Paper 083-2011

Developing User-Defined Functions in SAS®: A Summary and Comparison

Songfeng Wang, University of South Carolina, Columbia, SC
Jiajia Zhang, University of South Carolina, Columbia, SC

ABSTRACT

In this paper we review the methods for SAS® users to write their own functions. These methods include SAS macro, SAS/IML, SAS Component Language (SCL), and two recently available procedures: PROC FCMP and PROC PROTO. We give examples showing how to make user functions using each method. The computation time of different methods is investigated. The pros and cons of each method are summarized, which can give SAS users basic ideas about choosing the appropriate method according to their own purposes.

INTRODUCTION

SAS programs usually consist of a number of DATA steps or built-in PROC steps. Compared to other command-based or object-oriented programming languages, it is not straightforward for SAS users to write their own functions. SAS macro has played an important role for users to re-use some of their code without doing much cut-and-paste for a long time. SAS/IML and SAS Component Language (SCL) also provide possibilities for users to implement functions-like modules and methods into their programs. Nowadays, with the introduction of PROC FCMP and PROC PROTO, SAS users can have more options and flexibility in developing and programming the functions they want. In this paper, we go over the existing methods with examples to show how to use these methods, and then comparisons of these methods are made.

SAS MACRO

SAS macro has a long history of being used as the major alternative for user-developed functions. According to SAS Marco language reference, the macro facility is “a tool for extending and customizing the SAS system and for reducing the amount of text you must enter to do common tasks”.
The simplest type of SAS macro facility is macro variables, which are substituted into your program wherever the macro variables are referenced. So the basic idea behind macro variables is text substitution. Macro variables can be useful when common tasks are to be performed to different values by simply changing the values of the macro variables. Similar to macro variables, macro programs provide a way to substitute text into SAS programs. Macro programs can be written to run a set of DATA steps or procedures repeatedly, and therefore a batch of similar tasks can be done by reusing the same code. Conditional logic and decisions are available in macro programs, so the macro programs can be flexible and dynamic enough for complicated tasks. Furthermore, SAS macro facility provides a variety of macro statements and functions to make the coding easier and more efficient. There are four ways to make macros available to current programs: to compile a macro and use it for current sessions, to save it as a permanent macro and then use a **%INCLUDE** statement, to call it through the autocall facility, or stored it as a compiled macro. A macro variable is defined with **%LET** statement and referenced by preceding the macro variable name with an ampersand (&). A macro program usually starts with **%MACRO** statement, and then includes SAS data set and variable names, SAS or macro statements and procedures, and it ends up with a **%MEND** statement.

Example 1: Here is a simple example of using macro to find the larger value of two integers.
If you need to include source code:

```
%macro max(dat);
data _null_;
set &dat;
if x>=y then put x=;
else put y=;
%mend;

data temp;
  input x y;
  cards;
 7 3
;
run;
```

```
%max(temp);
```

The above macro program defines a macro program **max**, which can refer to a SAS dataset **dat**. When we call the macro with %***max***(temp), **&dat** is replaced with **temp** and the code looks like this:

```
data _null_;
set temp;
if x>=y then put x=;
else put y=;
```

It prints x=7 in the log window. However, due to the basic idea of macro is text substitution, developing SAS macro may not be very intuitive and straightforward for users who are more familiar with statistical languages like R/S-plus or STATA. Another issue is that although SAS macro can be easily shared by others, the code is not very easy to read, modified or maintained.

Example 2: This example is about calculating the value of π from the infinite series $\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} \cdots$. We are interested in a function that can take the length of the series as a parameter. A macro corresponding to this function is described as below.

```
%macro calpi(niter=);
data pi;
temp=0;
do i=1 to &niter;
temp=temp+4*(-1)**(i+1)*1/(2*(i-1)+1);
end;
put temp;
run;
%mend;

%calpi(niter=5000);
```

The above macro uses a series with a length of 5000, and it prints 3.1413926536 in the log window.

