

Customized Base SAS implemented solvers

Ruben Chiflikyan, Mila Chiflikyan, Jean Richardson, Renee Karlsen, Donna Medeiros
(RTI International, Research Triangle Park, NC)

ABSTRACT

Solvers are software tools or subroutines, designed for solving a problem in the area of numerical analysis. The use of efficient reusable mathematical software as the building blocks for complex programs is beneficial for creation of viable applications. Presently, there are numerous scientific software libraries created on the basis of known high-level programming languages. SAS also has a large number of solvers available in SAS/OR and SAS/IML modules, most of which are not available to SAS Base only users. Thus, the creation of Base SAS implemented scientific software repository of efficient customized solvers is of great importance for SAS community. The goal of this paper is a presentation of three customized Base SAS implemented solvers, based on well known mathematical algorithms. Particularly, the first solver finds a solution x to the equation $f(x) = 0$ using the bisection method as a root-finding algorithm. The second solver is intended for calculating of the numerical value of a one-dimensional definite integral $\int_a^b f(x)dx$, using the modified adaptive Simpson's method based on polynomial of order 2 with the adaptive increment. The third solver finds the roots of a system of linear equations $\{a_{ij}x_j = b_j\}$, where $i, j = 1, \dots, n$ using Cramer's rule and Leibniz formula. We hope that these solvers might be considered as prototypes for future enhancements, and spur other SAS users and developers to contribute to the future Base SAS solvers repository.

INTRODUCTION

There is a wide range of mathematical routines, solvers, subroutines, designed for solving a problem in the area of numerical analysis in applied mathematics. The use of efficient reusable mathematical software as the building blocks for complex programs is beneficial for creation of viable applications. Presently, there are numerous scientific software libraries created on the basis of known high-level programming languages (see, e.g., [1]). SAS also has a large number of solvers available in SAS/OR and SAS/IML modules, however, most of which are not available to SAS Base only users. Thus, the creation of Base SAS implemented scientific software repository of efficient customized solvers is of great importance for SAS community. The goal of this paper is a demonstration of using SAS language capabilities for creation of three customized Base SAS implemented solvers, based on well known mathematical algorithms. Particularly, the first solver finds a solution x to the equation $f(x) = 0$ using the bisection method as a root-finding algorithm. The second solver is intended for calculating of the numerical value of a one-dimensional definite integral $\int_a^b f(x)dx$, using the modified adaptive Simpson's method based on polynomial of order 2 with the adaptive increment. The third solver finds the roots of a system of linear equations $\{a_{ij}x_j = b_j\}$, where $i, j = 1, \dots, n$ using Cramer's rule and Leibniz formula. We hope that these solvers might be considered as prototypes for future enhancements, and spur other SAS users and developers to contribute to the future Base SAS solvers repository.

THE SOLVER FOR FINDING SOLUTION x TO THE EQUATION $f(x) = 0$

The first solver presented to your attention handles one of the most important topics in numerical analysis, namely finding a solution x to the equation $f(x) = 0$. From many root-finding algorithms developed for these methods (e.g., bisection method, Newton's method, the secant method, etc.) we will present SAS implemented code for the bisection method (see, e.g., [2]), which is based on dividing an interval in half and then selecting the subinterval in which the root exists. It works when the function is a continuous function, and it requires previous knowledge of 2 initial guesses, a and b , such that $f(a)$ and $f(b)$ have opposite signs. The bisection method is one of the bracketing methods for finding roots of equation since it keeps a

bounded interval where there is at least one root at each step. Given a function $f(x)$ and an interval which might contain a root, a predetermined number of iterations are carried out. A simple bisection procedure for iteratively converging on a solution which is known to lie inside some interval $[a, b]$ proceeds by evaluating the function in question at the midpoint of the original interval $x = (a + b)/2$ and testing to see in which of the subintervals $[a, (a + b)/2]$ or $[(a + b)/2, b]$ the solution lies. The procedure then repeated with the new interval as often as needed to locate the solution to desired accuracy. It is interesting to note that convergence is guaranteed. The advantages and limitations of this method are well documented in the existing mathematical literature (see, e.g., [3] and references therein).

Here are some explanations how the macro %Bisect(eqtn,x1,x2,eps) works. Eqtn variable represents the equation under consideration, e.g., $x^3 + 4 = 3x$ or $x^3 - 3x + 4 = 0$. x_1 is the starting point, and x_2 is the ending point of the interval, where the root will be searched, i.e., $x_1 \leq \text{root} \leq x_2$. These values are chosen in such a way that the condition $f(x_1)f(x_2) < 0$ is satisfied. The method terminates the interval bracketing the root has the length $|f(\text{root})| \leq \text{eps}$, where eps defines the accuracy of the calculation. The comments are presented in the code below.

