NLREG DLL Interface

The NLREG Dynamic Link Library (nlreg.dll) is designed to make it easy for production applications to call on NLREG as an "engine" to perform nonlinear regressions. Entry points are provided to pass in a NLREG source program, check for compile errors, pass in data values, perform the regression and fetch computed parameter and statistic values.

Entry points are provided for calls from Visual Basic (and VBA) applications and also C++ applications. The Visual Basic entry points are described first followed by the C++ entry points.

Integer values are passed as long (32-bit) values and floating point values are passed as double (64-bit) values. String values are passed and returned in the natural style that is appropriate for the language.

Visual Basic Entry Points

All of the Visual Basic entry points have names beginning with "nlregvb" (e.g., nlregvbCompile(), nlregvbCompute(), etc.). Values are passed by reference, which is the default method used by Visual Basic, so it is not necessary to specify the calling type. For example, the following declaration and code fragment could be used to retrieve the name and computed value for a parameter:

```
Private Declare Function nlregvbParameterName Lib "nlreg.dll" (index As Long) As
String

Private Declare Function nlregvbParameterValue Lib "nlreg.dll" (index As Long, valtype
As Long) As Double

Dim pvalue As Double
Dim pname As String

pname = nlregvbParameterName(1)
pvalue = nlregvbParameterValue(1,0)
```

Descriptions of the Visual Basic entry points follow.

nlregvbInitialize() -- Initialize for a new analysis

Private Declare Function nlregvbInitialize Lib "nlreg.dll" () As Long

The nlregvbInitialize() function resets NLREG for a new program. It should be called before starting a new analysis involving a new source program. You should not call it if you are changing the data values and re-computing using the same source program.

nlregvbCompile() -- Compile a NLREG program

Private Declare Function nlregvbCompile Lib "nlreg.dll" (*source* As String) As Long

The nlregvbCompile() function accepts a NLREG source program, compiles it and returns a status code indicating if there were any compile errors.

Arguments:

source = A string containing the NLREG source program. As usual, source statements are separated by semicolon (';') characters. You may place carriage-return and line-feed characters between the statements, but it is not necessary.

There are three ways to provide the data values:

1. You can pass the data records as part of the source program. In this case, the source program should end with a DATA statement of the form:

   DATA;

   and the data records should immediately follow it. You may separate the data records either with carriage-return/line-feed characters or by semicolons.

2. You can place the data in an external file. In this case the ending statement must be:

   DATA "*filename*";

   Where *filename* is the full specification of the file with the data records. Note: you should specify the directory along with the file name (for example, "c:\work\test.dat").

3. You can pass the data values in using the nlregvbSetDataArray() or nlregvbSetDataValue() functions described below. In this case, the program must end with the following statement:

   DATA;

   and no data records should follow it.

Returned value:

The value 1 is returned by nlregvbCompile() if the compilation was successful. A value of 0 (zero) is returned if there were compile errors. You can use the nlregvbGetAnalysisReport() function to obtain a listing of the compile errors.

nlregvbCompileFile() -- Compile a NLREG program from a file

Private Declare Function nlregvbCompileFile Lib "nlreg.dll" (*filespec* As String) As Long

The nlregvbCompileFile() function reads a NLREG source program from a file, compiles it and returns a status code indicating if there were any compile errors. The operation of this function is identical to nlregvbCompile() except that you specify the name of a file containing the NLREG program rather than passing the source program in as string variable.

Arguments:

filespec = A string containing the specification for the file containing the NLREG source program. You should specify the directory as well as the file name (for example, "c:\work\test.nlr").

nlregvbGetAnalysisReport() -- Get compiler and analysis report

Private Declare Function nlregvbGetAnalysisReport Lib "nlreg.dll" () As String

This function returns a string that is either (a) a list of the compile errors if nlregvbCompile() returned value of 0, or (b) a listing of the results of the NLREG analysis run if the compilation was successful and nlregvbCompute() was called.

nlregvbCompute() -- Perform a regression analysis

Private Declare Function nlregvbCompute Lib "nlreg.dll" () As Long

The nlregvbCompute() function is called after nlregvbCompile() or nlregvbCompileFile() to perform the regression analysis. If you are running multiple analyses using the same NLREG source program and different sets of data values, you can call nlregvbCompile() once and then call nlregvbCompute() repeatedly after providing each set of data values.

The following values are returned by nlregvbCompute():

6 = Both parameter and relative function convergence
5 = Parameter convergence
4 = Relative function convergence
3 = Absolute function convergence
2 = Singular convergence. (Possibly mutually dependent parameters)
1 = False convergence. (Answers may not be correct.)
-1 = Function did not converge before iteration limit reached
-2 = Function cannot be computed at starting parameter values
-3 = Bad parameter values
-4 = Jacobian could not be computed
-5 = Singular matrix. Possibly mutually dependent parameters
-6 = Function did not converge before max allowed function calls
-7 = There are fewer data observations than parameters.

nlregvbNumInputVariables() -- Get number of input variables

Private Declare Function nlregvbNumInputVariables Lib "nlreg.dll" () As Long

This function returns a count of the number of input variables that were specified in the NLREG program.

nlregvbVariableIndex() -- Get index number for variable or parameter

Private Declare Function nlregvbVariableIndex Lib "nlreg.dll" (*name* As String) As Long

The nlregvbVariableIndex() function looks up the name of a variable or parameter and returns an index number that NLREG uses to identify the variable or parameter. This index number can be used as an argument to other functions such as nlregvbInputVariableName() The *name* argument is the name of the variable or parameter. Both input and work variables may be specified as well as parameters. The names of variables and parameters are not case sensitive. The value returned by the function is the index number that corresponds to the name. A value of -1 is returned if the name cannot be found.

nlregvbInputVariableName() -- Get the name of an input variable

Private Declare Function nlregvbInputVariableName Lib "nlreg.dll" (*index* As Long) As String

The nlregvbInputVariableName() function returns the name of an input variable. The *index* argument is a 0-based index to select which variable name you want.

nlregvbNumParameters() -- Get number of computed parameters

Private Declare Function nlregvbNumParameters Lib "nlreg.dll" () As Long

This function returns a count of the number of parameters that were specified in the NLREG program.

nlregvbParameterName() -- Get the name of a computed parameter

Private Declare Function nlregvbParameterName Lib "nlreg.dll" (*index* As Long) As String

The nlregvbParameterName() function returns the name of a parameter. The *index* argument is a 0-based index to select which parameter name you want.

nlregvbSetParameterValue() -- Set the value for a parameter

Private Declare Function nlregvbSetParameterValue Lib "nlreg.dll" (*index* As Long, *valtype* As Long, *value* As Double) As Long

The nlregvbSetParameterValue() function sets the value for a parameter.

There are two situations where this function is useful:

1. You can set the initial (starting) value of a parameter prior to calling nlregvbCompute() to fit the model to the data. In this case, nlregvbSetParameterValue() should be called after nlregvbCompile() or nlregvbCompileFile() and before nlregvbCompute().

2. You can set the value for a parameter before calling nlregvbEvaluateFunction() if you want to use a specified parameter rather than the computed parameter value when evaluating the value of a function. In this case, nlregvbSetParameterValue() must be called after nlregvbCompute() and before nlregvbEvaluateFunction().

The *index* argument is a 0-based index to select which parameter you are setting the value for.

The *valtype* argument selects which type of value you want to set for the parameter. The following values may be specified for *valtype*:

The *value* argument is the value to which the parameter is to be set.

0 (PARSET_INITIAL) = Initial value for the parameter.

The following values are returned by nlregvbSetParameterValue():

0 (RVSDA_OK) = Success.
12 (RVSDA_BADVAR) = The *index* parameter argument is invalid.
13 (RVSDA_BADTYPE) = The *valtype* argument is invalid.

nlregvbParameterValue() -- Get the computed value for a parameter

Private Declare Function nlregvbParameterValue Lib "nlreg.dll" (*index* As Long, *valtype* As Long) As Double

The nlregvbParameterValue() function returns a value for a computed parameter value. The *index* argument is a 0-based index to select which parameter you are getting values for.

The *valtype* argument selects which type of value you want for the parameter. The following values may be specified for *valtype*:

0 (PARVAL_ESTIMATE) = Computed estimate for the parameter
1 (PARVAL_STDERR) = Standard error of the estimate
2 (PARVAL_TVALUE) = t value for the computed parameter value
3 (PARVAL_PROBT) = Probability of the t value (Prob(t))
4 (PARVAL_CONFLOW) = Lower value of the confidence interval
5 (PARVAL_CONFHI) = Upper value of the confidence interval
6 (PARVAL_INITIAL) = Initial value that was specified for the parameter

nlregvbSetDataArray() -- Set input data values using an array

Private Declare Function nlregvbSetDataArray Lib "nlreg.dll" (*dataarray*() As Double, *rows* As Long) As Long

There are four methods for specifying sets of data values for an analysis:

1. You can end your NLREG source program with a Data; statement and specify the data records starting with the next line of the program.
2. You can end your NLREG source program with a Data "*filename*"; statement and put the data in an external file.
3. You can use the nlregvbSetDataArray() function to specify the data values as a full array.
4. You can use the nlregvbSetDataRows() and nlregvbSetDataValue() functions to specify each element of the data array.

