Paper IS01

WPP, Worst Programming Practice – Real life pitfalls in every day clinical programming; why and how you should avoid them

Hannes Engberg Raeder, PRA Health Sciences, Mannheim, Germany
Rowland Hale, inVentiv Health Clinical, Berlin, Germany

ABSTRACT

Best Programming Practice should be the backbone of any company's endeavor to create efficient, easy to maintain and robust SAS® programs. Still, the concrete motive for the guidance offered is often overlooked. So, what this paper is aiming at is to be the missing link between what SAS programmers actually do inefficiently, and how it can be done more efficiently. Based on what we have seen during our daily work as SAS programmers in various positions, and underpinned by our own knowledge, various recurrent misconceptions about the SAS language have become crystallized. The paper covers both the aspect of performance as in code execution, and the performance related to maintaining code. Emphasis is placed on understanding of the DATA step, certain SAS procedures, aspects of the macro language and easily maintained SAS code. This is not a paper about fancy programming solutions, but about efficient, easy to maintain and robust SAS code solutions.

INTRODUCTION

With our combined experience in numerous SAS programming environments we have gathered examples from real life situations. The examples chosen relate both to programming efficiency and to maintenance efficiency. By programming performance we mean code solutions which in a given context do not execute efficiently, common examples being superfluous code or DATA steps and procedures used in a sub-optimal way. Maintenance performance on the other hand refers to the amount of resources spent on developing, debugging and understanding not only your own SAS code but others' too.

A common misconception is how and when records and variables are selected, dropped or kept in a DATA step or SAS procedure. Here we review frequently seen situations in which the programmer has implemented less-than-ideal solutions and thereby added considerable inefficiencies to the code.

In addition we discuss pitfalls relating to the DATA step merge, data transposition with the TRANSPOSE procedure, macro definitions (where not to do them) and falling foul of macro variable scope.

The amount of time needed while developing and testing tends to increase with the volume of data used. One common method to resolve this is to develop using a subset of the data. However, an insufficient subset might result in unexpected outcomes if the data used for testing doesn't reflect the whole data set's true structure. We take a look at various ways to reduce the data volume – strictly for code development and testing – and at what you need to consider when doing so.

Finally, a discussion about programming inefficiency and performance would not be complete without covering program comments through "self-documenting" code, and the well-known "being too smart" syndrome. Done properly, much can be communicated through the code itself, but unfortunately this way of implicitly commenting a SAS program is seen too seldom.

EFFICIENCY PITFALLS

In this section we discuss some common ways to manipulate data using the DATA step and SAS procedures used in daily programming. This is only a small portion of all the examples we have seen during our combined exposure to SAS programming but are some of the most frequently seen.

EFFICIENCY PITFALL 1: IF INSTEAD OF WHERE

Reading and manipulating data are common tasks for a clinical SAS programmer. A frequently seen example of inefficient programming results from confusion between the WHERE statement/data set option and the "subsetting" IF statement. In the below example, from a phase III study with almost 650 000 records in the laboratory data set, the task requires all the tests in hematology category to be written to TEMP1. The subset is created in a DATA step using a subsetting IF statement (A).

```
data TEMP1;
   set SDTM.LB;
(A) if LBCAT = 'HEMATOLOGY';
run;
```

However with a data set such as this containing well over a half million records this strategy will soon result in long coffee breaks. To understand this inefficiency it is crucial first to understand the role of the program data vector (hereafter PDV), the memory allocated by SAS for building the data set. The SET statement in the data step will read all rows in the data set SDTM.LB, one observation at a time, into the PDV (SAS 9.4 Language Reference, p. 396), including those where the LBCAT = 'HEMATOLOGY' condition is not met as well. In other words, the DATA step will loop through each of the 650 000 observations and test the condition for each row. The key here is to avoid bringing these rows into the PDV in the first place, and thereby significantly improve performance. This can be achieved by using a WHERE statement or the WHERE= data set option (B) which will examine the condition *before* the row reaches the PDV (Ibid., p. 187) and ignore it if the condition is not satisfied.

```
data TEMP1;
 (B) set SDTM.LB (where = (LBCAT = 'HEMATOLOGY'));
run;
```

