Paper 505-2013

Using SAS® PROC FCMP in SAS® System Development - Real Examples

Yves Deguire, Statistics Canada, Ottawa, Ontario, Canada
Xiyun (Cheryl) Wang, Statistics Canada, Ottawa, Ontario, Canada

ABSTRACT

At Statistics Canada, many complex applications have been built over the years using SAS®. These applications typically follow a modular design and leverage the SAS Macros facility. In some cases, the SAS/TOOLKIT® product has been used along with the C programming language to create user-written functions and procedures. With SAS®9.2 and beyond, the FCMP procedure gives the SAS programmer the ability to create user-written functions and CALL routines using DATA step syntax. Many projects within Statistics Canada have started adopting the FCMP procedure to encapsulate business logic and ease the system development process. In this paper, examples of user-written functions and CALL routines are presented to illustrate key concepts as well as several use cases.

INTRODUCTION

When developing SAS applications of any importance, a modular approach is essential to manage complexity. SAS developers normally implement modules as SAS® macros but some keen programmers have ventured with the SAS/TOOLKIT to develop routines using a 3rd generation programming language such as C. Starting with SAS 9.2, SAS programmers can now develop, using DATA step syntax, their own routines and in effect add them to the existing SAS built-in routines.

User-written routines are implemented via the PROC FCMP facility and their use should yield the following benefits:

- Reusability: A library of user-written routines can be built using DATA step syntax and then reuse in many other SAS programs within an application and even across multiple applications within an organization.
- Interoperability: User-written routines are independent from their use which means that any SAS program calling these routines is not affected by their implementation.
- Abstraction and encapsulation: Complex programs can be simplified by abstracting/wrapping common computations into routines. Programs that call these routines do not need to know anything about their implementation but just worry about how to interface with them.
- Easy maintainability: Developers can more easily read, write, and maintain complex code with independent and reusable routines.

The SAS macro facility and the SAS/TOOLKIT are still relevant because user-written routines are not a replacement for them but a powerful alternative. They are another tool in the developer's toolbox and should be well understood by any professional SAS developer.

KEY CONCEPTS

User-written routines are defined using PROC FCMP and can be either a function or a CALL routine:

- Function: is a subroutine that explicitly returns a value and accepts zero or more arguments. The arguments are always passed by value. This means that the function will receive a copy of the data passed as an argument and therefore any modifications made to that data by the function will not be reflected in the caller's copy.
- CALL routine: is a subroutine that does not return explicitly a value and normally receives one or more arguments. Arguments can be passed by value (same as with functions) but also by reference. Passing an argument by reference means that the subroutine receives the address of a data value which implies that one copy of the data is shared. CALL routines must have this capability to return information to the calling program. Indeed, any change made by the subroutine to the shared copy will be reflected in the environment of the caller.

Variables declare inside of a routine are local and not exposed to the calling program. This a desirable feature as it enforces encapsulation and reduces coupling. In other words, the routines tend to be self-contained with a very well-defined interface. Variables can be either character or numeric as is the case with DATA step variables. As well, arrays are available and, contrary to the arrays in a DATA step, they are dynamic. This is a very useful property when receiving data of unknown size.

A user-written routine can invoke most of the built-in SAS functions and CALL routines as well as a few special routines not available in a DATA step. Notably, there are routines to read and write arrays to a data set, to manipulate

matrices stored in multidimensional arrays, and to call BASE SAS code from within a routine. A user-written routine can also call another user-written routine which leads to modular design.

User-written routines must first be compiled and stored into packages before they can be used. Packages are special "containers" for compiled routines and they are stored inside a special data set. PROC FCMP is the BASE SAS procedure that handles this process and is only required at design time (i.e. when the routine is being defined). Most of the PROC FCMP statements are very similar to the DATA step statements and allows you to implement algorithms with various constructs (variables, constants, control flows, math operations, etc.). Exceptions do exist and they are formally documented in the SAS documentation.

Running a compiled routine is simple. First, you have to point to the PROC FCMP dataset with the package that contains the routine of interest. You do this with the CMPLIB option. Multiple PROC FCMP datasets can be listed creating a search path. Invoking a routine is similar to invoking a SAS built-in routine except that the path created by CMPLIB will be used to find the routine to be executed.

