Paper 3372-2015

Leads and Lags: Static and Dynamic Queues in the SAS® DATA STEP

Mark Keintz, Wharton Research Data Services

ABSTRACT

From stock price histories to hospital stay records, analysis of time series data often requires use of lagged (and occasionally lead) values of one or more analysis variable. For the SAS® user, the central operational task is typically getting lagged (lead) values for each time point in the data set. While SAS has long provided a LAG function, it has no analogous "lead" function – an especially significant problem in the case of large data series. This paper will (1) review the lag function, in particular the powerful, but non-intuitive implications of its queue-oriented basis, (2) demonstrate efficient ways to generate leads with the same flexibility as the lag function, but without the common and expensive recourse to data re-sorting, and (3) show how to dynamically generate leads and lags through use of the hash object.

INTRODUCTION

Have you ever needed to report data from a series of monthly records comparing a given month's results to the same month in the prior year? Or perhaps you have a set of hospital admission and discharge records sorted by patient id within date and been asked to find length of stay for each patient visit (or even time to next admission for selected DRGs). These are just two examples of the frequent need SAS programmers have to find lag or lead values. All too often programmers find themselves in a multi-step process of sorting, joining, and resorting data just to retrieve data items from some prior or subsequent record in a data set. But it is possible to address all these tasks in single step programs, as will be shown below.

SAS has a LAG function (and a related DIF function) intended to provide data values (or differences) from preceding records in a data set. This presentation will show the benefit of the "queue-management" character of the LAG function in addressing both regular and irregular times series, whether grouped or ungrouped. Since there is no analogous "lead" function, later sections show how to use multiple SET statements to find leads.

The final section will address cases in which you don't know in advance how many lag series or lead series will be needed. This problem of "dynamic" lags and leads can't be effectively handled through clever use of the lag function or multiple SET statements. Instead a straightforward application of a hash object provides the best solution.

THE LAG FUNCTION – RETRIEVING HISTORY

The term "lag function" suggests retrieval of data via "looking back" by some user-specified number of periods or observations, and that is what most simple applications of the LAG function seem to do. For instance, consider the task of producing 1-month and 3-month price "returns" for this monthly stock price file (created from the **sashelp.stocks** data – see appendix for creation of data set SAMPLE1):

Table 1
Five Rows From SAMPLE1
(re-sorted from sashelp.stocks)

Obs	DATE	STOCK	CLS
1	01AUG86	IBM	$138.75
2	02SEP86	IBM	$134.50

Table 1
Five Rows From SAMPLE1
(re-sorted from sashelp.stocks)

Obs	DATE	STOCK	CLS
3	01OCT86	IBM	$123.62
4	03NOV86	IBM	$127.12
5	01DEC86	IBM	$120.00

This program below uses the LAG and LAG3 (3 record lag) functions to compare the current closing price (CLS) to its immediate and it "third prior" predecessors, needed to calculate one-month and three-month returns (RTN1 and RTN3):

Example 1: Simple Creation of Lagged Values

```
data simple_lags;
  set sample1;
  CLS1=lag(CLS);
  CLS3=lag3(CLS);
  if CLS1 ^=. then RETN1 = CLS/CLS1 - 1;
  if CLS3 ^=. then RETN3 = CLS/CLS3 - 1;
run;
```

Example 1 yields the following data in the first 5 rows:

Table 2
Example 1 Results from using LAG and LAG3

Obs	DATE	STOCK	CLS	CLS1	CLS3	RETN1	RETN3
1	01AUG86	IBM	$138.75	.	.	.	.
2	02SEP86	IBM	$134.50	138.75	.	-0.031	.
3	01OCT86	IBM	$123.62	134.50	.	-0.081	.
4	03NOV86	IBM	$127.12	123.62	138.75	0.028	-0.084
5	01DEC86	IBM	$120.00	127.12	134.50	-0.056	-0.108

LAGS ARE QUEUES – NOT "LOOK BACKS"

At this point LAG functions have all the appearance of simply looking back by one (or 3) observations, with the additional feature of imputing missing values when "looking back" beyond the beginning of the data set. But actually the lag function **instructs SAS to construct a fifo (first-in/first-out) queue** with (1) as many entries as the specified length of the lag, and (2) the queue elements initialized to missing values. Every time the lag function is executed, the oldest entry is retrieved (and removed) from the queue and a new entry (i.e. the current value) is added. Why is this queue management vs lookback distinction significant? That becomes evident when the LAG function is executed conditionally, as in the treatment of BY groups below.