The nlregvbSetDataArray() function sets an array of values to be used by the next call of nlregvbCompute(). All previously specified data values are deleted and replaced by the values specified in the *dataarray* array.

The nlregvbSetDataArray() function must be called after nlregvbCompile(). If you want to run multiple analyses with the same NLREG program but with different sets of data values, the correct sequence of calls is:

```
nlregvbCompile()
nlregvbSetDataArray()
nlregvbCompute()
nlregvbSetDataArray()
nlregvbCompute()
…
```

If you call nlregvbCompile() to specify a new NLREG program, you must specify a new set of data values before calling nlregvbCompute().

The *dataarray* array is a two-dimensional array. Note that in the function declaration you must specify parentheses after the data variable (see above). The data is stored in the array so that the first dimension specifies the row (observation number) and the second dimension specifies the column (input variable). For example, if there were 100 observations (cases) and two input variables, the array would be dimensioned (0 To 99, 0 To 1). Note: you can also use a different base for the range such as (1 To 100, 1 To 2).

The *rows* argument specifies how many data observations there are in the array. If you specify 0 or –1 for *rows*, the actual size of the first dimension of the array will be used as the number of observations. The value of the *rows* argument should never exceed the size of the first dimension of the array. You may specify a value for *rows* that is less than the size of the first dimension of *dataarray* if you wish to use fewer observations than the array is dimensioned to store.

The array must be dimensioned so that there at least as many columns (second dimension) as there are input variables. When using the data values, the value in the first column is used for the first input variable, the second column for the second variable, etc. If the array is dimensioned with more columns than there are input variables, values are only used from as many columns as needed.

For example, if the NLREG program is:

```
Variables x,y;
Parameters a,b;
Function y = a*x + b;
Data;
```

Then there are two input variables, *x* and *y*. If there are 50 observations (cases), then the appropriate statements would be:

```
Dim data(0 To 49, 0 To 1) As Double
Dim result As Long
…
result = nlregvbSetDataArray(data, 50);
```

The following values are returned by nlregvbSetDataArray() and nlregvbSetDataValue():

0 (RVSDA_OK) = Success
1 (RVSDA_NUMDIM) = Data array does not have 2 dimensions
2 (RVSDA_ROWEXCESS) = Specified number of rows exceeds dimension size
3 (RVSDA_COLVAR) = Number of array columns is less than number of variables
4 (RVSDA_NOMEMORY) = Unable to allocate memory for the array
5 (RVSDA_NOPROGRAM) = No program has been specified yet
6 (RVSDA_BADROW) = Invalid row index number
7 (RVSDA_BADCOL) = Invalid column index number

nlregvbSetDataRows() and nlregvbSetDataValue() -- Set input data values

Private Declare Function nlregvbSetDataRows Lib "nlreg.dll" (*numrows* As Long) As Long

Private Declare Function nlregvbSetDataValue Lib "nlreg.dll" (*row* As Long, *col* As Long, *value* As Double) As Long

The nlregvbSetDataRows() and nlregvbSetDataValue() functions are another way to specify data values for an analysis. Rather than calling nlregvbSetDataArray() to specify the entire array of data values at once, you can call nlregvbSetDataRows() to declare how many rows (observations or cases) of data to reserve room for, then call nlregvbSetDataValue() repeatedly to specify a data value for each element of the data array.

The *numrows* argument to nlregvbSetDataRows() specifies how many rows (observations) of data there are. You must call nlregvbSetDataRows() after nlregvbCompile() and before nlregvbSetDataValue().

After you have called nlregvbSetDataRows() to specify how many rows of data there are, you can call nlregvbSetDataValue() to provide a value for each element of the data array. The *row* argument specifies the row index; it must be in the range from 0 to *numrows*-1. The *col* argument specifies the column index; it must be in the range from 0 to *numvar*-1 where *numvar* is the number of input variables in the program. The *value* argument is the data value to be set in the (*row*,*col*) element of the data array.

The nlregvbSetDataRows() and nlregvbSetDataValue() functions return the same set of result codes as nlregvbSetDataArray().

Here is a sample code fragment showing how nlregvbSetDataRows() and nlregvbSetDataValue() might be used:

```
Private Declare Function nlregvbSetDataRows Lib "c:\nlregdll\debug\nlreg.dll" (numrows
As Long) As Long
Private Declare Function nlregvbSetDataValue Lib "c:\nlregdll\debug\nlreg.dll" (row As
Long, col As Long, value As Double) As Long
Private Sub Setdata_Click()
Dim result As Long
nlregvbSetDataRows (4)
result = nlregvbSetDataValue(0, 0, 1.1)
result = nlregvbSetDataValue(0, 1, 2.1)
result = nlregvbSetDataValue(1, 0, 2.2)
result = nlregvbSetDataValue(1, 1, 3.9)
result = nlregvbSetDataValue(2, 0, 3.1)
result = nlregvbSetDataValue(2, 1, 6.2)
result = nlregvbSetDataValue(3, 0, 4.3)
result = nlregvbSetDataValue(3, 1, 7.8)
```

nlregvbGetDataValue() -- Get the value of an input variable

Private Declare Function nlregvbGetDataValue Lib "nlreg.dll" (*row* As Long, *col* As Long) As Double

The nlregvbGetDataValue() function returns the value of an input from a specified row and column position in the data matrix. The *row* argument specifies the observation number and must range from 0 to *numrows*-1. The *col* argument specifies which input variable the data value corresponds to and must be in the range 0 to *numvar*-1.

nlregvbSetVariableValue() -- Set the value of a work variable

Private Declare Function nlregvbSetVariableValue Lib "nlreg.dll" (*index* As Long, *value* As Double) As Long

This function sets the value of a work variable in the NLREG program. The variable will be reinitialized to this value at the beginning of each iteration of the program. Work variables in NLREG programs are declared by using the Double statement. This function must be called after the program has been compiled.

The *index* argument is the index number of the variable; it can be determined by calling nlregvbVariableIndex().

The *value* argument is the value to be assigned to the variable.

nlregvbInitializeArray() -- Initialize a work array

Private Declare Function nlregvbInitializeArray Lib "nlreg.dll" (*index* As Long, *InputValues()* As Double) As Long

This function initialize values for a work array in the NLREG program. It should be caused after the program is called and before it is executed.. Work arrays in NLREG programs are declared by using the Double statement. You can use the ARRAYSIZE() function in a NLREG program to determine the size of each dimension of an array.

The *index* argument is the index number of the array variable; it can be determined by calling nlregvbVariableIndex().

The *InputValues* argument is the name of an array containing the values that are to be transferred to the NLREG array variable. The values are moved to the NLREG array variable, and its dimension sizes are set to match those of the *InputValues* array.

Here is example code to initialize an array with 3 rows and 4 columns:

```
Dim data(0 To 2, 0 To 3) As Double
Dim index As Long
Dim result As Long
…
index = nlregvbVariableIndex(“ary”)
result = nlregvbInitializeArray(index, data);
```

nlregvbGetStatistic() -- Get a computed statistic value

Private Declare Function nlregvbGetStatistic Lib "nlreg.dll" (*valtype* As Long) As Double

This function can be called after nlregvbCompute() to obtain the values of statistics computed for the function.

The *valtype* argument specifies which statistic value is to be returned by the function. The following values for *valtype* may be specified:

0 (GSX_NUMOBS) = Number of observations
1 (GSX_ITERATIONS) = Number of iterations required to obtain convergence
2 (GSX_RESIDUAL) = Sum of squared deviations (residuals)
3 (GSX_SUMDEV) = Sum of deviations
4 (GSX_MAXDEV) = Maximum deviation for any observation
5 (GSX_STDERR) = Standard error of the estimate
6 (GSX_RSQUARED) = Proportion of variance explained (R^2)
7 (GSX_ADJRSQUARED) = Adjusted coefficient of multiple determination
8 (GSX_DURBINWATSON) = Value of Durbin-Watson test for autocorrelation
9 (GSX_RUNTIME) = Number of seconds to perform analysis
10 (GSX_NLREGVERSION) = Version of NLREG being used

11 (GSX_AVRDEV) = Average deviations for the observations

nlregvbGetCorrelationMatrix() -- Get a matrix of correlation values

Private Declare Function nlregvbGetCorrelationMatrix Lib "nlreg.dll" (*cormat*() As Double, *rowcol* As Long) As Long

If the correlate statement is included in the NLREG program, NLREG will compute a correlation matrix for the input variables. If you specify the correlate statement in the program without any variable names, like this:

```
variables x,y,z;
correlate;
```

then all of the variables (x, y and z in this example) will be included in the correlation matrix and there will be a row and a column in the matrix for each one. The order of the rows and columns corresponds to the order in which the variables are declared.

