Paper 326-2010

Adding Statistical Functionality to the DATA Step with PROC FCMP

Stacey M. Christian and Jacques Rioux, SAS Institute Inc., Cary, NC

ABSTRACT

How many times have you had to write a cumbersome workaround to do an analytical task inside a DATA step that would be easy if you could just call SAS® from the DATA step? With the advent of PROC FCMP in SAS 9.2, you can do exactly that. This paper demonstrates how to add statistical functionality to the DATA step through the definition of FCMP functions. We will provide specific examples of how to encapsulate SAS® Analytics within FCMP functions and thus make them callable from the DATA step.

INTRODUCTION

In SAS 9.2, the SAS Function Compiler (FCMP) procedure was introduced to provide a way to code reusable, independent programming units written in DATA step syntax. PROC FCMP provides the ability for users to write their own functions and CALL routines (subroutines), store them to disk, and recall them later from many different contexts within SAS. This enables programmers to more easily read, write, maintain, and reuse common code. Furthermore, these functions and subroutines allow more complex programming techniques such as recursion, variable scoping, and encapsulation.

However, in order to truly use the power of SAS Analytics, one needs the ability to call SAS itself from these functions as well. This paper will demonstrate that SAS can be used as a true functional programming language by encapsulating any analytical procedure (as well as the DATA step itself) within FCMP functions and subroutines.

More specifically this paper will develop solutions to two relevant statistical problems by encapsulating numerous analytical procedures as well as the DATA step inside FCMP functions using the specialized RUN_MACRO function provided by the FCMP procedure. The first example uses a recursive technique to segment time series data into piecewise linear approximations. The second example demonstrates an iterative algorithm to fit a proportional hazard survival model using a minimum distance method.

For enhanced readability, most of the code samples in this paper are incomplete. Complete code samples are available in the addendum to this paper (326-2010_samples).

ENCAPSULATING SAS PROCEDURES AND DATA STEPS IN FCMP FUNCTIONS

To enable calling any SAS program from within an FCMP subroutine or function, two special functions are provided by PROC FCMP: RUN_MACRO and RUN_SASFILE**.** The RUN_MACRO function executes a predefined SAS macro. The RUN_SASFILE function executes a predefined SAS program from a specified file. Both of these functions are quite powerful because they allow virtually any SAS program to be called from within the DATA step and any procedure that can use FCMP. Both pass the current value of local function variables to and from the executed macro or program through the use of macro variables.

These two functions work pretty much in the exact same way. For simplicity, in this paper we will demonstrate only the RUN_MACRO function. The syntax of the RUN_MACRO function is as follows:

rc = **run_macro***('macro_name', variable_1, variable_2, …*);

where

rc is 0 if the function was able to submit the macro. The return code indicates only that the macro call was executed.

macro_name specifies the name of the SAS macro to be executed. The macro itself should not have explicit arguments.

variable_n specifies optional PROC FCMP variables whose current values are passed to and from macro variables of the same name. Before PROC FCMP executes the macro, local SAS macro variables are defined with the same name and value as the FCMP variables. After PROC FCMP executes the macro, the macro variables' values are copied back to the corresponding FCMP variables.

This example creates a macro called TEST_MACRO, and then uses the macro within a PROC FCMP function to subtract two numbers:

```
/* Create a macro called testmacro */
%macro test_macro;
   %let difference = %sysevalf(&a - &b);
%mend test_macro;

/* Use testmacro within a function to subtract two numbers */
proc fcmp outlib = sasuser.ds.functions;
   function subtract_macro(a,b);
      rc = run_macro('test_macro', a, b, difference);
      if rc eq 0 then return(difference);
      else return(.);
   endsub;

   /* test the call */
   a = 5.3;
   b = 0.7;
   diff = subtract_macro(a, b);
   put diff=;
run;
```

Notice that we can execute and test the function directly from PROC FCMP. The following example demonstrates calling the function from within the DATA step:

```
options cmplib = (sasuser.ds);

data _null_;
   a = 5.3;
   b = 0.7;
   diff = subtract_macro(a, b);
   put diff=;
run;
```

The *cmplib* global option is used to allow the DATA step to find the FCMP function. Output from executing this program:

```
diff=4.6
```

This, of course, is a very simplistic example of how to call a SAS macro from an FCMP function. The important point is that through this mechanism, any SAS procedure or DATA step can be called from an FCMP function, and all the power of the SAS macro language can also be used.