As an illustration, consider observations 232 through 237 generated by Example 1 program, and presented in Table 3. This shows the last two cases for IBM and the first four for Intel. For the first Intel observation (obs 234), the lagged value of the closing stock price (CLS1=82.20) is taken from the IBM series. Of course, it should be a missing value, as should all the shaded cells. Lag values are not completely correct until the fourth Intel record (obs 237).

Table 3
The Problem of BY Groups for Lags

Obs	DATE	STOCK	CLS	CLS1	CLS3	RETN1	RETN3
232	01NOV05	IBM	$88.90	81.88	80.62	0.086	0.103
233	01DEC05	IBM	$82.20	88.90	80.22	-0.075	0.025
234	01AUG86	Intel	$23.00	*82.20*	*81.88*	*-0.720*	*-0.719*
235	02SEP86	Intel	$19.50	23.00	*88.90*	-0.152	*-0.781*
236	01OCT86	Intel	$20.25	19.50	*82.20*	0.038	*-0.754*
237	03NOV86	Intel	$23.00	20.25	23.00	0.136	0.000

The "natural" way to address this problem is to use a BY statement in SAS and (for the case of the single month return) avoid executing lags when the observation in hand is the first for a given stock. Example 2 is such a program (dealing with CLS1 only for illustration purposes), and its results are in Table 4.

Example 2: A "naïve" implementation of lags for BY groups

```
data naive_lags;
  set sample1;
  by stock;
  if not(first.stock) then CLS1=lag(CLS);
  else CLS1=.;
  if CLS1 ^=. Then RETN1= CLS/CLS1 - 1;
  format ret: 6.3;
run;
```

Table 4
Results of
"if not(first.stock) then CLS1=lag(CLS)"

Obs	STOCK	DATE	CLS	CLS1	RETN1
232	IBM	01NOV05	$88.90	81.88	0.086
233	IBM	01DEC05	$82.20	88.90	-0.075
234	Intel	01AUG86	$23.00	.	.
235	Intel	02SEP86	$19.50	*82.20*	*-0.763*
236	Intel	01OCT86	$20.25	19.50	0.038
237	Intel	03NOV86	$23.00	20.25	0.136

This fixes the first Intel record, setting both CLS1 and RETN1 to missing values. But look at the second Intel record (Obs 235). CLS1 has a value of 82.20, taken not from the beginning of the Intel series, but rather from the end

of the IBM series, generating an erroneous value for RETN1 as well. In other words, **CLS1 did not come from the prior record (lookback), but rather it came from the queue, whose contents were most recently updated in the last IBM record**.

LAGS FOR BY GROUPS (IFN AND IFC ALWAYS CALCULATE BOTH OUTCOMES)

The fix is eazy, unconditionally execute a lag, and then reset the result when necessary. This is usually done in two statements, but Example 3 shows a more compact solution (described by Howard Schrier – see References). It uses the IFN function instead of an IF statement - because IFN executes the lag function embedded in its second argument regardless of the status of first.stock (the condition being tested). Of course, IFN returns the lagged value only when the tested condition is true. Accommodating BY groups for lags longer than one period simply requires comparing lagged values of the BY-variable to the current values ("lag3(stock)=stock"), as in the "CLS3=" statement below.

Example 3: A robust implementation of lags for BY groups

```
data bygroup_lags;
  set sample1;
  by stock;
  CLS1 = ifn(not(first.stock),lag(CLS),.);
  CLS3 = ifn(lag3(stock)=stock,lag3(CLS),.) ;
  if CLS1 ^=. then RETN1 = CLS/CLS1 - 1;
  if CLS3 ^=. then RETN3 = CLS/CLS3 - 1;
  format ret: 6.3;
run;
```

The data set BYGROUP_LAGS now has missing values for the appropriate records at the start of the Intel monthly records.

