SESUG 2022 Paper 213

Why Write Base SAS® Code When the Macro Processor Can Do It for You

Andrew E. Walker, North Carolina State University

ABSTRACT

As SAS programmers, we need to infuse resiliency in our programs while reducing maintenance time and the likelihood of errors caused by faulty logic or typos. The macro processor is another tool in our toolbox to achieve such goals. This paper demonstrates how to use the macro processor to write dynamic Base SAS® code with two examples: 1) write a dynamic if/else if statement and 2) bin creation when completing a by-group analysis. This paper concludes with details on how to document resolved macros in the log and export the macro to an output file for documentation.

INTRODUCTION

There are three basic levels of macro programming: 1) define and call a macro variable (e.g., %LET), 2) create a macro program (e.g., %MACRO %MEND), and 3) create a macro that dynamically writes Base SAS® code. As Art Carpenter (2016) described, the learning curve from the first to the second level of the macro program is gradual and not all that difficult to overcome. However, going from the second to the third level can be steeper and takes more time and practice. This paper looks specifically at the third level of programming – dynamic SAS® programming.

Is this paper for you?
This paper is helpful to those unfamiliar with macros and those with moderate to high levels of macro experience. If you have not used %LET or %MACRO in a program, or they sound foreign, becoming more familiar with macro processing is highly recommended. However, no matter your experience or skill level, readers should be able to follow along and gain a better understanding of macro programming.

What does dynamic macro processing mean?
Dynamic macro SAS® programming changes based on facts external to the SAS® code. For example, the names of variables in a data set, the number of data sets in a SAS® library, or the number of data bins a user wants to generate. Art Carpenter and his published work Carpenter's Complete Guide to the SAS Macro Language, Third Edition, inspired the next two examples. The book gives dozens of examples of all three levels of SAS® programming. The first example dynamically builds if/else if statements based on a lookup table.

Hardcoded if/else if statements are common in SAS® programming. However, programs should be written dynamically to reduce the opportunity for error. To accomplish this goal, programmers should avoid hardcoded if/else if statements unless they are not subject to change over time.

Packaging programs is yet another strong opportunity for deploying dynamic programming. The programmer may not initially know values to include in the if/else if statement. To get around this the program can utilize the macro processor to look at an external source and write code based on the source. The programmer may be responsible for creating or reviewing long lists of if else/if statements. If the if/else if statements have predictable behavior, the program can use macro processing to create all of the if/else if statements. There are many more situations this method is useful. However, for brevity, let us start with the if/else if example.

DYNAMIC PROGRAMMING WITH AN IF/ELSE IF EXAMPLE

To get common data sets, run Figure 1 to generate two data sets: STUDENT_GRADES and GRADES_CROSSWALK. The data is simulated data sets from a college. At this example college, they only offer English 111, Math 101, Geology 200, and Psychology 100.

Figure 1 Base SAS® Code to Generate Project Data

```
Data Student_Grades (DROP=i a);
 ARRAY Coursez(4) $ _TEMPORARY_ ('ENG-111','MAT-101','GEO-200','PSY-
 100') ;
 DO i=1 TO 1000;
   student_ID = rand("integer", 1, 100);
     DO a=1 to 4;
     course = Coursez[a];
     grade_ID=(rand("integer",1,20)/2);
     IF mod(grade_ID,1)=0 THEN OUTPUT; END;END;
 Proc Sort Data=&SYSLAST Nodupkey; by Student_ID Course;
Run;

Data grades_crosswalk (DROP=i);
 ARRAY Gradez(10) $ _TEMPORARY_
 ('A','B','C','D','F','Z','X','W','.','WX');
 ARRAY Grade_IDz(10) _TEMPORARY_ (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
 ARRAY Succ(10) _TEMPORARY_ (1,1,1,0,0,0,0,0,.,.);
 DO i=1 to 10;
     grade_ID = grade_IDz[i]; grade=gradez[i]; success=succ[i];OUTPUT;
 END;
Run;
```

With the data generated, the next aspects to discuss are macro processing and project goals.

OVERALL GOAL

The overall goal is to create two variables called “grade” and “success,” where grade represents traditional alpha grades (e.g., “A” “B” “C” etc.) and success as “1” and unsuccessful as “0.” The next goal relates to programming methods. A program that requires maintenance is destined for failure and is a drain on future resources! It is bad practice to hardcode the if/else if statement into the code. The practice requires a person to review the code, look at an external crosswalk, and then recode the if/else if statement. We want to build base SAS® code that is self-sustaining. The program uses the GRADES_CROSSWALK to construct the if/else if statement.