THE CODE FOR FINDING THE ROOTS USING BISECTION METHOD:

```
%macro Bisect(eqtn,x1,x2,eps);
%global root;
* standardizing the initial equation,
i.e.,  $f_1(x)=f_2(x)$  is presented as  $f(x)=0$ ;
%let eqtn2=%scan(&eqtn,1,"=");
%let eqtn3=%scan(&eqtn,2,"=");
%let eqt=%str(y=&eqtn2-&eqtn3);

data one(keep=root);
flag=0;
x=&x1;
&eqt; y1=y; calculating  $f(x_1)$  and saving it as  $y_1$ ;
x=&x2;
&eqt; y2=y; calculating  $f(x_2)$  and saving it as  $y_2$ ;

x1=&x1;
x2=&x2;
*if the root is not between the initial values. In this case, one need
to change their values.

if y1*y2 > 0 then do; put 'No root within these borders!'; end;
*the initial range is divided in 2 sub ranges and one of them,
 $f(x_1)*f(x_2) < 0$  is chosen ;
else do until ( flag=1);
x3=0.5*(x1+x2);
x=x3;
&eqt; y3=y; calculating  $f(x_3)$  and saving as  $y_3$ ;

if y1*y3<0 then x2=x3; * sets the middle point x3 as the end point of the new interval , while the starting
point (x1) remains unchanged ;
else x1=x3; * sets the middle point x3 as the starting point of the new interval, while the ending point (x2)
remains unchanged ;
*the process repeats until the the required accuracy is achieved. If the middle point is less than the defined
accuracy then the corresponding value of x3 is considered as a root;
if abs(y3)< &eps then do; flag=1; root=x3;
call symput('root',root); end; * the root is presented in the form of a global macro variable ;
end;
run;
```

```
%mend Bisect;
%put &root;
```

EXAMPLES:

Let us demonstrate the accuracy of the method on the example for the $\sin(x/4) - \sqrt{2}/2 = 0$ trigonometric equation. The $\text{PI}=3.141592654..$ is one of the roots of this equation. Below, one can see how the root value depends on eps variable value, which defines the accuracy of the result. The range, where the root was searched, is (2.5,3.5).

```
%Bisect(sin(x/4)-sqrt(2)/2=0,2.5,3.5,1E-3); root=3.140625;
%Bisect(sin(x/4)-sqrt(2)/2=0,2.5,3.5,1E-5); root=3.141602;
%Bisect(sin(x/4)-sqrt(2)/2=0,2.5,3.5,1E-7); root=3.141593;
```

Here, is an example for finding root for quadratic equation: $x^2 + 1.5x - 1 = 0$, where one of the roots equals 0.5. The macro is used as follows: %Bisect($x^2+1.5x-1=0,1,10,1E-4$). The exact root is equal to 0.5, even the accuracy $\text{eps}=1E-4$ was not very high.

Here is the example of a simple transcendental equation in the form $\exp(-x)=x$, where the exact solution is equal to 0.5671432904.

```
%Bisect(exp(-x)=x,1,10,1E-5); root=0.5671405792;
%Bisect(exp(-x)=x,1,10,1E-7); root=0.5671432614;
%Bisect(exp(-x)=x,1,10,1E-9); root=0.5671432905;
```

Here is the example of another quadratic equation $x^2-1600=0$, where the exact root equals 40.

```
%Bisect(x**2=1600,1,100,1E-2); root=40.0000572;
%Bisect(x**2=1600,1,100,1E-3); root=40.0000100;
%Bisect(x**2=1600,1,100,1E-4); root=40.0000017;
%Bisect(x**2=1600,1,100,1E-5); root=40.0000000;
```

THE SOLVER FOR CALCULATING OF THE VALUE OF A DEFINITE INTEGRAL $\int_a^b f(x)dx$

The second solver is intended for calculating of the numerical value of a one-dimensional definite integral:

$$\int_a^b f(x)dx,$$

where the integrand $f(x)$ is a continuous function and the a and b are the limits of integration. A large class of quadrature rules (e.g., Newton-Cotes formula, Gaussian quadrature rule, Glenshaw-Curtis quadrature, etc) were developed and successfully used [4]. We implemented in SAS the algorithm based upon the modified adaptive Simpson's method using polynomial of order 2 with the adaptive increment. For details of this method with all its advantages and limitations see [5].