Table 5
Result of Robust Lag Implementation for BY Groups

Obs	STOCK	DATE	CLS	CLS1	CLS3	RETN1	RETN3
232	IBM	01NOV05	$88.90	81.88	80.62	0.086	0.103
233	IBM	01DEC05	$82.20	88.90	80.22	-0.075	0.025
234	Intel	01AUG86	$23.00	.	.	.	.
235	Intel	02SEP86	$19.50	23.00	.	-0.152	.
236	Intel	01OCT86	$20.25	19.50	.	0.038	.
237	Intel	03NOV86	$23.00	20.25	23.00	0.136	0.000

MULTIPLE LAGS MEANS MULTIPLE QUEUES – A WAY TO MANAGE IRREGULAR SERIES

While BY-groups benefit from *avoiding* conditional execution of lags, there are times when conditional lags are the best solution. In the data set SALES below (see Appendix for its generation) are monthly sales by product. But a

given product×month combination is only present when sales are reported. As a result each month has a varying number of records, depending on which product sales were reported. Table 6 shows 5 records for month 1, but only 4 records for month 2. Unlike the data in Sample 1, this time series is not regular, so comparing (say) sales of product B in January (observation 2) to February (7) would imply a LAG5, while comparing March (10) to February would need a LAG3.

Table 6
Data Set SALES (first 13 obs)

OBS	MONTH	PROD	SALES
1	1	A	4
2	1	B	20
3	1	C	11
4	1	D	7
5	1	X	20
6	2	A	16
7	2	B	12
8	2	D	2
9	2	X	8
10	3	B	14
11	3	C	4
12	3	D	7
13	3	X	6

The solution to this task is to use conditional lags, with one queue for each product. The program to do so is surprisingly simple:

Example 4: LAGS for Irregular Time Series[1]

```
data irregular_lags;
  set sales;
  select (product);
    when ('A') change_rate=(sales-lag(sales))/(month-lag(month));
    when ('B') change_rate=(sales-lag(sales))/(month-lag(month));
    when ('C') change_rate=(sales-lag(sales))/(month-lag(month));
    when ('D') change_rate=(sales-lag(sales))/(month-lag(month));
    otherwise;
  end;
run;
```

Example 4 generates four pairs of lag queues, one pair for each of the products A through D. Each invocation of a lag maintains a separate queue, resulting in eight distinct queues in Example 4. Because a given queue is updated only when the specified product is in hand (the "when" clauses), the output of the lag function must come from an earlier observation having the same PRODUCT as the current observation, no matter how far back it may be in the data set. Note that, for most functions, clean program design would collapse the four "when" conditions into one, such as

[1] A more compact expression would have used DIF instead of LAG, as in "change_rate=dif(sales)/dif(month).

```
when('A','B','C','D') change_rate= function(sales)/function(month);
```

Succumbing to this temptation would misuse the queue-management features of LAG (and DIF).

LEADS – USING EXTRA SET STATEMENTS TO LOOK AHEAD

SAS does not offer a lead function. As a result many SAS programmers sort a data set in descending order and then apply lag functions to create lead values. Often the data are sorted a second time, back to original order, before any analysis is done. For large data sets, this is a costly three-step process.

There is a much simpler way, through the use of extra SET statements in combination with the FIRSTOBS parameter and other elements of the SET statement. The following program generates both a one-month and three-month lead of the data from Sample1:

Example 5: Simple generation of one-record and three-record leads

```
data simple_leads;
  set sample1;

  if eof1=0 then
    set sample1 (firstobs=2 keep=CLS rename=(CLS=LEAD1)) end=eof1;
  else lead1=.;

  if eof3=0 then
    set sample1 (firstobs=4 keep=CLS rename=(CLS=LEAD3)) end=eof3;
  else lead3=.;

run;
```

The use of multiple SET statements to generate leads makes use of two "features" of the SET statement and two data set name parameters:

