LEADS AND LAGS IN SAS®
Mark Keintz, Wharton Research Data Services, University of Pennsylvania

ABSTRACT
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 non-intuitive implications of its queue-oriented basis and (2) demonstrate efficient ways to generate leads, without the common recourse to data re-sorting.

INTRODUCTION
SAS has a LAG function (and a related DIF) function intended to provide data values from preceding records in a data set. If may be something as simplistic as last month’s price in a monthly stock price data set (a regular time series), or as variable as the most recent sales volume of a product which sold only occasionally (irregular time series). The presentation will show the benefit of the “queue-management” orientation of the LAG function in addressing time series that are simple, sorted, or irregular. In the absence of a “lead” function, this presentation will show simple SAS scripts to produce lead values under similar conditions.

THE LAG FUNCTION – RETRIEVING HISTORY
The term “lag function” suggests the retrieval of data via “looking back” some user-specified number of periods or observations. 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>
<thead>
<tr>
<th>Obs</th>
<th>DATE</th>
<th>STOCK</th>
<th>CLOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01AUG86</td>
<td>IBM</td>
<td>$138.75</td>
</tr>
<tr>
<td>2</td>
<td>02SEP86</td>
<td>IBM</td>
<td>$134.50</td>
</tr>
<tr>
<td>3</td>
<td>01OCT86</td>
<td>IBM</td>
<td>$123.62</td>
</tr>
<tr>
<td>4</td>
<td>03NOV86</td>
<td>IBM</td>
<td>$127.12</td>
</tr>
<tr>
<td>5</td>
<td>01DEC86</td>
<td>IBM</td>
<td>$120.00</td>
</tr>
</tbody>
</table>

This program below uses the LAG and LAG3 (3 record lag) functions to compare the current close to its immediate predecessor and it “third prior” predecessor:
Example 1: Simple Creation of Lagged Values

data example1;
    set sample1;
    close_1=lag(close);
    close_3=lag3(close);
    if close_1 ^=. then return_1 = close/close_1 - 1;
    if close_2 ^=. then return_3 = close/close_3 - 1;
run;

which yields the following data in the first 5 rows:

<table>
<thead>
<tr>
<th>Obs</th>
<th>DATE</th>
<th>STOCK</th>
<th>CLOSE</th>
<th>CLOSE_1</th>
<th>CLOSE_3</th>
<th>RETURN_1</th>
<th>RETURN_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01AUG86</td>
<td>IBM</td>
<td>$138.75</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>02SEP86</td>
<td>IBM</td>
<td>$134.50</td>
<td>138.75</td>
<td></td>
<td>-0.031</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>01OCT86</td>
<td>IBM</td>
<td>$123.62</td>
<td>134.50</td>
<td></td>
<td>-0.081</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>03NOV86</td>
<td>IBM</td>
<td>$127.12</td>
<td>123.62</td>
<td>134.50</td>
<td>0.028</td>
<td>-0.084</td>
</tr>
<tr>
<td>5</td>
<td>01DEC86</td>
<td>IBM</td>
<td>$120.00</td>
<td>127.12</td>
<td>134.50</td>
<td>-0.056</td>
<td>-0.108</td>
</tr>
</tbody>
</table>

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 length of the lag period, 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 is added. The significance of this distinction 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 (CLOSE_1=82.20) is taken from the IBM series. Of course, it should be a missing value, as should all the shaded cells.

<table>
<thead>
<tr>
<th>Obs</th>
<th>DATE</th>
<th>STOCK</th>
<th>CLOSE</th>
<th>CLOSE_1</th>
<th>CLOSE_3</th>
<th>RETURN_1</th>
<th>RETURN_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>232</td>
<td>01NOV05</td>
<td>IBM</td>
<td>$88.90</td>
<td>81.88</td>
<td>80.62</td>
<td>0.086</td>
<td>0.103</td>
</tr>
<tr>
<td>233</td>
<td>01DEC05</td>
<td>IBM</td>
<td>$82.20</td>
<td>88.90</td>
<td>80.22</td>
<td>-0.075</td>
<td>0.025</td>
</tr>
<tr>
<td>234</td>
<td>01AUG86</td>
<td>Intel</td>
<td>$23.00</td>
<td>82.20</td>
<td>81.88</td>
<td>-0.720</td>
<td>-0.719</td>
</tr>
</tbody>
</table>
The “natural” way to address this problem is to use a BY statement in SAS and avoid executing lags when the observation in hand is the first for a given stock. Example 2 is such a program (dealing with CLOSE_1 only for illustration purposes), and its results are in Table 4.