In some instances, a user-written routine is a more appropriate than a macro especially in contexts where a true function is desirable if not required (for example: in a WHERE statement or with formats). User-written routines provide better encapsulation and reduces coupling much more so than it is possible with macros. Macros and user-written routines can also be used together in the development of a modular application:

- Calling a user-written routine from macros: as any built-in SAS routines, user-written routines can easily be called from the SAS code generated by a macro.
- Calling a macro from a user-written routine: surprisingly, it is possible to do so by using the special RUN_MACRO function. A complete example of RUN_MACRO is provided in a later section.

Calling a routine introduces an overhead that does not exist in macros. This overhead pertains to the management of the runtime stack which is used to push data values to the called routines. If large amount of data is passed and many calls to routines are triggered then the management of the stack becomes very important and could substantially slow down processing. Passing data values by reference is one technique to reduce the amount information moving in memory.

DEFINING USER-WRITTEN ROUTINES

The following examples illustrate the key concepts that were introduced. They revolve around the conversion of distances expressed in miles into their equivalent in kilometers. The conversion is performed by simply multiplying the distance in miles by 1.609344. All the routines are built around this basic conversion rule and are contained in a single package.

Invoking PROC FCMP and Defining the Conversion Function

```
proc fcmp outlib= work.SGF13Funcs.ConvertMIToKM ;

function convertMIToKM(inUnits);
    return(inUnits * 1.609344);
endsub;
```

This code invokes PROC FCMP and sets the output library using the OUTLIB option so that the routines are created in the package called ConvertMIToKM stored in a PROC FCMP dataset called *SGF13Funcs* in the WORK library (it is in fact a three level name). Our first routine is a function called convertMIToKM. The FUNCTION keyword starts the definition of the function which is terminated by the ENDSUB keyword. The function accepts a numerical argument called inUnits which represents a distance in miles to be converted in kilometers. The RETURN statement sets the return value which in this case is the result of the conversion itself. It is also possible to return the value of a local variable. As is the case with any good design, this is the only place where the calculation is done so any change to the calculation will require only one modification to this function.

Defining a CALL Routine and Calling a Function

```
proc fcmp;
    subroutine convertMIToKM_sub(inUnits, outUnits);
           outargs outUnits;
           outUnits = convertMIToKM(inUnits);
           return;
    endsub;
quit;
```

This is the definition of a CALL routine that invokes our basic conversion function. The keyword SUBROUTINE starts the definition of the CALL routine which ends with the ENDSUB keyword. The routine accepts two arguments: inUnits which is the distance in miles to be converted and outUnits where the result is stored. The first argument is passed by value as is the case with the function but outUnits is passed by reference as indicated by the OUTARGS keyword. This allows the routine to return the result in outUnits. The function defined in our first example is used to set the value in outUnits. The RETURN statement ends the execution of the routine but does not pass explicitly a value to the caller since this is a CALL routine.

Defining a CALL Routine with arrays

```
subroutine convertMIToKM_arrays(inUnits[*], outUnits[*]);
    outargs outUnits;

    do i = 1 to dim(inUnits);
           outUnits[i] = convertMIToKM(inUnits[i]);
    end;

    return;
endsub;
```

This is a variation of the CALL routine defined previously. This version accepts as input an array of numbers representing distances in miles. The routine stores the corresponding distances in the array specified in the second parameter for which the argument is passed by reference. A loop is used to process each element of the array with the previously defined function doing the calculation. The special function DIM returns the size of the array. Anyone familiar with DATA step programming should be at ease with the way the loop is coded.

Defining a Function with a Variable Argument List

```
function SumDistancesinKM (inUnits[*]) varargs;
    total=0;

    do i = 1 to dim(inUnits);
           total = total + convertMIToKM(inUnits[i]);
    end;

    return(total);
endsub;
```

This function accepts a variable argument list which means that the number of arguments passed to the function will only be known at run time. The option VARARGS of the FUNCTION statement enables this feature. The last formal parameter of the function must be an array and receives the individual arguments. In our example, this array receives the individual arguments representing distances to be converted from miles to kilometers. These are then summed up to produce a total distance in kilometers which is returned by the function.