If you specify a set of variables with the correlate statement, like this:

```
variables x,y,z;
correlate x,y;
```

Then only the specified variables are included in the correlation matrix, the matrix has a row and column for each variable and the order of the rows and columns corresponds to the order that the variables are specified with the correlate statement.

Since the correlation of a variable with itself is always 1.00, the diagonal elements of the correlation matrix will always have the value 1.00.

The *cormat()* argument is the name of a two-dimensional array of type Double into which NLREG.DLL will store the computed correlation matrix. You can declare the matrix with a static size using the Dim Basic declaration, or you can declare it with a dynamic size using ReDim. If you declare it with a static size, it must have enough (or more) rows and columns to hold the correlation matrix. For example, if your NLREG program has three declared variables, then you could declare the cormat array in your Basic program using this declaration:

```
Dim cormat(2,2) As Double
```

Note, by default Basic arrays have a starting index of 0, so declaring cormat(2,2) means that the rows and columns are numbered 0 to 2, for a total of 3.

The other way to declare the array is using the ReDim Basic declaration which specifies that the array size is dynamic. In this case, NLREG.DLL resizes the array so that it is exactly as large as needed to hold the correlation matrix. Since the array size will be set when you call **nlregvbGetCorrelationMatrix** you can declare the size as (0,0) like this:

```
ReDim cormat(0,0) As Double
```

Note: The first subscript for an array selects the row, and the second subscript selects the column. That is, cormat(*row*,*column*).

The *rowcol* argument receives a count of the number of rows and columns that the correlation matrix contains. Since the correlation matrix is square, the number of rows and columns are always equal; hence, a single returned value specifies both counts.

The value returned by the function itself indicates whether the correlation matrix was successfully retrieved. The following values can be returned:

0 (RVSDA_OK) = Success (the correlation matrix was stored in *cormat*)
1 (RVSDA_NUMDIM) = The *cormat* array argument does not have 2 dimensions
9 (RVSDA_ARRAYSIZE) = The *cormat* argument is of fixed size (declared with Dim), and it is not large enough to hold the correlation matrix.
10 (RVSDA_ARRAYTYPE) = The *cormat* argument is not of type Double.
11 (RVSDA_NODATA) = The correlation matrix was not computed. Either you did not put a correlation statement in the NLREG program, or this is a minimization analysis rather than a regression analysis.

Here is a program fragment that obtains the correlation matrix and writes its values to a file:

```
ReDim cormat(0,0) As Double
Dim rowcol As Long
Dim row As Long
Dim col As Long
Dim ivalue As Long
…
ivalue = nlregvbGetCorrelationMatrix(cormat(), rowcol)
Open "correlation.txt" For Output As #1   ' Open file for output.
For row = 0 To rowcol - 1
   For col = 0 To rowcol - 1
      Print #1, Format(cormat(row, col), "   ###.####");
   Next
   Print #1,
Next
Close #1
```

nlregvbGetCovarianceMatrix() -- Get a matrix of covariance values

```
Private Declare Function nlregvbGetCovarianceMatrix Lib "nlreg.dll" (covmat() As
Double, rowcol As Long) As Long
```

If the covariance statement is included in the NLREG program, NLREG will compute a covariance matrix for the computed parameter values. There will be a row and a column

in the matrix for each parameter. The order of the rows and columns corresponds to the order in which the parameters are declared.

For example, if the NLREG program contains the following declarations:

```
parameters a,b,c;
covariance;
```

Then the covariance matrix will have three rows and three columns. The entry for the first row and first column will correspond to the 'a' parameter.

The *covmat()* argument is the name of a two-dimensional array of type Double into which NLREG.DLL will store the computed covariance matrix. You can declare the matrix with a static size using the Dim Basic declaration, or you can declare it with a dynamic size using ReDim. If you declare it with a static size, it must have enough (or more) rows and columns to hold the covariance matrix. For example, if your NLREG program has three declared parameters, then you could declare the covmat array in your Basic program using this declaration:

```
Dim covmat(2,2) As Double
```

Note, by default Basic arrays have a starting index of 0, so declaring cormat(2,2) means that the rows and columns are numbered 0 to 2, for a total of 3.

The other way to declare the array is using the ReDim Basic declaration which specifies that the array size is dynamic. In this case, NLREG.DLL resizes the array so that it is exactly as large as needed to hold the covariance matrix. Since the array size will be set when you call **nlregvbGetCovarianceMatrix** you can declare the size as (0,0) like this:

```
ReDim covmat(0,0) As Double
```

Note: The first subscript for an array selects the row, and the second subscript selects the column. That is, covmat(*row*,*column*).

The *rowcol* argument receives a count of the number of rows and columns that the covariance matrix contains. Since the covariance matrix is square, the number of rows and columns are always equal; hence, a single returned value specifies both counts.

The value returned by the function itself indicates whether the covariance matrix was successfully retrieved. The following values can be returned:

0 (RVSDA_OK) = Success (the covariance matrix was stored in *covmat*)
1 (RVSDA_NUMDIM) = The *covmat* array argument does not have 2 dimensions
9 (RVSDA_ARRAYSIZE) = The *covmat* argument is of fixed size (declared with Dim), and it is not large enough to hold the covariance matrix.
10 (RVSDA_ARRAYTYPE) = The *covmat* argument is not of type Double.

11 (RVSDA_NODATA) = The covariance matrix was not computed for one of these reasons: (1) you did not put a covariance statement in the NLREG program; (2) this is a minimization analysis rather than a regression analysis; (3) a solution was not obtained for the analysis (e.g., the solution did not converge); (4) the constrain statement was used to constrain the computed values of the parameters.

Here is a program fragment that obtains the covariance matrix and writes its values to a file:

```
ReDim covmat(0,0) As Double
Dim rowcol As Long
Dim row As Long
Dim col As Long
Dim ivalue As Long
…
ivalue = nlregvbGetCovarianceMatrix(covmat(), rowcol)
Open "covariance.txt" For Output As #1    ' Open file for output.
For row = 0 To rowcol - 1
   For col = 0 To rowcol - 1
      Print #1, Format(covmat(row, col), "   ###.####");
   Next
   Print #1,
Next
Close #1
```

nlregvbEvaluateFunction() -- Evaluate the function with a set of data values

```
Private Declare Function nlregvbEvaluateFunction Lib "nlreg.dll" (inputvalues() As
Double) As Double
```

The nlregvbEvaluateFunction() function evaluates the current NLREG program function with a specified set of input variable values and the computed parameter values. It must be called after nlregvbCompute() has successfully completed the analysis. What nlregvbEvaluateFunction() does is "plug in" a set of input variable values and determine what the corresponding value of the function is using the parameter values computed by the regression analysis. Note: You can call nlregvbSetParameterValue() before evaluating the function if you can to set specific parameter values rather than using the computed parameter values.

The *inputvalues* argument is a one-dimensional array of values to be used for the input variables when the function is evaluated. There must be one value in the input array for each input variable. The index numbers for the variables correspond to the order in which the variables were declared on the variable statement(s) in the NLREG program. Note: One of the input variables is the dependent variable (i.e., the variable on the left side of the equal sign in the function). Since the purpose of the nlregvbEvaluateFunction() function is to compute the value of the dependent variable, its value in the *inputvalues* array is not used. However, you still must reserve room for the dependent variable value in *inputvalues* (you can assign it a value of 0.0 or anything

else), so the total number of elements in *inputvalues* will equal the total number of declared variables including the dependent variable.

Here is an example code fragment that evaluates the function at 101 points varying the value of the independent variable from 0 to 100:

```
Private Declare Function nlregvbEvaluateFunction Lib "nlreg.dll" (inputvalues() As
Double) As Double
Dim funval As Double
Dim inputvalues(2) As Double
Dim i As Long
Dim result As Long
…
result = nlregvbCompute()
For i = 0 To 100
        inputvalues(0) = i          ‘ Set independent variable value
        inputvalues(1) = 0.0        ‘ Set dependent variable value (not used)
        funval = nlregvbEvaluateFunction(inputvalues)
        …
Next
```

C++ Entry Points

NLREG.DLL contains a full set of entry points that can be used to perform regressions from C++ programs.

