Paper 093-2010

Stupid Human Tricks with PROC EXPAND®

David L. Cassell, Design Pathways, Corvallis OR

ABSTRACT

PROC EXPAND is one of the lesser-known procedures in SAS/ETS®. Its power and its purpose make it very useful for a host of basic, non-statistical, data management tasks. We will cover some tasks which may otherwise require lots of coding and/or macro programming… unless PROC EXPAND resides in your bag of tricks. We show how to: create lags and leads of variables; create moving maxima and minima of a given length; aggregate chunks of K records; and build moving-window averages with trimming to handle edge cases.

INTRODUCTION

SAS® tools like PROC EXPAND typically have a large number of features, to permit a wide range of applications. This often means that a given procedure has functionality that we wouldn't expect. In the case of PROC EXPAND, we can use it for some database management kinds of tasks that often crop up whenever we have objects we track through time, or across components.

In this paper, we will look at some simple uses of PROC EXPAND that are NOT statistical or related to econometrics, even though PROC EXPAND is in the SAS/ETS module. These uses may not be so common that they would be a natural part of some Base SAS procedures, but they do turn up when you least expect them.

TRICK 1: LAGS AND LEADS

We're all used to using lags to look at the previous value (or values) of a variable. The LAG(*variable-name*) and LAGn(*variable-name*) functions let us set up a queue that holds the last seen values of a specified variable. Unfortunately, these functions require extra programming in order to handle the usual situations that we see in database management. For example, how to we handle things when we have BY-groups and we cross from one by-group into the next? How do we handle things when we have to address border conditions, such as the value to use when we start out at the beginning of the data set (or at the beginning of a new by-group)? It turns out that PROC EXPAND can handle these situations for us.

And more importantly, PROC EXPAND can work with the reverse case: when we need, not the previous record's value of X, but the *next* record's value of X. LAGn() cannot get that for us directly, because the LAGn() functions work by having already read the values of the variable X and stored them in a queue for us. When we are interested in the *next* record, we usually call that a **lead** as opposed to a lag. There are several ways of working with leads in SAS, but all of them require reading the data set twice in some manner, or accessing it more than once, or re-defining our concept of 'current record'. PROC EXPAND does this for us, because it has a LEAD option.

Let's suppose we have a pharmaceutical database. We have our patients labeled with a PT variable, and our data are sorted (or indexed) by PT (patient ID) and then DT (a SAS datetime variable we won't need in our code below). We have a DOSEDT variable which tracks dosage by datetime. Suppose we need the *next* dosage as well, stored in a variable we'll call LEAD_DT. And we have to make sure that we never accidentally select a dosage from the next patient when we get to the last dose for the current patient in the database.

It turns out that PROC EXPAND is designed to do this for us. Let's look at some simple code.

```
proc expand data=YourData method=none;
```

```
    by pt;
    convert dosedt = lead_dt / transformout = (lead 1);
    run;
```

We are all used to the standard format for PROC steps, and PROC EXPAND is no exception. The DATA= option allows us to input our data set. There is an OUT= option, and since we are not using it, the new data set will also be called YourData (the name of the input data set), and have the new variables inserted along with all the old variables.

The METHOD= option provides procedure-level information. In this case, METHOD= tells the procedure how to do the interpolation between data points. We don't want any interpolation, so we specify METHOD=NONE. This is important, because the default interpolation method is cubic spline curves, which we don't want for these techniques. (Although, when you do want smooth interpolation of your data points, this is an invaluable thing to know about.) The BY statement works just as always, and ensures that the procedure will not use data from a prior or later value of your by-variable while you work on data for your current by-value.

The CONVERT statement lets us list the variable to be converted, and specify how we want to transform it. It typically takes the form:

CONVERT variable [=variable] [, …] / transformation-options ;

If the variable is not given the [=newvariable] optional part, then the converted value is placed into the current variable over the current value. Lists of variables can be put in a single CONVERT statement, or multiple CONVERT statements can be used. The transformation options are *very* extensive, so we'll only touch on a few of them. Two such options are TRANSFORMIN= and TRANSFORMOUT= . These perform data transformations before and after (respectively) the interpolating function is fit. As long as we specify METHOD=NONE, these two options are equivalent. And finally, the options for the transformation are in parentheses after TRANSFORMIN= (or TRANSFORMOUT= , it's up to you). Here, LEAD 1 asks for the first value of the lead. LEAD *n* would yield the nth lead, while LAG *n* would yield the nth lag.