Defining a CALL Routine That Implements Dynamic Arrays and Passes Results Back Via a SAS Data Set

```
subroutine convertMItoKM_ds( inDSName $,inColName $, outDSName $, outColName $ );
    array inUnits[1] / nosymbols;
    array outUnits[1] / nosymbols;

    rc=read_array(inDSName, inUnits, inColName);

    length=dim(inUnits);
    call dynamic_array(outUnits,length);

    /* Convert each value read in */
    do i = 1 to length;
           outUnits[i] = convertMIToKM(inUnits[i]);
    end;

    rc=write_array(outDSName,outUnits,outColName);
    return;
endsub;
```

This CALL routine converts distances stored in a SAS data set into an output dataset. The parameters specify the names of the input and output datasets as well as the corresponding column names to store the values. The routine defines two internal arrays called inUnits and outUnits. These arrays are defined as one element array with the NOSYMBOLS option which simply states that no individual variables are mapped to the array elements. We do this so that the arrays can grow dynamically. The special function READ_ARRAY is used to read data into the inUnits arrays from the input data set. The array grows automatically to accommodate the data that is being copied. The outUnits array receives the converted values and must be resized to match the size of inUnits. The special CALL routine DYNAMIC_ARRAY is used to achieved this. A loop is implemented to perform the conversion of the values read into the inUnits array.

Finally, the outUnits array is written to the output data set with the special routine WRITE_ARRAY. In this example, the information is passed back via a SAS dataset. This is different than our previous example where the information was passed back to the caller either explicitly with the RETURN statement or implicitly by reference.

CALLING USER-WRITTEN ROUTINES FROM SAS PROGRAMS

So far we have shown examples that illustrate the definition of routines within the confines of PROC FCMP. They pertain to design time activities. The purpose of defining these routines is to run them in one or many programs or in order words to be invoked at run time. In this section, we show a few use cases that leverage the routines that have been defined previously. They are not exhaustive: wherever a function or CALL routine can be used in the SAS system, chances are that a user-written routine can be used as well.

Calling From a DATA Step

```
options cmplib = work.SGF13Funcs;
data work.distancesWithKM;
    set work.distances;
    call convertMIToKM_sub(distanceInMI, distanceInKM);
run;

data _null_;
    CALL convertMItoKM_dataset( 'distances','inUnits',
                                'distancesWithKM','outUnits' );
run;
```

This example shows two DATA steps that call a user-written routine. The very first statement set the search path for the user-written routines with the option CMPLIB. The path contains a list of PROC FCMP datasets to be searched until a routine is matched by name.

The first DATA step simply converts the distances in miles as read from the distances data set. The converted values are stored in the dataset distancesWithKM into the column distanceInKM.

The second DATA step performs the exact same action but lets the CALL routine do all the IO operations. Somewhat simpler from the point of view of the caller but the routine is much more complex to implement.

Calling From a Macro

```
%macro Convert();
   %do i = 10 %to 50 %by 10;
      %let inUnits = %eval(&i);
      %put &inUnits. Miles = %sysfunc(convertMIToKM(%eval(&inUnits))) km;
   %end;
%mend;
%Convert();
```

A user-written function can be called directly in a macro with the %SYSFUNC facility. In this example, a macro loop calls the function convertMIToKM repetitively.

Calling from a WHERE Statement

```
title1 "--->subset of data where outUnits less than 40";
proc print data= work.Distances;
   where convertMIToKM(inUnits) < 40;
run;
```

A user-written function can be called in a WHERE statement. If the function implements complex logic this may not be possible to do with a macro.

Calling a Function from a Format

```
proc format;
    value convertMItoKM OTHER=[convertMItoKM()];
run;

data _null_;
    set work.Distances;
    putlog inUnits "Miles is DISPLAYED AS "  inUnits convertMItoKM.  " km";
run;
```

PROC FORMAT enables the creation of user-defined formats in a number of ways. A user-written function can be specified in the creation of such a format. The function will perform the mapping when the format is triggered at run time.

Calling a Routine with DATA Step Arrays

```
data _null_;
    array inUnits{5} _TEMPORARY_ (10 20 30 40 50);
    array outUnits{5}   _TEMPORARY_;

    call convertMIToKM_arrays(inUnits,outUnits);

    do i = 1 to 5;
          putlog inUnits[i] "Miles =" outUnits[i] " km";
    end;
run;
```

This example shows how to pass DATA step arrays to a routine. Contrary to PROC FCMP arrays, DATA step arrays are not dynamic so they have to be sized accordingly before calling a routine. Also, any DATA step array passed to a routine must be a temporary array otherwise the call will fail.