GENERAL STEPS

There are four general steps to accomplish the goals. First, the data must be accessible. Second, the programmer must adhere to the golden rule of data analysis - know thy data. Third, create macro variables from GRADES_CROSSWALK where each data point is represented by a macro variable. Lastly, create the macro program that uses the new macros to create an if/else if statement.

Access to data is step one. In this example, step one is accomplished by running Figure 1. Step two takes more time and should never be skipped. More often than not, mistakes in code are not a result of syntax errors but a result of not knowing thy data. On that note, student_ID, course, and grade_ID are in the STUDENT_GRADES data set, where student_ID represents a student identification number unique to each student, the course represents a course the student attempted, and grade_ID represents the key to a lookup table containing traditional grades and success outcomes. The grade_ID is the link between the STUDENT_GRADES table and the GRADE_CROSSWALK table.

The STUDENT_GRADES lookup table contains traditional grades, grade_ID represents the key to the table, and success indicates the outcome for the student where success equals 1, 0 is unsuccessful, and "." represents values to exclude from the calculation. The program uses the GRADE_CROSSWALK table to build the desired code.

The program needs to create a series of macro variables, so Figure 2 at ❶ _NULL_ tells the data step compiler not to create a data set. The program then points to the GRADES_CROSSWALK table at ❷ and utilizes the end option at ❹. The end option creates a new temporary variable indicating a "1" if the current row is the last row of the data set.

Steps ❸ utilize CALL SYMPUTX. This is where the magic happens. The data step compiler looks at ❸ and then calls the SYMPUTX function. Next, the concatenate function combines the literal text "grade" and the auto-generated _N_ variable into one macro variable. Once the concatenate function is complete, SYMPUTX looks at the first row of the GRADES_CROSSWALK table and assigns the macro variable &GRADE1 equal to "A." A grade of "A" is the first row of the table. After creating &GRADE1, &GRADE_ID1, and &SUCC1, are created, the Data Step Compiler evaluates if the current row, row 1, is the last record. If not, ❸ runs again, but each macro variable is now &GRADE2, &GRADE_ID2, and &SUCC2. ❺ is executed if it is the last record of the data set. &NUMROWS represents the total count of records in your data set. The program uses the macro variable &NUMROWS later to build a do loop.

Figure 2 Generate Macro Variables from a Crosswalk Table

```
❶ Data _NULL_;
      ❷SET grades_crosswalk ❹END=TheEnd;
      ❸ CALL symputx(CAT("grade",_N_),grade);
         CALL symputx(CAT("grade_ID",_N_),grade_ID);
         CALL symputx(CAT("Succ",_N_),success);
      ❺IF TheEnd THEN CALL SYMPUTX('Numrows',_N_);
Run;
```

The entire data set and each designated variable are now associated with a unique macro variable. For example, &GRADE6 resolves to grade "Z" because "Z" is on the sixth row of the table. &SUCC6 resolves to 0 because &SUCC6 is on the sixth row of the table. This program creates dynamic macro variables negating the necessity to hardcode if/else if statements. How is this dynamic? The code changes when GRADES_CROSSWALK changes.

Quick note, you can view the newly created macro variables with:

```
%PUT _USER_;
```

or with:

```
Proc SQL;
```

```
     SELECT name, value
     FROM dictionary.macros
     WHERE scope='GLOBAL' AND
     SUBSTR(name,1,2) IN ('GR','SU','NU');
Quit;
```

The next step is generating the actual if/else if code. In Figure 4 at ❶, the macro program "*TestIf*" is defined. At ❷, the data set called "Recode" is initialized. The process of creating the dynamic if/else if starts at ❸. In this example, the do loop has an iterator of "a" starting at 1 and goes to &NUMROWS. When the macro processor is done with ❸, the code is `%DO a=1 %To 10;` Where &NUMROWS resolves to 10.