Three files are provided with the distribution:

nlregdll.h – This is the header file that contains the prototypes for the nlreg.dll entry points. It should be included at the head of each source file that calls nlreg.dll routines by including a statement of the form:

```
#include "nlregdll.h"
```

nlreg.lib – This is a library file that must be added to your C++ project. It contains definitions of all of the modules to satisfy the linker, and it has information informing the C++ Windows runtime that the functions are part of nlreg.dll

nlreg.dll – This is the actual dynamic link library file. This file should be placed in the same directory as the application that is calling it or in a directory that is included in the search path. Windows searches for the DLLs in the following sequence:

1. The directory where the executable module for the current process is located.
2. The current directory.
3. The Windows system directory. The GetSystemDirectory() function retrieves the path of this directory.
4. The Windows directory. The GetWindowsDirectory() function retrieves the path of this directory.
5. The directories listed in the PATH environment variable.

All of the C++ entry points have names beginning with "nlregc" (e.g., nlregcCompile(), nlregcCompute(), etc.). Descriptions of the C++ entry points follow.

nlregcInitialize() -- Initialize for a new analysis

```
void nlregcInitialize(void);
```

The nlregcInitialize() function resets NLREG for a new program. It should be called before starting a new analysis involving a new source program. You should not call it if you are changing the data values and re-computing using the same source program.

nlregcCompile() -- Compile a NLREG program

long nlregcCompile(char *source);

The nlregcCompile() function accepts a NLREG source program, compiles it and returns a status code indicating if there were any compile errors.

Arguments:

source = A null-terminated string containing the NLREG source program. As usual, source statements are separated by semicolon (';') characters. You may place carriage-return and line-feed characters between the statements, but it is not necessary.

There are three ways to provide the data values:

1. You can pass the data records as part of the source program. In this case, the source program should end with a DATA statement of the form:

   DATA;

   and the data records should immediately follow it. You may separate the data records either with carriage-return/line-feed characters or by semicolons.

2. You can place the data in an external file. In this case the ending statement must be:

   DATA "*filename*";

   Where *filename* is the full specification of the file with the data records.

3. You can pass the data values in using the nlregcSetDataArray() or nlregcSetDataValue() functions described below. In this case, the program must end with the following statement:

   DATA;

   and no data records should follow it.

Returned value:

The value 1 is returned by nlregcCompile() if the compilation was successful. A value of 0 (zero) is returned if there were compile errors. You can use the nlregcGetAnalysisReport() function to obtain a listing of the compile errors.
source program.

nlregcCompileFile() -- Compile a NLREG program in a file

long nlregcCompileFile(char **filespec*);

The nlregcCompileFile() function reads a NLREG source program from a file, compiles it and returns a status code indicating if there were any compile errors. The operation of this function is the same as nlregcCompile() except that the source program is read from a file rather than being passed in as a string.

Arguments:

filespec = A null-terminated string containing the specification of the file with the NLREG source program.

nlregcGetAnalysisReport() -- Get compiler and analysis report

void nlregcGetAnalysisReport(char **buf*, long *bufsize*);

This function returns a null-terminated string that is either (a) a list of the compile errors if nlregcCompile() returned value of 0, or (b) a listing of the results of the NLREG analysis run if the compilation was successful and nlregcCompute() was called. The *buf* argument is a pointer to a buffer into which the analysis report will be stored. The *bufsize* argument is the size (in bytes) of the buffer. You can use the sizeof(*buf*) function to pass in the buffer size. If the specified buffer size is smaller than the space required by the report, the report is truncated to fit in the buffer. Typically, analysis reports are around 5,000 bytes long.

nlregcCompute() -- Perform a regression analysis

long nlregcCompute(void);

The nlregcCompute() function is called after nlregcCompile() to perform the regression analysis. If you are running multiple analyses using the same NLREG source program and different sets of data values, you can call nlregcCompile() once and then call nlregcCompute() repeatedly after providing each set of data values.

The following values are returned by nlregcCompute():

6 (COMPVAL_PARFUNCONVERGE) = Both parameter and relative function convergence.
5 (COMPVAL_PARCONVERGE) = Parameter convergence
4 (COMPVAL_FUNCONVERGE) = Relative function convergence
3 (COMPVAL_ABSFUNCONVERGE) = Absolute function convergence
2 (COMPVAL_SINGULARCONVERGE) = Singular convergence.
1 (COMPVAL_FALSECONVERGE) = False convergence. (Answers may not be correct.)
-1 (COMPVAL_NOCONVERGE) = Function did not converge before iteration limit reached.
-2 (COMPVAL_BADSTARTVALUE) = Function cannot be computed at starting parameter values
-3 (COMPVAL_BADPARAMETER) = Bad parameter values.

-4 (COMPVAL_NOJACOBIAN) = Jacobian could not be computed
-5 (COMPVAL_SINGULARMATRIX) = Singular matrix. Mutually dependent parameters?
-6 (COMPVAL_NOCONVERGEFUN) = Function did not converge.
-7 (COMPVAL_NOTENOUGHDATA) = There are fewer data observations than parameters.

nlregcNumInputVariables() -- Get number of input variables

long nlregcNumInputVariables(void);

This function returns a count of the number of input variables that were specified in the NLREG program.

nlregcVariableIndex() -- Get index number for variable

long nlregcVariableIndex(char *_name_);

The nlregcVariableIndex() function looks up the name of a variable and returns an index number that NLREG uses to identify the variable. This index number can be used as an argument to other functions such as nlregcInputVariableName() The *name* argument is the name of the variable. Both input and work variables may be specified. The names of variables are not case sensitive. The value returned by the function is the index number that corresponds to the name. A value of -1 is returned if the name cannot be found.

nlregcInputVariableName() -- Get the name of an input variable

void nlregcInputVariableName(long *index*, char **buf*);

The nlregcInputVariableName() function returns the name of an input variable. The *index* argument is a 0-based index to select which variable name you want. The *buf* argument is a pointer to a character buffer into which the variable name is stored as a null-terminated string. The buffer should be at least 31 characters long.

nlregcNumParameters() -- Get number of computed parameters

long nlregcNumParameters(void);

This function returns a count of the number of parameters that were specified in the NLREG program.

nlregcParameterIndex() -- Get index number for a parameter

long nlregcParameterIndex(char **name*);

The nlregcParameterIndex() function looks up the name of a parameter and returns an index number that NLREG uses to identify the parameter. This index number can be used as an argument to other functions such as nlregcParameterName() The *name* argument is the name of the parameter. Both input and work variables may be specified as well as parameters. The names of parameters are not case sensitive. The value returned by the function is the index number that corresponds to the name. A value of -1 is returned if the name cannot be found.

nlregcParameterName() -- Get the name of a computed parameter

void nlregcParameterName(long *index*, char **buf*);

The nlregcParameterName() function returns the name of a parameter. The *index* argument is a 0-based index to select which parameter name you want. The *buf* argument is a pointer to a character buffer into which the parameter name is stored as a null-terminated string. The buffer should be at least 31 characters long.

nlregcSetParameterValue() -- Set the value for a parameter

long nlregcSetParameterValue(long *index*, long *valtype*, double *value*);

The nlregcSetParameterValue() function sets the value for a parameter.

There are two situations where this function is useful:

1. You can set the initial (starting) value of a parameter prior to calling nlregcCompute() to fit the model to the data. In this case, nlregcSetParameterValue() should be called after nlregcCompile() or nlregcCompileFile() and before nlregcCompute().

2. You can set the value for a parameter before calling nlregcEvaluateFunction() if you want to use a specified parameter rather than the computed parameter value when evaluating the value of a function. In this case, nlregcSetParameterValue() must be called after nlregcCompute() and before nlregcEvaluateFunction().

The *index* argument is a 0-based index to select which parameter you are setting the value for. You can use the nlregcParameterIndex() function to convert a parameter name to an index number.

The *valtype* argument selects which type of value you want to set for the parameter. The following values may be specified for *valtype*:

The *value* argument is the value to which the parameter is to be set.

0 (PARSET_INITIAL) = Initial value for the parameter.

The following values are returned by nlregcSetParameterValue():

0 (RVSDA_OK) = Success.
12 (RVSDA_BADVAR) = The *index* parameter argument is invalid.
13 (RVSDA_BADTYPE) = The *valtype* argument is invalid.

nlregcParameterValue() -- Get the computed value for a parameter

double nlregcParameterValue(long *index*, long *valtype*);

The nlregcParameterValue() function returns a value for a computed parameter value. The *index* argument is a 0-based index to select which parameter you are getting values for.