Here are the steps used in the algorithm implemented in the SAS program below.

#1. The integration is carried out consequently by adding the new portions in the form of $\int_x^{x+step} f(x)dx$ to

the already calculated integral in the form of $\int_a^x f(x)dx$. Note that the value of the *step* variable is defined in the previous step.

#2. The integral $\int_x^{x+step} f(x)dx$ is calculated using Simpson's formula.

#3. The integral $\int_x^{x+step/2} f(x)dx$ is calculated also using Simpson's formula.

#4. If the difference between these two integrals exceeds a user-specified tolerance, then the program returns

to the step#2 and calls for subdividing the interval of integration in two, repeating the calculations with $step/2$, otherwise, the program returns to the step #1. These consequent steps are continued until the boundary b is reached. The program below takes care for careful integration in the closeness of b .

Notes to the program below.

The first macro %func(x) determines the integrand. There are no restrictions on the number of statements and the number of intermediate coefficients used in the constructing of the equation. The only requirement is the existence of the $y = f(x)$.

The second macro %Int determines the integration rule to be used in this program. Specifically here we use Simpson rule's scheme.

The third macro AdpSimps(from, to, eps) performs the calculation. Here from and to variables determine the integration range, while eps determines the accuracy.

IMPLEMENTATION OF ADAPTIVE SIMPSON'S METHOD FOR QUADRATURE SOLVER

* the specific form of the integrand;

```
%macro func(x);
```

```
y=f(x);
```

```
%mend func;
```

* the specific method of integration,

Simpson's rule used, particularly, here);

```
%macro Int(x1,x2,x3);
```

```
%func(x1); y1=y;
```

```
%func(x2); y2=y;
```

```
%func(x3); y3=y;
```

```
S=(x3-x1)/6*(y1+4*y2+y3);
```

```
%mend Int;
```

* this macro performs the calculation.

*Here from is the starting point of integration, to is the end point of integration, and eps is the accuracy;

```
%macro AdpSimps(from, to, eps);
```

```
%global integral;
```

```
data one;
```

```
*initialization;
```

```
st0=0.1; *initial step;
```

```
accum=0; * accumulator initialization;
```

```
Xst=&from; * the starting point initialization;
```

```
label1;;
```

```
* 2 regions;
```

```
st=st0;
```

```
x1=Xst;
```

```
x2=x1+st;
```

```
x3=x2+st;
```

```
x10=x1;
```

```
x20=x2;
```

```
x30=x3;
```

```
st0=st;
```

```
%Int(x1,x2,x3); S12=S;
```

```
* first part;
```

```
st=st/2;
```

```
x1=Xst;
```

```
x2=x1+st;
```

```
x3=x2+st;
```

```
%Int(x1,x2,x3); S1=S;
```

```
*second part;
```

```
x1=Xst+2*st;
```

```
x2=x1+st;
```

```

x3=x2+st;
%Int(x1,x2,x3); S2=S;

*comparison;
S1_2=S1+S2;
eps0=abs( (S12-S1_2)/S1_2 );
eps=&eps;
if eps0 <= eps then
do;
Accum=Accum+S12;
Xst=x30;
st0=2*st0;
if Xst < &to then goto label1; *regular mode;
else do; * subtracting surplus integral;
x1=&to;
x2=&to+0.5*(Xst-&to);
x3=Xst;

%Int(x1,x2,x3); Slast=S;
Accum=Accum-Slast;
goto label2;
end;
end;

else do;
Xst=x10;
st0=st0/2;
goto label1;
end;

label2;;
drop st0 xst st x1 x2 x3 x10 x20 x30
p1 p2 y y1 y2 y3 s s12 s1 s2 s1_2 slast eps0;
call symput('Integral',Accum);

run;
%mend AdpSimps;

%AdpSimps(0,3.14159265,1E-4);
%put &integral;

```

EXAMPLES:

Let us to illustrate practical usage of the method. For example, to calculate the definite integral $\int_0^{10} x^3 dx$ with

the accuracy $\text{eps}=10^{-5}$, one need to use the following scheme:

1. The %func macro takes the following form:

```

%macro func(x);
y=(&x)**3;
%mend func;

```

2. The %Int(x_1, x_2, x_3) macro remains unchanged;

3. The %AdpSimps(*to, from, eps*) takes the following form:

```

%AdpSimps(0,10,1E-5); The calculated value of the integral is 2500, which is the same as the
exact value;