At ❹, the macro processor builds the code dynamically. If it is the first loop, the do loop does not write "ELSE" to the compiler. However, if it is not the first time, "ELSE" is written to the if/else if statement. For example, on the first loop, "&a" resolves to 1. Next, the macro processor evaluates if 1 > 1. If the answer is no, it does not write "Else" as output. In this case, "Else" is **not** written because it is the first time through the loop.
So, the first line of code:

```
     if Grade_ID=1 Then DO; Grade= "A"; Success=1; end;
```

The next important concept to understand is the utilization of "&&" with "&a." At ❺, &&GRADE_&A is referenced. The macro processor can only process one & at a time. So, the macro processor looks at the first & of &&GRADE_&A and asks, is there a macro variable named "&Grade_?" The macro variable "&GRADE_" is nonexistent, so the "&" is dissolved. The word scanner continues down the text string and finds the third "&." It is recognized as a macro variable and triggers the macro processor to look for a macro called "a." As defined by the loop, there is a macro called "a" which resolves to 1. The macro processor only sends the code to the data step compiler until all macro variables are resolved. So, the code can't have an &. The second "&" was skipped the first time. Therefore, the code still has ampersands. After the first time, "&&GRADE_&A" now looks like "&GRADE_1." Remember, the SAS® program created all the macro variables in Figure 3 which has a macro variable called "&GRADE_1". The second time through the code, the macro processor encounters "&GRADE_1" which resolves to the letter grade of "A." It is vital to understand how the SAS® macro processor and tokenizer resolves ampersands. Lyons (2017) goes into detail as to how each character is tokenized, read by the word scanner, resolved by the macro processor, and finally sent to the input stack. It is worth the time to read.

On the second iteration or loop, "&a" resolves as 2 and therefore is greater than 1. So, the `%IF 2>1` evaluates as TRUE, and "Else" is written to the input stack before the rest of the if statement. The line of code written to the input stack on the second loop is:

```
     Else if Grade_ID=2 Then DO; Grade= "B"; Success=1; end;
```

The "Else" is now in front of "if Grade_..." The iteration continues until the end. In this case, 10. When the loop hits 11, it is out of bounds and ends. "Run" is sent through the word scanner and then to the input stack, and all the code is ready for execution.

Figure 3 Dynamic SAS® Code

```
❶%MACRO TestIf;
❷DATA Recode;
        SET STUDENT_GRADES;
       ❸%DO a=1 %To &Numrows;
       ❹ %IF &a>1 %THEN Else;
            ❺if Grade_ID=&&Grade_ID&a Then DO;
            Grade= "&&Grade&a";
            Success=&&SUCC&a; end;

      %END;
   Run;
%MEND TestIF;

%TESTIF;
```

The next logical question could be, why would I go through all this trouble if I can write a simple SQL statement, as seen below:

```
Proc SQL;
      CREATE TABLE Recode_SQL AS
      SELECT A.*, B.Grade, B.Success
      FROM STUDENT_GRADES A
      LEFT JOIN grades_crosswalk B
      ON A.grade_ID=B.grade_ID;
Quit;
```

That is a great question! Performance is the first response. After minimal testing with 20 million records by increasing the `do i=1 to 10000000` in Fig. 1, the macro method was three times faster than the SQL method. This could be just one of several processes in this program. Time adds up quickly when rows increase. If you mind the second, the seconds will mind the minutes, and the minutes will mind the hours. We need to do everything we can to reduce clock time and maximize computing resources.

DYNAMIC BIN CREATION

The inspiration for the next example originated from a SAS® on-demand SAS® Webinar titled Top 5 Handy PROC SQL Tips by Charu Shankar, 2021. In the presentation, the content detailed a very interesting way to create bins in PROC SQL. I appreciated the method, so I wanted to include the idea in my code base. However, I wanted to get away from hardcoding. When applying the method to the RECODE data set, the method grouped and counted all students with a success rate between 0 and 10 percent, then grouped all students with a success rate between 10 and 20. The bins would increase until 100. The example is hardcoded, which, if you cannot tell by now, is not my preferred method. Therefore, I started thinking about how SAS® can automate code generation and bin selection, produce several bin sizes automatically, and finally generate a bin count that is more appropriate based on Sterges Rule.

In Figure 5 at ❶, the code counts the number of observations within each bin. This code is nice, neat, and to the point. I appreciated the method. That is why I spent more time thinking about the code. However, this method requires maintenance if the bins change, which takes time, energy, and resources. We want to write code requiring little to no maintenance. Now, this is a great example if we never expect the bins to change. However, this may not be the best method if the bins change in the future.