Regarding the passing of FCMP variables to and from SAS macro variables, character variables are passed in quotation marks. It is often necessary to use the DEQUOTE function to remove quotation marks from these macro variables for proper usage inside the SAS macro language:

```
%let ds_name = %sysfunc(dequote(&ds_name));
```

RECURSIVE TECHNIQUE

Recursion, in mathematics and computer science, is a problem solving technique in which a function calls itself in a seemingly infinite loop until some stopping criterion is met. In the paper, "Segmenting Time Series: A Survey and Novel Approach," numerous recursive algorithms are presented in the domain of segmenting time series data. The main idea is that in order to reduce extremely large time series data sets, piecewise linear approximations of the data can be used instead. However, exactly how does one partition (or segment) the data? The simplest approach presented is the top-down algorithm in which the time series is recursively partitioned, from the top down, until some stopping criterion is met. In its simplest form, the algorithm goes like this:

```
Segment_TopDown ( currentSegment )
 {
     error = run_linear_approximation( currentSegment );
     leftError = run_linear_approximation ( leftSegment );
     rightError = run_linear_approximation ( rightSegment );
     combinedError = leftError + rightError;
     if ( combinedError < error ) then
        call Segment_TopDown ( leftSegment ) ;
        call Segment_TopDown ( RightSegment );
    else
        keep_segment(currentSegment);
}
```

Before PROC FCMP was introduced, this type of algorithm was extremely difficult to implement in SAS. However, PROC FCMP fully supports recursion, so a single function can be written that calls itself to partition the data. We translated this recursive algorithm into a fairly straight-forward FCMP subroutine called SEGMENT_TOPDOWN:

```
subroutine segment_topdown(data $, segdata $, var $,
                           start_in, end_in, error_in, threshold);

  /*- initial setup -*/
  if (start_in eq .) then start = 1; else start = start_in;
  if (end_in eq .) then end = get_nobs(data); and end = end_in;

  if (error eq .) error = linear_approximation(data,var,start,end);
  else error = error_in;

  mid = start + floor((end-start)/2);
  left_error = linear_approximation (data, var, start, mid);
  right_error = linear_approximation (data, var, mid+1, end);

  combined_error = left_error + right_error;
  improvement = (error - combined_error) / error;

  if (improvement > threshold) then do;
    call segment_topdown(data, segdata, var, start, mid, left_error, threshold);
    call segment_topdown(data, segdata, var, mid+1, end, right_error, threshold);
  end;
  else do;
    call append_segment(segdata, start, end, error);
  end;

endsub;
```

The SEGMENT_TOPDOWN subroutine is pretty easy to follow because it breaks the problem down into small independent sub-functions: LINEAR_APPROXIMATION and APPEND_SEGMENT. The main subroutine, SEGMENT_TOPDOWN**,** calls itself recursively until the combined error of the partitions at any point ceases to reduce the prior approximation error. All segments are written to a data set.

This leaves us with two very important issues: how to provide the two subroutines LINEAR_APPROXIMATION and APPEND_SEGMENT. Both of these routines require the use of a SAS procedure or a DATA step.

For the linear approximation and error calculation routine, we first define a macro called LINEAR_APPROXIMATION_MACRO, which uses the SAS REG procedure to perform a linear approximation of each segment:

```
%macro linear_approximation_macro;
   %let ds_in     = %sysfunc(dequote(&ds_in));
   %let var       = %sysfunc(dequote(&var));

   data _TEMP_;
      set &ds_in(firstobs=&first_obs obs=&last_obs);
      retain _TREND_ 0;
      _TREND_ = _TREND_ + 1;
   run;

   proc reg data=_TEMP_ outest=_EST_ noprint;
      model &var = _TREND_ / sse;
   run; quit;

   proc sql noprint; select _SSE_ into :ERROR from _est_; quit;

%mend linear_approximation_macro;
```

Then, we wrap this macro in the FCMP function called LINEAR_APPROXIMATION:

```
proc fcmp outlib=sasuser.examples.topdown;
    function linear_approximation(ds_in $, var $, first_obs, last_obs);
       error = .;
       ok = run_macro('linear_approximation_macro', ds_in, first_obs,
                      last_obs, var, error);
       return(error);
    endsub;
run;
```

Thus, whenever the LINEAR_APPROXIMATION function is called, the REG Procedure and the DATA step are executed. Notice that the current values of **ds_in, var, first_obs, last_obs**, and so on, are passed from the FCMP function to macro variables available in the LINEAR_APPROXIMATION_MACRO macro.