EFFICIENCY PITFALL 2: COPYING DATA WITH DATA STEP

Another common task for the clinical SAS programmer is altering data set metadata like labels and formats. The example below is taken from a program which derives an ADaM data set based on DM, LB and the custom SDTM domain XP. Now, making all three data sets available in work for later use throughout the program is not necessarily a bad strategy – it can be more efficient to access the data in the WORK library as opposed to always referring to a permanent library if you are using a local installation of SAS. However, this method of copying data sets into WORK is very inefficient as it iterates through all observations in order to copy them into the temporary WORK library, each observation being read into and out of the PDV en route.

```
/* Get the SDTM data sets */
data DM;
        set SDTM.DM;
run;
data LB;
        set SDTM.LB;
run;
data XP;
        set SDTM.XP;
run;
```

The most efficient way here is to use one of the SAS procedures created for an exercise like this, namely the COPY or DATASETS procedure. In the below example we have chosen to illustrate this using the DATASETS procedure which makes this an easy task. First of all the IN= and OUT= libraries are defined (C) where the IN= library is the source library and the OUT= library the target library. In the procedure's SELECT statement (D) we list all the items, in this case data sets, we would like to copy (Base SAS 9.4 Procedure Guide p. 427-8). The downside of this approach is you can't do any data manipulation at the same time. If this is what you need to do, use the DATA step.

```
/* Get the SDTM data sets */
proc datasets memtype = data;
  (C) copy in = SDTM out = WORK;
   (D) select DM LB XP;
quit;
```

EFFICIENCY PITFALL 3: CONCATENATING DATA WITH DATA STEP

Another typical task is to concatenate data sets, either "raw" data sets with an identical structure or as a final step to complete an ADaM data set for which different derived parameters are brought together. Below is an example from the creation of the vital signs ADaM data set ADVS, which contains parameters derived using different methods. Here the data sets (E) were all listed in the same SET statement, and no data manipulations were carried out as this was the final step.

```
/* Final step, create one data set */
data ADAM.ADVS;
   set VS_PARAM1
 (E)   VS_PARAM2
       ;
run;
```

Appending the data sets using the SET statement again makes SAS loop through every observation in both data sets, starting with VS_PARAM1. But as the data sets grow in size, as they tend to do especially for large phase III studies or integration projects, this will turn out to be less than efficient. So, instead, use either the APPEND procedure or the DATASETS procedure to append the data sets to each other as shown below. In this particular example these two procedures work in a similar manner.

```
proc append base = VS_PARAM1 data = VS_PARAM2;
run;

proc datasets;
    append base = VS_PARAM1 data = VS_PARAM2;
quit;
```

EFFICIENCY PITFALL 4: CHANGING METADATA WITH DATA STEP

A common method used to alter metadata, in this case adding a format and label and renaming a variable (F), is to do so in a DATA step. This is fine if you wish to carry out data manipulations as well, but it is less than ideal if all you need to do is change metadata, as there are much more efficient ways of doing this. The below example was taken from a program where, at some stage, formats and labels needed to be modified and a variable renamed. However, the data set ADLB contained around 900 000 observations and deploying this technique throughout several, maybe hundreds of programs on a study or a project, will certainly make for a very time-consuming batch run.

```
/* Change some metadata in ADLB */
data ADLB;
   set ADLB;
(F) format AVISTIN ZAVIS.;
   label AVISITN = 'Analysis Visit';
   rename TEMPSEQ = LBSEQ;
run;
```

Instead, a much more efficient way to work with the metadata without the need to iterate through a data set using a DATA step is to make use of the DATASETS procedure and the MODIFY statement (Ibid., p. 455) (G).

```
/* Change some metadata in ADLB */
proc datasets lib=work nolist;
   modify ADLB;
      format AVISTIN ZAVIS.;
  (G) label AVISITN = 'Analysis Visit';
      rename TEMPSEQ = LBSEQ;
   run;
quit;
```

PROGRAMMING PITFALLS

Badly written programs may eventually arrive at results which are correct, albeit at the cost of efficiency and performance. Next, we talk not about improving efficiency but about the sort of pitfall which will almost certainly result in frustration, or rather worse, an incorrect output which confuses the inexperienced SAS programmer because no ERROR or WARNING is generated.

PROGRAMMING PITFALL 1: VARIABLE GONE MISSING

The first pitfall does involve an error though, and at first glance, a confusing one at that. Have you, like us, ever seen a message which looks like this and thought, "But that variable is in the data set, I know it is!"?

```
ERROR: Variable NAME is not on file SASHELP.CLASS.
```

This is the direct result of using a variable in a WHERE clause after having dropped it first. So if this error appears in your log, first check the data set, then check the DROP and KEEP definitions! Remember – the DROP and KEEP data set options are processed before WHERE.