```

In another example, to calculate $\int_0^a [k_1 \cos(k_2 x) + k_3 \sin(k_4 x)] dx$, for $a = 3.14159265$,

$k_1 = 0, k_2 = 1, k_3 = 0$, and $k_4 = 1$ with the accuracies $1E-4, 1E-6, 1E-8$, and $1E-10$, one need to use the following scheme:

1. The %func macro takes the following form:

```
%macro func(x);
k1=0; k2=1; k3=1; k4=1; * coefficients of the function;
p1=k1*cos(k2*x); * intermediate calculations;
p2=k3*sin(k4*x); * intermediate calculations;
y=p1+p2; * final form of the integrand;
%mend func;
```

2. The %Int(x_1, x_2, x_3) macro remains unchanged;

3. The %AdpSimps(*to, from, eps*) takes the following form:

```
%AdpSimps(0,3.14159265,1E-4); Integral=2.0000174508;
%AdpSimps(0,3.14159265,1E-6); Integral=2.0000011115;
%AdpSimps(0,3.14159265,1E-8); Integral=2.0000000043;
%AdpSimps(0,3.14159265,1E-10); Integral=2.000000000.
```

THE SOLVER FOR FINDING UNKNOWN OF A SYSTEM OF LINEAR EQUATIONS

The third solver is intended for finding the roots of a system of linear equations:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2, \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n, \end{cases}$$

where $x_1, x_2, \dots$, and x_n stand for unknowns, the numbers $a_{11}, a_{12}, \dots, a_{nn}, b_1, b_2, \dots, b_n$ represent real constants. There are many different ways (e.g., Gaussian elimination, Gauss-Jordan elimination, etc.) to solve this system[6]. Particularly, we will use Cramer's rule, which states that the unknowns $x_1, x_2, \dots$, and x_n can be found according to the following formulas:

$$x_i = \frac{\Delta_i}{\Delta},$$

where $i=1, \dots, n$, and $\Delta, \Delta_1, \Delta_2, \dots, \Delta_n$ are the determinants of the following $n \times n$ matrices:

$$\Delta = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}, \Delta_1 = \begin{bmatrix} b_1 & a_{12} & \dots & a_{1n} \\ b_2 & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_n & a_{n2} & \dots & a_{nn} \end{bmatrix}, \Delta_2 = \begin{bmatrix} a_{11} & b_1 & \dots & a_{1n} \\ a_{21} & b_2 & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & b_n & \dots & a_{nn} \end{bmatrix}, \dots, \Delta_n = \begin{bmatrix} a_{11} & a_{12} & \dots & b_1 \\ a_{21} & a_{22} & \dots & b_2 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & b_n \end{bmatrix}.$$

As one can notice, Δ_k , where $k=2, \dots, n$ is basically Δ , where the k th column is replaced by the vector

$$\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

The determinants are calculated by using Leibniz formula(see, e.g., [7]):

$$\det(\Delta) = \sum_{\sigma \in S_n} \text{sgn}(\sigma) \prod_{i=1}^n A_{i, \sigma(i)},$$

here the sum is computed over all permutations σ of the numbers $\{1, 2, \dots, n\}$ and $\text{sgn}(\sigma)$ denotes the signature of the permutation σ : +1 if σ is an even permutation and -1 if it is odd. This formula contains $n!$ (factorial) summands, and it is therefore impractical to use it to calculate determinants for large n . For the current SAS implementation we have found that the program can effectively work for $n \leq 9$.

Below is the code for SAS implemented solver for the solution of a system of linear equations.

Note that constants from the system, namely, $a_{11}, a_{12}, \dots, a_{nn}, b_1, b_2, \dots, b_n$, are loaded into program via 'constants' dataset, which has the following structure:

```
%let rang=n;
Data constants;
q1      = a11;  q2      = a12;  K;  qn      = a1n;  qn+1    = b1;
qn+2    = a21;  qn+3    = a22;  K;  q2n+1    = a2n;  q2n+2    = b2;
      M      M      M      M ;
qn(n+1)-n = an1;  qn(n+1)-n+1 = an2;  K;  qn(n+1)-1 = ann;  qn(n+1) = bn;
run;
```