Likewise, in the creation of the segment data set subroutine, APPEND_SEGMENT, we encapsulate PROC APPEND and the DATA step as follows:

```
%macro append_segment_macro;
   %let seg_ds = %sysfunc(dequote(&seg_ds));
   data _temp_;
      start = &first_obs;
      end   = &last_obs;
      error = &error;
   run;
   proc append base=&seg_ds data=_temp_ force;
   run;
%mend append_segment_macro;

proc fcmp outlib=sasuser.examples.topdown;
    subroutine append_segment(seg_ds $, start, end, error);
        first_obs = start;
        last_obs = end;
        ok = run_macro( 'append_segment_macro', seg_ds, first_obs,
                        last_obs, error );
    endsub;
run;
```

In the following code, the SEGMENT_TOPDOWN subroutine was called from the DATA step on a time series data set containing the daily closing prices for the S&P 500 from Jan 3, 1950 to Jan 31, 2010. This data set contained 15,116 observations:

```
data _NULL_;
  call segment_topdown("sasuser.snp", "work.segds_20",
                       "close", ., ., ., 0.2);
  call segment_topdown("sasuser.snp", "work.segds_10",
                       "close", ., ., ., 0.15);
run;
```

Thus, the S&P 500 data was recursively partitioned until each new partition failed to reduce the prior approximation error by at least 20 percent (or 15 percent in the second call). The algorithm eventually chose 42 segments (with 20 percent error reduction threshold). The second call resulted in 113 segments (with 15 percent error reduction threshold).

The PLOT_SEGMENTS subroutine (see addendum for details) joined the linear segments and then used PROC SGPLOT to plot the linear segments against the original data:

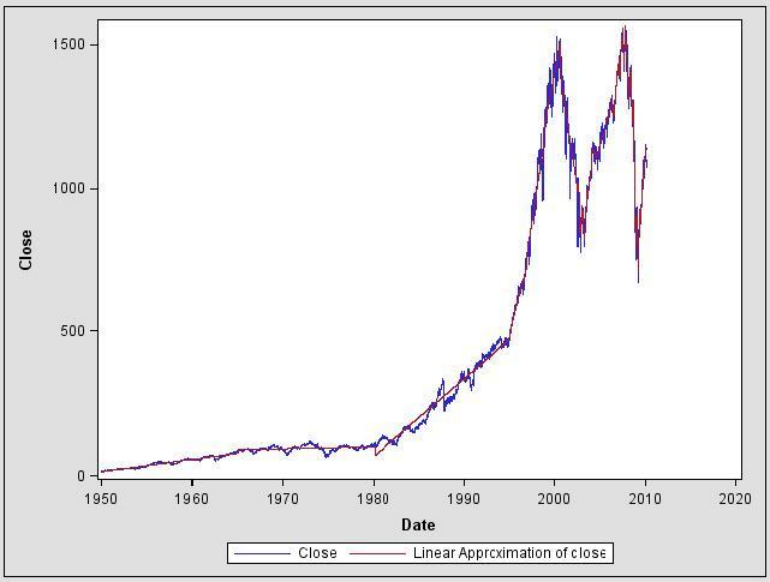

Figure 1 : S&P 500 - 42 Piecewise Linear Segments

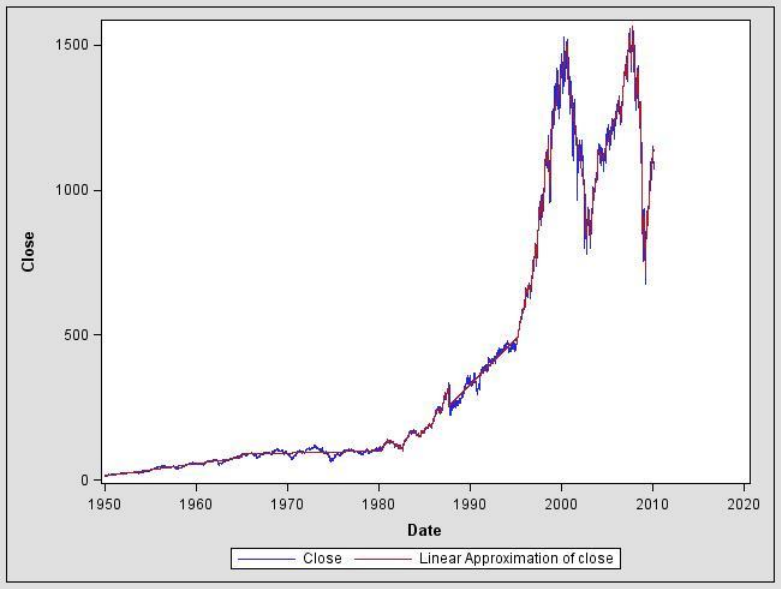

Figure 2: S&P 500 - 113 Piecewise Linear Segments

These plots demonstrate how accurately the time-series data can be approximated with significantly less data.

ITERATIVE TECHNIQUE

Iterative algorithms are numerous in statistics. One important class of such algorithms is the iteratively reweighted least squares method. This method is useful in cases where a modeling problem can be represented as a weighted linear regression problem where the weights themselves depend on the parameters. The solution is to start by running the regression routine with fixed weights. Based on the parameters obtained, the weights are recalculated. Then, the weighted regression problem is fit again with the new weights. At that point, some measure of distance between the two sets of fitted parameters is calculated. The fitting and reweighting steps can be repeated until the

distance between the two consecutive sets of parameters has reached a specified tolerance level. In its simplest form, the iteratively reweighted least squares algorithm can be represented as follows:

```
IterativelyReweightedLeastSquares
    initialize_weights( weights );
    params1 = run_regression( weights );
    maxRelativeDifference = 1.0;
    while (maxRelativeDifference > criteria)
    {
       update_weights(weights);
       params2 = run_regression( weights );
       maxRelativeDifference = calculate_difference(params1, params2);
       params1 = params2;
    }
```

To demonstrate an implementation of this algorithm in this paper, we choose a statistical model that is common and well understood but we apply a fitting method that may not be as common. A proportional hazard survival model is one for which the relationship between the base survival function $S_0(t)$ and the conditional survival function $S(t|\boldsymbol{x})$ is of the form $S(t|\boldsymbol{x}) = S_0(t)^{\exp(\mathbf{x}'\beta)}$. In SAS/STAT® software, PROC PHREG handles this kind of modeling using a likelihood based approach. Rioux and Luong (1996) consider the same model in the case where all the regressors are categorical and where the survival data is already grouped by intervals. They apply the log (-log()) transformation to the conditional probabilities of survival for each interval to obtain an expression of the form:

$$\log(-\log(P_j(\boldsymbol{x}))) = \boldsymbol{x}'\beta + \log(-\log(p_j)).$$

This leads to a minimum quadratic distance problem that can be solved by applying the iteratively reweighted least squares algorithm. For more information about the method and statistical details, see "Minimum Quadratic Distance Estimation for the Proportional Hazards Regression Model with Grouped Data" [Rioux, Luong].

The survival data on which this method is applied is such that each record contains a number of categorical variables, a time variable indicating the beginning of an interval, and a count variable representing the number of survivors at that time for the vector of covariates in that record. To simplify the code, we assume that counts of survivors are available for each vector of covariates and for each interval period. We disregard complications such as missing data.

The simulated sample data we use to demonstrate the algorithm is of the same form as the data used in the simulation study in [Rioux, Luong] . Specifically, it contains three regression variables with the following possible levels: $X_1 \in \{0,1\}$, $X_2 \in \{0,0.4,0.9\}$, $X_3 \in \{-1,1\}$ and covers twelve time intervals that are transformed into as many indicator variables. The macro that generates the data as described is called GENERATE_SURVIVAL_DATA_MACRO (see addendum).