Calling Directly Inside PROC FCMP

```
proc fcmp;
    total=SumDistancesinKM(30,40,50,60);
    put total=;
run;
```

We have seen that calling a user-written routine from another user-written routine is possible. But it is also possible to call a user-written routine from within PROC FCMP in open code. PROC FCMP provides a limited run time environment to do that. This can be useful to test a routine prior to its deployment. A routine that implements a variable argument list cannot be called directly in a DATA step. By running such a routine in PROC FCMP, it is easy to test it without having to create any special wrapper.

ERROR HANDLING IMPLEMENTED WITH A MACRO AND A USER-WRITTEN FUNCTION

A full fledge SAS application requires robust message handling to inform of any warnings or errors while processing as well as any notes that may useful in the review of the processing that took place. These messages are produced on top of any SAS messages and are specific to the application. The example presented in this section implements a rudimentary message handler that takes advantage of the capability of a user-written function to call a DATA step.

The diagram below provides an overview of the message handler. The system messages are centrally recorded in a message repository called msglib.msg_meta. At run time, when a message (a warning, a note, or an error) needs to be issued, a generic user-written function is invoked to retrieve the proper message from the messages repository, print it out to the SAS log and save an entry into the live system message table which accumulates all the messages generated in a given run.

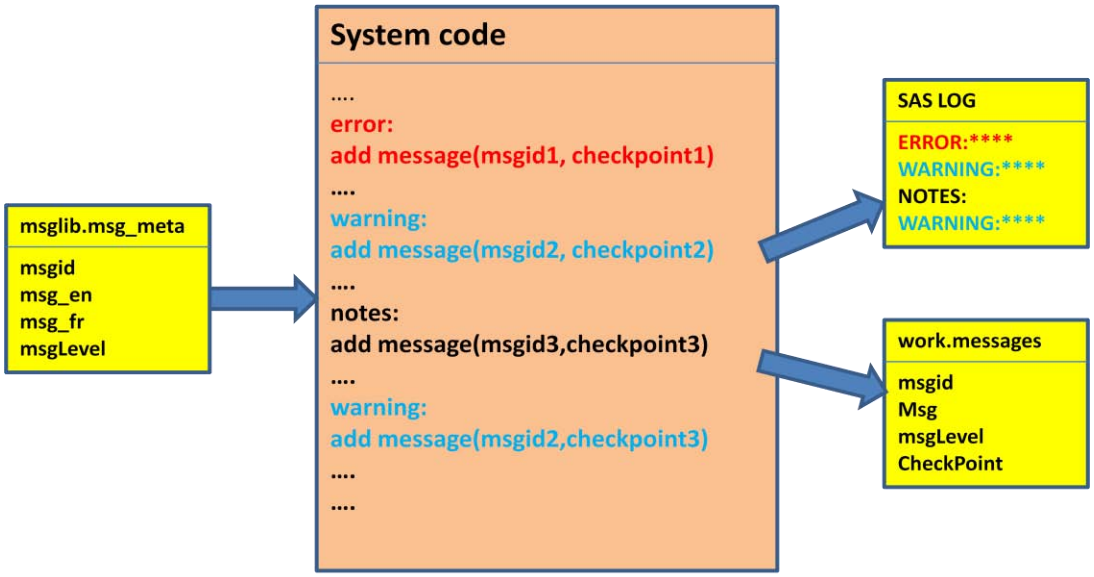

Figure 1 Generic Message Handler

The message handler itself is composed of two components: a user-written function and a macro.

Messaging Function

```
function insertMessage(msg_id $,checkpoint $);
    rc=run_macro('addmsg',msg_id,checkpoint);
    return(rc);
endsub;
```

The messaging function does not do any work; it is merely an interface to the messaging macro. Since the messaging macro requires a DATA step to retrieve a message and store it into a run runtime message table, it would be impossible to call it directly from another DATA step. This is not convenient. Most SAS applications need to issue messages from a DATA step. PROC FCMP provides the special function RUN_MACRO that enables a user-written routine to run a macro. There is no restriction as to what the macro can generate in terms of SAS code: it can include a number of DATA steps and PROC steps. Since a user-written routine is callable from a DATA step, then RUN_MACRO makes it possible to run a DATA step from another DATA step!

The messaging function is called insertMessage and accepts two arguments: a string that identifies the message to be issued (msg_id) and a label that corresponds to a checkpoint. It calls the special RUN_MACRO function in order to run the macro addmsg to which it passes the msg_id and checkpoint variables received as arguments. Before the macros runs, these variables are converted into local macro variables. This conversion is automatic and is the main communication channel between a user-written routine and a macro. Any change made by the invoked macro to the macro variables listed in the RUN_MACRO call is reflected back into the variables of the calling routine. This feature is not used in our message handler but is very useful to return information and new data values to the caller.

