NLREG

Nonlinear Regression Analysis Program

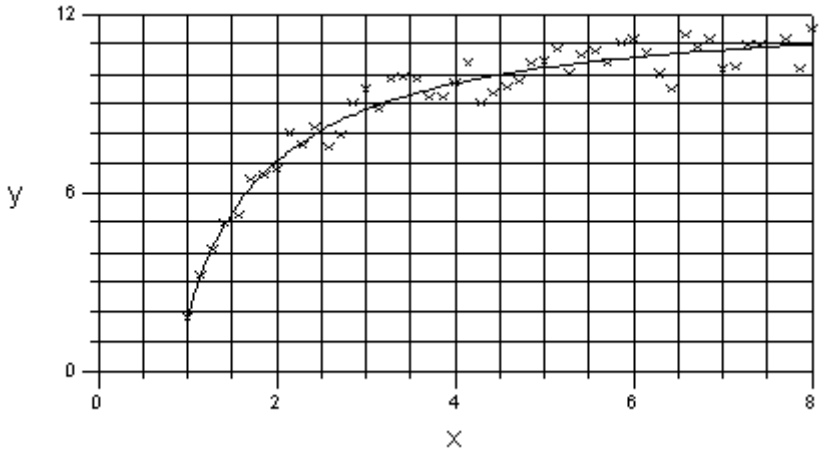
Phillip H. Sherrod

Copyright © 1991 – 2010
All rights reserved

www.nlreg.com

NLREG performs statistical regression analysis to estimate the values of parameters for linear, multivariate, polynomial, logistic, exponential, and general nonlinear functions. The regression analysis determines the values of the parameters that cause the function to best fit the observed data that you provide. This process is also called “curve fitting.”

NLREG allows you to specify the function whose parameters are being estimated using ordinary algebraic notation. In addition to determining the parameter estimates, NLREG can be directed to generate an output file with predicted values and residuals. It can also plot the data observations and the computed function. Although designed for regression analysis, NLREG can also be used to find the root (zero point) or minimum absolute value of a nonlinear expression. NLREG is in use at hundreds of engineering and research centers around the world.



Contents

Introduction

Introduction to Regression Analysis

The goal of regression analysis is to determine the values of parameters for a function that cause the function to best fit a set of data observations that you provide. In *linear* regression, the function is a linear (straight-line) equation. For example, if we assume the value of an automobile decreases by a constant amount each year after its purchase, and for each mile it is driven, the following linear function would predict its value (the dependent variable on the left side of the equal sign) as a function of the two independent variables which are age and miles:

```
value = price + depage*age + depmiles*miles
```

where *value*, the dependent variable, is the value of the car, *age* is the age of the car, and *miles* is the number of miles that the car has been driven. The regression analysis performed by NLREG will determine the best values of the three parameters, *price*, the estimated value when age is 0 (i.e., when the car was new), *depage*, the depreciation that takes place each year, and *depmiles*, the depreciation for each mile driven. The values of *depage* and *depmiles* will be negative because the car loses value as age and miles increase.

For an analysis such as this car depreciation example, you must provide a data file containing the values of the dependent and independent variables for a set of observations. In this example each observation data record would contain three numbers: value, age, and miles, collected from used car ads for the same model car. The more observations you provide, the more accurate will be the estimate of the parameters. The NLREG statements to perform this regression are shown below:

```
Variables value,age,miles;
Parameters price,depage,depmiles;
Function value = price + depage*age + depmiles*miles;
Data;
{data values go here}
```

Once the values of the parameters are determined by NLREG, you can use the formula to predict the value of a car based on its age and miles driven. For example, if NLREG computed a value of 16000 for *price*, -1000 for *depage*, and -0.15 for *depmiles*, then the function

```
value = 16000 – 1000*age – 0.15*miles
```

could be used to estimate the value of a car with a known age and number of miles.

If a perfect fit existed between the function and the actual data, the actual value of each car in your data file would exactly equal the predicted value. Typically, however, this is not the case, and the difference between the actual value of the dependent variable and its predicted value for a particular observation is the error of the estimate which is known as the "*deviation*" or "*residual*". The goal of regression analysis is to determine the values of the parameters that minimize the sum of the squared residual values for the set of observations. This is known as a "least squares" regression fit.

Introduction to NLREG

NLREG is a very powerful regression analysis program. Using it you can perform multivariate, linear, polynomial, exponential, logistic, and general nonlinear regression. What this means is that you specify the form of the function to be fitted to the data, and the function may include nonlinear terms such as variables raised to powers and library functions such as log, exponential, sine, etc. For complex analyses, NLREG allows you to specify function models using conditional statements (IF, ELSE), looping (FOR, DO, WHILE), work variables, and arrays. NLREG uses a state-of-the-art regression algorithm that works as well, or better, than any you are likely to find in any other commercial statistical packages.

As an example of nonlinear regression, consider another depreciation problem. The value of a used airplane decreases for each year of its age. Assuming the value of a plane falls by the same amount each year, a linear function relating value to age is:

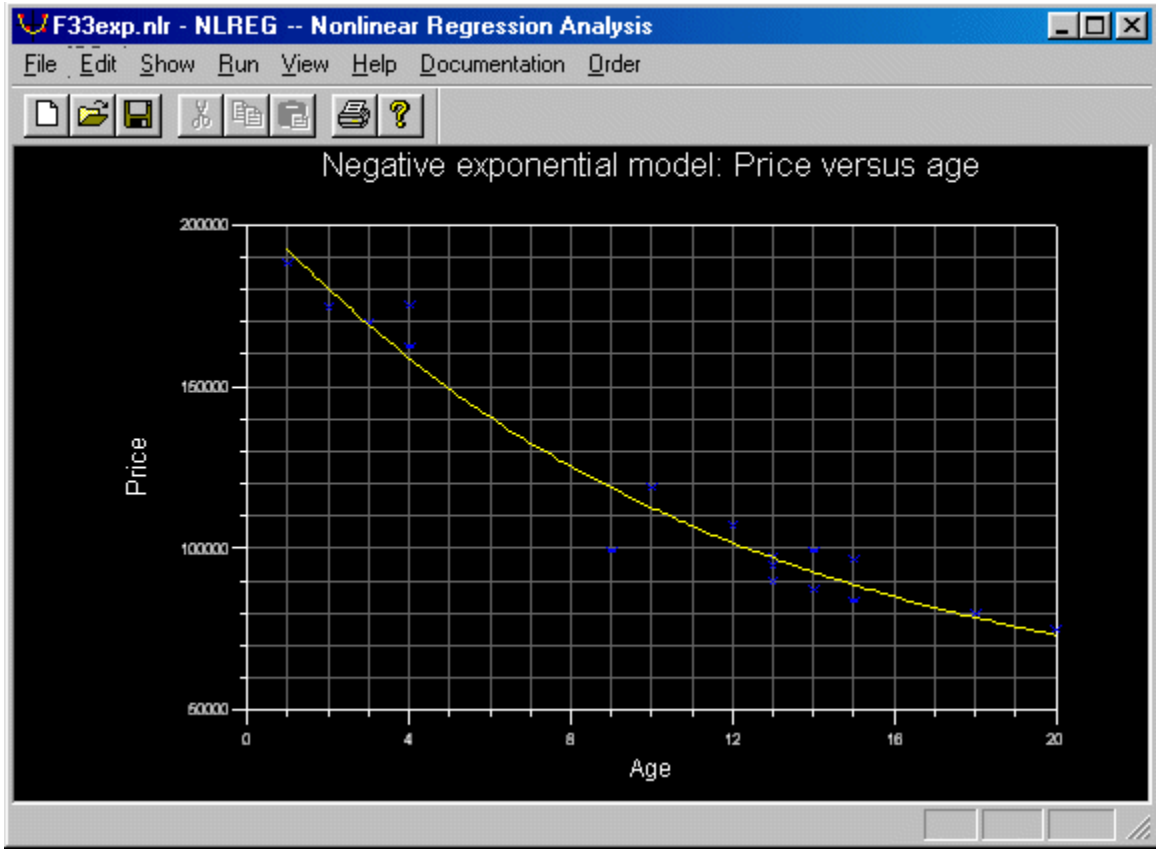
```
value = p0 + p1*Age
```

Where `p0` and `p1` are the parameters whose values are to be determined.