Thus, the iteratively reweighted least squares algorithm can be applied to this model within a DATA step program as follows:

```
%let numParams    = 15;
%let data         = "work.mdphdata";
%let preparedData = "work.outdata";
%let paramData    = "work.phresults";
%let regressors   = "x1 x2 x3";
%let regCount     = 3;
%let depVar       = "log_log_Pij";
%let weightVar    = "Weight";
```

```
data _NULL_;

   /*- Initial data preparation -*/
   length intervalVars $ 128;
   intervalCount = 0; intervalVars = "";
   call prepare_phdata(&data, "Count", "Quarter", &regressors,
                       &preparedData, intervalCount, intervalVars);
   independentVars = &regressors || " " || intervalVars;

   array params1[&numParams] _temporary_;
   array params2[&numParams] _temporary_;

   /*- initial regression run with initial weights -*/
   call run_regression(&preparedData, &depVar, independentVars,
                       &weightVar, &paramData, params1);

   /*- Iterative loop -*/
   maxRelativeDifference = 1;
   do while( maxRelativeDifference > 0.0001 );
      call update_weights(&preparedData, &paramData, &regressors,
                          &weightVar, intervalCount);

      call run_regression( &preparedData, &depVar,independentVars,
                           &weightVar, &paramData, params2 );

      maxRelativeDifference = calc_max_relative_diff(params1,params2);
   end;

   call update_weights(&preparedData, &paramData, &regressors,
                       &weightVar, intervalCount);
run;
```

As in the previous example, this algorithm is quite elegant. However, we now need to provide various FCMP functions to complete this code. And these functions can be quite complicated in implementation.

The PREPARE_PHDATA subroutine must transform the data to bring it to the form needed to apply the iterated regression algorithm. This means that for each vector of covariates, and for each time interval, we need to compute the probability of survival from the beginning to the end of the interval (Pij). The log (-log()) transform must be applied to that probability, and the result will be the used as the independent variable in the regression problem generated. We also need to encode the time interval information by creating indicator variables for each of them. These indicators along with the original regression variables account for all independent variables in the model:

```
subroutine prepare_phdata(data $, count $, time $, regressors $,
                          outData $, intervalCount, intervalVars $);

      outargs intervalCount, intervalVars;
      intervalVars = "";
      intervalCount = .;
      rc = run_macro ( 'prepare_phdata_macro',
                       data, count, time, regressors,
                       outData, intervalCount, intervalVars);
endsub;
```

As in the previous example, this subroutine utilizes the RUN_MACRO function in order to call the macro PREPARE_PHDATA_MACRO. This macro is quite long, using the DATA step, PROC SQL, and PROC SORT to accomplish the data transformation. The full source code for this macro can be found in the addendum to this paper.

The next part is the first regression step. At this point, the data is in the form required to apply a regression routine to it. We write the RUN_REGRESSION subroutine to encapsulate calls to PROC REG and DATA step:

```
%macro run_regression_macro;
   %let data        = %sysfunc( dequote(&data)        );
   %let params      = %sysfunc( dequote(&params)      );
   %let dependent   = %sysfunc( dequote(&dependent)   );
   %let independent = %sysfunc( dequote(&independent) );
   %let weight      = %sysfunc( dequote(&weight)      );

   proc reg data=&data outest=&params NOPRINT;
      model  &dependent = &independent/noint;
      weight &weight;
   quit;

   data &params;
      set &params;
      keep &independent;
   run;
%mend run_regression_macro;

subroutine run_regression( data $, dependent $, independent $, weight $,
                           params $, paramArray[*]);
      outargs paramArray;

      array tmpArray[1] _temporary_;
      call dynamic_array(tmpArray,dim(paramArray));

      rc = RUN_MACRO ('run_regression_macro', data, params, dependent,
                      independent, weight) ;
      rc = read_array(params, tmpArray);
      do i = 1 to dim(paramArray);
         paramArray[i] = tmpArray[1,i];
      end;
 endsub;
```

Notice that the RUN_REGRESSION subroutine uses the READ_ARRAY function of PROC FCMP to load the parameters fit by PROC REG from the parameter data set into an array that is returned to the caller.
We do not reproduce all options that PROC REG provides. We only implement those that are needed to have a quick and efficient regression routine reusable for this particular example. In reality one could write a more generic wrapper for the regression function, encapsulating many more options, and then reuse it in many other applications.