The *valtype* argument selects which type of value you want for the parameter. The following values may be specified for *valtype*:

0 (PARVAL_ESTIMATE) = Computed estimate for the parameter
1 (PARVAL_STDERR) = Standard error of the estimate
2 (PARVAL_TVALUE) = t value for the computed parameter value
3 (PARVAL_PROBT) = Probability of the t value (Prob(t))
4 (PARVAL_CONFLOW) = Lower value of the confidence interval
5 (PARVAL_CONFHI) = Upper value of the confidence interval
6 (PARVAL_INITIAL) = Initial value that was specified for the parameter

nlregcSetDataArray() -- Set input data values using an array

long nlregcSetDataArray(double **dataarray*, long *rows*, long *cols*);

There are four methods for specifying sets of data values for an analysis:

1. You can end your NLREG source program with a Data; statement and specify the data records starting with the next line of the program.
2. You can end your NLREG source program with a Data "*filename*"; statement and put the data in an external file.
3. You can use the nlregcSetDataArray() function to specify the data values as a full array.
4. You can use the nlregcSetDataRows() and nlregcSetDataValue() functions to specify each element of the data array.

The nlregcSetDataArray() function sets an array of values to be used by the next call of nlregcCompute(). All previously specified data values are deleted and replaced by the values specified in the *dataarray* array.

The nlregcSetDataArray() function must be called after nlregcCompile(). If you want to run multiple analyses with the same NLREG program but with different sets of data values, the correct sequence of calls is:

```
nlregcCompile();
nlregcSetDataArray();
nlregcCompute();
nlregcSetDataArray();
nlregcCompute();
…
```

If you call nlregcCompile() to specify a new NLREG program, you must specify a new set of data values before calling nlregcCompute().

The *dataarray* array is a two-dimensional array. The data is stored in the array so that the first dimension specifies the row (observation number) and the second dimension specifies the column (input variable). For example, if there were 100 observations (cases) and two input variables, the array would be declared

```
double dataarray[100][2];
```

The *rows* argument specifies how many data observations there are in the array. The value of the *rows* argument should never exceed the size of the first dimension of the array. You may specify a value for *rows* that is less than the size of the first dimension of *dataarray* if you wish to use fewer observations than the array is allocated to store.

The array must have at least as many columns (second dimension) as there are input variables. When using the data values, the value in the first column is used for the first input variable, the second column for the second variable, etc. If the array is allocated with more columns than there are input variables, values are only used from as many columns as needed.

For example, if the NLREG program is:

```
Variables x,y;
Parameters a,b;
Function y = a*x + b;
Data;
```

Then there are two input variables, *x* and *y*. If there are 50 observations (cases), then the appropriate statements would be:

```
double data[50][2];
long result;
```

```
…
// Store data values into the array
…
result = nlregcSetDataArray((double *)data, 50, 2);
```

The following values are returned by nlregcSetDataArray() and nlregcSetDataValue():

0 (RVSDA_OK) = Success
3 (RVSDA_COLVAR) = Number of array columns is less than number of variables
4 (RVSDA_NOMEMORY) = Unable to allocate memory for the array
5 (RVSDA_NOPROGRAM) = No program has been specified yet
6 (RVSDA_BADROW) = Invalid row index number
7 (RVSDA_BADCOL) = Invalid column index number

nlregcSetDataRows() and nlregcSetDataValue() -- Set input data values

long nlregcSetDataRows(long *numrows*);

long nlregcSetDataValue(long *row*, long *col*, double *value*);

The nlregcSetDataRows() and nlregcSetDataValue() functions are another way to specify data values for an analysis. Rather than calling nlregcSetDataArray() to specify the entire array of data values at once, you can call nlregcSetDataRows() to declare how many rows (observations or cases) of data to reserve room for, then call nlregcSetDataValue() repeatedly to specify a data value for each element of the data array.

The *numrows* argument to nlregcSetDataRows() specifies how many rows (observations) of data there are. You must call nlregcSetDataRows() after nlregcCompile() and before nlregcSetDataValue().

After you have called nlregcSetDataRows() to specify how many rows of data there are, you can call nlregcSetDataValue() to provide a value for each element of the data array. The *row* argument specifies the row index; it must be in the range from 0 to *numrows*-1. The *col* argument specifies the column index; it must be in the range from 0 to *numvar*-1 where *numvar* is the number of input variables in the program. The *value* argument is the data value to be set in the (*row*,*col*) element of the data array.

The nlregcSetDataRows() and nlregcSetDataValue() functions return the same set of result codes as nlregcSetDataArray().

Here is a sample code fragment showing how nlregcSetDataRows() and nlregcSetDataValue() might be used:

```
long result;
nlregcSetDataRows(4);
result = nlregcSetDataValue(0, 0, 1.1);
result = nlregcSetDataValue(0, 1, 2.1);
result = nlregcSetDataValue(1, 0, 2.2);
```

```
result = nlregcSetDataValue(1, 1, 3.9);
result = nlregcSetDataValue(2, 0, 3.1);
result = nlregcSetDataValue(2, 1, 6.2);
result = nlregcSetDataValue(3, 0, 4.3);
result = nlregcSetDataValue(3, 1, 7.8);
```

nlregcGetDataValue() -- Get the value of an input variable

double nlregcGetDataValue(long *row*, long *col*);

The nlregcGetDataValue() function returns the value of an input from a specified row and column position in the data matrix. The *row* argument specifies the observation number and must range from 0 to *numrows*-1. The *col* argument specifies which input variable the data value corresponds to and must be in the range 0 to *numvar*-1.

nlregcSetVariableValue() -- Set the value of a work variable

void nlregcSetVariableValue(long *index*, double *value*);

This function sets the value of a work variable in the NLREG program. The variable will be reinitialized to this value at the beginning of each iteration of the program. Work variables in NLREG programs are declared by using the Double statement. This function must be called after the program has been compiled.

The *index* argument is the index number of the variable; it can be determined by calling nlregcVariableIndex().

The *value* argument is the value to be assigned to the variable.

nlregcInitializeArray() -- Initialize a work array

void nlregcInitializeArray(long *index*, double **InputValues*, long *dimsize1*, long *dimsize2*, long *dimsize3*);

This function specifies the size for each dimension of an array work variable and specifies the initial values for the array. Work arrays in NLREG programs are declared by using the Double statement. You can use the ARRAYSIZE() function in a NLREG program to determine the size of each dimension of an array. The nlregcCompile() function must be called to compile the program before nlregcInitializeArray() is used.

The *index* argument is the index number of the array variable; it can be determined by calling nlregcVariableIndex().

The *InputValues* argument is a pointer to a vector or array of initial values that are to be set for the array. The number of values must match the number of elements in the array as specified by *dimsize1*, *dimsize2* and *dimsize3*. For arrays with more than one dimension, the right-most dimension index varies the fastest.

The *dimsize1*, *dimsize2* and *dimsize3* arguments specify the first, second and third dimension sizes for the array. If the array has only one dimension, then only *dimsize1* is used, and you should specify 0 (zero) for *dimsize2* and *dimsize3*. Similarly, an array with two dimensions will use only *dimsize1* and *dimsize2*.

Here is example code to initialize an array with 2 rows and 3 columns:

```
double ArrayValue[2][3] = {
   {1, 2, 3},
   {4, 5, 6} };
long varindex = nlregcVariableIndex("calibration");
nlregcInitializeArray(varindex, ArrayValue, 2, 3, 0);
```

nlregcSetArrayValue() -- Set a value in a work array

long nlregcSetArrayValue(double *value*, long *VarIndex*, long *index1*, long *index2*, long *index3*);

This function sets a value in a specified element of a work array. If you want to initialize all elements in an array, you can use the nlregcInitializeArray() function. The nlregcCompile() function must be called to compile the program before nlregcSetArrayValue() is used.

The *value* argument is the value that is to be stored into the specified array element.

The *VarIndex* argument is the index number of the array variable; it can be determined by calling nlregcVariableIndex().

The *index1, index2* and *index3* arguments are the first, second and third dimension array indexes of the array element (i.e., the array subscripts). Their values must be in the range (0..*dimsize*-1). If the array has fewer than three dimensions, specify 0 for the unused indexes.

The function returns 0 (zero) if it executes correctly. If an error occurs, a non-zero error code is returned.

Here is example code to store 123.4 in the (2,3) element of an array named "calibration":

```
long varindex = nlregcVariableIndex("calibration");
nlregcSetArrayValue(123.4, varindex, 2, 3, 0);
```

nlregcGetArrayValue() -- Get a value from a work array

double nlregcGetArrayValue(long *VarIndex*, long *index1*, long *index2*, long *index3*);

This function gets a value from a specified element of a work array. The nlregcCompile() function must be called to compile the program before nlregcGetArrayValue() is used.

The *VarIndex* argument is the index number of the array variable; it can be determined by calling nlregcVariableIndex().