Example 3: Let's take a look at a more complicated example. Suppose we would like to perform a Monte Carlo simulation to investigate the coverage probability of 95% confidence intervals for coefficients in a linear regression model. The model is specified as $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \epsilon$, where $X_1 \sim Uniform(0,1)$, $X_2 \sim Binomial(1, 0.5)$, and $\epsilon \sim Normal(0,1)$. A macro named **covsim** is created, which has the following parameters: the number of datasets (**nsim**), the sample size of each dataset (**nsize**), the true values of parameters (**beta0**, **beta1**, **beta2**), and the seed used to generate the random error (**seed**). The macro uses **PROC GENMOD** to fit a regression model for each simulated dataset, and then uses **ODS OUTPUT** to save the 95% CI from each simulation into a dataset named **parestimate**. The coverage probability can then be derived from this dataset accordingly.

```
%macro covsim(nsim=,  nsize=, beta0=, beta1=,beta2=,  seed=, simout=);
data &simout;
* generate the datasets ;
do sim = 1 to &nsim;
 do index=1 to &nsize;
x1=RANUNI(12345);
x2=RANBIN(12345,1,0.5);
y = &beta0 + &beta1*x1+ &beta2*x2+RANNOR(&seed);
output;
end;
end;
run;

ods output "Analysis Of Parameter Estimates"=parestimate;
proc genmod data=&simout;
class x2 /descending;
model y=x1 x2 /d=n;
```

```
by sim;
run;
ods output close;
run;

data temp;
set parestimate;
if Parameter="x1" & LowerWaldCL<=&beta1 &  &beta1<=UpperWaldCL    then cover1=1; else
cover1=0;
if Parameter="x2" & LowerWaldCL<=&beta2 &  &beta2<=UpperWaldCL    then cover2=1; else
cover2=0;
run;

/**the coverage of beta1**/
proc freq data=temp;
table cover1;
where Parameter="x1";
run;
/**the coverage of beta2**/
proc freq data=temp;
table cover2;
where Parameter="x2" & level1='1';
run;
%mend;
```

If we would like a simulation with 1000 datasets generated, 200 observations in each dataset, true coefficients of (1, 3, 2), and a seed of 123. A call of the macro %***covsim***(nsim=**1000**, nsize=**200**,beta0=**1**, beta1=**3**,beta2=**2**, seed=**123**, simout=results) will output a coverage probability of 0.945 for β_1 and 0.952 for β_2, and both are reasonably close to 0.95.

SAS/ IML (INTERACTIVE MATRIX LANGUAGE)

A SAS/IML module enables user to reuse statements by passing matrices into the module as input data. Generally speaking, there are two types of modules: function modules that return values, and subroutine modules that usually perform some operations without returning any values. Therefore, SAS users can use SAS/IML modules to define their own functions or subroutines. A function module can be used in assignment statements or expressions while a subroutine module can be called by using the **CALL** or **RUN** statement. A module is defined with the **START** and **FINISH** statements. The **START** statement defines the name and the arguments of the module; the **RETURN** statement returns a value from a function module, and then the **FINISH** statement indicates the end of user-defined module.

Example 1 (continued): Here is an example of using SAS/IML for a user-defined function to find the larger one of two integers.

```
proc iml;
start max(x,y);
if x>=y then return(x); else return(y);
finish max;
a=20;
b=40;
n=max(a,b);
print n;
quit;
```

A module named **max(x,y)** is defined in the above SAS/IML code. The module is called in the assignment statement **n=max(a,b)** and the value n is then printed out. The above code will print 40 in the log window.

Example 2 (continued): The same example to calculate the value of π from the infinite series.

```
proc iml ;
start calpi(niter);
temp=0;
do i=1 to niter;
```

```
temp=temp+4*(-1)**(i+1)*1/(2*(i-1)+1);
end;
return (temp);
finish;
store module=calpi;
quit;

proc iml;
load module=calpi;
pi=calpi(5000);
print pi;
quit;
```

The above code first defines a module named **calpi**, which has a parameter called **niter** that indicates the length of the infinite series used. Then the module is called in another **PROC IML** with **niter** set to be 5000. The code will print 3.1413927 in the output window.