The third step in the algorithm is the calculation of the weights according to the regression parameters. This calculation is done in the UPDATE_WEIGHTS subroutine. In this routine we do more than what is minimally required. We actually compute and update the estimates for the base survival function. That is why we call this routine one last time at the end of iteration loop so that the survival function estimates as well as the estimates of the regressors correspond to the parameters reported in the parameter dataset. This routine is again quite long. It encapsulates DATA step as well as PROC SQL. The details can be found in the addendum to this paper.

The final function CALC_MAX_RELATIVE_DIFF simply calculates the maximum relative difference between the two sets of parameters:

```
function calc_max_relative_diff(params1[*],params2[*]);
     outargs params1;
     maxRelativeDifference = 0;
     numParams = DIM(params1);
     do i = 1 to numParams;
        maxRelativeDifference = max( maxRelativeDifference,
                                     abs((params1[i]-params2[i])/params2[i]));
        params1[i] = params2[i];
     end;
     return(maxRelativeDifference);
endsub;
```

So now our algorithm is complete.

We generated the survival data and ran the DATA step algorithm to fit the model. The algorithm continually reweighted until the distance between the two consecutive sets of parameters reached the specified tolerance level (0.0001). It required five iterations to fit this model.

THE TRUE GLORY OF REUSEABLE FUNCTIONS

At this point, some readers may want to claim that this iterative technique could have been accomplished using only the macro language facility. While this might be partially true, we think that the level of encapsulation and reuse provided by PROC FCMP is far better than what could ever be accomplished with macros alone. To demonstrate this even further, let's extend the reach of our problem a bit.

First let's wrap the iterative algorithm that was initially written as a DATA step in another FCMP subroutine called FIT_PH_MODEL (fit proportional hazard model):

```
subroutine fit_ph_model(data $, paramDataOut $,
                        countVar $, timeVar $, depVar $, weightVar $,
                        regressors $);
  /*- initial data preparation -*/
  preparedData= "_prepdat_";
  length intervalVars $ 128;
  intervalCount = 0; intervalVars = "";
  call prepare_phdata( data, "Count", "Quarter", regressors, preparedData,
                       intervalCount, intervalVars );
  independentVars = regressors || " " || intervalVars;

  /*- create param arrays dynamically */
  numParams = intervalCount  + count(regressors, " ")+1;
  array params1[1] _temporary_;
  array params2[1] _temporary_;
  call dynamic_array(params1, numParams );
  call dynamic_array(params2, numParams );

  /*- initial regression run with initial weights -*/
  call run_regression( preparedDataOut, depVar,independentVars,
                       weightVar, paramData, params1 );

  /*- iterative loop -*/
  maxRelativeDifference = 1;
  do while( maxRelativeDifference > 0.0001 );
     call update_weights(preparedDataOut, paramData, regressors, weightVar,
                         intervalCount);

     call run_regression( preparedData, depVar, independentVars,
                          weightVar, paramDataOut, params2 );

     maxRelativeDifference = calc_max_relative_diff(params1,params2);
  end;

  call update_weights(preparedData, paramDataOut, regressors, weightVar,
                      intervalCount);
endsub;
```

This single subroutine both prepares the proportional hazard data and then runs the iteratively reweighted least squares algorithm. Now that we have coded a new fitting routine for the Proportional Hazard Model, we can reuse it to consider a simulation study that will ascertain how well the asymptotic results of the fit compare with what can be observed in practice.

To accomplish that, we write two helper subroutines. One is called SIMULATE_PHDATA which handles, as the name suggests, the simulation of a survival data for input to our fitting routine. It is simply a wrapper to the macro code we

originally used to generate the data in our previous example (GENERATE_SURVIVAL_DATA_MACRO). The second helper subroutine is called APPEND_DATA. It's a very thin wrapper to PROC APPEND:

```
proc fcmp outlib=sasuser.example2.sim ;
   subroutine simulate_phdata( dsname $);
      rc = run_macro('generate_survival_data_macro', dsname);
   endsub;

   subroutine append_data( base $, data $ );
      rc = run_macro('append_data_macro', base, data );
   endsub;
run;
```

With this in hand, it is straight forward to write a loop that simulates data, fits the Proportional Hazard Model, and accumulates all the fitted parameter results into a single data set:

```
%let NSIM=1000;
proc fcmp;
   do i=1 to &NSIM;
       call simulate_phdata ("work.simdata");
       call fit_ph_model ("work.simdata", "work.params",
                          "Count", "Quarter", "log_log_Pij", "Weight",
                          "x1 x2 x3" );
       call append_data("work.simresults", "work.params");
   end;
run;
```

Notice in this example that we actually run our final simulation algorithm in PROC FCMP rather than use the DATA step. This is just to demonstrate that PROC FCMP can be used directly to call functions as well as calling them from the DATA step. PROC FCMP supports the DATA= and OUT= options in the case that you want to run your algorithm on an input data set or write out your results to an output data set.