Example 2: A “naïve” implementation of lags for BY groups

data example2;
  set sample1;
  by stock;
  if first.stock=0 then close_1=lag(close);
  else close_1=.;
  if close_1 ^=. Then return_1= close/close_1;
  format ret: 6.3;
run;

This fixes the first Intel record, setting both CLOSE_1 and RETURN_1 to missing values. But look at the second Intel record (Obs 235). CLOSE_1 has a value of 82.20, taken not from the first Intel record, but rather from the last IBM record, generating an erroneous value for RETURN_1 as well. In other words, CLOSE_1 did not come from the prior record, but rather it came from the queue, whose contents were most recently updated prior to the first Intel record.

LAGS FOR BY GROUPS

The usual fix is to unconditionally execute a lag, and then reset the result when necessary. Example 3 shows just such a solution (described by Howard Schrier – see References). It uses the IFN function instead of an IF statement - because IFN executes the embedded lag regardless of the status of first.stock (the condition being tested).
IFN keeps the lagged value only when the tested condition is true. Accommodating BY groups for lags longer than one period requires comparing lagged values of the BY-variable to the current values ("lag3(stock)=stock").

**Example 3: A robust implementation of lags for BY groups**

```
data example3;
  set sample1;
  by stock;
  close_1 = ifn(first.stock=0,lag(close),.);
  close_3 = ifn(lag3(stock)=stock,lag3(close),.);
  if close_1 ^=. then RETURN_1 = close/close_1 - 1;
  if close_3 ^=. then RETURN_3 = close/close_3 - 1;
  format ret: 6.3;
run;
```

The data set EXAMPLE3 now has missing values for the appropriate records at the start of the Intel monthly records.

**Table 5**

<table>
<thead>
<tr>
<th>Obs</th>
<th>STOCK</th>
<th>DATE</th>
<th>CLOSE</th>
<th>CLOSE_1</th>
<th>CLOSE_3</th>
<th>RETURN_1</th>
<th>RETURN_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>232</td>
<td>IBM</td>
<td>01NOV05</td>
<td>$88.90</td>
<td>81.88</td>
<td>80.62</td>
<td>0.086</td>
<td>0.103</td>
</tr>
<tr>
<td>233</td>
<td>IBM</td>
<td>01DEC05</td>
<td>$82.20</td>
<td>88.90</td>
<td>80.22</td>
<td>-0.075</td>
<td>0.025</td>
</tr>
<tr>
<td>234</td>
<td>Intel</td>
<td>01AUG86</td>
<td>$23.00</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>235</td>
<td>Intel</td>
<td>02SEP86</td>
<td>$19.50</td>
<td>23.00</td>
<td>.</td>
<td>-0.152</td>
<td>.</td>
</tr>
<tr>
<td>236</td>
<td>Intel</td>
<td>01OCT86</td>
<td>$20.25</td>
<td>19.50</td>
<td>.</td>
<td>0.038</td>
<td>.</td>
</tr>
<tr>
<td>237</td>
<td>Intel</td>
<td>03NOV86</td>
<td>$23.00</td>
<td>20.25</td>
<td>23.00</td>
<td>0.136</td>
<td>0.000</td>
</tr>
</tbody>
</table>

**MULTIPLE LAGS MEANS MULTIPLE QUEUES**

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 generation of SALES) 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 January 2010, but only 4 records for February 2010. 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.
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**

```sas
data example4;
  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 queues, one for each of the products A through D. 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 observation having the same PRODUCT as the current observation, no matter how far back it may be in the data set.

**LEADS – HOW 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. In the case of large data sets, this is a costly technique.
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**