1. SET Feature 1: Multiple SET statements reading the same data set do not read from a single series of records (unlike a collection of INPUT statements). Instead they produce separate streams of data (much like each LAG function generating its own independent queue). It is as if the three SET statements above were reading from three different data sets. In fact the log from Example 5 displays these notes reporting three incoming streams of data:

   ```
   NOTE: There were 699 observations read from the data set WORK.SAMPLE1.
   NOTE: There were 698 observations read from the data set WORK.SAMPLE1.
   NOTE: There were 696 observations read from the data set WORK.SAMPLE1.
   ```

2. Data Set Name Parameter 1 (FIRSTOBS): Using the "FIRSTOBS=" parameter provides a way to "look ahead" in a data set. For instance the second SET has "FIRSTOBS=2 (the third has "FIRSTOBS=4"), so that it starts reading from record 2 (4) while the first SET statement is reading from record 1. This provides a way to synchronize leads with any given "current" record.
3. Data Set Name Parameter 2 (RENAME, and KEEP): All three SET statements read in the same variable (CLS), yet a single variable can't have more than one value at a time. In this case the original value would be overwritten by the subsequent SETs. To avoid this problem CLS is renamed (to LEAD1 and LEAD3) in the additional SET statements, resulting in three variables: CLS (from the "current" record, LEAD1 (one pe-

riod lead) and LEAD3 (three period lead). To avoid overwriting any other variables, only variables to be renamed should be in the KEEP= parameter.

4. SET Feature 2 (EOF=): Using the "IF EOFx=0 then set" in tandem with the END=EOFx parameter avoids a premature end of the data step. Ordinarily a data step with three SET statements stops when any one of the SETs attempts to go beyond the end of input. In this case, the third SET ("FIRSTOBS=4") would stop the DATA step even though the first would have 3 unread records remaining. The answer is to prevent unwanted attempts at reading beyond the end of data. The "end=" parameter generates a dummy variable indicating whether the record in hand is the last incoming record. The program can test its value and stop reading each data input stream when it is exhausted. That's why the log notes above report differing numbers of observations read.

The last 4 records in the resulting data set are below, with lead values as expected:

Table 7:
Simple Leads

Obs	STOCK	DATE	CLS	LEAD1	LEAD3
696	Microsoft	01SEP05	$25.73	$25.70	$26.15
697	Microsoft	03OCT05	$25.70	$27.68	.
698	Microsoft	01NOV05	$27.68	$26.15	.
699	Microsoft	01DEC05	$26.15	.	>

LEADS FOR BY GROUPS

Just as in the case of lags, generating lead in the presence of BY group requires a little extra code. A single-period lead is relatively easy – if the current record is the last in a BY group, reset the lead to missing. That test is shown after the second SET statement below. But in the case of leads beyond one period, a little extra is needed – namely a test comparing the current value of the BY-variable (STOCK) to its value in the lead period. That's done in the code below for the three-period lead by reading in (and renaming) the STOCK variable in the third SET statement, comparing it to the current STOCK value, and resetting LEAD3 to missing when needed.

Example 6: Generating Leads for By Groups

```
data bygroup_leads;
  set sample1;
  by stock;

  if eof1=0 then
    set sample1 (firstobs=2 keep=CLS rename=(CLS=LEAD1)) end=eof1;
  if last.stock then lead1=.;
  if eof3=0 then
    set sample1 (firstobs=4 keep=stock CLS
           rename=(stock=stock3 CLS=LEAD3)) end=eof3;
  if stock3 ^= stock then lead3=.;

  drop stock3;
run;
```

The result for the last four IBM observations and the first three Intel observations are below, with LEAD1 set to missing for the final IBM, and LEAD3 for the last 3 IBM observations.