Figure 4 Hardcoded Method

```
Proc SQL;
Create Table Hardcode As
Select ❶SUM(Total_Credits Between 0 and 19) as 'From 0 to 19'n,
      SUM(Total_Credits Between 19 and 38) as 'From 19 to 38'n,
      SUM(Total_Credits Between 39 and 58) as 'From 39 to 58'n
From Total_Credit_By_Student;
Quit;
```

The next example takes the same idea and leverages SAS® to generate code. In this example, the code assumes 10 equal bins. Figure 6 generates a data set with 1,000 observations and a variable called TOTAL_CREDITS representing the total sum of credits by student.

Figure 5 Data Generation for Bin Creation

```
Data Total_Credit_By_Student;
   Do i=1 to 1000;
   Total_Credits =FLOOR((rand('Uniform')*100));
   Academic_Term ='2024FA'; Output;
   End;
   Rename I = StudentID;
Run;
```

The goal is to count the number of students within each bin. To do this, the macro needs the maximum number of credits earned, the number of bins to create, and the number of credits in each bin. Figure 7 looks at the TOTAL_CREDIT_BY_STUDENT data set, finds the MAX TOTAL_CREDITS, and places it into a macro variable called MAX_CREDIT. A %PUT statement places the value in the log for review.

Figure 6 Find MAX Credits by Student

```
Proc SQL NoPrint;
  Select MAX(Total_Credits)
  Into :Max_Credit
  From TOTAL_CREDIT_BY_STUDENT;
Quit;

%PUT &Max_Credit;
```

The next task is to detail bin count as seen below:

```
%LET BinCount=10;
```

The next step determines how many credits are in each bin. For example, if the max number of credits is 99 and a later process requires ten bins, then each bin needs a width of nine credits. `%EVAL` is used next. `%EVAL` immediately evaluates mathematical calculation or logical expression. For example, `%EVAL(`2+1), the macro processor immediately completes the equation and resolves the equation to 3. As seen below, `%EVAL` is used to divide &MAX_CREDIT by &BINCOUNT. The `%PUT` statements detail the results:

```
%LET Bin=%EVAL(&MAX_CREDIT/&BinCount);
%PUT Bin count is &BinCount;
```

```
%PUT Bin size equals &Bin;
```

With the required calculation completed, the next step is to review the macro. In Figure 8 at ❶, the do loop starts at zero and goes to &MAX_CREDITS (e.g., is 99 in this example) and increases by &BIN size (e.g., nine credits). Notice the absence of a set, merge, or alike. Because the program is creating a table containing the code. The program then embeds the code in a subsequent SQL procedure. The results of the data step are in Figure 9.

Figure 8 Complete Bin Creation SAS® Code

```
Data Recode_Code;
❶ Do i=0 to &Max_Credit by &Bin;
   Low=i;
   High = i + &Bin;
   Statement=CATX(' ','SUM ( Total_Credits Between' ,PUT(Low,8.),
   'and',PUT(CATS(High-1,'.99'),8.),')',COMPRESS(QUOTE(CAT(PUT(Low,8.)
   ,'-',PUT(CATS(High-1,'.99'),8.)))),',');
   If i <= &Max_Credit and high > &Max_Credit Then Statement = CATX('
   ','SUM ( Total_Credits Between' ,PUT(Low,8.), 'and',
   PUT(High,8.),')',COMPRESS(QUOTE(PUT(Low,8.))));
   Output; End; Drop i;
Run;
```

Figure 9 Bins

```
SUM ( Total_Credits Between 0 and 8.99 ) "0-8.99" ,
SUM ( Total_Credits Between 9 and 17.99 ) "9-17.99" ,
SUM ( Total_Credits Between 18 and 26.99 ) "18-26.99" ,
SUM ( Total_Credits Between 27 and 35.99 ) "27-35.99" ,
SUM ( Total_Credits Between 36 and 44.99 ) "36-44.99" ,
SUM ( Total_Credits Between 45 and 53.99 ) "45-53.99" ,
SUM ( Total_Credits Between 54 and 62.99 ) "54-62.99" ,
SUM ( Total_Credits Between 63 and 71.99 ) "63-71.99" ,
SUM ( Total_Credits Between 72 and 80.99 ) "72-80.99" ,
SUM ( Total_Credits Between 81 and 89.99 ) "81-89.99" ,
SUM ( Total_Credits Between 90 and 98.99 ) "90-98.99" ,
SUM ( Total_Credits Between 99 and 108 ) "99"
```

Figure 10 places the table contents into a macro variable called “Groups.” Figure 10 creates the macro variable “Groups” from `Recode_Code` where each row is combined into one long text string separated by a space.