However, it is a well-known fact that planes (and automobiles) lose more value the first year than the second, and more the second than the third, etc. This means that a linear (straight-line) function cannot accurately model this situation. A better, nonlinear, function is:

```
value = p0 + p1*exp(-p2*Age)
```

Where the 'exp' function is the value of *e* (2.7182818...) raised to a power. This type of function is known as "negative exponential" and is appropriate for modeling a value whose rate of decrease is proportional to the difference between the value and some base value. The F33YEAR.NLR example program file fits a linear function to the value of used airplanes. The F33EXP.NLR example fits a negative exponential function to the same data. Run both examples and compare the fitted functions. See F33.NLR for an example of multiple regression using three independent variables. Here is a plot produced by the f33exp.nlr function which shows a negative exponential function fitted to a set of data values:



Much of the convenience of NLREG comes from the fact that you can enter complicated functions using ordinary algebraic notation. Examples of functions that can be handled with NLREG include:

```
Linear:         Y = p0 + p1*X

Quadratic:      Y = p0 + p1*X + p2*X^2

Multivariate:   Y = p0 + p1*X + p2*Z + p3*X*Z

Exponential:    Y = p0 + p1*exp(X)

Periodic:       Y = p0 + p1*sin(p2*X)

Misc:           Y = p0 + p1*Y + p2*exp(Y) + p3*sin(Z)
```

In other words, the function is a general expression involving one dependent variable (on the left of the equal sign), one or more independent variables, and one or more parameters whose values are to be estimated.

Because of its generality, NLREG can perform all of the regressions handled by ordinary linear or multivariate regression programs as well as nonlinear regression.

Note: Some other regression programs claim to perform nonlinear regression but actually do it by transforming the values of the variables such that the function is converted to linear form. They then perform a linear regression on the transformed function. This technique has a major flaw: it

determines the values of the parameters that minimize the squared residuals for the transformed, linearized function rather than the original function. This is different than minimizing the squared residuals for the actual function and the estimated values of the parameters may not produce the best fit of the original function to the data. NLREG uses a true nonlinear regression technique that minimizes the squared residuals for the actual function. Also, NLREG can handle functions that cannot be transformed to a linear form.

Versions of NLREG

There are four versions of NLREG: Standard GUI, Advanced GUI, DLL library and COM object. The features and limitations of each version are described in the following table.

	Standard	**Advanced**	**DLL Library**	**COM Object**
Maximum variables	5	2000	2000	2000
Maximum parameters	5	2000	2000	2000
Maximum data records	500	(unlimited)[1]	(unlimited)[1]	(unlimited)[1]
Line function plots	Yes	Yes	No	No
Line residual plots	Yes	Yes	No	No
Normal prob. plots	Yes	Yes	No	No
3D surface plots	No	Yes	No	No
3D residual plots	No	Yes	No	No
Surface contour plots	No	Yes	No	No
Call from C++	No	No	Yes	Possible[4]
Call from Visual Basic	No	No	Possible[5]	Yes
Call from ASP	No	No	No	Yes

Notes:
[1] The maximum data records depend on available memory space.
[4] The COM object version of NLREG can be called from Visual C++ programs, but you are responsible for moving data to and from Variant objects. The DLL library version is easier to use because natural C data types can be passed and received.
[5] Visual Basic is designed to call COM objects, so the COM object version is recommended over the DLL library for this type of application.

The **demonstration version** of NLREG may be used for up to 30 days to evaluate the product. If you continue to use NLREG beyond the evaluation period, or if you wish to use it for commercial applications, you must order the commercial version of NLREG. The demonstration version of NLREG has the following restrictions that are not present in the commercial version:

- A maximum of 50 data observations may be processed.
- A maximum of 5 variables may be used.
- A maximum of 5 parameters may be fitted.
- It will operate for only 30 days.

Installing NLREG

NLREG is distributed as a self-extracting executable file named **nlrsetup.exe**. To install NLREG, execute nlrsetup.exe, and the setup program that will guide you through the installation.

The installation will create a directory with the executable program (NLREG.EXE) and a set of example analysis files (they have the extension ".NLR"). You may want to create on your desktop a shortcut to the NLREG.EXE program. A full manual for NLREG in Microsoft Word format is also installed; it is named **nlreg.doc**.

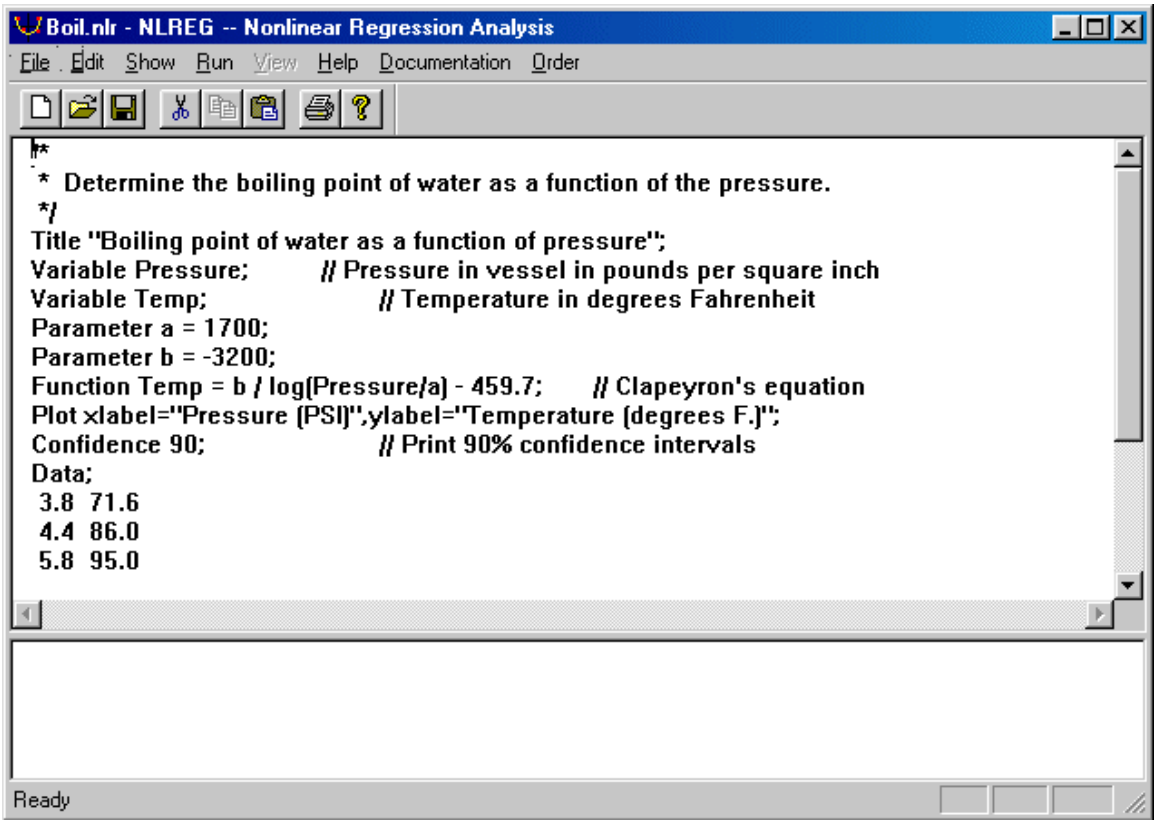
Using NLREG

Running NLREG

Once NLREG has been installed, you can start it in either of two ways:

Double click the icon representing the executable program (NLREG.EXE).

Double click a NLREG analysis file which has the extension ".NLR". If you start the NLREG.EXE program, it will begin with an empty workspace. You can then either enter a new analysis program or click the open toolbar item to open an existing program file. If you start NLREG by double clicking a program file, the program file will be loaded in the workspace when NLREG begins executing. The sections that follow describe the statements that you can use in a program file.



Performing an Analysis

Once an analysis has been specified either by typing it into the workspace or by opening an existing program file, you can execute it by clicking the "Run" button on the main menu.

