Paper SA-15-2015

Good SAS Programming Practices

Pat O’Meara, Pat O’Meara Associates, Inc., Lincoln, NE

ABSTRACT

The PhUsewiki website is a collaboration of PhUse, FDA, and pharmaceutical industry members to provide a means to share information related to clinical trials informatics. Among the documents on the website is a white paper describing principles of good programming practice (GPP). This presentation describes the principles of GPP as they relate to SAS programming. Several “before and after” examples taken from real SAS programs are presented.

INTRODUCTION

A Google search of “good programming practices” produced 118,000,000 hits. Narrowing the search by adding the “in SAS” reduced the number of 385,000 hits. Both searches indicate there is a great deal of interest in good programming practice. On the first page of the second search is a link to www.phusewiki.org and a white paper entitled “Good Programming Practice Guidance.” The PhUse organization is a collaboration of PhUse, the FDA, and industry to provide a platform to share information and ideas focused on clinical trial informatics.

The web site contains several articles about GPP, including an article entitled “Robustness” that describes elements of good programming design. We will introduce four concepts of programming design that go hand-in-hand with GPP: clarity, simplicity, transparency, and robustness.

THE GUIDANCE

The Good Programming Practice Guidance (GPPG) not only touches on many aspects of the job of a statistical programmer in either the medical device or pharmaceutical industry, but provides much good advice for programmers. When you start a new project, gather and read all the background material that you can find. In the clinical trials arena the documents include the study protocol, a statistical analysis plan, analysis data sets and the corresponding specification documents. Also there may be journal articles or previously written study reports.

English is the language used in the programming code and comments. Each program should contain a standard header that identifies the project, the program name, a concise description of the program description, and a revision history. Comments should be complete sentences.

While these are important topics, our focus will programming conventions specifically applied to programming with SAS. The GPPG divides the conventions into 14 required and 5 recommended conventions. The conventions fall into two broad categories: conventions that improve readability; conventions that describe good technical coding practices. Here is the list:

- **Conventions that improve readability**
    - Use of all upper case letters should be avoided
    - Separate data and procedure steps with at least one blank line
    - Split data steps into logical parts
    - Put each statement on a separate line
    - Left justify global statements
    - Left justify data and procedure statements and their corresponding run and quit statements
    - Indent statements belonging to a level by 2-5 columns
    - For do loops, place the end statement in the same position as the do statement

- **Conventions that describe good coding practices**
    - Do not over write existing datasets
    - Use the “data=*dataset*” option in procedure statements
    - Remove unnecessary code left over from previous projects
    - End data and procedure steps with either run or quit
    - Do not use tabs for indentation
    - Use self-explanatory names for variables and datasets
    - Use put and input functions when convention from character variables to numeric and vice-versa
    - Structure programs to read all external data at the top, do processing, then produce output

- o Insert parentheses in meaningful places

The conventions that improve readability should not require further explanation. Unless company standards require otherwise, SAS® code should be written in lower case to improve readability. It is also good practice to left justify data and procedure statements and their corresponding run and quit statement. Indentation allows the reader to follow the logic of a complex data step. Ending an indented block of code with a blank line tells the reader that the section of logic is complete. Similarly, for do-loops, aligning the end-statement with the do-statement provides a definite boundary for the reader.

The first rule of good technical programming is to never over write an existing data set. This most frequently happens in a data step when the programmer is adding new variables or modifying existing ones. For an example consider the following code where an index variable, case, is added to the existing data set demog and the existing variable, weight, is change from kilograms to pounds.

Poor SAS Coding Practice

```
data demog;
     set demog;
     case + 1;
     weight = weight*2.54;
run;
```

Good SAS Coding Practice

```
data demog1;
     set demog;
     case + 1;
     weightlb = weight*2.54;
run;
```

In the example on the left, the original value of weight is lost; other program segments may no longer work properly. In a program with several hundred lines of code, the place where weight was changed from kilograms to pounds may be difficult to find. In the code on the right, a new dataset, demog1, containing two new variables, case and weight, is created; the original dataset is preserved.

Each of following sets of code that reads a dataset containing randomization codes and merges it with a dataset containing demographic information. The first statement is a %LET left over from a previous version of the program; it should be removed. The second line is using a macro variable to &SIP to replace SUBJID, an unnecessary complication. By removing obsolete code and changing to lower case we have improved readability. Note the use of the format procedure to improve the efficiency of creating the new variable, treat. Also, note the use of the "in=" options in the final data step, where the names of the variables indicate the source of the observations if the if statement.