Let us illustrate this for the system of rang=3. To solve the system

$$\begin{cases} 2x_1 + 3x_2 + 3x_3 = 17, \\ 3x_1 + x_2 + 2x_3 = 11, \\ x_1 + 3x_2 + 2x_3 = 13. \end{cases}$$

the 'constants' dataset should be written in the following way:

```
%let rang=3;
data constants;
q1=2; q2 =3; q3 =3; q4=17;
q5=3; q6 =1; q7 =2; q8=11;
q9=1; q10=3; q11=2; q12=13;
run;
```

SAS IMPLEMENTATION OF LINEAR EQUATION SOLVER BASED ON CRAMER'S RULE AND LEIBNIZ FORMULA

```
%let rang=n; * put the rang of the matrix;
Data constants;
q1      = a11;  q2      = a12;  K;  qn      = a1n;  qn+1    = b1;
qn+2    = a21;  qn+3    = a22;  K;  q2n+1    = a2n;  q2n+2    = b2;
      M      M      M      M ;
qn(n+1)-n = an1;  qn(n+1)-n+1 = an2;  K;  qn(n+1)-1 = ann;  qn(n+1) = bn;
run;
```

```

%let str=;
%let str7=z=zz(1,1);
%let str6=;
%let str4=keep=Z ;

%macro selmid;
%global str14 str1 str3 str13 str5 str8 str9 str10
str11 cnt str12;
%local l mf ms;
%do l=1 %to &rang;

%let str1= &str1 if %eval(&l*&rang-&rang+1) <= l <=
%eval(&l*&rang) then sub(l)=all(l+%eval(&l-1))
%str(;);

%let str14= &str14 all(%nrstr(&l)+ %eval(-1+(&rang+1)*(&l-1)) )=
all(%eval(&l*(&rang+1)))
%str(;);

%let md=%eval(&l+1);
%let str3= &str3
x&l=%nrstr(&det)&md/%nrstr(&det1)
%str(;);

%let str4=&str4 l&l;
%let str13=&str13 do l&l=1 to %nrstr(&rang)
%str(;);

%let str5=&str5 end
%str(;);

%let str6=&str6 l&l;

%if &l >=2 %then %do;
%let str7=&str7 * zz(&l,l&l);
%end;

%end;

%let mal=%eval(&rang*&rang+1));
%do mf=1 %to &mal;
%let str8= &str8 call symput ("q&mf",sub&mf)
%str(;);
%let str9= &str9 q&mf
;
%if &mf <= %eval(&rang+1) %then %do;
%let str10= &str10 det&mf
;
%end;

%end;

%let cnt=0;
%do mf=1 %to &rang;
%do ms=1 %to &rang;

%let cnt=%eval(&cnt +1);
%let str11= &str11 zz(&mf,&ms)=%nrstr(&q)&cnt%str(;);

%if &mf < &ms %then %do;
%let str=&str and l&mf ^= l&ms ;
%end;

%end;
%end;

```

```

%let str12=%substr(&str,5);

%mend selmid;
%selmid;

%macro det;
%global det;
* keeping only l1 through l&rang and Z ;
data zero(&str4);
array zz[1:&rang,1:&rang];

* filling the array elements with
coefficient values;
%unquote(&str11);

%unquote(&str13);
* creating &rang times repeating do loop header;
* the criterion of selecting from 1 to &rang nonreptitive
numbers from each row and column;
    if &str12
        then do;

* calculation the product of elements corresponding to each
to each of the numbers defined above;
        &str7;
        output;
    end;
* creating &rang times repeating do loop ending;
&str5;
run;

* calculating the number of inversions for a given
of set of numbers, then calculating the sum of determinants ;
data one (keep=det);
set zero end=eof;
retain det 0;
cntr=0; sign=-1;
* creating an array elements from 1 through &rang;
    array ll(&rang) &str6;
    * repeat inversion &rang times;
    do m=1 to &rang;
        * inversing all neighbors;
        do n=1 to &rang-1;
            if ll(n)>ll(n+1) then
                do;
                    temp=ll(n);
                    ll(n)=ll(n+1);
                    ll(n+1)=temp;
                    * counter works each time inversion occurs;
                    cntr=cntr+1;
                end;
            end;
        end;
    end;
    * checking the parity of inversion number;
    if mod(cntr,2)=0 then sign=1;
z1=sign*z; * calculating the determinant;
det=det+z1;
if eof;
call symput('det',det);
run;
%mend det;

* defining macro variables with prefixes det (determinants)