PROGRAMMING PITFALL 2: OVERWRITE VARIABLES WITH MERGE

Most of the time, you merge two data sets because you want to add variables from one data set to the other. Observations are combined in accordance with the defined BY-groups. But what happens when the two data sets contain the same but non BY-group, variables? Here, the general rule for common variables is that the second ("right") data set on the MERGE statement overwrites common variables on the first ("left"). Most of the time, such "overwriting" behaviour will be undesirable, however it is easily avoided – by explicitly keeping, on the second data set, only the BY-group variable(s) and the variable(s) to be added as shown below.

```
data adx2;
    merge adx1 adsl (keep=subjidn age);
    by subjidn;
run;
```

Note that messages about overwriting variables are only written to the log if the MSGLEVEL system option is set to I, which is *not* the SAS default, although where you work this may well have been changed. So, if your log is clear of such messages, either no variables are being overwritten, or MSGLEVEL is set to N. If, after all, you do wish your merge to result in data being overwritten you should be aware that one more pitfall rears its head with match-merges, namely that overwriting only affects the number of observations in the BY-group on the left data set as there are observations in the same BY-group on the right data set.

PROGRAMMING PITFALL 3: WRONG DATA CARRIED FORWARD ON MERGE

This code is taken from a very common real world situation in which an analysis data set for vital signs had to be updated. Specifically the variable which flagged high systolic blood pressure was updated but in a way that caused unexpected results. Figure 1 below shows the DM data set which already contains the as yet unpopulated high systolic blood pressure flag variable. In figure 2 we have the VS data set with the actual systolic blood pressure measurements.

	USUBJID	AGE	ARMCD	HIGH_FLAG
1	001-651877	52	DRUG A	
2	001-651879	31	DRUG B	
3	001-651898	33	DRUG A	
4	001-651899	40	DRUG B	
5	001-651901	61	DRUG B	

Fig. 1. DM data set

	USUBJID	VSTESTCD	VSSTRESN
1	001-651877	SYS	120
2	001-651879	SYS	151
3	001-651879	SYS	121
4	001-651898	SYS	139
5	001-651898	SYS	134
6	001-651898	SYS	127
7	001-651898	SYS	129
8	001-651899	SYS	131
9	001-651899	SYS	141
10	001-651901	SYS	153

Fig. 2. VS data set

And here, in figure 3, is the final result of the merge and the condition which states that we want systolic blood pressure equal to or above 135 to be flagged as high.

	USUBJID	VSTESTCD	VSSTRESN	AGE	ARMCD	HIGH_FLAG
1	001-651877	SYS	120	52	DRUG A	
2	001-651879	SYS	151	31	DRUG B	Y
3	001-651879	SYS	121	31	DRUG B	Y
4	001-651898	SYS	139	33	DRUG A	Y
5	001-651898	SYS	134	33	DRUG A	Y
6	001-651898	SYS	127	33	DRUG A	Y
7	001-651898	SYS	129	33	DRUG A	Y
8	001-651899	SYS	131	40	DRUG B	
9	001-651899	SYS	141	40	DRUG B	Y
10	001-651901	SYS	153	61	DRUG B	Y

Fig. 3. Final result

As we can see, the merge with the condition to flag the records which meet the threshold did not work out as planned. For the subject on the second and third rows, USUBJID 001-651879, the first measurement is correctly flagged as high as it is 151, but the second measurement of 121 is also flagged as high! The same goes for the subject on rows 4, 5, 6 and 7 - here 3 out 4 measurements are mistakenly flagged as high. The code which produces this result is a basic merge with a BY statement and a seemingly innocuous IF statement (H).

```
data ADVS1;
   merge VS
         DM;
   by USUBJID;
(H) if VSSTRESN >= 135 then high_flag = 'Y';
    [Additional manipulations not displayed]
run;
```

The reason for this merge and condition going wrong is however rather simple and is a result of how SAS handles a merge with BY-groups. SAS retains the values of all pre-existing variables in the PDV until a new BY-group is encountered (SAS Language Reference 9.4, p. 486). Because the variable HIGH_VALUE already existed in the DM data set it was retained within each BY-group (USUBJID). Creating HIGH_FLAG as a new variable in the DATA step would have solved this problem as the values would not then have been retained in the PDV (for a more detailed explanation see Szilagyi, B., Binder, C., 2006).