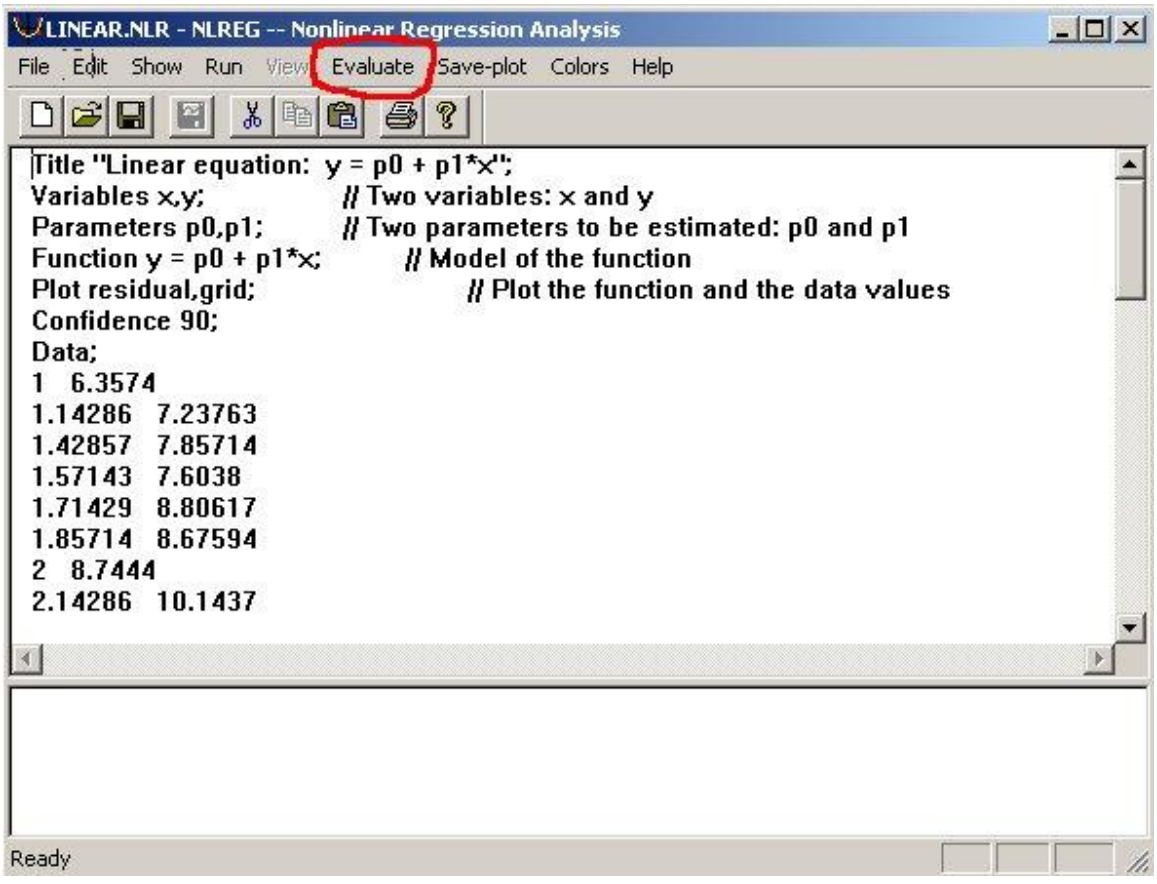
Viewing the Results and Plots

When the analysis completes, the results of the analysis will be displayed. The "Run" menu button will be grayed out and the "View" menu button will be enabled. You can click "View" and select which view you wish to display. If the analysis program specified one or more plots, you can display them by selecting the "Plot" option from the View list. Select "Program source" from the View list to display and edit the current program file. Anytime the program file is edited, the "Run" menu button will be re-enabled and the "View" button will dim.

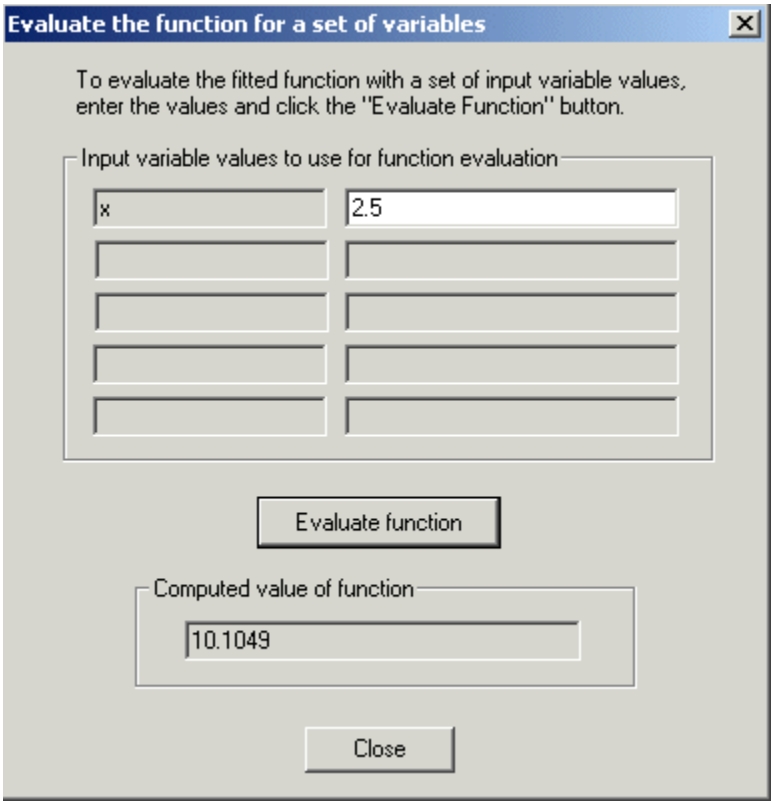
At this point, I suggest you pause in your reading and try running a NLREG example to get a feel for how it works. Several example files with the extension ".NLR" are provided with the distribution. LINEAR.NLR is a good one to start with followed by AIDS.NLR and TREND.NLR.

Evaluating the Function at Specific Points

Once NLREG has fitted a function to a set of data points, you can use the "Evaluate" function on the menu bar to compute the value of the fitted function at a specified data point whose values you provide.



When you click “Evaluate” a menu will appear where you can type the values of the dependent variables for which you want the function evaluated. Enter the values and then click “Evaluate Function”. The computed value of the function will be shown in the lower window.



Statement Syntax

The syntax for NLREG statements follows the style of the C programming language. Each statement must end with a semicolon character, you may place more than one statement on a line, and text strings and file names are enclosed in quote marks. NLREG has the same arithmetic and logical operators as C, and brace characters ('{' and '}') are used to group statements.

The following keywords are reserved by NLREG and may not be used as the names of variables or parameters: angletype, basecontour, break, confidence, connect, connect2, constant, constrain, continue, contourplot, correlate, covariance, cplot, data, datacount, dataskip, do, domain, double, else, expresidual, filldensity, for, function, grid, if, iteration, iterations, legend, model, nobasecontour, noecho, nofun, nofunction, nogrid, nolegend, nopause, notitle, noxlabel, noylabel, nozlabel, nplot, output, parameter, parameters, plot, poutput, presolution, print, register, residual, rplot, splot, stop, sweep, title, to, tolerance, values, var, variable, variables, viewpitch, viewroll, viewyaw, while, width, xlabel, xvar, xvar2, ydomain, ylabel, yvar, yvar2, zlabel, zvar.

Variables and Parameters

NLREG allows you to use four types of variables: input, computed, system, and constant.

Input variables are variables whose values are set by observations read from your data file. Input variables are declared using the VARIABLES statement. For each step of the analysis, NLREG cycles through each data observation and executes the statements in your NLREG program file. Each execution begins by setting the input variables to the values for a specific observation. If you want to transform input variables you should assign the transformed values to computed variables rather than modifying the values of input variables.

Computed variables are variables whose values are computed and assigned in the NLREG program file. They are declared using the DOUBLE statement. You can use computed variables to hold intermediate results of calculations or to hold transformed values of input variables. Computed variables may hold single values (scalars), lists of items (vectors), or arrays of values. You may assign initial values to computed variables when they are declared with the DOUBLE statement and you may modify the values during the course of execution.