The *index1, index2* and *index3* arguments are the first, second and third dimension array indexes of the array element (i.e., the array subscripts). Their values must be in the range (0..*dimsize*-1). If the array has fewer than three dimensions, specify 0 for the unused indexes.

The function returns the value stored in the specified element of the array.

Here is example code to get the value in the (2,3) element of an array named "calibration":

```
long varindex = nlregcVariableIndex("calibration");
double curvalue = nlregcGetArrayValue(varindex, 2, 3, 0);
```

nlregcSetArraySize() -- Set the size of a work array

void nlregcSetArraySize(long *index*, long *dimsize1*, long *dimsize2*, long *dimsize3*);

This function specifies the size for each dimension of an array work variable. The *index* argument is the index number of the array variable; it can be determined by calling nlregcVariableIndex(). The *dimsize1*, *dimsize2* and *dimsize3* arguments specify the first, second and third dimension sizes for the array. If the array has only one dimension, then only *dimsize1* is used, and you should specify 0 (zero) for *dimsize2* and *dimsize3*. Similarly, an array with two dimensions will use only *dimsize1* and *dimsize2*.

nlregcGetArraySize() -- Get the dimension size of a work array

long nlregcGetArraySize(long *VarIndex*, long *DimIndex*);

This function gets the size of a specified dimension of a work array. The nlregcCompile() function must be called to compile the program before nlregcGetArraySize() is used.

The *VarIndex* argument is the index number of the array variable; it can be determined by calling nlregcVariableIndex().

The *DimIndex* argument specifies which dimension size is to be gotten. The value must be 0, 1 or 2 corresponding to the first, second or third dimension of the array.

The function returns the size of the specified dimension.

Here is example code to get the size of the first dimension of an array named "calibration":

```
long varindex = nlregcVariableIndex("calibration");
long dimsize = nlregcGetArraySize(varindex, 0);
```

nlregcGetStatistic() -- Get a computed statistic value

double nlregcGetStatistic(long *valtype*);

This function can be called after nlregcCompute() to obtain the values of statistics computed for the function.

The *valtype* argument specifies which statistic value is to be returned by the function. The following values for *valtype* may be specified:

0 (GSX_NUMOBS) = Number of observations
1 (GSX_ITERATIONS) = Number of iterations required to obtain convergence
2 (GSX_RESIDUAL) = Sum of squared deviations (residuals)
3 (GSX_SUMDEV) = Sum of deviations
4 (GSX_MAXDEV) = Maximum deviation for any observation
5 (GSX_STDERR) = Standard error of the estimate
6 (GSX_RSQUARED) = Proportion of variance explained (R^2)
7 (GSX_ADJRSQUARED) = Adjusted coefficient of multiple determination
8 (GSX_DURBINWATSON) = Value of Durbin-Watson test for autocorrelation
9 (GSX_RUNTIME) = Number of seconds to perform analysis
10 (GSX_NLREGVERSION) = Version of NLREG being used

nlregcGetCorrelationMatrix() -- Get a matrix of correlation values

long nlregcGetCorrelationMatrix(double ****cormat*, long **rowcol*);

If the correlate statement is included in the NLREG program, NLREG will compute a correlation matrix for the input variables. If you specify the correlate statement in the program without any variable names, like this:

```
variables x,y,z;
correlate;
```

then all of the input variables (x, y and z in this example) will be included in the correlation matrix and there will be a row and a column in the matrix for each one. The order of the rows and columns corresponds to the order in which the variables are declared.

If you specify a set of variables with the correlate statement, like this:

```
variables x,y,z;
correlate x,y;
```

Then only the specified variables are included in the correlation matrix, the matrix has a row and column for each variable and the order of the rows and columns corresponds to the order that the variables are specified with the correlate statement.

Since the correlation of a variable with itself is always 1.00, the diagonal elements of the correlation matrix will always have the value 1.00.

The *cormat* argument is the address of a pointer to type double that should be declared as follows:

```
double *cormat;
```

The nlregcGetCorrelationMatrix function uses malloc() to allocate heap memory to store the correlation matrix and returns a pointer to the allocated array in the *cormat* argument. It is the responsibility of the calling program to call free() to deallocate the correlation matrix once it has finished using it.

The *rowcol* argument is a pointer to a long into which the nlregcGetCorrelationMatrix function stores the number of rows and columns in the correlation matrix. Since the matrix is square, it has the same number of rows and columns.

The returned correlation matrix is a two-dimensional array with a variable number of rows and columns (as specified by the value returned in *rowcol*). Use the following formula to compute a linear (one-dimensional) index into *cormat* corresponding to a specified row and column:

index = (*row***rowcol*) + *column*;

For example, the following code would obtain the value from the correlation matrix corresponding to row 2 column 3 (note that indexes in C are 0 based):

```
long row,column,index;
long rowcol;
double *cormat,covalue;
…
nlregcGetCorrelationMatrix(&cormat,&rowcol);
row = 2;
column = 3;
index = (row*rowcol) + column;
covalue = cormat[index];
```

The value returned by the function itself indicates whether the correlation matrix was successfully retrieved. The following values can be returned:

0 (RVSDA_OK) = Success (the correlation matrix was computed)
11 (RVSDA_NODATA) = The correlation matrix was not computed. Either you did not put a correlation statement in the NLREG program, or this is a minimization analysis rather than a regression analysis.

Here is a program fragment that obtains the correlation matrix and writes its values to a file:

```
FILE *f;
long value,row,col,rowcol,index;
double *cormat;
…
value = nlregcGetCorrelationMatrix(&cormat, &rowcol);
if (value == RVSDA_OK) {
   f = fopen("correlation.txt","wt");
   for (row=0; row<rowcol; row++) {
      for (col=0; col<rowcol; col++) {
         index = (row * rowcol) + col;
         fprintf(f,"  %2.6lf",cormat[index]);
      }
      fprintf(f,"\n");
   }
   fclose(f);
   free(cormat);
}
```

nlregcGetCovarianceMatrix() -- Get a matrix of covariance values

long nlregcGetCovarianceMatrix(double ***covmat*, long **rowcol*);

If the covariance statement is included in the NLREG program, NLREG will compute a covariance matrix for the computed parameter values. There will be a row and a column in the matrix for each parameter. The order of the rows and columns corresponds to the order in which the parameters are declared.

For example, if the NLREG program contains the following declarations:

```
parameters a,b,c;
covariance;
```

Then the covariance matrix will have three rows and three columns. The entry for the first row and first column will correspond to the ‘a’ parameter.

The *covmat* argument is the address of a pointer to type double that should be declared as follows:

```
double *covmat;
```

The nlregcGetCovarianceMatrix function uses malloc() to allocate heap memory to store the covariance matrix and returns a pointer to the allocated array in the *covmat* argument. It is the responsibility of the calling program to call free() to deallocate the covariance matrix once it has finished using it.

The *rowcol* argument is a pointer to a long into which the nlregcGetCovarianceMatrix function stores the number of rows and columns in the covariance matrix. Since the matrix is square, it has the same number of rows and columns.

The returned covariance matrix is a two-dimensional array with a variable number of rows and columns (as specified by the value returned in *rowcol*). Use the following formula to compute a linear (one-dimensional) index into *covmat* corresponding to a specified row and column:

index = (*row***rowcol*) + *column*;

For example, the following code would obtain the value from the covariance matrix corresponding to row 2 column 3 (note that indexes in C are 0 based):

```
long row,column,index;
long rowcol;
double *covmat,covalue;
…
nlregcGetCovarianceMatrix(&covmat,&rowcol);
row = 2;
column = 3;
index = (row*rowcol) + column;
covalue = covmat[index];
```

The value returned by the function itself indicates whether the covariance matrix was successfully retrieved. The following values can be returned:

0 (RVSDA_OK) = Success (the covariance matrix was computed)
11 (RVSDA_NODATA) = The covariance matrix was not computed for one of these reasons: (1) you did not put a covariance statement in the NLREG program; (2) this is a minimization analysis rather than a regression analysis; (3) a solution was not obtained for the analysis (e.g., the solution did not converge); (4) the constrain statement was used to constrain the computed values of the parameters.

Here is a program fragment that obtains the covariance matrix and writes its values to a file:

```
FILE *f;
long value,row,col,rowcol,index;
double *covmat;
…
value = nlregcGetCovarianceMatrix(&covmat, &rowcol);
if (value == RVSDA_OK) {
   f = fopen("covariance.txt","wt");
   for (row=0; row<rowcol; row++) {
      for (col=0; col<rowcol; col++) {
         index = (row * rowcol) + col;
         fprintf(f,"  %2.6lf",covmat[index]);
      }
      fprintf(f,"\n");
   }
   fclose(f);
   free(covmat);
}
```

nlregcEvaluateFunction() -- Evaluate the function with a set of data values

double nlregcEvaluateFunction(double *_inputvalues_);

The nlregcEvaluateFunction() function evaluates the current NLREG program function with a specified set of input variable values and the computed parameter values. It must be called after nlregcCompute() has successfully completed the analysis. What nlregcEvaluateFunction() does is "plug in" a set of input variable values and determine what the corresponding value of the function is using the parameter values computed by the regression analysis.