Thus, we simulated the survival data 1000 times, ran the iteratively reweighted least squares algorithm to fit the Proportional Hazard Model 1000 times, and combined the final results of each simulation into a single data set. All 1000 simulations were run with parameters x1 = 0.1, x2 = 0.3, x3 = 0.2. Below we show histograms of the estimated values for each of parameter. We also computed an estimate of p1, the probability of survival in the first interval, and we plot the distribution of that value as well. In the table below, we compare the real values of x1, x2, x3 and p1 with summary information from their estimators. All estimators look like they behave as they should.

Coefficient	Real Value	Mean	StDev
X1	0.1	0.102454	0.036917
X2	0.3	0.307029	0.050375
X3	0.2	0.205464	0.017793
P1	0.97530991	0.975794	2.48196 E-6

Table 1 – Summary of Simulation Results

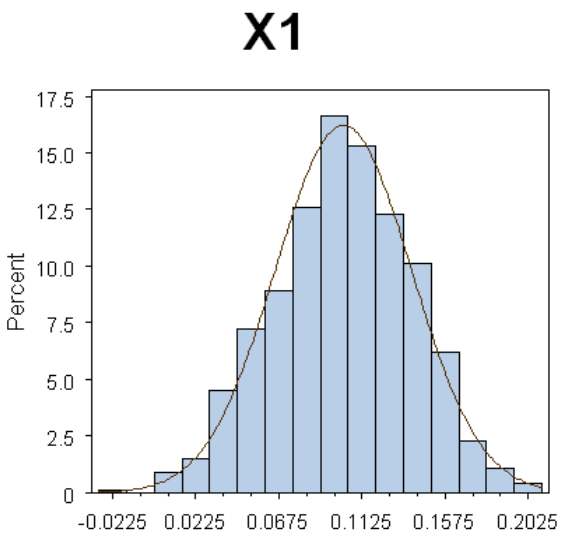

Figure 3: Histogram for Fits of X1

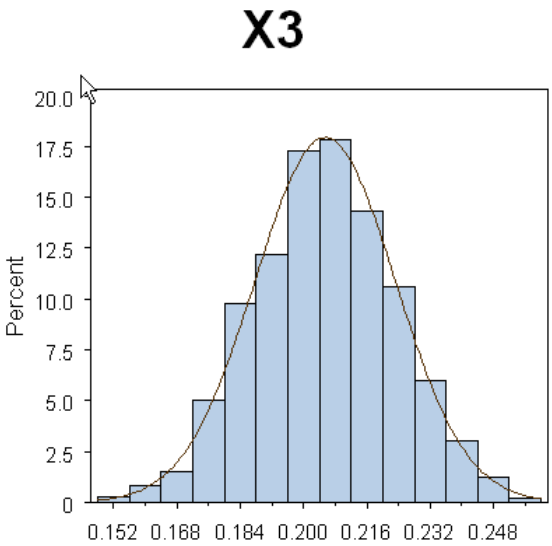

Figure 5: Histogram for Fits of X3

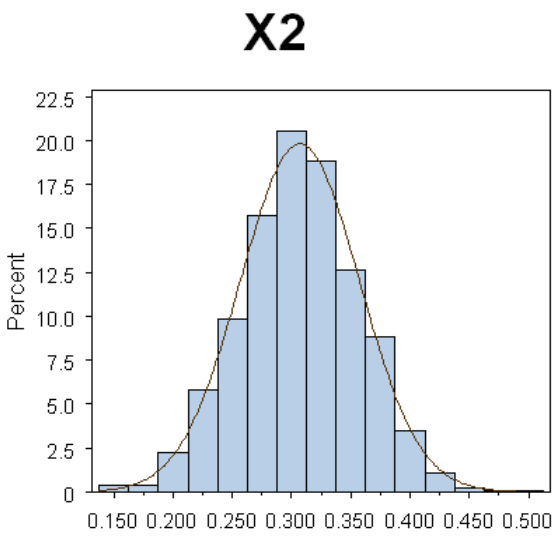

Figure 4: Histogram for Fits of X2

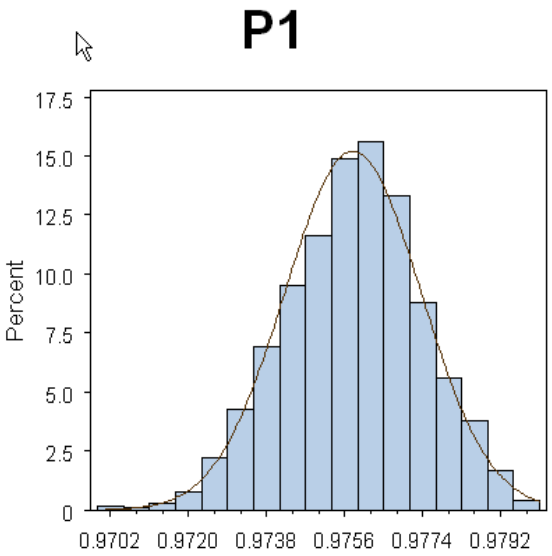

Figure 6: Histogram for Fits of P1

The point we wish to make here is that creating the numerous copies of the fitted parameters corresponding to the different samples drawn can be done in the same loop in which the estimation is done. Everything is done at the same level of programming and functions we wrote earlier can be reused in a more complex problem. While the code inside the FCMP functions themselves may resort to calling macros, the fact that they can all be called from within a single programming environment allows for much more readability, more reusability, and code that is easier to maintain over time.

CONCLUSION

This paper demonstrated that SAS can be used as a functional programming language. PROC FCMP provides the vehicle to write reusable, independent program units (functions and subroutines). These units can be written and tested independently. This enables programmers to more easily read, write, and maintain complex code.