PROGRAMMING PITFALL 4: TRANSPOSE WITHOUT ID

Once in a while we need to transpose data and most of us tend to utilise the TRANSPOSE procedure which, with minimal familiarisation, is fairly easy to use and gets the job done. But it does contain one potential pitfall, namely when we fail to identify rows. The below code was used to transpose raw data into a non-normal structure. But in this particular case there was no Visit 2 in the source data, as can be seen in figure 4.

```
proc transpose data = RAW.PICCO out = TR_PICCO (drop = _NAME_)
   prefix = VISIT;
   by USUBJID;
   var XPORRES;
run;
```

VISIT	USUBJID	XPORRES
1	1001	5
3	1001	5
4	1001	.
5	1001	9
1	1002	5
3	1002	6
4	1002	.
5	1002	5

Fig. 4 Raw data set called PICCO

Without a specific link to the rows we want transpose, the TRANSPOSE procedure will corrupt the data as can be seen below (Carpenter 2012, p. 62), so instead of having a Visit 5 (which is what we had) we now have a Visit 2 (see figure 5).

USUBJID	visit1	visit2	visit3	visit4
1001	5	5	.	9
1002	5	6	.	5

Fig. 5

The solution to this problem is to identify the set of variables that results in a unique row. To do this we add the ID statement and use the VISIT variable to achieve this (I). With a proper validation process and statistical review an error like this will not go undetected for long, but it can be cumbersome and time-consuming for a programmer to figure out what went wrong.

```
proc transpose data = test out = tr_test(drop = _NAME_)
   prefix = VISIT;
   by USUBJID;
   var XPORRES;
(I) id visit;
run;
```

And the result we were looking for, using the ID statement in the code above, looks like this (fig. 6).

USUBJID	VISIT1	VISIT3	VISIT4	VISIT5
1001	5	5	.	9
1002	5	6	.	5

Fig. 6

SAS MACRO LANGUAGE

The macro language is one of the most powerful features of the SAS language when it comes to carrying out repetitive tasks (amongst other things). The main job of the macro facility is to generate SAS code, so it does not manipulate data as the DATA step does (although it can of course be used to generate DATA step code). However, there are still significant performance advantages that can be gained from a deeper understanding about how it works. This section describes various scenarios related to performance as well as programming/de-bugging inefficiencies.

MACRO PITFALL 1: THE NESTED MACRO DEFINITION

The need to repeat certain tasks inside a macro is something every SAS programmer comes across frequently. The simplified example below shows how the programmer achieved repetition within the outer macro *%first* (A) by defining an inner macro *%second* (B). This meant that the first time *%first* was called *%second* was compiled, and for each subsequent call of *%first*, *%second* was needlessly *re*compiled, and efficiency was compromised accordingly.

```
(A) %macro first;

(B)     %macro second(param=);

                proc sort data=vs
                        (where=(vstestcd = "&param"))
                        out=vs_temp;
                        by vscat vstestcd;
                run;

                data vs_temp1;
                        set vs_temp;
                        [Additional manipulations not displayed]
                run;

                [Additional code not displayed]

        %mend second;
        %second(param=SYS)
        %second(param=DIAB)
        %second(param=TEMPE)
        %second(param=BMI)
        %second(param=PO)
        %second(param=PO2)

        [Additional code not displayed]

    %mend first;
```

If we assume *%first* was called eight times then SAS compiled the definition of the *%second* eight times also (Carpenter 2004, p. 71, SAS 9.4 Macro Language Reference, p. 143). The examples in this section are unfortunately a poor representation of the real world and not seldom the inner macro not only contains rather more code – which means more to compile – but further levels of macro definition nesting (however, we have yet to come across four levels of macro definitions)!

It is easy to lose track of all the additional compilations needed but say a macro for a table program contains a macro definition which in turn contains a further macro definition, then each time the outer table macro is called the nested macro definition is compiled and each time that macro is called the innermost macro is recompiled as well! This is repeated for all calls to the table macro as SAS does not know that you are compiling the exact same macro definition over again. Fortunately, the SAS macro language is a powerful tool when it comes to handling repetitive tasks, and there are numerous ways to avoid using nested macros. Here are two:

```
%macro first;

  %let params = SYS DIAB TEMPE BMI
                PO PO2;

(D) %do i = 1 %to %sysfunc(countw(&params));

    proc sort data=vs
(C)   (where=(vstestcd = "%scan(&params,&i)"))
      out = vs_temp&i;
      by vscat vstestcd;
    run;

  %end;

%mend first;
```

```
%macro first;

  proc sql no print;
(E)  select distinct vstestcd into
    :param1-:param99
    from vs;
    select count(distinct vstestcd) into :no
    from vs;
  quit;

  %do i = 1 %to &no;

    proc sort data = vs
      (where = (vstestcd = "&&param&i"))
       out = vs_temp&i;
       by vscat vstestcd;
    run;

    data vs_temp1_&i;
      set vs temp &i;
      [Additional manipulations not displayed]
    run;

  %end;

%mend first;
```