The *inputvalues* argument is a one-dimensional array of values to be used for the input variables when the function is evaluated. There must be one value in the input array for each input variable. Note: One of the input variables is the dependent variable (i.e., the variable on the left side of the equal sign in the function). Since the purpose of the nlregcEvaluateFunction() function is to compute the value of the dependent variable, its value in the *inputvalues* array is not used. However, you still must reserve room for the dependent variable value in *inputvalues* (you can assign it a value of 0.0 or anything else).

Here is an example code fragment that evaluates the function at a number of points:

```
double funval;
double inputvalues[2];
long i;
long result;
…
result = nlregcCompute()
for (i=0; i<100; i++) {
        inputvalues(0) = (double)i
        inputvalues(1) = 0.0
        funval = nlregcEvaluateFunction((double *)inputvalues)
        …
}
```

nlregcSetExternalFunction() -- Declare external function evaluation routines

```
double nlregcSetExternalFunction(
    void *pUserData,
    int (*pExternalEvalFunction)(void *pUserData, long IterNum, long NumVar, long NumParam,
        double *DataVector, double *ParamValues, double *CalculatedValue),
    int (*pExternalResidualFunction)(void *pUserData, long IterNum, long NumVar, long NumParam,
        long NumObs, double *DataArray, double *ParamValues, double *CalcResiduals),
    void (*pInitializationFunction)(void *pUserData, long NumVar, long NumParam,
        long NumObs, double *DataArray),
    void (*pCompletionFunction)(void *pUserData, long StatusCode, long NumVar, long NumParam,
        long NumObs, double *DataArray, double *FinalParamValues)
);
```

For some nonlinear regression analyses, it may be desirable to compute the predicted value of the function being fitted by using an external function rather than using the NLREG programming language. This could be required if the function is very complex or if it requires computational techniques not available in the NLREG language such as solving differential equations.

The nlregcSetExternalFunction() function is called to declare a set of external functions that are called by NLREG during the analysis to compute the predicted value of the dependent variable and the residual values that are to be minimized.

When using an external function, you must place the EXTERNALFUNCTION statement in the NLREG program in place of the usual FUNCTION statement. For example, here is a NLREG program that specifies that the function evaluation is external:

```
Variables X, Y;
Parameters m, c;
ExternalFunction;
Data;
```

If you use a FUNCTION statement in the NLREG program rather than EXTERNALFUNCTION, then the function will be computed internally by the NLREG

program, and any external functions declared by calling nlregcSetExternalFunction() will not be called.

The nlregcSetExternalFunction() procedure is called with one pointer variable argument and pointers to four call-back functions that NLREG will call as the analysis is performed. The *pInitializationFunction* and *pCompletionFunction* arguments are optional. You may specify 0 (zero) for their addresses if you don't need them.

pUserData is an argument you can specify to pass data to the call-back functions. NLREG stores the value you specify for *pUserData* and passes it as the first parameter to each of the call-back functions.

int (*_pExternalEvalFunction_)(void *_pUserData_, long _IterNum_,
long _NumVar_, long _NumParam_,
double *_DataVector_, double *_ParamValues_, double *_CalculatedValue_)

The *pExternalEvalFunction* argument is the address of a function that NLREG will call to compute the value of the dependent variable given a set of data values and a set of parameter values. This function must be defined in the application program that is calling NLREG, and it must have 7 parameters:

pUserData – (input) NLREG passes in a (void *) pointer to a data item that is specified as the first argument to the nlregcSetExternalFunction() function. You can use this argument to reference a data item in your program.

IterNum – (input) NLREG passes in a count of the number of iterations that have been performed by the convergence process. The first time the function is called, *IterNum* will have a value of 0.

NumVar – (input) NLREG passes in a count of the number of input variables (i.e., the number of independent and dependent variables).

NumParam – (input) NLREG passes in a count of the number of parameters whose values are being adjusted to make the function fit the data.

DataVector – (input) This is a pointer to a vector (one-dimensional array) that has a value for each variable. There are *NumVar* values in *DataVector*. The values are passes as type double. You can use the nlregcVariableIndex() function to convert the name of a variable to an index into the *DataVector* vector.

ParamValues – (input) This is a pointer to a vector (one-dimensional array) that has the current values of the parameters. There are *NumParam* values in the *ParamValues* array. The values are passes as type double. You can use the nlregcParameterIndex() function to convert the name of a parameter to an index into the *ParamValues* vector.

CalculatedValue – (output) This is a pointer to a memory location of type double where the computed value of the dependent variable is to be stored. The evaluation function

must use the values of the independent variables provided in *DataVector* and the values of the parameters in *ParamValues* to calculate the predicted value of the dependent variable and then store that value into the cell pointed to by *CalculatedValue*.

The evaluation function is of type int, and it must return one of the following values when it exits:

+1 ==> The function has converged. Stop the analysis.
 0 ==> Continue normal processing.
-1 ==> The predicted value could not be computed with these parameter values.

Note: If it is not possible to evaluate the function, specify 0 (zero) as the value of the ExternalEvalFunction parameter. In this case, NLREG will use only the residual values computed by ExternalResidualFunction. The calculated parameter values will be correct, but statistics such as R^2 will not be calculated.

int (*_pExternalResidualFunction_)(void *_pUserData_, long _IterNum_, long _NumVar_, long _NumParam_, long NumObs, double *_DataArray_, double *_ParamValues_, double *_CalculatedResiduals_)

The *pExternalResidualFunction* argument is the address of a function that NLREG will call to compute the residual values that are to be minimized by the regression process. This function must be defined in the application program that is calling NLREG, and it must have 7 parameters:

pUserData – (input) NLREG passes in a (void *) pointer to a data item that is specified as the first argument to the nlregcSetExternalFunction() function. You can use this argument to reference a data item in your program.

IterNum – (input) NLREG passes in a count of the number of iterations that have been performed by the convergence process. The first time the function is called, *IterNum* will have a value of 0.

NumVar – (input) NLREG passes in a count of the number of input variables (i.e., the number of independent and dependent variables).

NumParam – (input) NLREG passes in a count of the number of parameters whose values are being adjusted to make the function fit the data.

NumObs – (input) NLREG passes in a count of the number of data rows (observations) to which the function is being fitted.

DataArray – (input) This is a pointer to a two-dimensional array that has one row for each data record and one column for each variable. Hence, its size is [*NumObs*,*NumVar*]. The values are passes as type double. You can use the nlregcVariableIndex() function to convert the name of a variable to a column index into the *DataArray* array. Values in the array are stored by rows, so all of the variable values for one row occur in sequence. To

access the value of a variable whose index is VarColIndex on row RowNum in the array, use code similar to this:

```
VarColIndex = nlregcVariableIndex("X1");
ArrayIndex = (RowNum * NumVar) + VarColIndex;
DataValue = DataArray[ArrayIndex];
```

ParamValues – (input) This is a pointer to a vector (one-dimensional array) that has the current values of the parameters. There are *NumParam* values in the *ParamValues* array. The values are passes as type double. You can use the nlregcParameterIndex() function to convert the name of a parameter to an index into the *ParamValues* vector.

CalculatedResiduals – (output) This is a pointer to a vector (one dimensional array) of type double where the computed residual values are to be stored. The residual function must use the values of the data records provided in *DataArray* and the values of the parameters in *ParamValues* to calculate the residual values and then store those values into the *CalculatedResiduals* vector. A residual value must be calculated and stored in *CalculatedResiduals* for each data row, so there are *NumObs* entries in the *CalculatedResiduals* vector.

Normally, a residual value is calculated by subtracting the computed (predicted) value of the dependent variable from the actual value of the dependent variable stored in *DataArray*. Note that you should *not* square the residual values; NLREG's computation routine squares the residual values and adjusts the parameter values to minimize the sum of the squared residual values. If data rows have different weights, you can weight the residual values by multiplying them by the row weights.

If the goal of the analysis is to minimize the value of a function rather than fit the function to a set of dependent variable values, then the "residual" values are just the computed value of the function itself for each data row.