Finally, RUN_MACRO returns a code that indicates whether or not the macro was called. It is passed back to the calling program.

addMessage Macro

```
%macro addmsg();
    data work.newMsg(keep=msg_id msgTime msg checkPoint msglevel);
           attrib msgTime format=datetime21.;
           length checkpoint $150;
           set work.msg_meta;
                  where msg_id = &msg_id;
           msg = resolve(msg_&lang);
           msgTime = today();
           checkpoint = &checkpoint;
           msgLevel1=left(trim(msgLevel))||":";
           put msgLevel1 " " msgTime datetime21. "  "  msg  "at " &checkpoint;
    run;

    %if %eval(%sysfunc(exist(work.messages,data))=0) %then
           %do;
                  data work.messages;
                         set work.newMsg;
                  run;
           %end;
           %else
           %do;
                  proc append base=work.messages data=work.newMsg FORCE;
                  run;
           %end;
%mend addmsg;
```

The addMessage macro does all the work. It uses two steps: one DATA step to retrieve and output a message based on msg_id and another step (a DATA step or a PROC APPEND step) to add the retrieved message into the runtime system table. Although it is invoked by a user-written function, there is nothing special about this macro except for the fact that it cannot be defined with formal parameters. If so, these will not be set by the RUN_MACRO facility even if they match the names specified in the call to RUN_MACRO.

Global macro variables are available to a macro invoked by RUN_MACRO and any change made to these variables is reflected in the SAS global environment. In our example, the global macro variable &lang is used to retrieve the message text in the selected language.

Finally, another method of communication with the RUN_MACRO facility is with SAS data sets. User-written routines can easily store and access data in SAS data sets with the special arrays functions covered earlier in this paper.

USING PROC FCMP TO BUILD CUSTOMIZED SAS FUNCTIONS FOR PROC OPTMODEL

At Statistics Canada, a Generalized Sampling System called G-Sam is being implemented and will be used to draw samples for many business surveys. One of the modules in G-Sam is called Sample Allocation and computes the number of units to sample in each of the stratum identified for a given survey. The module takes into account the sampling costs and the variance. There are different algorithms for sample allocation. If the total sampling cost is fixed, then the allocation module produces minimum variance for the estimated population total of some target variable. Alternatively, if the user needs to obtain a specified CV for the estimate of one or more variables, there is an algorithm that determines the allocation while minimizing the overall cost. In summary, sample allocation is an optimization problem.

The G-Sam allocation module takes advantage of PROC OPTMODEL, a SAS/OR procedure that includes a powerful modeling language and a series of solvers to handle mathematical problems. The system specifications for the allocation module as documented by survey methodologists provide details of the optimization models to be created along with a number of formulas that has to be implemented inside PROC OPTMODEL. Implementing those formulas in SAS macros is cumbersome and makes the PROC OPTMOPDEL code difficult to read. User-written functions are callable in PROC OPTMODEL programs and are a natural fit for the formulas such as these described in the following figure.

Method	Functions used by PROC OPTMODEL (h is stratum ID)
method=1	$F_h(x) = (\rho_h x + 1.4 - 0.4\rho_h)/(\rho_h^2 x(x+1))$
method =2	$F_h(x) = ((N_h - 0.4)\rho_h x + 1.4)/(x^2 \rho_h^2 N_h(N_h+1))$
function Fh(x, Rho_h, big_Nh, method$); *programming-statements;* **endsub;**	
method =2	$S_h(k) = InverseCDF(N_h, k, q_h)$ Where $q_h = CDF(\text{"BINOMIAL"}, k, S_h(k), N_h)$
function Sh(k, big_Nh, quantile_h_r); *programming-statements;* **endsub;**	

Figure 2 Natural Mapping of Formulas to User-Written Functions

There are dozens of these formulas in the sample allocation specifications and each is quite complex. Their encapsulation into user-written functions makes the PROC MODEL code quite readable and easy to maintain. During the construction phase of the G-Sam software, creating separate functions for each formula allowed the project team to divide up the work into smaller tasks and work in parallel.