System variables hold values calculated by NLREG during the execution of your program. There are three system variables:

`PREDICTED` -- the value of the dependent variable computed by executing your function using the current parameter and input variable values.

`RESIDUAL` -- the difference between predicted and actual value of the dependent variable of the function.

`OBS` -- the number of the observation record from your data file that is currently being executed, the first record is number 1. The PREDICTED and RESIDUAL variables only have defined values after the FUNCTION statement has been executed. They are primarily useful in SPLOT and OUTPUT statements.

Symbolic constants are declared using the CONSTANT statement and assigned values with their declaration. The values may *not* be altered after the declaration. Symbolic constants may not appear on the left side of an equal sign; otherwise, they may be used wherever variables or constants may be used.

There are two other designations for variables that should be mentioned. An "independent" variable is a variable that appears in the FUNCTION statement on the right side of the equal sign. You may have many independent variables. The "dependent" variable appears in the FUNCTION statement on the left side of the equal sign. There will be only one dependent variable. If you use multiple FUNCTION statements, the same dependent variable must be used in each one. Input and computed variables may be used as independent and dependent variables.

In addition to variables, you will use the PARAMETERS statement to declare parameters whose values are to be calculated by NLREG. Parameters are used like variables but there values are neither read from the input file nor computed by statements in your program, rather their value is determined by NLREG so as to cause the function to best fit your data observations.

Plots

The standard version of NLREG allows you to generate four types of plots as part of the analysis. The Advanced version includes three additional 3D plot types. Any combination of these plot types may be requested and you can generate multiple scatter plots by including more than one SPLOT statement. With the registered version of NLREG you can cause hard copy images of the plots to be printed.

The Advanced version of NLREG can produce 3D surface and contour plots of functions with two independent variables of the form:

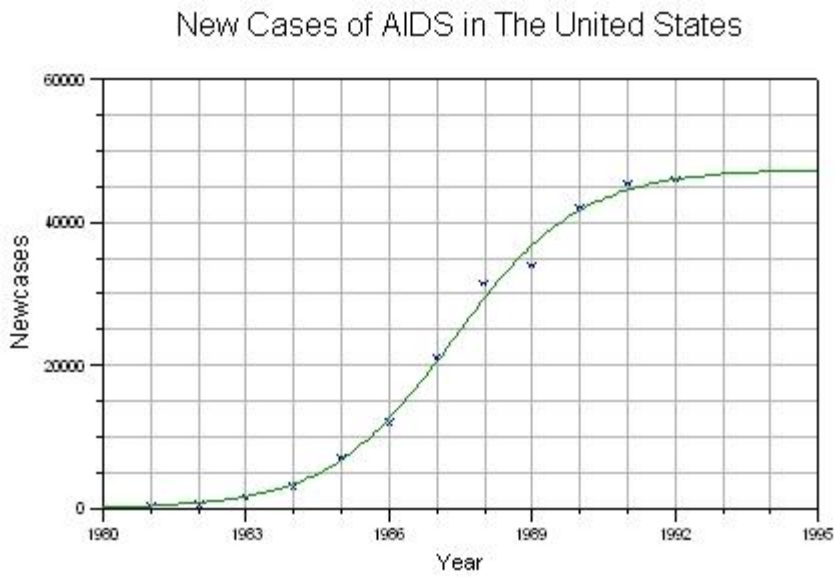
$$z = f(x,y)$$

The plane formed by the X and Y axes is the base of the displayed function. The Z value determines the height above this plane. When you request a surface plot, NLREG computes the value of the function (Z) over the X and Y domains using small X and Y steps, and plots the corresponding 3D surface. NLREG performs hidden-point elimination so you see the visible (front) side of the surface. Color bands are used to denote ranges of the Z variable values.

The Advanced version of NLREG also can generate contour plots for functions with two independent variables. A contour plot shows the plane formed by the X and Y axes. The values of the Z variable are represented by color bands on this plane.

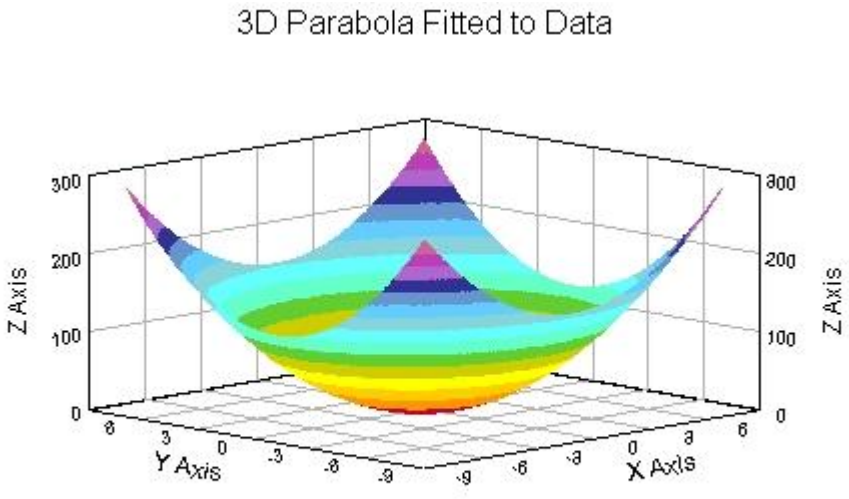
The plots are requested using the following statements:

PLOT -- Generate a plot showing the computed value of the function superimposed on a scatter plot of the input data values. NLREG evaluates the function using the computed values of the parameters at many data points over the domain and plots it as a smooth curve. Because NLREG must evaluate the function at points between the observations the PLOT statement can only be used if your function has a single independent variable which is an input variable (not a computed variable). You may use symbolic constants in the function. Here is an example of a function plot:

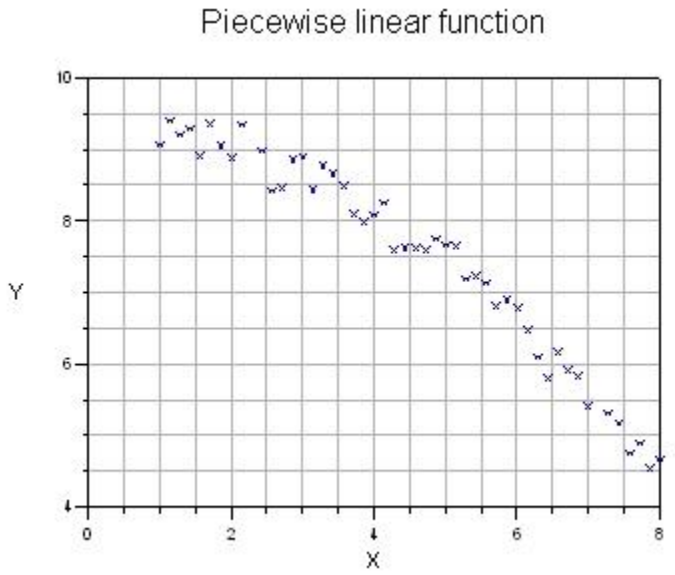


If you have the Advanced version of NLREG, you can use the PLOT statement to generate 3D surface plots for functions with two independent variables. Here is an example of a NLREG program that produces a surface plot and the plot it generated:

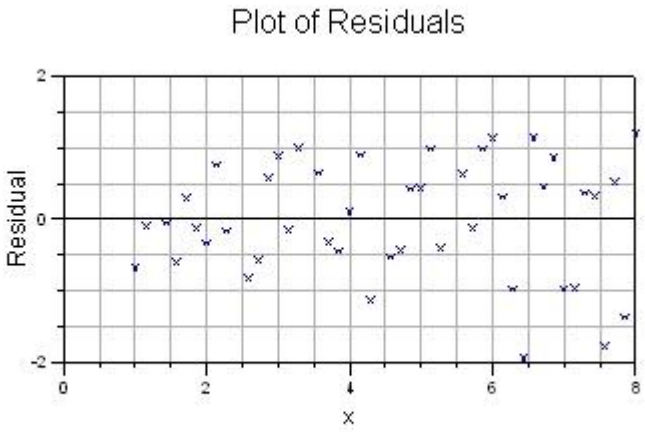
```
Title "3D Parabola Fitted to Data";
Variables X,Y,Z;
Parameters a,b;
Function Z = a*X^2 + b*Y^2;
Plot xlabel="X Axis", ylabel="Y Axis", zlabel="Z Axis",
        viewpitch=80,viewroll=0,viewyaw=45,FillDensity=0.7,
        Xdomain=-8,8,Ydomain=-8,8;
Data;
```