Table 8
Leads With By Groups

Obs	STOCK	DATE	CLS	LEAD1	LEAD3
230	IBM	01SEP05	$80.22	$81.88	$82.20
231	IBM	03OCT05	$81.88	$88.90	.
232	IBM	01NOV05	$88.90	$82.20	.
233	IBM	01DEC05	$82.20	.	.
234	Intel	01AUG86	$23.00	$19.50	$23.00
235	Intel	02SEP86	$19.50	$20.25	$21.00

GENERATING IRREGULAR LEAD "QUEUES" – USING PRECEDENCE OF WHERE OVER FIRSTOBS

Generating lags for the SALES data above required the utilization of multiple queues – a LAG function for each product, resolving the problem of varying "distances" between successive records for a given product. Developing leads for such irregular time series requires the same approach – one "queue" for each product. Similar to the use of several LAG functions to manage separate queues, generating leads for irregular times series requires a collection of SET statements – each "filtered" by an appropriate WHERE condition. The relatively simple program below demonstrates:

Example 7: Generating Leads for Irregular Time Series

```
data irregular_leads;
  set sales;

  select (product);
   when ('A') if eofa=0 then set sales (where=(product='A') firstobs=2
              keep=product sales rename=(sales=LEAD1)) end=eofa;

   when ('B') if eofb=0 then set sales (where=(product='B') firstobs=2
              keep=product sales rename=(sales=LEAD1)) end=eofb;

   when ('C') if eofc=0 then set sales (where=(product='C') firstobs=2
              keep=product sales rename=(sales=LEAD1)) end=eofc;

   when ('D') if eofd=0 then set sales (where=(product='D') firstobs=2
              keep=product sales rename=(sales=LEAD1)) end=eofd;

   otherwise lead1=.;
  end;
run;
```

The logic of Example 7 is straightforward. If the current product is 'A' [WHEN('A')] and the stream of PRODUCT "A" records is not exhausted (if eofa=0), then read the next product "A" record, renaming its SALES variable to LEAD1. The technique for reading only product "A" records is to use the "WHERE=" data set name parameter. Most important to this technique is the fact that the ***WHERE parameter is honored prior to the FIRSTOBS parameter***. So "FIRSTOBS=2" means the second product "A" record, not the second record overall.

The first 8 product A and B records are as follows, with the LEAD1 value always the same as the SALES values for the next identical product.

Table 9
Lead produced by Independent Queues

Obs	MONTH	PRODUCT	SALES	LEAD1
1	1	A	4	16
2	1	B	20	12
6	2	A	16	15
7	2	B	12	14
10	3	B	14	15
14	4	A	15	3
15	4	B	15	16
19	5	A	3	20

Of course this approach could become burdensome if the intent is to produce not only a 1 period lead for each product, but also a 2 period and 3 period lead. Although this might at first suggest issuing 12 SET statements (4 products and 3 differing lead levels), one can still work with one SET per product. A complete program to address that issue is in Appendix 2. An abridged version follows.

Example 8: Shorter Leads = Lag of Longer Leads

```
data irregular_leads2 (drop=i);
  set sales;

  select (product);

    when ('A') do i=1 to ifn(lag(product) =' ',3,1);
      if eofa=0 then set sales (where=(product='A') firstobs=2
               keep=product sales rename=(sales=LEAD3))  end=eofa;
      else lead3=. ;
      lead2=lag(lead3);
      lead1=lag2(lead3);
    end;

    when ('B') ...
    when ('C') ...
    when ('D') ...

    otherwise lead3=. ;
  end;
run;
```

The approach above gets the 3-period lead (LEAD3), and then simply applies lag functions to it to produce shorter leads. To do this of course, the program must initially read, for each PRODUCT, observations 2, 3, and 4 initially, and subsequently once for each reference record. This is done with the

```
do i=1 to ifn(lag(product) =' ',3,1);
```

which merely utilizes the fact the lag(product) queue is initialized to a missing value. Consequently the ifn condition is true (and 3 observations are read) only for the first instance of each SET statement.

DYNAMICALLY GENERATING LAGS AND LEADS

In earlier sections on generating lags (or leads) for irregular series, a LAG function (or SET statement) was issued for each product. But what if you don't know in advance how many products were in the data set? If you don't, then a static approach is no longer sufficient. A technique is needed for dynamically establishing a time series for each product as it is discovered in the data set. Hash objects fit this need. Their content, unlike ordinary variables in the data set, persists from record to record, and can be expanded and revised dynamically as well.