Example 3 (continued):
For the same example about using Monte Carlo simulation to investigate the coverage probability of 95% confidence intervals, the following SAS/IML code creates a user module named **CoverageSimu** and save it for further use:

```
proc iml ;
start CoverageSimu(nsim,nsize,beta,seed);
cover={0,0,0};
do sim = 1 to nsim;
x1=RANUNI(J(nsize,1,12345));
x2=RANBIN(J(nsize,1,12345),1,0.5);
x=j(nsize,1,1)|| x1 ||x2;
y = x*beta+RANNOR(J(nsize,1,seed));
 xpxi=inv(t(x)*x);          /* inverse of X'X           */
  ebeta=xpxi*(t(x)*y);       /* parameter estimate       */
   yhat=x*ebeta;              /* predicted values         */
    resid=y-yhat;                 /* residuals            */
    sse=ssq(resid);               /* SSE                  */
     dfe=nrow(x)-ncol(x);          /* error DF             */
 mse=sse/dfe;                  /* MSE                     */
 stdb=sqrt(vecdiag(xpxi)*mse); /* std of estimates        */

lower=ebeta-1.96*stdb; /Lower Bound of 95% CI/
upper=ebeta+1.96*stdb;  /Upper Bound of 95% CI/
aa=lower<=beta & upper>=beta;
cover=cover+aa;
end;
return (cover/nsim);
finish;
store module=CoverageSimu;
quit;

proc iml;
load module=CoverageSimu;
cover=CoverageSimu(1000,200,{1,3,2},123);
print "Simulation Results", {'beta0', 'beta1','beta2'} cover;
quit;
```

The above SAS/IML code uses the matrix operations to derive the standard deviations of fitted regression parameters, and then calculates the coverage probability based on the 95% CI for each simulated dataset. When the code is submitted, the user-defined module **CoverageSimu** will be saved in the default SAS library **Work.Imlstor**, and then the module can be called in other SAS/IML procedures.

Although they give the same results, the above SAS/IML program is much faster than using the macro **covsim**. This owes to the ability of SAS/IML to handle matrix, which provides a more efficient way to deal with some complex tasks than many SAS procedures. Furthermore, a wide range of built-in subroutines are available within SAS/IML, which

can be helpful to make the programming process flexible and easier. However, the user modules defined in SAS/IML can only be used in the IML procedure and can not be used by DATA step or other procedures.

SAS COMPONENT LANGUAGE (SCL)

SCL was called Screen Control Language in earlier release of SAS. With the introduction of several new features that enable the object-oriented applications, the name is changed to SAS Component Language since SAS version 8. SCL is the scripting language behind SAS/AF, SAS/FSP, and SAS/EIS software, and it provides a way to develop interactive and menu-driven applications using SAS procedures, DATA steps and SAS macros. Similar to Base SAS language and many other programming languages, SCL contains statements, functions and CALL routines, which enhance the readability of SCL code for SAS users. Moreover, SCL provides other features that facilitate object-oriented and interactive applications. By taking advantage of these features, users can develop their own functions in a flexible way. The most important feature that makes SCL different from base SAS is class, which defines a set of attributes, methods, events, event handlers and interfaces that we can perform on data set.

Example 1 (continued): Here is an example of SCL code to compare two integers. We first create a program called **max.SCL** which integrates a method named **compare** to find the larger one of two integers, and then use the **SAVECLASS** command to generate the **max** class.

```
class max;
Compare: public method a:num b:num return=num;
dcl num value;
if a>=b then value = a;
else  value = b;
return value;
endmethod;
endclass;
```

Then we can create another program named **testdata.SCL** that contains an **INIT** section, which calls the **compare** method of the **max** class. After we compile and run **testdata.SCL**, it instantiates the **max** class and calls the method.

```
INIT:
dcl num n;
dcl max xobject = _new_ max();
n = xobject.Compare(7,3);
put n=;
```

The **PUT** statement produces n=7 in the log window.

Example 2(continued): We could also simply use the **CALL METHOD** routine which enables passing parameters to the method. For the example about calculating π from the infinite series, we can create a method named **PI** to perform this function, and save the method in a catalog entry of type SCL with the name **calcupi.SCL**. The parameter **nsim** is the length of the series. The method is then called in the **MAIN** section, where the **call method** specify the name of the SCL entry **calcupi.SCL**, the name of the method **PI**, and the parameter value of 5000 for **nsim**.

```
PI: method  nsim:num return=num;
     dcl num temp=0;
do i=1 to nsim;
temp=temp+4*(-1)**(i+1)*1/(2*(i-1)+1);
end;
put temp;
endmethod;

MAIN:
call method('calcupi.SCL','PI',5000);
return;
```

The method module performs its operations, and then returns modified values to the calling routine. The above program will print 3.14139265359179 in the log window.

When it's difficult or impossible to perform some tasks using only features available in SCL, we can use **SUBMIT** statements to execute both DATA steps and all the procedures in any product in SAS software, including macro. For

example, for the Monte Carlo simulation example we've seen before, we can include the macro between the **SUBMIT** and **ENDSUBMIT** statement and run it in SCL. However, it's worth pointing out that it is more efficient to use equivalent SCL features than to submit SAS statements. SCL is a powerful tool and developing customized functions is only one of its many functions. However, compared to macro or IML, the way to write SCL code might look more complicated for beginners. For users who are not familiar with object-oriented programming, developing functions with SCL may take more efforts.