```

```

and g (all coefficients);
%global
&str10
&str9
;

%macro sep;
%local sm i;
%let sm=%eval(&rang*(&rang+1));
%do i=1 %to %eval(&rang+1);
data minus(keep=sub1-sub&cnt);
set constants;
array all[&sm] q1-q&sm;
array sub[&cnt] sub1-sub&cnt;

* substituting the free member for each row
in the ith position;
%if &i ^=1 %then %do;
%unquote(&str14);
%end;

do l=1 to %eval(&rang*&rang);

* filling the 'sub' array with appropriate
values from 'all' array leaving only
coefficients are variables (not free members) ;
%unquote(&str1);
end;

* creating macro variables from all
elements in 'sub' array;
%unquote(&str8);
run;

* creating macro variables
with determinants value;
%det;
%let det&l=&det;

%end;
%mend sep;
%sep;

*calculating the roots x1, x2, etc ;
data comb;
%unquote(&str3);
run;

* "deleting" global macro variables created during execution
of this program;
%macro dlt;
%local l;
%let str=;
%let tot=%eval(&rang*(&rang+1));
%do l=1 %to &tot;
%let q&l=;

%if &l <= 8 %then %do; %let det&l=; %end;

%if &l <= 14 %then %do; %let str&l=; %end;

%end;
%let rang=;

```

```
%mend dlt;
%let str14=; %let str1=; %let str3=; %let str13=;
%let str5=; %let str8=; %let str9=; %let str10=;
%let str11=; %let cnt=; %let str12=;
%dlt;
```

We conducted numerical simulations to estimate the real time needed to solve the system of linear equations using a computer the following characteristics: Intel®, Pentium® 4CPU, 3 GHZ, 504 MB of RAM. The calculation time to find unknowns $x_1, x_2, \dots, \text{ and } x_n$ was found to be less than 1 sec for the rang ≤ 6 , less than 2 sec for rang=7, less than 8 sec for rang =8, and less than 260 sec for rang=9. The exponentially increasing time is a consequence of using time-consuming Leibniz formula in calculatig the determinants of the matrices.

CONCLUSION

We presented in this paper three solvers, which are intended for finding the real root of an equation, the values of a definite integral, and the unknowns of a system of linear equations. The great advantage of the SAS implemented solvers is their ability to conduct calculations and generate results *without* leaving the SAS environment. We are aware of the limitations of the presented implementation of known algorithms but hope that they could be considered as the prototypes for the future high performance scalable state-of-the-art SAS implemented algorithms. We hope that SAS users and developers will be encouraged to create other effective SAS implemented solvers for various problems in applied mathematics.

REFERENCES

- Netlib repository at UTK and URNL (www.netlib.org/); GNU repository of various types of solvers (www.gnu.org/); ALGLIB (<http://www.alglib.net/>).
1. 'The Bisection Method'. <http://math.fullerton.edu/mathews/n2003/BisectionMod.html>; 'Bisection', (<http://mathworld.wolfram.com/Bisection.html>).
 2. Richard L. Burden, J. Douglas Faires (2000), "Numerical Analysis, (7th Ed)", Brooks/Cole.
 3. David Kincaid, Ward Cheney (2002), "Numerical Analysis, Mathematics of Scientific Computing" (3rd Ed)", Brooks/Cole.
 4. William M. McKeeman: Algorithm 145: Adaptive numerical integration by Simpson' rule. Commun. ACM5(12):604(1962).
 5. David C. Lay(2002),"Linear Algebra and Its Applications(3rd Ed)", Addison Wesley.
 6. Determinant. (2007, April 20). In *Wikipedia, The Free Encyclopedia*. Retrieved 18:43, April 20, 2007, from <http://en.wikipedia.org/w/index.php?title=Determinant&oldid=124361682>

ACNOWLEDGMENTS

The authors are thankful to Tonya Chapman from the SAS Institute Technical Support Group for answering several questions during development of this program.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. For more information (including electronic copies of these programs) contact authors at:

Ruben Chiflikyan	rchiflikyan@rti.org	(919)541-6064
Mila Chiflikyan	milachif@rti.org	(919)336-3660
Jean Richardson	jer@rti.org	(919)541-6107
Renee Karlsen	rbk@rti.org	(919)541-7115
Donna Medeiros	djm@rti.org	(919)541-1064

Mailing address for all authors:

RTI International
P.O. Box 12194
Research Triangle Park, NC 27709-2194
Web:<http://www.rti.org>.
Fax: (919) 541-6178

SAS and all other SAS Institute Inc. product or service names are registered trademarks are trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.