Fh Function

```
function Fh(x,Rho_h,big_Nh,method);
    temp = .;

    if (x <> 0) then
           do;
           if (method = 1) then
                  temp = ((big_Nh-
                         0.4)*Rho_h*x+1.4)/(x**2*Rho_h**2*big_Nh*(big_Nh+1));
           else if(method = 2) then
                  temp = (Rho_h*x+1.4-0.4*Rho_h)/(Rho_h**2*x*(x+1));
           end;

    return(temp);
endsub;
```

This function is a straight implementation of the formula provided in the specifications above. It can be used directly in PROC OPTMODEL. Although this function implements more complex calculations than our previous examples, it does not use any new features that would have not been covered so far.

OPTMODEL Program with User-Written Functions

```
proc optmodel;
    /*declare and initialize array or constants*/
    set <string> stratum =/"strat1" "strat2" "strat3" "strat4"/;
    num rh{stratum}=[0.5 0.5 0.6 0.6];
    num csth{stratum} init 10;
    num ch{stratum} init 100;
    num big_Nh{stratum} init 100;
    num Rho_h{stratum} init 1;
    num p=2;
    num quantile_h_r{h in Stratum} = quantile('NORMAL',rh[h]);
    num Fh_Sh_csth_ch{stratum};

    /*using PROC FCMP functions to do calculation*/
    for {h in stratum}
          Fh_Sh_csth_ch[h]
          = Fh(Sh(round(csth[h]/ch[h]),big_Nh[h],quantile_h_r[h])
          ,Rho_h[h],big_Nh[h],2);

    /*build proc optmodel constraints*/
    var decisionVars{stratum};
    con consLB{h in stratum}:
          decisionVars[h] >=
          (big_Nh[h]*Fh_Sh_csth_ch[h] -1 )**p;
    expand;
quit;
```

This is a PROC OPTMODEL program that invokes the Fh and Sh user-written functions. It is interesting to note that function Sh is used to calculate the first argument passed to function Fh directly in the function call itself. There is a caveat to using user-written functions in PROC OPTMODEL. Because of issues with numerical precision, they should not be used in objective functions.

CONCLUSION

In this paper, we have presented key concepts that should be mastered in order to take advantage of the PROC FCMP user-written routines. Several examples and use cases have also been studied to illustrate these concepts.

There are other features that have not been addressed in this paper. Notably, it is possible to invoke, from user-written routines, external functions written in C or C++ as well as interact with complex C structures. To do so, PROC PROTO must first be used to register the external functions and structures.

User-written routines are an important addition to the SAS developer's toolbox and when use properly they yield modular applications that are easy to maintain. SAS macros are still an important facility that can be used hand-in-hand with user-written routines. It is possible to integrate them in a single application.

REFERENCES

Eberhardt, Peter. "Functioning at an Advanced Level: PROC FCMP and PROC PROTO." *Proceedings of SAS Global Forum 2010 Conference.* Available at http://support.sas.com/resources/papers/proceedings10/024-2010.pdf.

SAS Institute Inc. 2012. Base SAS® 9.3 Procedures Guide, Second Edition. Cary, NC: SAS Institute Inc.

Secosky, Jason. "User-Written DATA Step Functions." *Proceedings of SAS Global Forum 2007 Conference.* Available at http://www2.sas.com/proceedings/forum2007/008-2007.pdf.

Secosky, Jason. "Executing a PROC from a DATA Step." *Proceedings of SAS Global Forum 2012 Conference.* Available at http://support.sas.com/resources/papers/proceedings12/227-2012.pdf.

Wang, Songfeng; Zhang . Jiajia. " Developing User-Defined Functions in SAS®: A Summary and Comparison." *Proceedings of SAS Global Forum 2011 Conference.* Available at http://support.sas.com/resources/papers/proceedings11/083-2011.pdf.

ACKNOWLEDGMENTS

The authors would like to thank Pierre Lafrance for its continued support as well as the members of the SAS Technology Centre at Statistics Canada for their numerous suggestions.

CONTACT INFORMATION

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

Yves DeGuire
Statistics Canada
100 Tunney's Pasture Driveway
Ottawa, Ontario, K1A 0T6
Canada
Phone : 613-951-1282
E-mail: Yves.Deguire@statcan.gc.ca
Web: http://www.statcan.gc.ca

Xiyun Wang
Statistics Canada
100 Tunney's Pasture Driveway
Ottawa, Ontario, K1A 0T6
Canada
Phone 613-951-0843
E-mail: CherylXiyun.Wang@statcan.gc.ca
Web: http://www.statcan.gc.ca