PROC FCMP

Since SAS 9, the FCMP procedure has been available for users to create SAS functions and subroutines that can be used by other SAS procedures. In SAS 9.1.3, the user function defined by PROC FCMP can only be used in the following procedures: CALIS, COMPILE, DISTANCE, GA, GENMOD, MODEL, NLIN, NLMIXED, NLP, PHREG, RISKDIMENSIONS, ROBUSTREG, SIMILAR, SYLK. Since SAS 9.2, these user functions can be called from a DATA step like other SAS functions. Therefore, the functions and subroutines defined by PROC FCMP can be independent from the main program, which makes it different from SAS macro and IML. This feature also greatly enhanced the ability for SAS programs to be read and maintained.

Example 1(continued): Here is an example of using PROC FCMP to compare two integers:

```
proc fcmp outlib=sasuser.userfuncs.mymath;
function maxx(x,y);
if x>=y then return(x);
else return(y);
endsub;
options cmplib=sasuser.userfuncs;
data _null_;
a= 5;
b=7;
maxval= maxx(a,b);
put maxval=;
run;
```

The above code creates a function named **maxx** in a function package **mymath,** which is saved in the library **Sasuser.userfuncs**. The function **maxx** has two numeric parameters: x and y, which refer to the two numbers to be compared. The program compares the two numbers and then returns the larger one. The end of the user function is indicated by the **endsub** statement. The **CMPLIB=** system option in the **OPTIONS** statement specifies the location of the function package to be searched, and the DATA step calls the function and will output maxval=7 in the log window.

Example 2 (continued): The same example to calculate the value of π from the infinite series.

```
proc fcmp outlib=sasuser.userfuncs.mymath;
FUNCTION pi(nsim);
temp=0;
do i=1 to nsim;
temp=temp+4*(-1)**(i+1)*1/(2*(i-1)+1);
end;
return(temp);
endsub;
run;

options cmplib=sasuser.userfuncs;
data _null_;
estpi= pi(5000);
put estpi=;
run;
```

The code will print 3.1413927 in the output window.

From these simple examples, we can see that the way to write functions in PROC FCMP is quite straightforward and the code is easy to read and understand. Unfortunately, when writing functions within PROC FCMP, users can neither take advantage of other existing SAS procedures nor using DATA steps, which greatly limit the flexibility and

usage of the user functions developed. Therefore, using PROC FCMP for relatively complicated tasks, like the one to perform Monte Carlo simulation, would be really challenging. However, PROC FCMP is relatively new, it can be expected that the procedure can find more usage in the future.

PROC PROTO

The PROTO procedure has the capability to call a C/C++ functions from within a DATA step. According to SAS knowledge base, such capability provides several benefits:

- To directly use the existing C/C++ functions without the need to rewrite the function in SAS code.
- To leverage C libraries with functions which are not available in SAS.
- Using C instead of SAS to write functions might be preferred by some users who are more familiar with C.
- C functions might be more efficient than SAS functions under some circumstances.

For simple C functions, we can include the C source code in PROC PROTO and compile it in SAS. For example 1 where two integers are compared, the SAS code is:

```
proc proto package = Sasuser.userfuncs.mymath ;
int Maxx(int x, int y);
externc Maxx;
int Maxx(int x, int y)
{
if(x>=y) return x;
else return y;
}
externcend;
run;
```

Here the **PACKAGE=** option stores the prototypes in a function package. Then we can use **EXTERNC** statement to specify the beginning of C source code, and **EXTERNCEND** specify the end of the C source code. The above method can compile the external C code in a limited way. For complete C code, we may also compile the C code and generate Dynamic Link Library (DLL) files, and then call the DLL file from PROC PROTO. For example, we can compile the following C code and generate a DLL file named **maxxC.dll** and then save it in the folder 'C:\temp'.

```
int Maxx(int x, int y)
{ if(x>=y) return x; else return y;}
```

Then we can use PROC PROTO to generate a package named **mymath** and save it in the library **Sasuser.userfuncs**, and then use the **LINK** statement to link SAS to the DLL that contains the C function **maxx**.

```
proc proto package = Sasuser.userfuncs.mymath
label = "package of math functions";
link "C:\temp\maxxC.dll";
int Maxx(int a , int b);
run;
```

