This addendum to the BIOS 669 simulation notes covers some additional useful topics related to simulation, some simple and some not so simple.

- A review of using PROC SURVEYSELECT to select random samples
- Creating fake IDs
- The RANPERK call routine (example: creating random dates)
- An extension of simulation notes section 4 to cover procedures that make multiple output data sets when you use BY

A review of using PROC SURVEYSELECT to select random samples

The "miscellaneous" chapter at the end of the BIOS 511 notes covers using PROC SURVEYSELECT to select either a simple or stratified random sample without replacement. Here is a reminder of that syntax being used to select 50 random METS participants from the DEMA data set. In the first case, the 50 participants are unrestricted. In the second case, 25 males and 25 females are selected.

```
* simple random sample without replacement;
proc surveyselect data=mets.dema_669 method=srs n=50 out=sample1 seed=58291;
run;


* stratified random sample without replacement, same number in each stratum;
* here the stratum is gender (dema2);
proc sort data=mets.dema_669 out=dema;
    by dema2;
run;
proc surveyselect data=dema method=srs n=25 out=sample2 seed=901132;
    strata dema2;
run;
```

Specifying a positive SEED value means that the same collection of records will be selected each time the code is run. This can be very helpful when this step is only a small part of a complicated program that you are working on.

Creating fake IDs

A common need is to give someone a copy of a data set in which real identifiers have been replaced by a fake ID. A nice, simple routine for creating such fake ID values involves assigning

each record a random number, sorting the data by that random number variable, and then constructing the fake ID values from the resulting _N_ values in the sorted data. Let's do this for the demog669 data used in the PROC REPORT unit.

```
data demog;
    set demog.demog669;
    call streaminit(55818);
    sorter = rand('UNIFORM');
run;

proc sort data=demog;
    by sorter;
run;

data for_distribution;
    FakeID = put(_N_,z4.);
    set demog;
    * drop subjid sorter;
run;
```

Here are the first 10 records of the resulting data set.

Obs	FakeID	subjid	trt	gender	race	age	sorter
1	0001	110	1	1	1	52	0.03503
2	0002	205	0	1	3	47	0.07830
3	0003	107	0	1	1	57	0.08377
4	0004	410	1	1	1	62	0.10014
5	0005	306	0	2	1	25	0.10615
6	0006	712	0	1	1	56	0.12081
7	0007	703	1	1	2	40	0.13669
8	0008	711	0	1	1	32	0.14889
9	0009	605	0	1	1	35	0.16108
10	0010	401	0	2	1	64	0.18871

You can see that the FakeID values are based on _N_. The purpose of the z4. format used in the PUT statement is to add leading 0's to fill in up to the specified length of 4. The FakeID assignment statement is located before the SET statement so that FakeID is the first variable in the new data set. I left variables subjid and sorter in the new data set for display purposes only

– so that you could see that the sort by the random number worked. Before making the final data set for distribution, of course you would want to drop those variables. Finally, it was important that a seed was used in the initial step that created the random numbers so that if I re-ran this program, I would get exactly the same new data set. When you assign fake IDs, you as the owner of the original data MUST have a way to know which original ID goes with which fake ID value.

The RANPERK call routine

The RANPERK call routine, which is available in either base SAS or SAS/IML, can be used to generate random permutations of *k* items that are chosen from a set with *n* elements. A call routine is sort of like a function, but it does not appear in an assignment statement. Rather, the returned value or values replace arguments used to call the routine.

The RANPERK syntax is

CALL RANPERK(seed, k, variable-1<, variable-2, …>);

The variable-1 … items make up the n elements, and depending on the magnitude of k, the first k values of the variable list are replaced with the permutation selections. That explains the oddity (to me!) in the code below.

A simple use (k always 1) could be to simulate random date values. Suppose we wanted to simulate 25 dates from the years 2010-2015 and were willing to ignore leap years.

```
data sim_dates;

    array m m1-m12 (1:12);
    array p p1-p28 (1:28);
    array q q1-q30 (1:30);
    array r r1-r31 (1:31);
    array y y1-y6  (2010:2015);

    seed = 449961;

    do i = 1 to 25;

        * this call will replace m1 with the randomly selected item from m1-
m12;
        * use that selection to set month;
        call ranperk(seed, 1, of m1-m12);
        month = m1;
```

```
        * based on what month was selected, select a day from the appropriate
range;
        if m1=2 then do;
            call ranperk(seed, 1, of p1-p28);
            day = p1;
        end;
        else if m1 in (4, 6, 9, 11) then do;
            call ranperk(seed, 1, of q1-q30);
            day = q1;
        end;
        else if m1 in (1, 3, 5, 7, 8, 10, 12) then do;
            call ranperk(seed, 1, of r1-r31);
            day = r1;
        end;

        * finally, select year;
        call ranperk(seed, 1, of y1-y6);
        year = y1;

        * construct date of birth variable from the current selections and
output the record;
        DOB = mdy(month,day,year;
        output;
    end;

    label DOB='Date of birth';
    format DOB date9.;
    keep DOB;
run;
```

