	1.72	0.06	0.06	1.68	7.32	1.65	1.71
1,000,000	6	1000	0.34	0.34	18.97	99.45	26.44	17.06	0.19	0.21	14.26	99.05	26.27	16.98
10,000,000	6	1000	2.14	2.24	182.30	EX	CR	169.39	1.46	1.55	137.71	EX	CR	168.44
10,000	12	1000	0.07	0.07	0.98	0.49	0.32	0.35	0.07	0.07	0.69	0.48	0.30	0.35
100,000	12	1000	0.13	0.13	4.27	14.70	3.50	3.39	0.12	0.13	3.21	14.63	3.44	3.37
1,000,000	12	1000	0.49	0.51	38.03	200.03	54.13	34.01	0.39	0.41	28.58	199.15	53.76	33.86
10,000,000	12	1000	3.81	3.99	379.94	EX	CR	337.96	3.01	3.18	275.03	EX	CR	335.85
10,000	2	2000	0.03	0.04	0.33	0.16	0.10	0.12	0.02	0.02	0.23	0.16	0.10	0.11
100,000	2	2000	0.04	0.04	1.44	4.86	1.10	1.12	0.04	0.04	1.08	4.84	1.08	1.12
1,000,000	2	2000	0.09	0.09	12.34	65.69	17.37	11.19	0.08	0.08	9.52	65.48	17.32	11.16
10,000,000	2	2000	0.78	0.81	134.49	EX	CR	111.84	0.49	0.51	105.60	EX	CR	111.53
10,000	6	2000	0.07	0.07	1.01	0.49	0.30	0.34	0.07	0.07	0.75	0.48	0.29	0.34
100,000	6	2000	0.12	0.12	4.47	14.50	3.29	3.39	0.11	0.11	3.40	14.45	3.26	3.38
1,000,000	6	2000	0.40	0.40	38.70	197.25	54.03	33.86	0.24	0.25	29.79	196.62	52.12	33.66
10,000,000	6	2000	2.11	2.32	461.33	EX	CR	336.08	1.50	1.64	372.84	EX	CR	334.72
10,000	12	2000	0.15	0.14	2.22	0.96	0.61	0.69	0.14	0.14	1.64	0.96	0.60	0.68
100,000	12	2000	0.22	0.22	9.82	29.20	6.79	6.72	0.21	0.22	7.60	28.98	6.74	6.70
1,000,000	12	2000	0.57	0.60	84.37	393.25	106.57	67.28	0.49	0.52	66.94	391.97	106.21	67.01
10,000,000	12	2000	3.79	4.07	EX	EX	CR	EX	3.02	3.33	EX	EX	CR	EX
7,500,000	2	2000	0.42	0.47	102.38	EX	148.37	84.48	0.37	0.40	79.23	EX	146.70	84.20
7,500,000	6	2000	1.31	1.52	317.34	EX	438.96	254.20	1.14	1.26	247.98	EX	434.72	253.33
25,000,000	12	500	1.36	1.58	EX	EX	CR	EX	1.13	1.21	EX	EX	CR	EX
50,000,000	12	500	2.84	3.38	EX	EX	CR	EX	2.26	2.42	EX	EX	CR	EX
100,000,000	12	500	6.12	6.41	EX	EX	CR	EX	4.64	4.85	EX	EX	CR	EX

Appendix C

For the convenience of the reader, a proof is provided below for the validity of the SRSWOR procedure used by OPDN, which is essentially the approach of Goodman & Hedetniemi (1982). The algorithm as implemented in OPDN is shown again below.

OPDN implementation of Goodman & Hedetniemi (1982) for Permutation Tests:

*** temp[] is the array filled with all the data values, for current stratum, of the variable being permuted
*** psums[] is the array containing the permutation sample statistic values for every permutation sample

```
do m = 1 to #permutation tests
   x ← 0
   tot_FREQ_hold ← # records in current stratum
   tot_FREQ ← tot_FREQ_hold
   do n = 1 to # records in smaller of Control and Treatment samples
      cell ← uniform random variate on 1 to tot_FREQ
      x    ← temp[cell] + x
      hold ← temp[cell]
      temp[cell] ← temp[tot_FREQ]
      temp[tot_FREQ] ← hold
      tot_FREQ ← tot_FREQ -1
   end;
   psums[m] ← x
end;
```

For a sampling algorithm to be a valid SRSWOR procedure, the probability of selecting any without-replacement sample of n items from a population of N items ($N > n$) needs to equal the probability of selecting any other without-replacement sample of n items, and that probability is:

Pr(drawing any particular sample of n items from larger group of N items) = $1 \Big/ \binom{N}{n} = n!(N-n)!/N!$ (C1)

because there are N – choose – n possible without-replacement samples.