DYNAMIC LAGS – DROP THE LAG FUNCTION

In the case of lags, this problem is most readily addressed by use of two hash objects, one (CURRSEQ) for maintaining a running count of records for each product, and one (LAGS) to hold data for later retrieval as lagged values. The program below, after declaring these two objects, has three elements:

Example 9: Dynamically Generating Lags

```
data dynamic_lags (drop=_:);
  set sales;
  _seq=0;

  if _n_=1 then do;
    /* Hash object CURRSEQ: track current count by product     */
    declare hash currseq();
      currseq.definekey('product');
      currseq.definedata('_seq');
      currseq.definedone();

    _sl=. ;  _ml=. ;                  /* Lagged Sales and Month */
    /* hash object LAGS: _SL and _ML keyed by PRODUCT and _SEQ */
    declare hash lags();
      lags.definekey('product','_seq');
      lags.definedata('_sl','_ml');
      lags.definedone();
  end;

  /* Part 1: Update _SEQ for the current PRODUCT                */
  _rc=currseq.find();      /* Retrieve _SEQ for current PRODUCT */
  _seq=_seq+1;
  currseq.replace();       /* Replace old value with new        */

  /* Part 2: Add SALES and MONTH (as data elements _SL and _ML) */
  /* to the LAGS object, keyed on PRODUCT and _SEQ              */
  lags.add(key:product,key:_seq,data:sales,data:month);
```

```
  /* Part 3: Retrieve historical value and make the monthly rate*/
  _rc = lags.find(key:product,key:_seq-1);
  if _rc=0 then change_rate=(sales-_sl)/(month-_ml);
run;
```

In the program above, the CURRSEQ hash object maintains only one item (i.e. one "row") per product, containing the variable _SEQ, the current number of records read for that product. In part 1, the "currseq.find()" method retrieves _SEQ for the current product (only if the product is already in CURRSEQ, otherwise _SEQ is unmodified). _SEQ is then incremented and replaced into CURRSEQ with every incoming record for the product.

The LAGS object (keyed on PRODUCT and _SEQ), gets one item for each incoming record, and contains a running history of the variables _SL and _ML (so named to allow later retrieval without overwriting the current value of SALES and MONTH). In part 2, the new data is added to LAGS ("lags.add(…)"). The use of the two "data:.." arguments assigns the values of SALES and MONTH to _SL and _ML as they are stored in LAGS.

The third major element of this program simply checks whether the product in hand is beyond its first occurrence, and if so, retrieves lagged values from LAGS and generates the needed variable. Note the "key:seq-1" argument tells lag.find() to retrieve the item (with variables _SL and _ML) corresponding to a single period lag. An n-period lag would require nothing more than the argument "key:seq-n".

DYNAMIC LEADS – ORDERED HASH OBJECT PLUS HASH ITERATOR

In the case of leads, if you don't know how many PRODUCTs are expected, building a history record-by-record as in the case of lags won't work. Instead the hash object needs to have a complete history. That problem is solved in Example 10 below, a data step with two parts:

Example 10: Dynamically Generating Leads

```
data dynamic_leads (drop=_:) ;

  /* Part 1: Populate LEADS hash object, and assign an iterator */
  if _n_=1 then do;
    if 0 then set sales (keep=product sales month
              rename=(product=_PL sales=_SL month=_ML));
    declare hash leads
      (dataset:'SALES (keep=PRODUCT MONTH SALES
                       rename=(PRODUCT=_PL MONTH=_ML SALES=_SL)'
      ,ordered:'A') ;


      leads.definekey('_PL','_ML');
      leads.definedata('_PL','_ML','_SL');
      leads.definedone();
    declare hiter li ('leads');
  end;

  /* Part 2: Read the dataset and retrieve leads */
  set sales ;
  /* Point hash iterator at hash item for current product/month */
  li.setcur(key:PRODUCT,key:MONTH);
  _rc=li.next();         /* Advance one item in the hash object */
  if _rc=0 and _PL=PRODUCT then changerate=(_SL-sales)/(_ML-month);
 run;
```

Before reading the data set record-by-record, the leads object is fully populated with all the PRODUCT, SALES and MONTH values (renamed to _PL, _SL and _ML), stored in ascending order by _PL and _ML. Because the object is ordered, moving from one item to the next within the hash is equivalent to traversing the series for a single PRODUCT. The hash iterator li supports this process.