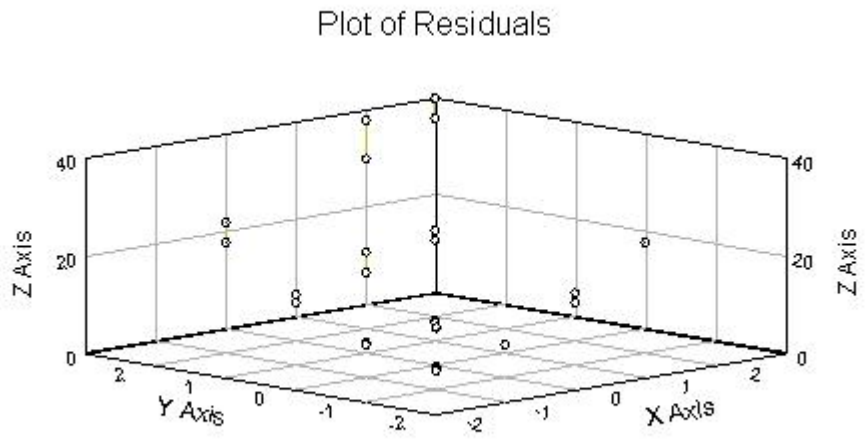
SPLOT -- Generate a scatter plot with marks at (X,Y) coordinates of data points. You may plot two sets of points on the same graph for comparison purposes. Unlike the PLOT statement, you may use any type of variable with the SPLOT statement: input, computed, system, and constant. Options are available to cause the plots to be connected by straight-line segments but unlike the PLOT statement NLREG will not compute curved segments between the points. You can use multiple SPLOT statements in your program to cause multiple scatter plots to be generated. Here is an example of a scatter plot:



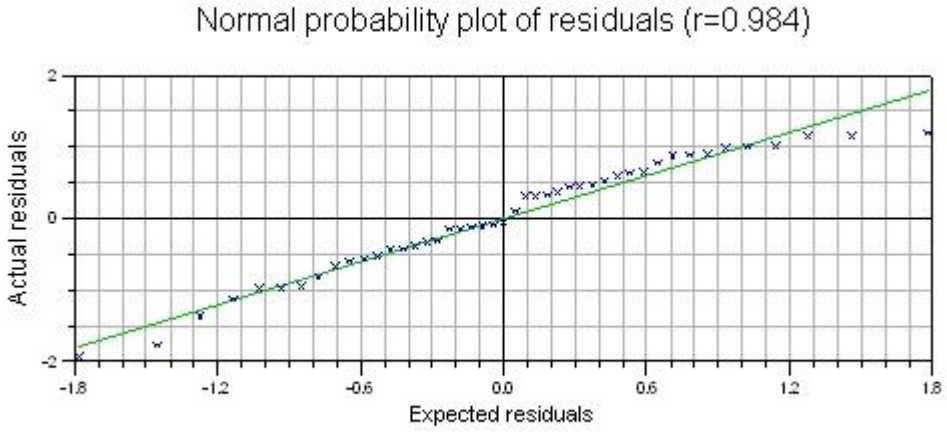
RPLOT -- Generate a plot showing the residual values of the function on the vertical (Y) axis. A "residual" value (or error deviation) is the difference between an actual value of the dependent variable of the function for an observation and the predicted value based on the function fitted by the regression analysis. A residual plot is useful for determining where, and by how much, the fitted function fails to predict the actual observations. Here is an example of a residual line plot:



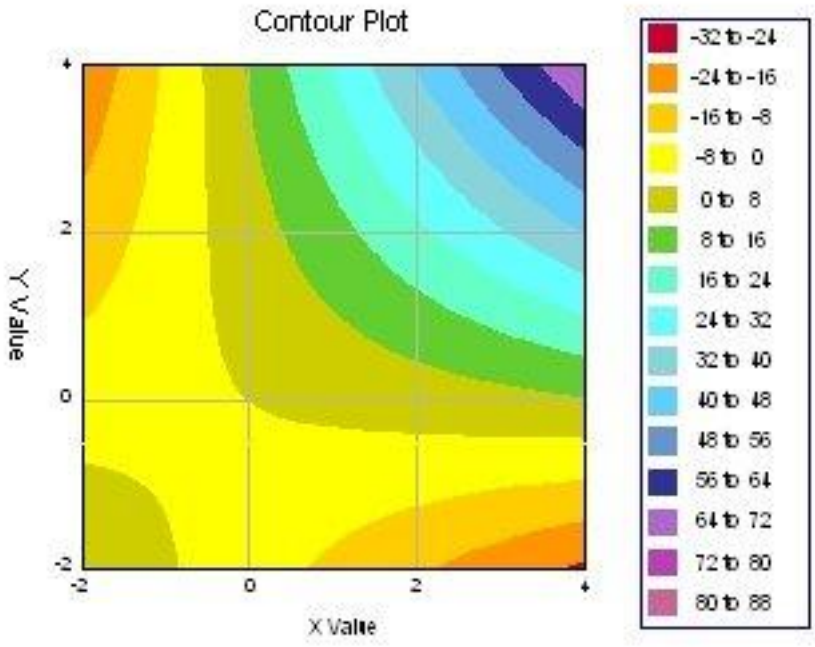
If you have the Advanced version of NLREG, you can use the RPLOT statement to generate 3D plots of residuals for functions with two independent variables. Here is an example of a 3D residual plot:



NPLOT -- Display a normal probability plot of the residual values. In this plot, the actual value of each residual is plotted on the vertical (Y) axis and the expected value of the residual, assuming the residuals are normally distributed, is plotted on the horizontal (X) axis. If the residuals are normally distributed, the resulting plot will be a straight line passing through the origin with a slope of 1. A normal probability plot is useful for determining whether the residuals are normally distributed. If they are not normally distributed then the form of the function being fitted may be inappropriate for the data. Here is an example of a normal probability plot:



CONTOURPLOT -- (Available only in the Advanced version) Display a contour plot showing the outline of the values of the dependent ("Z") variable on a plane. Here is an example of a contour plot:



Overview of Computation Process

Before getting into the details of how you present an analysis to NLREG, it is a good idea to review the basic computation process. You prepare a program file containing NLREG statements to declare variables, perform computations, and compute the value of a function that you wish to have fitted to your data observations. The function will have a dependent variable on the left side of the equal sign and one or more independent variables and parameters on the right side of the equal sign. Either input or computed variables may be used as dependent and independent variables.

For each observation record in your data file, NLREG executes your statements and computes the value of the function. The computed value of the function is assigned to the PREDICTED system variable. The predicted value is then subtracted from the actual value of the dependent variable and this difference is assigned to the RESIDUAL system variable. This process is repeated for each observation in the data file and the squared residual values are added together. After all of the observations have been processed NLREG adjusts the values of the parameters whose values are to be determined and repeats the process attempting to minimize the sum of the squared residuals.

It is important to understand what happens when NLREG executes a FUNCTION statement such as

```
Function y = a + b*x;
```

Unlike an assignment statement, this FUNCTION statement does *not* assign a new value to the y variable. Rather, it computes the value of the expression on the right side of the equal sign using

the current values of the a and b parameters and assigns this value to the PREDICTED system variable. It then subtracts this from the current value of the y variable to determine the residual.

Function Specification

Much of the power of NLREG comes from its ability to estimate the value of parameters that are part of complicated functions that you enter in ordinary algebraic form. NLREG provides you with great power in defining the function. Not only can you use a complicated expression to specify the function, you can also use multiple statements complete with intermediate work variables, conditional control (IF, ELSE), and looping. The only requirement is that a FUNCTION statement must be executed to define the estimated value of the dependent variable. The following section explains the arithmetic operators and built-in functions that are used to specify a function.