The residual calculation function is of type int, and it must return one of the following values when it exits:

+1 ==> The function has converged. Stop the analysis.
 0 ==> Continue normal processing.
-1 ==> The predicted value could not be computed with these parameter values.

void (pInitializationFunction*)(void **pUserData*, long *NumVar*,**
long *NumParam*, long *NumObs*, double **DataArray*)

The *pInitializationFunction* argument is the address of a function that NLREG will call once at the beginning of each analysis. You can place calls to nlregcVariableIndex() and nlregcParameterIndex() in the initialization function to avoid the overhead of calling these functions for each iteration. If you don't want NLREG to call an initialization function, specify 0 (zero) as the address of the function. If an initialization function is used, the function must have 5 arguments:

pUserData – (input) NLREG passes in a (void *) pointer to a data item that is specified as the first argument to the nlregcSetExternalFunction() function. You can use this argument to reference a data item in your program.

NumVar – (input) NLREG passes in a count of the number of input variables (i.e., the number of independent and dependent variables).

NumParam – (input) NLREG passes in a count of the number of parameters whose values are being adjusted to make the function fit the data.

NumObs – (input) NLREG passes in a count of the number of data rows (observations) to which the function is being fitted.

DataArray – (input) This is a pointer to a two-dimensional array that has one row for each data record and one column for each variable. Hence its size is [*NumObs*,*NumVar*]. The values are passes as type double. You can use the nlregcVariableIndex() function to convert the name of a variable to a column index into the *DataArray* array. Values in the array are stored by rows, so all of the variable values for one row occur in sequence. To access the value of a variable whose index is VarColIndex on row RowNum in the array, use code similar to this:

```
VarColIndex = nlregcVariableIndex("X1");
ArrayIndex = (RowNum * NumVar) + VarColIndex;
DataValue = DataArray[ArrayIndex];
```

void (pCompletionFunction*)(void **pUserData*, long *StatusCode*,**
long *NumVar*, long *NumParam*, long *NumObs*, double **DataArray,*
double **FinalParamValues*)

The *pCompletionFunction* argument is the address of a function that NLREG will call once at the end of the analysis. If you don't want NLREG to call a completion function, specify 0 (zero) as the address of the function. If a completion function is used, the function must have 7 arguments:

pUserData – (input) NLREG passes in a (void *) pointer to a data item that is specified as the first argument to the nlregcSetExternalFunction() function. You can use this argument to reference a data item in your program.

StatusCode – (input) is a value that indicates the final result of the analysis. See the description of the nlregcCompute() function for a list of the status code values.

NumVar – (input) NLREG passes in a count of the number of input variables (i.e., the number of independent and dependent variables).

NumParam – (input) NLREG passes in a count of the number of parameters whose values are being adjusted to make the function fit the data.

NumObs – (input) NLREG passes in a count of the number of data rows (observations) to which the function is being fitted.

DataArray – (input) This is a pointer to a two-dimensional array that has one row for each data record and one column for each variable. Hence its size is [*NumObs*,*NumVar*]. The values are passes as type double. You can use the nlregcVariableIndex() function to convert the name of a variable to a column index into the *DataArray* array. Values in the array are stored by rows, so all of the variable values for one row occur in sequence. To access the value of a variable whose index is VarColIndex on row RowNum in the array, use code similar to this:

```
VarColIndex = nlregcVariableIndex("X1");
ArrayIndex = (RowNum * NumVar) + VarColIndex;
DataValue = DataArray[ArrayIndex];
```

FinalParamValues – (input) This is a vector (one-dimensional array) containing the final parameter values computed by the regression analysis. There are *NumParam* values in the *FinalParamValues* array. The values are passes as type double. You can use the nlregcParameterIndex() function to convert the name of a parameter to an index into the *FinalParamValues* vector.

Here is an example program that uses external functions to fit a function,
Y = m*X + c to a set of data values.

```
  << Part of main program >>

/*
 *  Local function prototypes.
 */
int EvaluationFunction(void *pUserData, long IterNum, long NumVar,
         long NumParam, double *DataVector, double *ParamValues,
         double *PredictedValue);

int ResidualFunction(void *pUserData, long IterNum, long NumVar,
         long NumParam, long NumObs, double *DataArray,
         double *ParamValues, double *CalcResiduals);

void InitializationFunction(void *pUserData, long NumVar,
         long NumParam, long NumObs, double *DataArray);

void CompletionFunction(void *pUserData, long StatusCode,
         long NumVar, long NumParam, long NumObs,
         double *DataArray, double *ParamValues);

/*
 *  Global variables.
 */
long IndexX = 0;   /* Index of X variable */
long IndexY = 0;   /* Index of Y variable */
long IndexM = 0;   /* Index of M parameter */
long IndexC = 0;   /* Index of C parameter */
long MyGlobalData = 1; /* External data to be passed */

/*
 *  Set the call-back functions.
 */
      nlregcSetExternalFunction((void *)&MyGlobalData,
                          EvaluationFunction,ResidualFunction,
                          InitializationFunction, CompletionFunction,
                          (void *)&MyGlobalData);
/*
 *  Do the analysis.
 */
      status = nlregcCompute();
```

```
/*---------------------------------------------------------------------
 *  This is a call-back function that is called by NLREG.dll at the
 *  beginning of an analysis.
 */
void InitializationFunction(void *pUserData, long NumVar,
                  long NumParam, long NumObs, double *DataArray)
{

/*
 *  Get the index numbers of the input variables which are X and Y
 *  and store them in global variables.
 */
      IndexX = nlregcVariableIndex("x");
      IndexY = nlregcVariableIndex("y");
/*
 *  Get the index numbers of the parameters which are M and C
 *  and store them in global variables.
 */
      IndexM = nlregcParameterIndex("m");
      IndexC = nlregcParameterIndex("c");
/*
 *  Finished
 */
      return;
}

/*---------------------------------------------------------------------
 *  This is a call-back function that is called by NLREG.dll to
 *  compute the predicted dependent variable value for the next
 *  step of the analysis.
 *
 *  Function return values:
 *  +1 ==> The function has converged. Stop the analysis.
 *   0 ==> Continue normal processing.
 *  -1 ==> The predicted value could not be computed.
 */
int EvaluationFunction(void *pUserData, long IterNum,
                   long NumVar, long NumParam,
                   double *DataVector, double *ParamValues,
                   double *PredictedValue)
{
      double m,c,x;

/*
 *  Using the variable and parameter indexes we stored from the
 *  initilization routine,  compute the value of the dependent
 *  variable using the function:
 *      Y = m*X + c
 */
/*
 *  Get the current parameter values, m and c.
 */
      m = ParamValues[IndexM];
      c = ParamValues[IndexC];
/*
 *  Get the value of the independent variable, X.
 */
```

```
      x = DataVector[IndexX];
/*
 *  Compute the value of the function.
 */
      *PredictedValue = m * x + c;
/*
 *  Finished.  Return code 0 to continue the analysis.
 */
      return(0);
}

/*-----------------------------------------------------------------
 *  This is a call-back function that is called by NLREG.dll to
 *  compute the residual values for the next step of the analysis.
 *
 *  Function return values:
 *  +1 ==> The function has converged.  Stop the analysis.
 *   0 ==> Continue normal processing.
 *  -1 ==> The residuals could not be computed.
 */
int ResidualFunction(void *pUserData, long IterNum,
                  long NumVar, long NumParam,
                  long NumObs, double *DataArray,
                  double *ParamValues, double *CalcResiduals)
{
      long ix,iy,obs;
      double m,c,x,y,PredictedY;

/*
 *  Using the variable and parameter indexes we stored from the
 *  initilization routine,  compute the residuals using the function:
 *      Y = m*X + c
 */
/*
 *  Get the current parameter values, m and c.
 */
      m = ParamValues[IndexM];
      c = ParamValues[IndexC];
/*
 *  Compute and store residual values for each observation.
 */
      for (obs=0; obs<NumObs; obs++) {
/*
 *  Compute a linear index into the data array to access the
 *  value of X and Y for a particular observation.
 */
            ix = (obs * NumVar) + IndexX;
            iy = (obs * NumVar) + IndexY;
/*
 *  Get the X and Y values for this observation.
 */
            x = DataArray[ix];
            y = DataArray[iy];
/*
 *  Compute the predicted value of the function for this observation.
 */
            PredictedY = m * x + c;
```

```
/*
 *  Compute and store the residual for this observation.
 */
            CalcResiduals[obs] = PredictedY - y;
      }
/*
 *  Finished
 */
      return(0);
}

/*---------------------------------------------------------------------
 *  This is a call-back function that is called by NLREG.dll when
 *  the analysis is completed.
 */
void CompletionFunction(void *pUserData, long StatusCode,
                        long NumVar, long NumParam,
                        long NumObs, double *DataArray,
                        double *ParamValues)
{
      double m,c;

/*
 *  Get the final parameter values.
 */
      m = ParamValues[IndexM];
      c = ParamValues[IndexC];

/*
 *  Finished
 */
      return;
}
```

Index

P

R

S

V