In the example on the left (again less complex that most real world scenarios), the vital signs parameters of interest are listed in a single macro variable PARAMS. A macro %DO loop is then used in conjunction with the %SCAN macro function to read and, here using a SORT procedure, deal with each parameter in turn. The number of %DO loop iterations is dynamic because the loop's *stop* value (%sysfunc(countw(&params)) (C)) is determined at runtime to be the number of items in the PARAMS list (D). So this solution is reasonably flexible - any change in the required parameters means only the parameter list assigned to PARAMS has to be updated.

The example on the right offers yet more flexibility as the macro variables created PARAM*n* are data driven, that is all the values in the VSTEST data set are used and if VSTEST changes such that it subsequently contains additional parameters, the macro remains valid without any updates required (Carpenter, 2004, p. 127-32) (E). In detail, the SQL procedure is used to create a macro variable, PARAM*n* for each distinct value in VSTESTCD starting with PARAM1 and if needed up to PARAM99 (only as many macro variables as needed will be created – SAS 9.4 SQL Procedure User's Guide, p. 255). Secondly, to make the loop data driven, we count the distinct rows in the data set for that variable and put the result into a macro variable called NO. And so we have an entirely data driven solution which will ebb and flow with the data (should the data do so).

MACRO PITFALL 2: MACRO VARIABLE SCOPE

There are not too many efficiency drawbacks limited to the SAS macro language itself, the nesting of definitions discussed above being one of them. However, there are a few pitfalls to be aware of which can be frustrating and time-consuming to resolve. One in particular relates to confusion about local and global macro variables (Virgile, 2013, p.77-80). As with the other macro code illustrations, we present here a simplified example:

```
/* From autoexec.sas */
(F) %let var1 = 10;

/* Anotherprog.sas */
   %put &var1;

   %macro _TEST;

     (I) %let var1 = PH;

   %mend _TEST;

   %_TEST;

   %put &var1;
```

The macro variable named VAR1 (F) is a macro variable defined in a totally different program (the autoexec) and happens to be set to a constant value of 10. As it is defined outside a macro definition it will automatically be assigned to the global symbol table (SAS 9.4 Macro Language Reference, p. 49). So far so good, but inside the macro definition for the macro %_TEST a macro variable with the (same) name VAR1 is assigned a new value (I), but before the variable is created the macro processor searches the global and local symbol tables for the name and if present in the global symbol table assigns the value to the variable there (Ibid., p. 53). And that is what is happening here. The macro variable VAR1 already exists in the global symbol table which means the value 10 is overwritten with the value PH, despite this *appearing* to take place in a local context inside a macro. And then the problems begin: other programs in the study which expect VAR1 to hold the value 10 and not the PH which it does hold will cause untold warnings, errors and other unexpected results and lead to significant time lost when it comes to de-bugging (if you do not know what you are looking for) – at best. The worst case of course is when results get altered and neither the QC process nor the visual review spots the error(s) in time.

Fortunately this problem is easily overcome. Starting with SAS 9.4 user-defined macro variables can be set to "read only" (K) in either the global or the local statement – that is they cannot be overwritten – and optionally assigned a value at the same time as a true constant. This is especially handy if you need to reserve certain macro variable names in an autoexec file (Ibid. p. 304-5). Other ways to avoid this overwriting are a) to explicitly define macro variables inside macro definitions as local (L) so that these are treated separately from any macro variables with the same name in the global symbol table as the macro processor now assigns these to the local table specific to this macro, and b) additionally to label local macro variables with a suffix such as L_ (M)

```
(K) %global / readonly var = 10;

    Or

    %macro get_LBTEST;

     (L) %local var1;
         %let var1 = PH;

    %mend get_LBTEST;

    Or

    %macro get_LBTEST;

     (M) %local l_var1;
         %let l_var1 = PH;

    %mend get_LBTEST;
```