The function set up by PROC PROTO can not be used directly by DATA step. Therefore, an intermediate step that uses PROC FCMP can be used to bridge the gap. Now using PROC FCMP, we can define a wrapper function **SAS_Maxx**. When the DATA step calls **SAS_Maxx**, the C function **Maxx** set up by PROC PROTO is then called. The **INLIB=** option is used to specify the SAS data set that contains the function package, while the **OUTLIB=** option specifies the function package in which to store the wrapper function.

```
proc fcmp inlib=sasuser.userfuncs
outlib=sasuser.userfuncs.mathfun1;
function SAS_Maxx(a,b);
x=Maxx(a, b);
return(x);
endsub;
quit;
```

Now we can call the wrapper function **SAS_Maxx** from a DATA step. The **CMPLIB=** system option in the **OPTIONS** statement indicates the path containing the functions to be searched.

```
options cmplib=sasuser.userfuncs;
```

```
data _null_;
a=sas_Maxx(12,7);
put a;
run;
quit;
```

The above code will print 12 in the log window.

COMPARISON OF COMPUTATION TIMES

To get a better understanding of the computational efficiency of different methods, we compare the real time and CPU time using the example of calculating π with infinite series. We investigate three different numbers of iterations (i.e., the length of the series): 10000, 100000 and 1000000. The performance of SCL is not included because it's not straightforward to monitor computational time of SCL. The comparison is performed in SAS 9.2, and the computer has Intel Core 2 Duo E6400 2.4GHz CPU and 3G RAM.

Table 1 Computational times of different methods (Unit: Seconds)

Iteration		Macro	PROC IML	PROC FCMP
10000	real time	0.03	0.03	0.24
	CPU time	0	0.01	0.05
100000	real time	0.04	0.17	0.26
	CPU time	0.01	0.15	0.05
1000000	real time	0.07	1.54	0.31
	CPU time	0.06	1.53	0.13

We can see that the SAS macro takes the least time regardless of the number of iterations. The computational time of PROC IML is also very short when the number of iteration is small (e.g., 10000). However, with the increase of iteration the time increases dramatically. For PROC FCMP, although it takes longer time to finish 10000 iterations, it is more robust to the increase of iterations and its computational times are less than PROC IML for 1000000 iterations. Therefore, PROC IML may not be as efficient as SAS macro and PROC FCMP where a DATA step is involved. However, as we've seen in the Monte Carlo simulation example, if we can take advantage of the ability of SAS/IML to manipulate matrix, SAS/IML may have better performance.

CONCLUSION

This paper provides a review and summary of the five methods to develop user-defined functions in SAS. We can see that each method can handle simple tasks very well. However, for complicated task, macro might still be the best way available given its ability to integrate DATA step and other existing procedures. It's worth pointing out that macro can be used in combination with all the other methods discussed above, which makes it really attractive. Although it can only be used within an IML procedure, SAS/IML modules can be an efficient method if the function involves matrix operation. On the other hand, SCL is a power tool to develop functions, but programming in SCL may not be straightforward for SAS end-users. Even though the FCMP and PROTO procedures provide easy and straightforward way for users to write functions, they lack the ability to interact with DATA step and other SAS procedures, which may limit their applications in practice.

REFERENCES

SAS Institute Inc. 2004. *SAS® 9.1 Macro Language: Reference*. Cary, NC: SAS Institute Inc.

SAS Institute Inc. 2004. *SAS/IML 9.1 User's Guide*. Cary, NC: SAS Institute Inc.

SAS Institute Inc. 2004. *SAS® Component Language 9.1: Reference*. Cary, NC: SAS Institute Inc.

SAS Institute Inc. 2003. *The FCMP Procedure*. Cary, NC: SAS Institute Inc. Available at http://support.sas.com/documentation/onlinedoc/base/91/fcmp.pdf

Secosky, Jason. *User Written DATA Step Functions*. Proceedings of the SAS Global Forum 2007 Conference

Peter Eberhardt, *Functioning at an Advanced Level: PROC FCMP and PROC PROTO*. SAS Global Forum 2010, Paper 024-2010

CONTACT INFORMATION

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

Songfeng Wang
Department of Epidemiology and Biostatistics
800 Sumter St Room 205
Columbia, SC 29208
Work Phone: (803)777-3813
E-mail: songfeng@gmail.com