Poor SAS Coding practice

```
*%LET SIP = STUDYID INVID SUBJID;
%LET SIP = SUBJID;
DATA RN; SET &DB..RN;
     LENGTH TREAT $30.;
/*     IF TRTGRP^=RANDGRP THEN PUT
'WAR' 'NING *** TRTGRP ^= RANDGRP. '
STUDYID= SUBJID= TRTGRP= RANDGRP=;*/
/*     RENAME TRTGRP = TREAT;
*/
     IF  RANDL = 'P' THEN TREAT='PLACEBO';
     IF  RANDL = 'A' THEN TREAT='XYZ-45G';
     IF  RANDL = 'B' THEN TREAT='XYZ-87F';
     KEEP &SIP RANDL RANDGRP TREAT ;*
TRTGRP;
PROC SORT; BY &SIP;
RUN;
DATA DEMOG; MERGE DEMOG(IN=A) RN(IN=B);
     BY &SIP;
     *IF A;
     IF A AND B;
     GROUP=INT(SUBJID/100);
RUN;
```

Good SAS Coding Practice

```
   proc format;
        value $trt ‘P’=’PLACEBO’
                   ‘A’=’XYZ-45G’
                   ‘B’=’XYZ-87F’;
   run;

   data rn(keep=subjid randl randgrp
             treat);
        set &db..rn;
        length treat $30;

        treat = put(randl,$trt.);
   run;

   proc sort data=rn; by subjid;
   run;

   data demogrn;
        merge demog(in=indemog)
              rn   (in=inrn); by subjid;
        if indemog and inrn;
        group = int(subjid/100);
   run;
```

PROGRAM DESIGN

An important aspect of good programming practice is to have a well-planned approach to writing a program. The program should with a standard header that lists the name and location of the program, the author, the purpose, the date that the program was put into production and the description and dates of any modification. After the header, add library references for source data and results, as well as, any user-defined macros. The source code should flow naturally from the steps that access the data, then modifications, analysis, and finally printing results.

Three rules listed in the robustness article (http://www.phusewiki.org/wiki/index.php?title=Robustness) follow

- Rule of Simplicity: Design for simplicity; add complexity only where you must.
- Rule of Transparency: Design for visibility to make inspection and debugging easier.
- Rule of Clarity: Clarity is better than cleverness.

Consider the task of computing area under the concentration-time curve (AUC), common task when programming for a pharmacokinetic study. Figure 1 demonstrates the solution to the problem using the linear trapezoidal rule. The first step is to partition the area under curve into trapezoids. Next, compute the areas of the trapezoids. Then final step is to sum the areas.

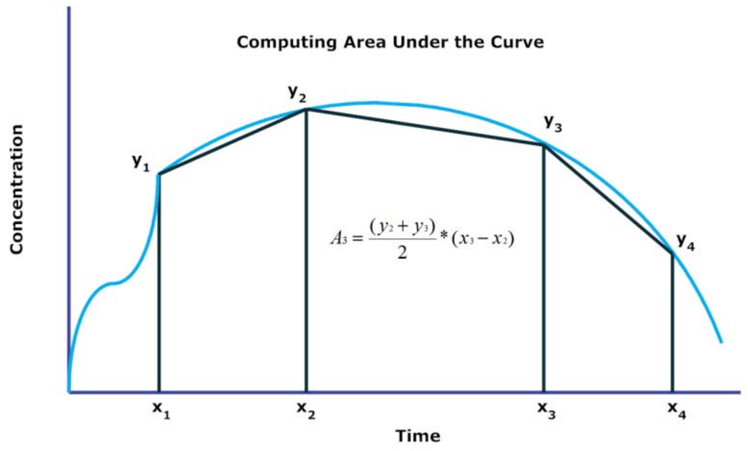
Computing Area Under the Curve
Concentration
Time
y_1
y_2
y_3
y_4
x_1
x_2
x_3
x_4
$A_3 = \frac{(y_2 + y_3)}{2} * (x_3 - x_2)$

Figure 1

Two macros that compute AUC are presented below. The first one is a standard part of a standard macro library. Although there is technically nothing wrong with it, we may be able to improve it by applying the programming conventions from the guidance and applying the three rules of program design. First let's analyze the code. The goal of the macro is to produce a dataset with one observation per subject that includes SORTORDER, SUBJID, AUC, TLAST, and CLAST.