MACRO PITFALL 3: CONFUSING THE MACRO LANGUAGE WITH THE DATA STEP

Another common source of confusion arises between how the DATA step and the SAS macro language handle character and numeric variables. It is important to know that normal SAS variables can be one of two types: character or numeric, and that macro variables, on the other hand, and with a few specific exceptions (macro loop counter variables being one), contain SAS code, because that is what SAS macros generate. The resemblance between the macro language and the DATA step might be there inasmuch as they both use common programming constructs ('e.g. %IF-%THEN-%ELSE vs. IF-THEN-ELSE), but the DATA step needs to distinguish between character variables and numeric variables, whereas for the SAS macro processer (Ibid. p. 21) the content of macro variables is only pieces of code in the form of characters, digits or text (unless you tell it explicitly to evaluate the value as a numeric item). The piece of code below is a shortened example from a recent macro debugging exercise:

```
%macro concat;
      [Additional manipulations not displayed]
  (N) %if &input = "" %then %put "missing";
      [Additional manipulations not displayed]
%mend concat;
```

The passage from the %concat macro above in which the condition that INPUT is empty (N) is tested will never be true (unless INPUT literally contains two double quotes "" – and then it is obviously not missing anymore). Had this

been a DATA step it would have been a just method to use (personally we think that the MISSING function is a more elegant solution in a DATA step). However the quotes on the right side of the equals sign is telling SAS to compare whatever value we have in INPUT with “”. Many junior and more senior SAS programmers have spent a good many hours trying to figure out why a condition that is apparently true or false does not act in the way they would expect, but in fact SAS is doing exactly what it has been told – in our example comparing whatever is stored in the macro variable with “”.

There are of course a great many more things that can go wrong using the macro language – macro quoting being another source of confusion. But based on our experience, the nesting of macros, the confusion between global and local macro variables and macro variables being treated as normal data variables are the most common mistakes in the daily use of macros when generating databases, analysis data sets or tables, listings and figures.

COMMENTS, CODE DEVELOPMENT AND EXCELLENT PROGRAMMING

In this section we are switching the focus to another type of issue which adversely affects performance – namely the layout of a program and what that it can do or not do for future maintenance of the code.

COMMENTS IN GENERAL

Done properly, much can be communicated through the code itself, but unfortunately this way of implicitly commenting a SAS program is seen too seldom. What is seen rather too often, though, is either no comments at all, or the other extreme – endless comments stating the already obvious. For obvious reasons a complex program with no comments will not be displayed here, but the implications of such a practice is easy to forecast. All forthcoming maintenance work, including that done by the original programmer, will often be highly prolonged and inefficient as the programmer will always need to spend time at a later date getting into the code once again.

The below example, which to some extent is simplified here, contains only comments stating what is obvious from the code itself. First the data is read in and then it is sorted before being merged with another data set.

```
/* Read in the DM dataset */
data dm;
   set sdtm.dm (drop = armcd actarmcd r:);
   [Additional manipulations not displayed]
run;

/* Sort the dataset before merge */
proc sort data = dm;
   by usubjid  derived_value ;
run;
```

For someone new to the program, or for the original programmer going back to the program after a few weeks, these types of comment do not add anything of value to help the person trying to understand what is going on. With this example we would like to introduce what we would like to call implicit comments and conceptual comments. What we mean by implicit comments are comments through intuitive code. The code itself is written in such a way that it reveals the key details about what is actually going on.

```
(B) data dm (keep = usubjid siteid derived_value);
(A)    set sdtm.dm (keep = usubjid siteid armcd actarmcd);
       [Additional manipulations not displayed]
   run;

(C) proc sort data = dm out = sorted_dm;
     by usubjid  derived_value;
   run;
```

By using the data set option KEEP (A) instead of DROP (we are aware that this not always feasible but much code encountered uses only a fraction of the variables contained in the input data sets) we make the code much more transparent by stating explicitly which four variables are going to be present after the DATA step has executed. What is not displayed in the above example, in order to save space, are the derivations themselves but it is important to mention that implicit commenting means too that cryptic variable and data set names such as _1, _temp, etc., should be avoided in favour of meaningful ones, including for intermediate sections of a derivation, and that this can greatly help to demystify the code without the need for additional comments. On the DATA statement we are using the data

set KEEP option as well (B) to show the variables we are retaining for the next step in the program, and to ensure that superfluous variables are not kept beyond the point they are needed. This way it can be made clear without any comments which variables are used, what is done, and what is kept. Finally, the SORT procedure (C) was defined with an explicit OUT data set to help us distinguish between the DM data set and its sorted version. (To ensure you don't inadvertently overwrite any source data, defining an OUT data set within a SORT procedure should be your default programming practice, so that any decision not to do so is a conscious decision.)