Furthermore, these functions and CALL routines are now able to encapsulate virtually any analytical procedure (as well as the DATA step itself) using the RUN_MACRO function. Users can now easily call analytical procedures within

both recursive and iterative algorithms. This enables users to encapsulate preexisting analytical procedures as building blocks for even larger, more complex statistical analysis methods.

REFERENCES

Full examples for this paper can be found in Addendum 326-2010_samples: http://www.sascommunity.org/mwiki/images/1/18/326-2010_samples.pdf.

Keogh, Eamonn, et. al. 2004. "Segmenting Time Series: A Survey and Novel Approach." In *Data Mining in Times Series Databases*, eds. Mark Last, Abraham Kandel, and Horst Bunke, 1-21. London: World Scientific Publishing Co. Pte. Ltd. Chapter is available at http://www.ics.uci.edu/~pazzani/Publications/survey.pdf.

Rioux, Jacques, and Andrew Luong. 1996. "Minimum Quadratic Distance Estimation for the Proportional Hazards Regression Model with Grouped Data." *Actuarial Research Clearing House*, 1:99-109. Available at http://www.soa.org/library/research/actuarial-research-clearing-house/1990-99/1996/arch-1/arch96v110.pdf.

SAS Institute Inc. 2009. "The FCMP Procedure." *Base SAS 9.2 Procedures Guide*. Cary, NC: SAS Institute Inc. Available at http://support.sas.com/documentation/cdl/en/proc/61895/HTML/default/a002890483.htm.

SAS Institute Inc. 2009. "The REG Procedure." *SAS/STAT 9.2 User's Guide, Second Edition*. Cary, NC: SAS Institute Inc. Available at http://support.sas.com/documentation/cdl/en/statug/63033/HTML/default/reg_toc.htm

RECOMMENDED READING

Eberhardt, Peter. 2009. "A Cup of Coffee and Proc FCMP: I Cannot Function Without Them." *Proceedings of the SAS Global Forum 2009 Conference.* Cary, NC: SAS Institute Inc. Available at http://support.sas.com/resources/papers/proceedings09/147-2009.pdf.

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

CONTACT INFORMATION

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

Stacey Christian
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
919-531-4135
Stacey.Christian@sas.com
http://www.sas.com

Jacques Rioux
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
919-531-9386
Jacques.Rioux@sas.com
http://www.sas.com