Figure 7 Create Macro Variable with SQL

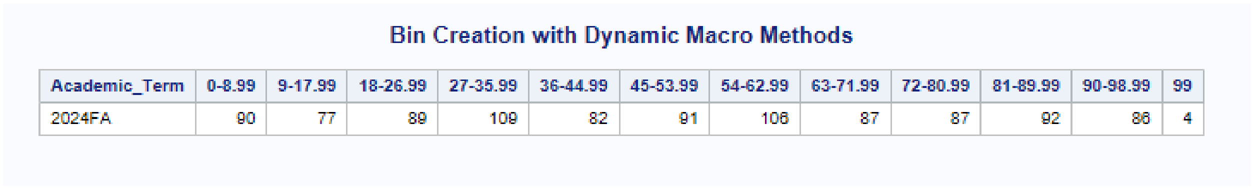
```
Proc SQL NoPrint;
      Select Statement
      Into :Groups separated by ' '
      From Recode_Code;
Quit;
```

Now, all that needs to be done is to call the macro in the appropriate place in the SQL procedure as seen in Figure 11 at ❶;

Figure 8 Call MACRO

```
Proc SQL;
      Select Academic_Term,
             ❶ &Groups
      From Total_Credit_By_Student
             Group by Academic_Term;
Quit;
```

There are ten bins each with a width of 8.99 and a correlated label.



Bin Creation with Dynamic Macro Methods

Academic_Term	0-8.99	9-17.99	18-26.99	27-35.99	36-44.99	45-53.99	54-62.99	63-71.99	72-80.99	81-89.99	90-98.99	99
2024FA	90	77	89	109	82	91	106	87	87	92	86	4

With the report output, the next step is to look at key features of the program responsible for generating the bins. Figure 12 at ❶, concatenates all the text into one statement with all the pieces separated by a space. At ❷, the known pieces of the equation are prescribed: "SUM (Total_Credits Between." At ❸, `Low` is dynamically placed for each bin. When the code runs, the programmer does not know the third bin's low band is 18. Next an "and" is placed and at ❹ the high limit of the bin is added. At ❺, the program creates the label for the report. Remember, the goal is to avoid hardcoding ten lines of code for each bin. We want SAS® to create the actual code. We want to make the skeleton and SAS® fill out-out the rest.

Figure 9 Building Block One for SQL SUM Statement

```
Statement=❶CATX(' ', ❷'SUM ( Total_Credits Between',
❸PUT(Low,8.),'and', ❹PUT(CATS(High-1,'.99'),8.),')',
❺COMPRESS(QUOTE(CAT(PUT(Low,8.),'-',PUT(CATS(High-
1,'.99'),8.)))),',');
```

The first time through the code produces:
SUM (Total_Credits Between 0 and 8.99) "0-8.99" ,

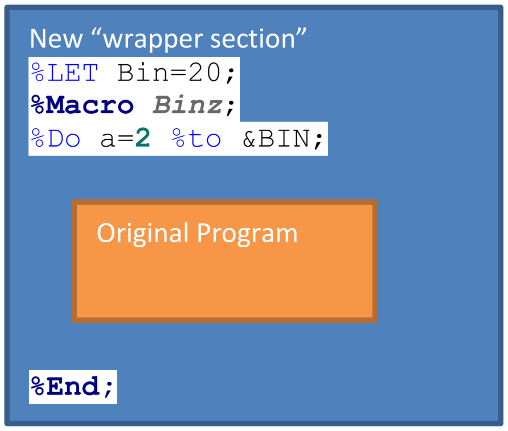
The do loop creates each bin with special attention to the last bin. With PROC SQL, if a comma is at the end of the last statement, the code will produce an error. Figure 13 at ❶ demonstrates how SAS® determines the last bin and runs the code excluding the comma.

Figure 10 Building Block Two for SQL SUM Statement

```
If ❶ high > &Max_Credit Then
Statement = CATX(' ','SUM ( Total_Credits Between',PUT(Low,8.),
'and',PUT(High,8.),')' ,COMPRESS(QUOTE(PUT(Low,8.))));
```

The current macro requires a bin count (e.g., `%LET BinCount=10;`). So, if the program needs to run for multiple bin sizes (e.g., 10, 12, 25), the current program must run multiple times to get the desired results. We do not want to do that, we want SAS® to do that for us. Therefore, the original program is placed in a do loop as illustrated in Figure 14. The inner box represents the original macro program, and the outer box represents the wrapper do loop.