The input and output data sets are implied by the macro call. The datastep overwrites &INDAT. Note that the macro assumes the data in &INDAT have already be sorted in ascending order by SORTORDER, SUBJID, and X. Next, the LAG1 function is used to obtain the previous values of X and Y. When FIRST.SUBJID=1, the variables XDIFF and YSUM are set to 0. Otherwise, XDIFF and YAVG are computed. The result of the first data step is a new version of &INDAT that contains the original variables in the dataset plus the variables PREVX, PREVY, XDIFF, and YAVG. The quantities needed for computing AUC are present. The actual computation is accomplished by PROC SQL. It is the last clause that limits &OUTDAT to one observation per SUBJID. To see this recall the observations are sorted in ascending values of X, which means the max(X) is the last value of X from each SUBJID.

Was it necessary to use PROC SQL to compute AUC? Is there a simpler way to compute AUC that is transparent? Consider the second macro.

The second macro demonstrates the computation of AUC without resorting to PROC SQL. AUC is set to 0 at the first observation for each subjid and its value is retained for the next cycle. When the observation is not the first observation for the subject, the sum function is used to add the area of the trapezoid (yavg*xdiff) to the previous value of AUC. The value of AUC is saved in &OUTDAT only when the last observation for the subject has been processed.

The second method applies the design rules of simplicity, transparency and clarity to obtain a macro that easy to read and understand. Included in the method are some improvements in the code: lower case is used; “do” and “end” statements are aligned; keyword parameters have been used in the macro statement; &INDAT is no longer overwritten.

Poor SAS Coding practice	Good SAS Coding practice
`%MACRO GETAUC(INDAT,OUTDAT);`	`%macro getauc(indat=, outdat=);`
`  DATA &INDAT;`	`  data &outdat.(keep  =sortorder subjid auc x y`
`     SET &INDAT;`	`                rename=(x  =tlast`
`     BY SORTORDER SUBJID X;`	`                        y  =clast));`
`     PREVX=LAG1(X);`	`       set &indat.;`
`     PREVY=LAG1(Y);`	`       by sortorder subjid hour;`
`     IF FIRST.SUBJID THEN DO;`	
`        XDIFF=0;`	`       prevx=lag1(x);`
`        YSUM=0;`	`       prevy=lag1(y);`
`     END;`	
`     ELSE DO;`	`       if first.subjid`
`          XDIFF=X-PREVX;`	`          then do;`
`          YAVG=(Y+PREVY)/2;`	`                 xdiff=0;`
`     END;`	`                 yavg=0;`
`  RUN;`	`                 auc=0;`
	`               end;`
`  PROC SQL;`	`          else do;`
`     CREATE TABLE &OUTDAT AS`	`                 xdiff=x-prevx;`
`     SELECT SORTORDER, SUBJID,`	`                 yavg =mean(y,prevy);`
`            SUM(XDIFF*YAVG) AS AUC,`	`                 auc  =sum(auc, yavg*xdiff);`
`            X AS TLAST,`	`               end;`
`            Y AS CLAST`	`       if last.subjid then output;`
`     FROM &INDAT`	`retain auc;`
`     GROUP BY SORTORDER,  SUBJID`	`run;`
`     HAVING X=MAX(X);`	`%mend;`
`%MEND GETAUC;`	

CONCLUSION

SAS is a very flexible language that allows the programmer a great deal of leeway when writing a program. Even poorly written code can get the job done. www.phusewiki.org offers the interested SAS program sound advice on good programming practices. With thought and planning, the rules of program design can help you write code that is easy to read, easy to review and robust.

REFERENCES

http://www.phusewiki.org/wiki/index.php?title=Good_Programming_Practice

http://www.phusewiki.org/wiki/index.php?title=Robustness

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name:	Pat O’Meara
Enterprise:	Pat O’Meara Associates, Inc.
Address:	2810 Kucera Drive
City, State ZIP:	Lincoln, NE 68502
Work Phone:	402.420.9099
E-mail:	pat@patomeara.com
Web:	www.patomeara.com