Arithmetic Operators
The following arithmetic operators may be used in expressions:

`++`	add 1 to a variable
`--`	subtract 1 from a variable
`+`	addition
`-`	subtraction or unary minus
`*`	multiplication
`/`	division
`%`	modulo
`** or ^`	exponentiation

The "++" and "--" operators may be used either immediately before or after a variable name. If they are used before the name, the increment or decrement is performed before the value of the variable is used in the expression. If they are used after the name, the value of the variable before being modified is used in the expression and then the increment or decrement takes place. For example, the sequence:

```
a = 3;
b = 3;
x = ++a;
y = b++;
```

assigns the value 4 to x and 3 to y. At the end of the sequence, both a and b have the value 4.

The following assignment operators can be used in expressions:

```
variable = expression;     // Assign expression to variable
variable += expression;    // Add expression to variable
variable -= expression;    // Subtract expression from variable
variable *= expression;    // Multiply variable by expression
variable /= expression;    // Divide variable by expression
```

The following operators compare two values and produce a value of 1 if the comparison is true, or 0 if the comparison is false:

==	Equal
!=	Not equal
<=	Less than or equal
>=	Greater than or equal
<	Less than
>	Greater than

The following logical operators may be used:

!	Logical NOT (negates true and false)
&&	AND
||	OR

The conditional operator has the form:

operand1 ? *operand2* : *operand3*

The value of *operand1* is evaluated. If it is true (not zero) then the value of *operand2* is the result of the expression. If the value of *operand1* is false (zero) then *operand3* is the result of the expression.

There are two other special operators: "[...]" (square brackets) which enclose subscripts on arrays (see the description of the DOUBLE statement for information about arrays), and "," (comma) which is used to specify left-to-right, sequential evaluation of a list of expressions and is most commonly used in FOR statements.

Operator precedence, in decreasing order, is as follows: subscript, unary minus, logical NOT, ++ and --, exponentiation, multiplication, division and modulo, addition and subtraction, relational (comparison), logical AND, logical OR, conditional, assignment, comma. Parentheses may be used to group terms.

Numeric Constants

Numeric constants may be written in their natural form (1, 0, 1.5, .0003, etc.) or in exponential form, *n.nnnEppp,* where *n.nnn* is the base value and *ppp* is the power of ten by which the base is multiplied. For example, the number 1.5E4 is equivalent to 15000. All numbers are treated as "floating point" values (actually, double precision), regardless of whether a decimal point is specified or not.

Symbolic Constants

You can use the CONSTANT statement to associate symbolic names with constant numeric values. When you use the symbolic name in the function the numeric value is substituted for the symbolic name. The PIECE.NLR example contains a symbolic constant.

Built-in Constant

The symbolic name "`PI`" (written without quotes) is equivalent to the value of π, 3.14159... You may write `PI` using either upper or lower case letters.

Built-in Functions

The following functions are built into NLREG and may be used in expressions. Most of the trig functions have forms that take angles in units of radians and another form (same name ending with 'D') that take angles in units of degrees. The DEG() and RAD() functions can be used to convert between degrees and radians.

ABS(*x*) -- Absolute value of *x*.

ACOS(*x*) -- Arc cosine of *x*. The returned value is the angle in radians.

ACOSD(*x*) -- Arc cosine of *x*. The returned value is the angle in degrees.

ACOSH(*x*) -- Arc hyperbolic cosine of *x*.

ASIN(*x*) -- Arc sine of *x*. The angle must be specified in radians.

ASIND(*x*) -- Arc sine of *x*. The angle must be specified in degrees.

ASINH(*x*) -- Arc hyperbolic sine of *x*.

ATAN(*x*) -- Arc tangent of *x*. The returned angle is in units of radians.

ATAND(*x*) -- Arc tangent of *x*. The returned angle is in units of degrees.

ATANH(*x*) -- Arc hyperbolic tangent of *x*.

BETAI(*x,a,b*) -- Incomplete beta function: Ix(*a,b*). The incomplete beta function can be used to compute a variety of statistical functions. For example, the probability of Student's t with *df* degrees of freedom can be computed with BETAI(*df*/(*df*+t2),.5**df*,.5). The probability of the F statistic with *df1* and *df2* degrees of freedom can be computed with 2*BETAI(*df2*/(*df2*+*df1***f*),.5***df2*,.5***df1*).

CEIL(*x*) -- Ceiling of x (an equivalent name for this function is INT). Returns the smallest integer that is at least as large as *x*. For example, CEIL(1.5)=2; CEIL(4)=4; CEIL(-2.6)=-2.

COS(*x*) -- Cosine of *x*. The angle must be specified in radians.

COSD(*x*) -- Cosine of *x*. The angle must be specified in degrees.

COSH(*x*) -- Hyperbolic cosine of *x*.

COT(*x*) -- Cotangent of x. (COT(*x*) = 1/TAN(*x*)). The angle must be specified in radians.

COTD(*x*) -- Cotangent of x. (COTD(*x*) = 1/TAND(*x*)). The angle must be specified in degrees.

CSC(*x*) -- Cosecant of x. (CSC(*x*) = 1/SIN(*x*)). The angle must be specified in radians.

CSCD(*x*) -- Cosecant of x. (CSCD(*x*) = 1/SIND(*x*)). The angle must be specified in degrees.

CTOP(*angle*) -- Convert an angle in the compass coordinate system to a polar coordinate angle. The polar coordinate system has the origin of an angle along the positive X axis and the angle increases in a counter-clockwise direction. The compass coordinate system has the positive Y axis as the origin (i.e., north) and the angle increases in a clockwise direction. The PTOC function performs the reverse transformation. The angle is in units of radians.

CTOPD(*angle*) -- Same as ctop() except the angle is in units of degrees.

DEG(*x*) -- Converts an angle, *x*, measured in radians to the equivalent number of degrees. Note, the built-in trig functions provided in NLREG have both radian and degree forms. The degree versions have the letter "D" appended to the end of their names such as sin(*radians*) and sind(*degrees*).

EI1(*alpha,phi*) -- Elliptic integral of the first kind. Computes the integral from 0 to *phi* (degrees or radians) of the function d.*phi*/sqrt(1-*k***2*sin(*phi*)**2), where *k* = sin(*alpha*). *alpha* and *phi* must be in the range 0 to 90 degrees or π/2 radians. The angle must be specified in radians.

EI1D(*alpha,phi*) -- Same as ei1() except the angle is specified in degrees.

EI2(*alpha,phi*) -- Elliptic integral of the second kind. Computes the integral from 0 to *phi* radians of the function sqrt(1-*k***2*sin(*phi*)**2)*d.*phi*, where *k* = sin(*alpha*). *alpha* and *phi* must be in the range 0 to π/2 radians.

EI2D(*alpha,phi*) -- Same as ei2() except the *alpha* and *phi* angles must be specified in degrees between 0 and 90.

EIC1(*alpha*) -- Complete elliptic integral of the first kind. Computes the integral from 0 to π/2 radians of the function d.*phi*/sqrt(1-*k***2*sin(*phi*)**2), where *k* = sin(*alpha*). *alpha* must be specified in radians and must be in the range 0 to π/2 radians.

EIC1D(*alpha*) -- Same as eic1() except the angle must be specified in degrees and must be in the range 0 to 90.

EIC2(*alpha*) -- Complete elliptic integral of the second kind. Computes the integral from 0 to or π/2 radians of the function sqrt(1-*k***2*sin(*phi*)**2)*d.*phi*, where *k* = sin(*alpha*). *alpha* must be specified in radians and must be in the range 0 to π/2.

EIC2D(*alpha*) -- Same as eic2() except *alpha* must be specified in degrees and must be in the range 0 to 90.

ERF(*x*) -- Standard error function of *x*.

EXP(*x*) -- e (base of natural logarithms) raised to the *x* power.

FAC(*x*) -- x factorial (*x*!). Note, the FAC function is computed using the GAMMA function (FAC(*x*)=GAMMA(*x*+1)) so non-integer argument values may be computed.

FLOOR(*x*) -- Floor of *x*. Returns the largest integer that is less than or equal to *x*. For example, FLOOR(2.5)=2; FLOOR(4)=4; FLOOR(-3.6)=-4.

GAMMA(*x*) -- Gamma function. Note, GAMMA(*x*+1) = *x*! (i.e., *x* factorial).

GAMMAI(*x*) -- Reciprocal of GAMMA function (GAMMAI(*x*) = 1/GAMMA(*x*)).