Using the algorithm shown above, the probability of drawing the first item is $1/N$, and the probability of drawing the second item is $1/(N\text{-}1)$ and the probability of drawing the third item is $1/(N\text{-}2)$, and so on. Because each of these draws is independent of the others, the probability of drawing a sample of any n items is the product of these probabilities:

Pr(drawing any particular sample of n items from larger group of N items) =

$$\frac{1}{N} \bullet \frac{1}{(N-1)} \bullet \frac{1}{(N-2)} \bullet \cdots \bullet \frac{1}{(N-n+2)} \bullet \frac{1}{(N-n+1)}$$ * the number of permutations of these n items (because we do not care about the ordering of the n items, only that a particular set of n items is drawn), which is $n!$.

So Pr(drawing any particular sample of n items from larger group of N items) = $n! \Big/ \prod_{i=0}^{n-1}(N-i)$ (C2)

But in fact, (C1) = (C2), as shown below.

$$(\mathrm{C1}) = \frac{n!(N-n)!}{N!} = \frac{\left[n(n-1)(n-2)\cdots 2\cdot 1\right]\left[(N-n)(N-n-1)(N-n-2)\cdots 2\cdot 1\right]}{N(N-1)(N-2)\cdots(N-n+1)(N-n)(N-n-1)\cdots 2\cdot 1} =$$

$$= \frac{\left[n(n-1)(n-2)\cdots 2\cdot 1\right]}{N(N-1)(N-2)\cdots(N-n+1)} = \frac{n!}{\prod_{i=0}^{n-1}(N-i)} = (\mathrm{C2})$$

References

Bebbington, A. (1975), "A Simple Method of Drawing a Sample Without Replacement," *Journal of the Royal Statistical Society, Series C (Applied Statistics)*, Vol. 24, No. 1, 136.

Bentley, J. L. and Floyd, R. (1987), "A Sample of Brilliance," *Communications of the Association for Computing Machinery*, 30, 754–757.

Bentley, J. L. and Knuth, D. (1986), "Literate Programming," *Communications of the Association for Computing Machinery*, 29, 364–369.

Cassell, D. (2007), "Don't Be Loopy: Re-Sampling and Simulation the SAS® Way," Proceedings of the SAS Global Forum 2007 Conference, Cary, NC: SAS Institute Inc.

Ernvall, J., & O. Nevalainen, "An Algorithm for Unbiased Random Sampling," *The Computer Journal*, Vol.25 (1), p.45-47.

Fan, C. T., Muller, M. E., and Rezucha, I. (1962), "Development of Sampling Plans by Using Sequential (Item by Item) Selection Techniques and Digital Computers, " *Journal of the American Statistical Association*, 57, 387–402.

Fisher, Sir R.A. (1935), *Design of Experiments*, Edinburgh, Oliver & Boyd.

Goodman, S. & S. Hedetniemi (1977), *Introduction to the Design and Analysis of Algorithms*, McGraw-Hill, New York.

Mielke, P. & K. Berry (2001), *Permutation Methods: A Distance Function Approach*, Springer-Verlag, New York.

Opdyke, J.D. (2010), "Much Faster Bootstraps Using SAS®," *InterStat*, October, 2010.

Pesarin, F. (2001), *Multivariate Permutation Tests with Applications in Biostatistics*, John Wiley & Sons, Ltd., New York.

SAS Institute Inc. (2007), *SAS OnlineDoc® 9.2*, Cary, NC: SAS Institute Inc.

Tillé, Y. (2006), *Sampling Algorithms*, New York, NY, Springer.