Figure 11 Nested Macro



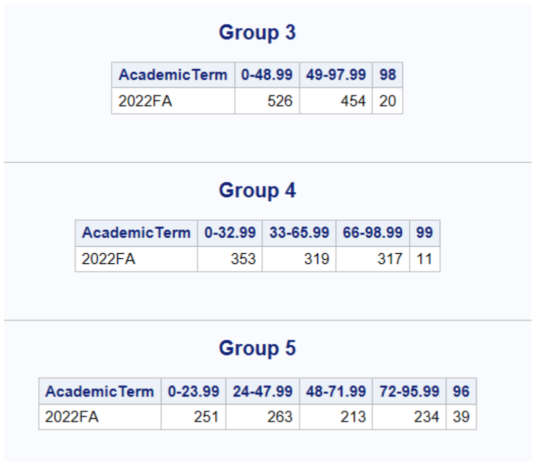
```
%LET Bin=20;
%Macro Binz;
%Do a=2 %to &BIN;

%End;
```

Figure 12 Entire Program - Nested Macro Program

```
%LET Bin=20;

%Macro Binz;
%Do a=2 %to &BIN;
   %LET Bin=%EVAL(&Max_Credit/&a);
Data Code;
   Do i=0 to &Max_Credit by &Bin;
   Low=i;
   High = i + &Bin;
   Statement=CATX(' ','SUM ( Total_Credits Between',
   PUT(Low,8.),'and',PUT(CATS(High-1,'.99'),8.),')'
   ,COMPRESS(QUOTE(CAT(PUT(Low,8.),'-',PUT(CATS(High-1,
   '.99'),8.)))),',');
   If i <= &Max_Credit and high > &Max_Credit Then Statement =
   CATX(' ','SUM ( Total_Credits Between',PUT(Low,8.),
   'and',PUT(High,8.),')',COMPRESS(QUOTE(PUT(Low,8.))));
      Output; End; Drop i;
Run;
Proc SQL NoPrint;
   Select Statement
   Into :Groups separated by ' '
   From Code;
Quit;

Title Group %EVAL(&a+1);
Proc SQL;
   Select Academic_Term,
   &Groups
   From Total_Credit_By_Student
   Group by Academic_Term;
Quit;
%End;
```

```
Title Group &a;
Proc SQL;
   Select Academic_Term,
   &Groups
   From Total_Credit_By_Student
   Group by Academic_Term;
Quit;
%Mend Binz;

%BINZ;
```

Figure 13 Sample Output



Group 3

AcademicTerm	0-48.99	49-97.99	98
2022FA	526	454	20

Group 4

AcademicTerm	0-32.99	33-65.99	66-98.99	99
2022FA	353	319	317	11

Group 5

AcademicTerm	0-23.99	24-47.99	48-71.99	72-95.99	96
2022FA	251	263	213	234	39

As seen in Figure 16, this one report produces all possible bin sizes. Very useful. The next idea to discuss is having SAS® determine the appropriate number of bins. For example, preparing data for a histogram. A histogram should not be too simple but should not overwhelm the reader with too many bins. We want just the right number of bins. So, what is the recommended number of bins? Up to now, the program has yet to determine bin counts. The user defined the number of bins as they see fit. However, it is easy to adapt the code to meet any statistical requirements.

For producing the proper number of bins, it can be appropriate to replace a hardcoded macro variable with a statistical method. For example, when the programmer does not know the size of the population before the analysis. Using a bin size of nine would be cumbersome with a total population variance of 9,000. That would result in 1,000 bins. To get around hardcoded bin sizes, Sturges rule is applied. It is a mathematical method to determine the appropriate number of bins for an analysis.

Figure 14 Sturges Rule

Optimal Bins = $[\log_2 n + 1]$

To make the calculation SAS® must know the total number of observations. Figure 18 uses PROC SQL to find the total count and places the count into a macro variable called totobs – short for total observations.