GAMMALN(*x*) -- Log (base e) of the GAMMA function.

GAMMP(*a*,*x*) -- Incomplete Gamma function.

$$gammp(a,x) \equiv \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$$

HAV(*x*) -- Haversine of *x*. (HAV(*x*) = (1-COS(*x*))/2). The angle must be specified in radians.

HAVD(*x*) -- Haversine of *x*. (HAVD(*x*) = (1-COSD(*x*))/2). The angle must be specified in degrees.

INCOMPLETEGAMMA(a,x) – Incomplete Gamma.

INT(*x*) -- Ceiling of *x* (an equivalent name for this function is CEIL). Returns the smallest integer that is at least as large as *x*. For example, INT(1.5)=2; INT(4)=4; INT(-2.6)=-2.

INTEGRAL(*variable*, *LowLimit*, *HighLimit*, *expression*[, *tolerance*]) – Compute the integral of an expression. The INTEGRAL function computes the definite integral of an expression over a specified range. Romberg's method of numerical integration is used by NLREG to compute the integral.

The Integral function computes $\int_{variable=LowLimit}^{variable=HighLimit} expression$

Variable is the name of a work variable that has been declared earlier in the program using a DOUBLE statement. The expression is integrated over this variable. *Variable* must consist only of a single variable name – not an expression or constant – and the variable must be a work variable, not a parameter or the dependent or independent variable.

LowLimit is the lower limit of the integration range, and *HighLimit* is the upper limit of the integration range. Full expressions may be used to specify the low and high limits. These expressions may include operators, functions, parameters whose values are being calculated and dependent variables.

Expression is the expression or function whose integral is to be computed. It may contain parameters and dependent variables. Usually it will contain the variable declared by the first argument (*variable*).

Tolerance is an optional argument that may be omitted; it specifies the accuracy to which the integral will be computed. If you omit the *tolerance* parameter, a default value of 1E-8 is used. If you specify the *tolerance* parameter, its value must be in the range 1E-3 to 1E-14. As you decrease the tolerance value (i.e., approach 1E-14), the accuracy of the integral increases, and the computation time also increases.

Please see the section beginning on page 73 for additional information about the Integral function.

J0(*x*) -- Bessel function of the first kind, order zero.

J1(*x*) -- Bessel function of the first kind, order one.

JN(*n,x*) -- Bessel function of the first kind, order *n*.

LOG(*x*) -- Natural logarithm (base e) of *x*.

LOG10(*x*) -- Base 10 logarithm of *x*.

LOG2(*x*) -- Base 2 logarithm of *x*.

MAX(*x1,x2*) -- Maximum value of *x1* or *x2*.

MIN(*x1,x2*) -- Minimum value of *x1* or *x2*.

NORMAL(*x*) -- Normal probability distribution of *x*. *X* is in units of standard deviations from the mean. See also the NPD function. NORMAL(*x*) = NPD(*x*,0,1);

NPD(*x,mean,std*) -- Normal probability distribution of *x* with specified mean and standard deviation. *X* is in units of standard deviations from the mean.

PAREA(*x*) -- Area under the normal probability distribution curve from $-\infty$ to *x*. That is, $\int_{-\infty}^{x} normal(x)$

PRINTF("*format*",*value1,value2,...*) -- Format and print a series of values. The NLREG printf function has the same syntax and function as the printf function in the C language. It causes a string to be written to your terminal and also the listing file for the analysis. Printf is primarily useful as a diagnostic tool to give you a way to observe what is happening during an analysis. Note: since your statements are executed for each data observation and each iteration, the printf may generate a great deal of output.

The first argument to printf (*format*) is a quoted string that contains characters to be printed, control codes, and (if values are to be printed) formatting specifications. If you are familiar with the C programming language, the NLREG formatting string has the same form and control codes.

Ordinary characters and numbers in the format string are printed just as they appear. Use the control code '`\n`' to cause a carriage-return, line-feed sequence to be printed to terminate a line. For example, the following statement prints a line of text:

```
printf("Beginning of analysis\n");
```

If you wish to insert formatted values in the string, specify one or more expressions after the format string. Place in the format string at the location where you want to insert the formatted value the sequence '`%lf`' (percent sign, lower case L, f) if you want the number formatted in the style *nnnn.nnnn*or; use '`%lE`' if you want exponential notation (*nnn.nnn*E*nnn*). Optionally, you may specify the width of the formatted value and the number of decimal places between '%' and 'l'. For example, the following sequence produces a formatted value with 8 total characters and 4 decimal places: %8.4lf. Here are several examples:

```
printf("Processing observation %lf\n",obs);
printf("X = %lf, Y = %lf\n",x,y);
printf("Predicted = %14.6lE\n",predicted);
```

PTOC(*angle*) -- Convert an angle in the polar coordinate system to a compass coordinate angle. The polar coordinate system has the origin of an angle along the positive X axis and the angle increases in a counter-clockwise direction. The compass coordinate system has the positive Y axis as the origin (i.e., north) and the angle increases in a clockwise direction. The CTOP function performs the reverse transformation. The angle is in units of radians.

PTOCD(*angle*) -- Same as ptoc() except the angle is in units of degrees.

PTORX(*angle,distance*) -- Convert a position in polar coordinates to the corresponding rectangular coordinate. This function returns the X coordinate of the position; use PTORY to obtain the Y coordinate. Note: polar coordinates are specified with the positive X axis being the origin for the angle and with the angle increasing in the counter-clockwise direction. The angle must be specified in units of radians.

PTORXD(*angle,distance*) -- Same as ptorx() except the angle must be specified in units of degrees.

PTORY(*angle,distance*) -- Convert a position in polar coordinates to the corresponding rectangular coordinate. This function returns the Y coordinate of the position; use PTORX to obtain the X coordinate. Note: polar coordinates are specified with the positive X axis being the origin for the angle and with the angle increasing in the counter-clockwise direction. The angle must be specified in units of radians.

PTORYD(*angle,distance*) -- Same as ptory() except the angle must be specified in units of degrees.

PULSE(*a,x,b*) -- Pulse function. If the value of *x* is less than *a* or greater than *b*, the value of the function is 0. If *x* is greater than or equal to *a* and less than or equal to *b*, the value of the function is 1. In other words, it is 1 for the domain (*a,b*) and zero elsewhere. If you need a function that is zero in the domain (*a,b*) and 1 elsewhere, use the expression (1-PULSE(*a,x,b*)).

RAD(*x*) -- Converts an angle measured in degrees to the equivalent number of radians. Note, the built-in trig functions provided in NLREG have both radian and degree forms. The

degree versions have the letter "D" appended to the end of their names such as sin(*radians*) and sind(*degrees*).

RANDOM() -- Returns a random value uniformly distributed in the range 0 to 1.

ROUND(*x*) -- Rounds *x* to the nearest integer. For example, ROUND(1.1)=1; ROUND(1.8)=2; ROUND(-2.8)=-3;

RTOPA(*x,y*) -- Convert a rectangular coordinate (*x,y*) to the corresponding polar coordinate (*angle,distance*). This function returns the angle (in radians), use RTOPD to get the distance coordinate. Note: polar coordinates are specified with the positive X axis being the origin for the angle and with the angle increasing in the counter-clockwise direction.

RTPOPAD(*x,y*) – Same as RTOPA() except the angle is in degrees.

RTOPD(*x,y*) -- Convert a rectangular coordinate to the corresponding polar coordinate. This function returns the distance from the origin, use RTOPA to get the angle. Note: polar coordinates are specified with the positive X axis being the origin for the angle and with the angle increasing in the counter-clockwise direction.

SEC(*x*) -- Secant of *x*. (SEC(*x*) = 1/COS(*x*)). The angle must be specified in radians.

SECD(*x*) -- Secant of *x*. (SECD(*x*) = 1/COSD(*x*)). The angle must be specified in degrees.

SEL(*a1,a2,v1,v2*) -- If *a1* is less than *a2* then the value of the function is *v1*. If *a1* is greater than or equal to *a2*, then the value of the function is *v2*.

SIN(*x*) -- Sine of *x*. The angle must be specified in radians. See TREND.NLR for an example of a function with a sin term.

SIND(*x*) -- Sine of *x*. The angle must be specified in degrees.