The use of intuitive code and the implicit commenting which then results as a matter of course will not be sufficient as the derivations grow more complex. It should be considered necessary to complement the code with comments which describe either what a certain block of code is doing and/or why it is being done. Imagine a fairly complex analysis lab data set with several derived parameters. Stating only that '*Parameter X is derived here*' will not be particular helpful, but extending the comment to explain the concept behind a certain derivation will greatly help the original and other programmers to update and maintain the code at a later date.

DEVELOPING, TESTING AND DEBUGGING

Unless you are always working on very small data sets, developing, testing and debugging your program using a subset of the input data will surely speed up the process. SAS offers plenty of techniques to reduce the amount of data for testing and developing, not all equally efficient or necessarily suitable for the task in hand. The simplest ones, displayed below, select a specific subset of records using the WHERE statement/data set option (D), selecting the first 100 observations or a range of observations using the OBS= data set option (E, F) or the POINT= option to select, say, every fifth observation (G) (SAS 9.4 Language Reference, p. 463).

```
data test_data;
  (D) set ADLB (where=(SUBJIDN between 10001 and 10050));
run;

data test_data;
  (E) set ADLB (obs=100);
run;

data test_data;
  (F) set ADLB (firstobs=101 obs=200);
run;

data test_data;
    do i = 1 to nobs by 5;
  (G)  set ADLB nobs=nobs point=i;
         output;
    end;
    stop;
run;
```

The four solutions above should be considered "quick and dirty" ways to reduce the amount of data, but with these methods there is a good chance that such a sample will be too systematic. In other words, the selection of test data does not represent the true nature of the data in terms of variety of data scenario and subsequent testing on the full set of the data might reveal new cases that need attention and you will have to start all over again.

To be on the safe side it is recommended to put more effort into the process of selecting your sample. In the below examples we turned to the RANUNI function (Cody, Ron. 2010, p. 290-2) to generate a random sample. This will surely not guarantee an ideal sample which reflects the diverse situations that good or bad data can throw up, but it will certainly be an improvement on the first four options we presented. In the first example below we use RANUNI to generate a random variable (H). The argument we use here is the seed, in this case 1, which is used to initialise the seed stream (a positive seed results in the same sample each time you run the code, however a seed of 0 or less means the time of day will be used to initialise the seed stream (Ibid.) and a different sample will be generated each time you run the code, that is unless you run your code precisely every 24 hours!). The data is sorted by the random RAN variable, OBS= is used to take the first 100 observations of the randomly sorted data, and the data is then sorted by the temporary _n variable created (I) to preserve the original sort order of the data.

```
data test_data;
       set ADLB;
   (H) ran = ranuni(1);
   (I)  _n = _n_;
   run;

   proc sort data=test_data;
       by ran;
   run;

   /* Re-sort by _n to obtain original sort order */
   proc sort data=test_data (obs=100) out=test_data (drop=_n);
       by _n;
   run;
```

Or

```
(J) data test_data (where = (ran <= 0.1));
       set ADLB;
       ran = ranuni(1);
   run;
```

The second example also utilises the RANUNI function with the seed 1 but instead of selecting a sample based on the number of observations it is selecting a portion of the original data (J), in this case approximately 10%.

But why go to the trouble of generating a sample with a DATA step and the SORT procedure when you can use the SURVEYSELECT procedure? The METHOD= option used here, SRS, specifies Simple Random Sampling (K) and, as can be seen, the SURVEYSELECT procedure can be used to select a random sample either of a) n observations, or b) n% of observations (SAS/STAT 9.2 User's Guide to the SURVEYSELECT Procedure, p. 6638).

```
/* Select 100 observations */
proc surveyselect data=ADLB out=test_data
        seed = 1
     (K) method=SRS
        sampsize=100
        noprint;
run;

/* Select 10% of observations */
proc surveyselect data=ADLB out=test_data
       seed = 1
       method=SRS
       samprate=0.1
       noprint;
run;
```

It is also good practice to put a temporary "error" (L) to the log as a safety net when reducing input data for code development and debugging. This ensures that the use of test data stays temporary and is not overlooked later by you, or indeed somebody else who may suddenly be required to take over the code.

```
(L) %put %str(ER)ROR: Sampled input data in use! Not valid for production runs!
```

EXCELLENT PROGRAMMING

Insufficient or bland comments are not the only way in which excess maintenance and re-work negatively impacts the chance of meeting a tight deadline. “Being too smart” refers to the exaggerated use of complex code where a simpler, easier-to-follow solution would suffice. We are both aware that for the healthy SAS programmer there is always an interest in trying out new techniques which may well be result in fewer lines of code and which may well be more efficient when it comes to run time. But from a project point of view it can be very tiresome when even the smallest code updates – as a result of new data – keep programmers busy for days as they need to understand and re-write large sections of code to manage the changes in the data.