In part 2, a record is read, and the li.setcur method points the iterator at the data item in the object corresponding to the record. Moving the iterator forward by one item ("_rc=li.next()") retrieves the _PL, _ML, and _SL values for a one period lead.

CONCLUSION

While at first glance the queue management character of the LAG function may seem counterintuitive, this property offers robust techniques to deal with a variety of situations, including BY groups and irregularly spaced time series. The technique for accommodating those structures is relatively simple. In addition, the use of multiple SET statements produces the equivalent capability in generating leads, all without the need for extra sorting of the data set.

The more complex problem is how to address irregular time series dynamically, i.e. how to handle situations in which the user does not know how many queues will be needed. For this situation, the construction of hashes used as tools to "look up" lagged (or lead) data for each queue provides a graceful solution.

REFERENCES

Matlapudi, Anjan, and J. Daniel Knapp (2010) "Please Don't Lag Behind LAG". In the Proceedings of the North East SAS® Users Group.

Schreier, Howard (undated) "Conditional Lags Don't Have to be Treacherous". URL as or 7/1/2013: http://www.howles.com/saspapers/CC33.pdf.

ACKNOWLEDGMENTS

CONTACT INFORMATION

This is a work in progress. Your comments and questions are valued and encouraged. Please contact the author at:

Author Name: Mark Keintz
Email: mkeintz@wharton.upen.edu

APPENDIX: CREATION OF SAMPLE DATA SETS FROM THE SASHELP LIBRARY

```
/*Sample 1: Monthly Stock Data for IBM, Intel, Microsoft for Aug 1986 - Dec 2005 */
proc sort data=sashelp.stocks (keep=stock date close)
  out=sample1 (rename=(close=cls)) ;
  by stock date;
run;


/*Sample 2: Irregular Series: Monthly Sales by Product,  */
data SALES;
  do MONTH=1 to 24;
    do PRODUCT='A','B','C','D','X';
      if ranuni(09481098)< 0.9 then do;
        SALES =ceil(20*ranuni(10598067));
        output;
      end;
    end;
  end;
run;
```

APPENDIX 2: EXAMPLE PROGRAM OF MULTI-PERIOD LEADS FOR IRREGULAR TIME SERIES

```
/*To get leads for each product of interest ('A','B','C','D') for three different
periods (1, 2, and 3), you might issue 12 SET statements, as per the section on
leads for irregular series.  However, only 4 SET statements are needed when com-
bined with lag functions, as below */


data irregular_leads2 (drop=_:
   label='1, 2, & 3 period leads for products A, B, C, and D');
  set sales;

  select (product);

    when ('A') do _I=1 to ifn(lag(product) =' ',3,1);
      if eofa=0 then set sales (where=(product='A') firstobs=2
               keep=product sales rename=(sales=LEAD3))  end=eofa;
      else lead3=. ;
      LEAD2=lag(lead3);
      LEAD1=lag2(lead3);
    end;

    when ('B') do _I=1 to ifn(lag(product) =' ',3,1);
      if eofb=0 then set sales (where=(product='B') firstobs=2
               keep=product sales rename=(sales=LEAD3))  end=eofb;
      else lead3=. ;
      lead2=lag(lead3);
      lead1=lag2(lead3);
    end;

    when ('C') do _I=1 to ifn(lag(product) =' ',3,1);
      if eofc=0 then set sales (where=(product='C') firstobs=2
               keep=product sales rename=(sales=LEAD3))  end=eofc;
      else lead3=. ;
      lead2=lag(lead3);
      lead1=lag2(lead3);
    end;

    when ('D') do _I=1 to ifn(lag(product) =' ',3,1);
      if eofd=0 then set sales (where=(product='D') firstobs=2
               keep=product sales rename=(sales=LEAD3))  end=eofd;
      else lead3=. ;
      lead2=lag(lead3);
      lead1=lag2(lead3);
    end;

    otherwise lead3=. ;
  end;

run;
```