A SAS Macro for Construction of Symbol Statements

Submitted by James E. Blum on behalf of the Spring 2004 Advanced SAS Programming class,
UNC-Wilmington, Wilmington, NC.

Abstract:
This paper focuses on the construction of a SAS Macro that allows the user to define a number of SYMBOL statements (up to the maximum of 255 permitted by SAS) without explicitly writing each. The design allows for significant control and flexibility for the user; they may input lists of parameters to define options, a data set where these parameters are defined, or both. Provisions are made to allow the user to save their specifications so that they may be reused with a simple two parameter macro call. A series of pre-defined parameter sets are included.

1. Introduction

Background

It is probably worth while to begin with some discussion of how this project originated. Extensive interest in SAS programming at UNC-Wilmington led to the offering of the Advanced SAS Programming course for the first time in early 2004. The major topic for the semester was SAS Macro Language, and students were given the task of creating a SAS macro that would write SYMBOL statements for use with SAS/GRAPH. There were few parameters for this assignment, only that it be sufficiently flexible and easy enough to use as to have some practical value. It is not necessarily difficult to write a macro that will define SYMBOL statements; however, writing one that saves the user a significant amount of coding time while still allowing for full control of options is a greater challenge. What follows is a discussion of several ideas that came from this work and the author/instructor's efforts to put them together into one coherent package.

2. Methods

2.1 Basic Behavior

One of the most important aspects in the development of the macro is handling the passing of parameters. Since SYMBOL statements are not much more than a list of parameters and their values, it would be possible to have an extensive list of parameters to pass to this macro, so we would like to keep this as simple as possible. A common theme was the development of a set of defaults for all parameters, which may or may not be the same as the SAS defaults, and allow the user to pass parameters they wish to modify in lists—together with the number of SYMBOL statements they wish to create. A typical call might look something like this:

```
%symbols(number=3,colors=magenta blue red);
```

This would generate three SYMBOL statements, with the respective colors listed, and other options set to their defaults by the macro. These may be the SAS defaults; "plus" symbol, no

interpolation, *etc.*; but they may be something else entirely. Aligning default parameter specification with the user preferences is an important way to simplify the use of the macro, and it will be taken up in more detail later.

Presuming for the moment that all other defaults are the SAS defaults, suppose the user wishes to have triangular plotting symbols and a connect-the-dots (join) interpolation. Obviously, we could create a call that looks like the following:

```
%symbols(number=3,colors=magenta blue red, v=triangle triangle
                triangle, i=join join join);
```

However, the goal is to make this *easier* for the user, so any unnecessary redundancy that can be eliminated should be. Here we are treading on a situation where the gap in work to create the SYMBOL statements and to call the macro is not as wide as it should be. To alleviate this, a particular default behavior is introduced whereby a single option specification is taken as global and is applied to all SYMBOL statements generated. Hence the call:

```
%symbols(number=3,colors=magenta blue red, v=triangle, i=join);
```

is sufficient for the previous example.

Extending the previous example, suppose the user has two response variables to plot against one predictor for three different categories; for example, a plot call as such:

```
plot response1*predictor=category;
plot2 response2*predictor=category;
```

Further suppose the user wishes to have the plot of a given response be the same color across categories and the plotting symbol to be common within each category. This will require two different colors, one for each response, and three different plotting symbols for the three categories, for a total of six SYMBOL statements. The following call will give the desired result:

```
%symbols(groups=3, gsize=2, colors=blue red, v=triangle square
                diamond, gby=symbol, i=join);
```

Here, the gby= option notes which characteristic should be constant across a group, the symbol or color. If the user wishes to have colors reflect the category and plotting symbol reflect the response, a possible call would be:

```
%symbols(groups=3, gsize=2, colors=blue red green, v=triangle
                square, gby=color, i=join);
```

It should be noted that proper use of this utility requires an appropriate plot call corresponding to it due to the fact that SYMBOL statements are applied corresponding to the order in which SAS "sees" the plots. The behavior of the macro corresponds to creating symbol definitions on the

first response for all levels of the first category, then continuing for the next response, and so on. This was chosen due to the behavior of PLOT1 and PLOT2 calls like the one shown above. If the user in the previous example had a data set with six response variable columns, an appropriate plot call would be:

```
plot (response1A response1B response1C response2A
    response2B response2C)*predictor/overlay;
```

presuming A, B and C denote differing categories.