The example below, which makes copious use of nested select statements and regular expressions, made for a tricky update when the raw data changed in a way that the original programmer had not planned for.

```
proc sql
       create table &table.start as
       select a.*,
              b.p,
              b.t,
              b.trtlbl,
              compba(tranwrd(catt(max(b.pref), strip(scan(b.lbl,2,'('))), '@@', b.trta)) as trtlbl
         from &table.2(drop=trta) a,
              (select *,
                      case
                        when prxmatch('/\dTT/', trt) then prxchange('s/TOTAL\s*\|*//', -1, trtb)
                        else trtb
                      end as lbl,
                      case
                      when prxmatch('/TT/', trt) then
                           prxchange('s/^(\s*[^(]*)(\()(\s*[^(]*)$/$1 Main=@@| $2/', -1,
                           calculated lbl)
                        else ''
                      end as pref length=200,
                      case
                        when prxmatch('/TT/', trt) then 'ALL Main Groups'
                        when prxmatch('/PBO/', trt) then 'PBO'
                        else 'TREAT'
                      end as trta
                 from (select *,
                              1 as flag,
                              prxchange('s/(TRT\d*)(.*)/$1/', -1, trt) as p,
                              prxchange('s/TOT//', -1, trt) as t
                         from popu
                         where prxmatch('/\d/', trt)
                       outer union corr
                       select *,
                              2 as flag,
                              prxchange('s/(TRT\d*)(.*)/$1/', -1, trt) as p,
                              prxchange('s/TT//', -1, trt) as t
                         from popu
                         where not prxmatch('/\d/', trt)
                      )
              ) b
         where a.trtn = b.trtn and
               a.flag = b.flag
         group by a.flag, b.p
         ;
quit;
```

From a SAS programming standpoint we are both advocators for adding new skills and techniques to a programmer’s tool box but from a lead programmer perspective we favour the easy and robust code solution that can be maintained and updated by a SAS newbie or a DATA step die-hard. The message is: think again before “being too smart”. Quite apart from consideration for others, it may be you who needs to battle with that code you had so much fun writing six months previously!

CONCLUSION

SAS is a powerful language and the master of data manipulation and statistical analysis. Thanks to a number of potential pitfalls though, SAS can sometimes leave less experienced programmers scratching their heads when results don't look the way they should, runs take hours to complete or a none too complex code update takes days to implement. In this paper we have brought together a variety of common pitfalls we have both experienced ourselves and seen in the code of other programmers – beware of these pitfalls and learn from them and you'll enjoy using SAS so much the more!

REFERENCES

Base SAS 9.4 Procedure Guide. SAS Institute
Carpenter, Art. 2012. Carpenter's Complete Guide to the SAS Macro Language, 2nd ed. SAS Institute
Carpenter, Art. 2012. Carpenter's Guide to Innovative SAS Techniques. SAS Institute
Cody, Ron. 2010. SAS Functions by Example, 2nd ed. SAS Institute
SAS 9.4 Macro Language Reference 4th ed. SAS Institute
SAS 9.4 Language Reference 4th ed. SAS Institute
SAS 9.4 SQL Procedure User's Guide, 2nd ed. SAS Institute
SAS/STAT 9.2 User's Guide to the SURVEYSELECT Procedure. SAS Institute
Szilagyi, B., Binder, C., 2006. Watch out, a MERGE ahead! PhUSE 2006
Virgile, Robert, 2013. SAS Macro Language Magic – Discovering Advanced Techniques. SAS Institute

RECOMMENDED READING

Carpenter, Art. 2012. Carpenter's Complete Guide to the SAS Macro Language, 2nd ed. SAS Institute
Carpenter, Art. 2012. Carpenter's Guide to Innovative SAS Techniques. SAS Institute

CONTACT INFORMATION

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

Rowland Hale, Principal Statistical Programmer (Systems Development)
inVentiv Health Clinical
Joachimsthaler Strasse 10-12
10719 Berlin, Germany
rowland.hale@inventivhealth.com
inventivhealthclinical.com

Hannes Engberg Raeder, Senior Clinical Programmer
PRA Health Sciences
Gottlieb-Daimler-Strasse 10
68165 Mannheim, Germany
engbergraederhannes@prahs.com
prahs.com

Brand and product names are trademarks of their respective companies.