We should also note the importance of missing values in our new variable LEAD_DT. The final record for each patient PT has no following record for that value of PT. When we reach the end of the records for a given value of PT, PROC EXPAND here will automatically fill the value of LEAD_DT with a missing value, just as we want. If we were doing this work by hand, we would have to make sure that our code performed this way. PROC EXPAND saves us that problem.

So… what if we need something more than merely the next record? PROC EXPAND lets us access any lag, and any lead. All we have to do is specify the number of the LAG or LEAD in the TRANSFORMOUT options. So let's get the next dosage in LEAD1_DT, the dosage after that in LEAD2_DT, the previous dosage in LAG1_DT, and the dosage two records back in LAG2_DT. And let's make sure that we don't accidentally snag any dosage information from other patient IDs.

```
proc expand data=YourData method=none;
   by pt;
   convert dosedt = lead1_dt / transformout = (lead 1);
   convert dosedt = lead2_dt / transformout = (lead 2);
   convert dosedt = lag1_dt / transformout = (lag 1);
   convert dosedt = lag2_dt / transformout = (lag 2);
   run;
```

We can use as many CONVERT statements as we want, and create as many variants of our data as we require.

TRICK 2: MAX OR MIN OF THE LAST K RECORDS

We know what to do if we need the maximum or minimum of all the non-missing values in the data set (or for each value of the by-variable in the data set. PROC MEANS and PROC SUMMARY and PROC

TABULATE (and many other procedures) can compute that easily. But what if we need the maximum and/or minimum of the last 12 records, or the last 50 records, at every record of our data, within each value of our by-variable? Let's say that we have a series of factories, and at each factory we have a sequence of production line records for which we need these data. It's possible to code that up in a DATA step using an array, but it's not exactly trivial.

On the other hand, it *is* trivial if we can use PROC EXPAND. If we want the max or min of a moving window of the data, then the transformations we want are called MOVMAX and MOVMIN, respectively. So all that we need to do in order to look at the max and min of the last 50 records, for every value of X, within each factory, is:

```
proc expand data=YourData out=YourMax method=none;
   by factory;
   convert x = max_x / transformin=(movmax 50);
   convert x = min_x / transformin=(movmin 50);
   run;
```

This will create moving maxima and moving minima of length 50, but will also create computed max and min values in the first 49 records of each factory where the max and min are all based on less than the 50 records we specified. So the first record will have the min and max of only one record, the tenth record will have the min and max of only ten records, and so on.

If we need to have some of these early computed values set to a missing value, PROC EXPAND can do that too. The TRIM or TRIMLEFT option will do the job for us. (Since we only use values 'to the left' of the current record in these computations, they are equivalent in this case.) TRIM *n* will set the values to missing for the first *n* records, and do the computations thereafter.

So, if we want moving max and min values of size 50, but we don't want to do the computations until we have at least 30 records for each value of our by-variable FACTORY, then we need only modify the code like this:

```
proc expand data=YourData out=YourMax method=none;
   by factory;
   convert x = max_x / transformin=(movmax 50 trim 29);
   convert x = min_x / transformin=(movmin 50 trim 29);
   run;
```

With 29 or fewer records, we set the max and min to missing. Starting at 30 records, we compute the max and min. It's far simpler than working out the necessary DATA step code.

TRICK 3: AGGREGATING CHUNKS OF RECORDS

Let's suppose that we have stock market data or some similar sequential data. We would like to aggregate the first three values of the variable VOL (our volume variable), and we would like to get the last value of PRICE for that three-record chunk. Similarly, we want the sum of the 4th through 6th values of VOL together with the value of PRICE from record 6, and so on. Here's a data set to show what we are after:

```
data temp1;
   input obs vol price;
   datalines;
1 2 11
2 2 11
3 2 12
4 3 13
5 3 11
6 3 12
7 4 14
```

```
8 4 12
9 4 12
10 5 11
11 5 16
12 5 14
13 6 10
   ;
   run;
```

Records 1-3 are aggregated, then records 4-6, records 7-9, records 10-12, and so on. Since there is not a complete set of three records after record 12, we will not continue the aggregation there.