Figure 15 Find Total Observations With PROC SQL

```
proc sql;
```

```
      select nobs into :totobs
      from dictionary.tables
      where libname='WORK' and memname='Total_Credit_By_Student';
quit;
```

In Figure 19 at ❶, %SYSFUNC is used to run the LOG2 function, at ❷ it is used again to round the LOG2 results as a whole number, then at ❸, %EVAL is used to add 1 to the product. Simply stated, %SYSFUNC allows users to immediately resolve data step functions outside the data step but while inside the confines of macro activities. As previously stated, %EVAL tells the macro processor to evaluate integer mathematical equations or logical expressions immediately.

Figure 16 Dynamically Create BinCount

```
%LET BinCount =
❸%EVAL(❷%SYSFUNC(ROUND(❶%SYSFUNC(LOG2(&totobs))))+1);
```

All of the pieces are now in place to run the original macro program from Figure 9, but have SAS® determine the appropriate number of bins for analysis. The program is self-sustaining and ready for any sized data set with which it can create the appropriate number of bins.

DEBUGGING MACROS

Sometimes we want to see what is going on with the macro processor. Fortunately, there are options to write macro processor activities to the log. They include MPRINT, SYMBOLGEN and MLOGIC. MPRINT prints all code generated by the macro processor. SYMBOLGEN details when a macro variable resolves. For example, when "&GRADE_1" resolves to 1. Lastly, MLOGIC writes all execution steps to the log. For example, when %IF/%ELSE %IF is evaluated as true for false. These options are vital for beginner, intermediate, expert, and master macro users. The default setting is off. To turn each option on run:

OPTIONS MLOGIC MPRINT SYMBOLGEN;

The options will increase the amount of information in the log. After debugging, if you need to turn them back off, run:

OPTIONS NOMLOGIC NOMPRINT NOSYMBOLGEN;

WRITING MACRO VARIABLES AS OUTPUT

The advantage of writing dynamic code is now available to you. However, when the code changes, there may be a need to document how things are resolved. There is an easy method is to write all the user-defined SAS macros to a txt file.

Figure 20 details how to output a simple txt file with the naming convention today's date as YYYYMMDD followed by "_Generated_Macros.txt." There are two lines of data. The "@1" places the output "Bin Count is" followed by the macro variable "&BINCOUNT" at the first position of the first line. The "/" is a carriage return and tells SAS® to go to the subsequent line. It then outputs "Bin size equals" followed by the resolved "&BINCOUNT" macro.

Figure 17 Export Macro Variables

```
DATA _NULL_;
 FILE "C:\My
Location\%Sysfunc(Today(),YYMMDD10.)_Generated_Macros.txt";
PUT
@1 'Bin count is '  "&BinCount" /
@1 'Bin size equals ' "&Bins" ;
Run;
```

```
Bin count is 10
Bin size equals 9
```

This is a very basic example. However, these methods can document the entire log, all macro activities, or specific macro output. These tools allow a Base SAS® programmer to debug and document their macro activities. Very powerful!

CONCLUSION

In summary, creating dynamic adaptive Base SAS® code saves time, energy, and reduces maintenance issues. The if/else if macro method allows the programmer to create adaptive code thereby ensuring the program is up-to-date. Bin creation is a common activity in data analysis. This paper demonstrates how Base SAS® code generates a specific number of desired bins without hardcoding. The next level demonstrated how SAS® can loop through a sequence of bin sizes.

Lastly, the paper demonstrated statistical methods to determine the appropriate number of bins. The exact application may not apply to your specific situation, however, the principles of building code dynamically, creating loops, and having SAS® determine bin size and then creating the appropriate code are vital for all SAS® programs.

Each method presents positives and negatives, and each method is another tool in our SAS® toolbox.

REFERENCES

Carpenter, A. 2016. *Carpenter's Complete Guide to the SAS Macro Language*. 3rd ed. Carry, NC: SAS Institute Inc.

Lyons, L. 2017. "Going Under the Hood: How Does the Macro Processor Really Work?" as Northeast SAS Users Group Conference 2017.

Shanker, C. (2021). Top 5 Handy PROC SQL Tips from. Presented as SAS - Ask the Expert.

RECOMMENDED READING

- *Carpenter's Complete Guide to the SAS Macro Language*. 3rd ed. *SAS® For Dummies®*

ACKNOWLEDGMENTS

Thank you Kelly Smith and Anne Oxenreider for your continued support as I travel along my SAS journey!

CONTACT INFORMATION

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

Andrew E. Walker
aew5044@gmail.com