```plaintext
data lead_example5;
set sample1;
    if eof1=0 then
        set sample1 (firstobs=2 keep=close rename=(close=LEAD1)) end=eof1;
    else lead1=.;
    if eof3=0 then
        set sample1 (firstobs=4 keep=close rename=(close=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 interleaved records. Instead they produce separate streams of data. 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 (CLOSE), 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 CLOSE is renamed (to LEAD1 and LEAD3) in the additional SET statements, resulting in three variables: CLOSE (from the “current” record, LEAD1 (one period 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=“ 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 while the first would have 3 unread records remaining. The way to work around this 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:
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 example6;
  set sample1;
  by stock;

  if eof1=0 then
    set sample1 (firstobs=2 keep=close rename=(close=LEAD1)) end=eof1;
  if last.stock then lead1=.;

  if eof3=0 then
    set sample1 (firstobs=4 keep=stock close
      rename=(stock=stock4 close=LEAD3)) end=eof3;
  if stock4 ^= stock the lead3=.;

  drop stock4;
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

<table>
<thead>
<tr>
<th>Obs</th>
<th>STOCK</th>
<th>DATE</th>
<th>CLOSE</th>
<th>LEAD1</th>
<th>LEAD3</th>
</tr>
</thead>
<tbody>
<tr>
<td>230</td>
<td>IBM</td>
<td>01SEP05</td>
<td>$80.22</td>
<td>$81.88</td>
<td>$82.20</td>
</tr>
<tr>
<td>231</td>
<td>IBM</td>
<td>03OCT05</td>
<td>$81.88</td>
<td>$88.90</td>
<td></td>
</tr>
<tr>
<td>232</td>
<td>IBM</td>
<td>01NOV05</td>
<td>$88.90</td>
<td>$82.20</td>
<td></td>
</tr>
<tr>
<td>233</td>
<td>IBM</td>
<td>01DEC05</td>
<td>$82.20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Table 8**
Leads With By Groups

<table>
<thead>
<tr>
<th>Obs</th>
<th>STOCK</th>
<th>DATE</th>
<th>CLOSE</th>
<th>LEAD1</th>
<th>LEAD3</th>
</tr>
</thead>
<tbody>
<tr>
<td>234</td>
<td>Intel</td>
<td>01AUG86</td>
<td>$23.00</td>
<td>$19.50</td>
<td>$23.00</td>
</tr>
<tr>
<td>235</td>
<td>Intel</td>
<td>02SEP86</td>
<td>$19.50</td>
<td>$20.25</td>
<td>$21.00</td>
</tr>
</tbody>
</table>

**GENERATING MULTIPLE LEAD “QUEUES”**

Generating lags for the SALES data above required the utilization of multiple queues – a LAG function for each product. This resolved 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. However, instead of depending on the several LAG functions to manage separate queues, generating leads require a collection “filtered” SET statements. The relatively simple program below demonstrates:

```
Example 7: Generating Leads for Irregular Time Series

data sales_lead;
set sales;

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

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

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

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

else lead1=.;
run;
```

The logic of Example 7 is straightforward. If the current product is ‘A’ (if PRODUCT=A) and the set of PRODUCT “A” records is not finished (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 filter is honored prior to the FIRSTOBS parameter. So “FIRSTOBS=2” means the second product “A” record.

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**

<table>
<thead>
<tr>
<th>Obs</th>
<th>MONTH</th>
<th>PRODUCT</th>
<th>SALES</th>
<th>LEAD1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>A</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>B</td>
<td>20</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>A</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>B</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>B</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>A</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>B</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>19</td>
<td>5</td>
<td>A</td>
<td>3</td>
<td>20</td>
</tr>
</tbody>
</table>

### CONCLUSIONS

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.

### REFERENCES:


### ACKNOWLEDGMENTS

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

### 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
- **Company:** Wharton Research Data Services
- **Address:** 305 St. Leonard’s Court
  3819 Chestnut St
  Philadelphia, PA 19104
- **Work Phone:** 215.898-2160
- **Fax:** 215.573.6073
- **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 out=sample1;
  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;