2.2 Specialized Behaviors

One area that can be problematic is the use of multiple symbol fonts. Suppose the user wishes to use a circle, square, triangle and star, each of which is filled in, as their symbol set. This is easy enough to accomplish based on what has been discussed thus far, using the *special* font:

```
%symbols(n=4, v=J K L M, f=special);
```

Suppose instead that the user wanted to use an open square, closed square, open triangle and closed triangle as their four symbols. Needing to mix fonts for plotting symbols is relatively common, and fonts and symbol values are tied firmly together. Thus when specifying a plotting value, it would be helpful to be able to specify a font with it, if necessary, at the same time:

```
%symbols(n=4, v=square K{special} triangle L{special});
```

So the user can link their plotting symbols and fonts in a relatively simple manner.

One area that can be automated, to the potential advantage of the user, is color selection. If you are like the author (hopefully not), it is easy to run out of ideas for colors after you pass about three or four. Also, it is very easy to choose colors that look awful together or are hard to distinguish (black and green in the default color list are notorious for this). Having an algorithm choose colors, based on the number of different colors desired, can make the plotting tasks much easier. Although no particular algorithm(s) has been decided upon as yet, several things can be said as to their potential nature.

A couple of suggestions are based on the HLS (hue, lightness and saturation) color system. One suggestion is to divide up the “color wheel” (in HLS, the visible spectrum is placed in a circle) based on the number of SYMBOL statements requested. If the number of requests is relatively small (12 or less), then this gives a fairly diverse array of colors, though not necessarily aesthetically pleasing. Another option is to let the user choose the base hue, or two or three base hues, and vary the lightness level, again based on the number of SYMBOL statements requested. This would not be unlike the style of coloring in a heatmap, which is often based on a couple of complementary hues with varying degrees of brightness. Another suggestion is to systematically work through RGB color codes. This allows for some extra control in avoiding problem colors; yellow, for example. Hopefully, with the assistance of someone who understands color mixing and has a keen aesthetic sense, these suggestions can be ironed into a useable set of utilities.

3. Usability and Re-Usability

One of the most important notions to arrive in the development of this utility was the idea of giving the user the ability to reproduce any result obtained by invoking the macro. Of course, we could simply save the SYMBOL statements as generated by the macro, or save the macro call itself, but a different approach was implemented. The option of creating a data set, where the variables in the data set correspond to the options and each record corresponds to a separate SYMBOL statement, was chosen as a means of storing user input and also as a means of storing the default values for the macro. This means that the macro can be run with only one input, a data set name (all data sets used by the macro are presumed to be stored in the SASUSER library) and a SYMBOL statement will be written for every record in that data set. This also means that the user does not need to invoke the macro to set up options. They can, if they so choose, define a set of options by creating a data set and referring to it in the macro call. If they use the macro call to set up their set of options, they may also pass a data set name to the macro where the chosen options will be stored.

Some consideration has been given to handling changes or overrides when the user wishes to use a particular data set, but make some modifications. As noted in the beginning of section 2, user inputs will override the defaults and, in fact, any data set specification. The question becomes, what if the user wants to make one change on one SYMBOL statement, or two or three changes here or there? At this point, no provisions have been made to do such localized changes, for two main reasons. First, the additive nature of SYMBOL statements makes these types of changes fairly simple. For example, if I want the 9th symbol to be a triangle, then:

```
symbol9 v=triangle;
```

modifies the plot symbol without any other changes. Second, if I wish to make a few minor changes and save the result, it is likely easier to make a copy of the data set and make the modifications there.

4. Comments

Instructions, updates, examples, pictures and such are available on the web at http://people.uncw.edu/blumj/projects . Further development of this utility will be ongoing, with the possibility of including some links between the styling of symbols and legends and axes. For example, when using PLOT and PLOT2 a common trick is to match text coloring on the vertical axes to the symbol color of the corresponding plot. One could also easily automate a matching between text color for legend values and their corresponding symbol color. Many ideas for applying a complete plotting style in one macro call will be explored.

Finally, members of the Advanced SAS Programming course at UNC-Wilmington, Spring 2004, and participants in this project are:

Eddie Bakewell, Emilea Grove, Michael Hanson, Stephanie Herring, Peter Hocking, Kelley Honeycutt, Yuan Liu, Lisa Olsen, Marvin Rothwell, Aaron Thatcher, Shauna Turner, Deidre Wood, Meng Wu, Yusheng Zhai and Ryan Ziemiecki.

5. References

SAS V8 On-Line Documentation.

6. Contact Information

Your comments and questions are valued and encouraged.
Contact the author at:
James Blum
University of North Carolina-Wilmington
601 S. College Road
Wilmington, NC 28403-5970
Work: (910) 962-4299
Fax: (910) 962-7107
Email: blumj@uncw.edu
Web: http://people.uncw.edu/blumj