It turns out that PROC EXPAND has an option that lets us do this sort of aggregation directly. The only requirement is that we specify the aggregation factor in the procedure statement.

```
proc expand data=temp1 out=exp1 factor=(1:3);
   convert vol=aggrvol / observed=total;
   convert price=new_price / observed=end;
   run;
```

The FACTOR=(n:m) option describes the rate of our aggregation. So here we get 1 new record for each three old ones. This very conveniently does precisely what we were looking for.

The two CONVERT statements use a different type of transformation: the OBSERVED= option. This option lets you specify how to convert from the input series into the output series. The usual values are options like TOTAL, AVERAGE, BEGINNING, MIDDLE, and END, although there are more complex options available. Here we see that the OBSERVED=TOTAL option gives us the total of all values of VOL for the three-record chunk of interest, and the OBSERVED=END option gives us the value of PRICE from the last record of the three-record chunk. This yields the result:

```
TIME       AGGRVOL       NEW_PRICE
 0            6             12
 3            9             12
 6           12             12
 9           15             14
```

Note that the partial value at the end of the initial stream is lost, since there aren't three more records to use in order to create the aggregate value or the 'last of three' value.

TRICK 4: THE MOVING AVERAGE

Suppose we want to look over some sequential data, only we have been asked to compute a mean at every point. The mean is for each of two variables, which we will call NUMBER1 and NUMBER2. Each mean is supposed to be the average of the current value and the four previous values. So we have a 'window' that slides along with our record pointer, and we compute a mean of five values at each record. This is known in the trade as a moving average, or a moving-window average. This can be done by transposing all the data and loading the values into a long array, so that all the averages can be computed from the array using a nested do-loop. This is not particularly clean, and requires attention to detail at the boundaries.

Or we can use PROC EXPAND, which is designed to compute statistics like this. In fact, the TRANSFORMIN= and TRANSFORMOUT= options have the MOVAVE transformation which specifically does these moving averages. There are other choices which will let you perform a centered moving average, an n-step forward-looking moving average, a weighted average with your choice of weights, etc. So, to compute a moving average of width five, as we have just discussed, all that we need to do is:

```
proc expand data=temp1 out=YourOut method=none;
   convert number1=mean1 number2=mean2 / transformout=(movave 5);
   run;
```

Note that we have put two different variables into the CONVERT statement. We can actually put an entire list of variables in the CONVERT statement, as long as they will all be receiving the same transformation (or sequences of transformations).

Note that if you use the above code, the first average is just the first number; the second average is the average of the first and second numbers; the third average is computed from the first three numbers; and so on. Only after five records have been read is the average computed from five numbers. This is how a moving average would normally be computed. If you want to make the average be set to missing when the number of values is small, then you can use the TRIM transformation too. The following code will set the first three moving averages to missing, and only compute moving averages starting with the fourth record:

```
proc expand data=temp1 out=YourOut method=none;
   convert number1=mean1 number2=mean2 / transformout=(movave 5 trim 3);
   run;
```

This means that you can generate a fairly complicated algorithm very simply here. If you need (backward) moving averages of length 12, but you are required to assign a missing value if the average is based on 5 or fewer records, then you can create the entire system like this:

```
convert x = moveave12 / transformout = (moveave 12 trim 5);
```

This also handles by-processing if you insert a BY statement, so it can be a lot easier than trying to work everything with a data step.

CONCLUSIONS

It is always important to have a 'bag of tools' when programming. A rich environment like SAS gives you the opportunity to have a bag of tools the size of a Winnebago. A procedure like PROC EXPAND may not seem to have much relevance to areas like database management and data manipulation, but (not too surprisingly) the complexity of SAS procedures gives us the opportunity to turn them into new and ingenious tools for our toolbag. These four 'tricks' are but a tiny subset of the possibilities available with a tool like PROC EXPAND.

REFERENCES

SAS OnlineDoc® 9.1.3, Copyright © 2002-2005, SAS Institute Inc., Cary, NC, USA; All rights reserved. Produced in the United States of America.

ACKNOWLEDGMENTS

CONTACT INFORMATION

The author welcomes questions and comments. He can be reached at his private consulting company, Design Pathways:

David L. Cassell
Design Pathways
3115 NW Norwood Pl.
Corvallis, OR 97330

DavidLCassell@msn.com
541-754-1304