Randomly-constructed dates

Obs	DOB
1	03DEC2012
2	07APR2012
3	17JUN2010
4	13MAY2011
5	13FEB2012
6	21SEP2011
7	18MAY2013

Obs	DOB
8	13AUG2010
9	06MAY2010
10	30MAY2013
11	26DEC2014
12	05OCT2013
13	22NOV2013
14	08MAR2010
15	11AUG2014
16	19JUN2012
17	05SEP2010
18	19MAY2010
19	26AUG2013
20	11APR2010
21	31MAR2011
22	15FEB2014
23	11DEC2012
24	19APR2011
25	28APR2013

An extension of section 4 to cover procedures that make multiple output data sets when you use BY

I've gotten email from several former students indirectly pointing out a hole in their SAS learning, and this is an appropriate place to fill it. The lack of knowledge appears when they need to run the same procedure many, many times with different input data sets (coded using BY) and then aggregate the results. The symptom of their problem is that their output data contains the results from only one of the runs (the last run). How can this problem be solved?

Well, this sounds kind of like section 4 of the simulation notes, where we estimated sampling distributions using a Monte Carlo method. A key in this section was that we initially generated *m* collections of data, with each *m* identified with a different SampleID, and then we used

SampleID as a BY variable in the procedure call where we computed statistics for each sample. Here's a reminder of that code:

```
/* 2. Compute statistics for each sample */
PROC MEANS DATA=SimUni NOPRINT;
    BY SampleID;
    VAR x;
    OUTPUT OUT=OutStatsUni MEAN=SampleMean;
RUN;
```

In this case, PROC MEANS was able to write the output from each SampleID to the same OutStatsUni data, which we could than analyze with PROC UNIVARIATE to get our sampling distribution.

But with ODS output data sets, the procedure doesn't always keep adding to the same output data set when you use BY – sometimes it does, and sometimes it doesn't. I tried the code above but writing to an ODS output data set instead of using the OUTPUT statement, and output data set ODSStatsUni DID contain the stacked results from all BY values. You could try this yourself:

```
ods output summary=ODSStatsUni;
proc means data=simuni;
    by sampleid;
    var x;
run;
```

But what if you try a more complicated procedure and SAS does not automatically stack all ODS results in the same data set? In that case you can use special ODS OUTPUT syntax to request that each iteration's output go to a data set with a patterned name and a consistent set of variables, and then you can easily stack the data sets in the end to get the collection of data that you want to analyze further to compare run results across samples.

I'm abashed to say that as I'm writing these notes I can't find an example of when this special syntax is needed. But I will describe it to you anyway. The sign that you need this special syntax is that the output data set produced is only for the last iteration through the loop.

Let's imagine that we are running PROC CRAZY and want to produce the CRAZYESTIMATES table as an output data set. Our BY variable is X. When we run the code directly below, we find that our output data set OUT contains only estimates for the final value of X.

```
ods select CrazyEstimates;
proc crazy data=all_reps;
    model Resp = cov1 cov2 cov3;
    ods output CrazyEstimates=out;
    by X;
run;
```

How can we change the code to get estimates for all values of X stacked up in an output data set? Change the PROC CRAZY code to the following, where (MATCH_ALL=LIST) has been added as shown:

```
ods select CrazyEstimates;
proc crazy data=all_reps;
    model Resp = cov1 cov2 cov3;
    ods output CrazyEstimates(match_all=list)=out;
    by X;
run;
```

This run actually produces as many output data sets as there are values of the BY variable X, and the data sets have the names OUT1 – OUTi (where i is the number of values of X). Magically, that thing called list (in match_all=list) is actually a macro variable that contains the expression OUT1 – OUTi. So if i were 100, &list would be OUT1 – OUT100. This allows us to follow the step above with

```
data allout;
    set &list;
run;
```

And now data set allout is the data set containing all of our stacked parameter estimates, ready for further analysis.