SINH(*x*) -- Hyperbolic sine of *x*.

SQRT(*x*) -- Square root of *x*.

STEP(*a,x*) -- Step function. If *x* is less than a, the value of the function is 0. If *x* is greater than or equal to *a*, the value of the function is 1. If you need a function which is 1 up to a certain value and then 0 beyond that value, use the expression STEP(*x,a*).

T(*n,x*) -- Chebyshev polynomial of order *n*.

TAN(*x*) -- Tangent of *x*. The angle must be in units of radians.

TAND(*x*) -- Tangent of *x*. The angle must be in units of degrees.

TANH(*x*) -- Hyperbolic tangent of *x*.

VARMAX(*x*) Computes the maximum value of an input variable. The single argument, *x*, must be the name of an input specified on the "Variables" command. The argument cannot be an expression, parameter name, or computed variable.

VARMEAN(*x*) Computes the mean value of an input variable. The single argument, *x*, must be the name of an input specified on the "Variables" command. The argument cannot be an expression, parameter name, or computed variable.

VARMIN(*x*) Computes the minimum value of an input variable. The single argument, *x*, must be the name of an input specified on the "Variables" command. The argument cannot be an expression, parameter name, or computed variable.

VARSTDDEV(*x*) Computes the standard deviation of an input variable. The single argument, *x*, must be the name of an input specified on the "Variables" command. The argument cannot be an expression, parameter name, or computed variable.

Y0(*x*) -- Bessel function of the second kind, order zero.

Y1(*x*) -- Bessel function of the second kind, order one.

YN(*n*,*x*) -- Bessel function of the second kind, order *n*.

NLREG Program Files

The statements described in this section are placed in a program file. A program file can be created using the editor built into NLREG or using any other editor of your choice. However, the file must a simple text file (with carriage-return and line-feed characters separating lines). It may not be a formatted word-processor file. NLREG program files should be given the extension ".NLR".

To execute a program file, either double click the name of the file, or start NLREG and then open the program file.

Comments

The beginning of a comment is denoted with "//" (two consecutive slash characters). Everything from the "//" sequence to the end of the line is treated as a comment. Comments may be on lines by themselves or on the ends of other statements. You can also specify a comment by beginning the comment with the "/*" character sequence. All characters following this are treated as comments up to the matching "*/" sequence. The following lines illustrate both types of comments:

```
//  Function to be fitted
y = a + b*x;  //  Simple linear equation
/*
 *  This is a comment.
 */
z = y / 5;                  /* This is a comment too */
```

Required Statements

Every program file must contain the following statements: VARIABLES, PARAMETERS, FUNCTION, and DATA. The DATA statement introduces the data for the analysis and must be the last statement in the file (data records may follow it). Other, optional, statements may be interspersed in the program file. The following is an example of a complete program file:

```
title "Depreciation Example";
variables  value,age,miles;
parameters base,depage,depmiles;
function value = base + depage*age + depmiles*miles;
data;
(data records follow)
```

NLREG Program Statements

The following is a list of the valid NLREG statements that can be placed in a NLREG program file. NLREG statements are not case sensitive. Remember to end each statement with a semicolon.

TITLE

TITLE "*string*"; (optional) -- Specifies a title line that is printed with the results of the analysis. Note: the title string must be enclosed in quote signs.

VARIABLES

VARIABLES *var1*,*var2*,...; (required) -- Specifies the names of the input variables whose values will be read from your data file. The order of the variable names must match the order of the data values on each observation record. You may define more variables than you actually use in the function specification. A maximum of 2000 variables may be specified. The length of a variable name is limited to 30 characters. Capitalize the variable names as you want them displayed in the results. The keyword "VARIABLE" may be used instead of "VARIABLES".

You may specify all of the variables on a single statement or you may use multiple VARIABLES statements. If you use multiple statements, the order in which they appear in the program file must match the order of the variable values on each observation record. The VARIABLES statement must precede the FUNCTION statement. See F33.NLR for an example of multiple regression using three independent variables. Here are examples of the VARIABLES statement:

```
Variables x,y;
Variable z;
```

You can also use the DOUBLE statement to declare variables (see below). The difference is that the VARIABLES statement declares variables that are read from the input file, whereas the DOUBLE statement declares variables whose values will be computed by statements in your program file.

PARAMETERS

PARAMETERS *param1*[=*initial1*],*param2*[=*initial2*],...; (required) -- Specifies the names of the parameters whose values are to be determined by NLREG. NLREG is capable of handling up to 2000 parameters. The parameter names may not exceed 30 characters in length. Do not specify any parameters that are not used in the analysis. The PARAMETERS statement must precede the FUNCTION statement. The keyword "PARAMETER" may be used instead of "PARAMETERS".

Optionally, an initial estimate of the parameter value may be specified by following the parameter name with an equal sign and the value. If no value is specified, 1 is used by default. Specifying an initial value that is near the actual value usually speeds up the operation of NLREG and may enable it to successfully converge to a solution. If NLREG is unable to converge to a solution, try specifying different starting values for the parameters. Try to specify a value that at least has the correct sign as the expected final value. Here are examples of the PARAMETERS statement:

```
Parameters p0,p1;
Parameter Temperature=22;
Parameter Mass=1E-5;
```

The CONSTRAIN statement can be used to limit the range of values for parameters. The SWEEP statement can be used to perform the regression analysis with a range of parameter initial values.

DOUBLE

DOUBLE *var1*[=*value*],*var2*[=*value*],...; (optional) -- Specifies the names of computed variables that you may use subsequently to hold calculated values. NLREG allows you to define up to 1000 computed variables. All variables hold double precision (64 bit) floating point values. Optionally, the name of a variable may be followed by an equal sign and a value to which the variable is initialized. If you do not specify an initial value, the variable is initialized to 0. The following are examples of DOUBLE statements:

```
double t1,t2;
double roomtemp=73;
```

It is convenient to use computed variables for intermediate calculations such as transformed values of input variables.

NLREG allows you to declare arrays with one, two or three dimensions. To do this, follow the name of the variable with number of array elements enclosed in square brackets. If the array has two dimensions specify the number of rows, then the number of columns separated by a comma. (Note: this is different than the C language syntax for declaring a two-dimensional array). You can use the ARRAYSIZE() function in a NLREG program to determine the size of an array dimension. The following statements declare a one dimensional array (i.e., a vector) with 20 elements, a two dimensional array with 5 rows and 10 columns and a three dimensional array with 10, 20 and 30 elements:

```
double xvec[20];
double ya[5,10];
double spaceval[10,20,30]
```

You may assign initial values to arrays by following the variable declaration with an equal sign and a list of values enclosed in curly braces. In the case of a multidimensional array, the right-most index varies the fastest. The following are examples of array declarations with initializations:

```
double xvec[5] = {2,5,7,1,0};
double xa[2,3] = {2.3,7.5,1.2,4.4,2.6,7.3};
double calibrate[2,2,2] = {1,2,3,4,5,6,7,8} ;
```

When used in expressions the subscript values are 0 based. That is, the first element of the array is referenced using a subscript value of 0 and the last element is referenced using a subscript value equal to one less than the number of elements in the array. For example, the following statements would declare an array with 100 elements and initialize it:

```
double xsq[100],i;
for (i=0; i<100; i++) {
    xsq[i] = i;
}
```

The ARRAYSIZE() function can be used within NLREG programs to determine the size of an array dimension. The form of the function is:

ARRAYSIZE("*arrayname*",*dimindex*)

Where *arrayname* is the name of the array variable enclosed in quote marks and *dimindex* is 0, 1 or 2 to specify which dimension you want the size of. Here is an example:

```
double calibrate[3,4], i, j;
for (i=0; i<ArraySize("calibrate",0) {
      for (j=0; j<ArraySize("calibrate",1) {
            calibrate[i,j] = 0.0;
      }
}
```

CONSTANT

CONSTANT *variable=value*; (optional) -- Specifies the name of a symbolic constant and associates a numeric value. You can then use the symbolic name where you would use a number and the corresponding constant numeric value will be substituted. This is useful when you are trying out different models and want to easily be able to change a constant value for each run. The following is an example of a symbolic constant named "Roomtemp" that causes the value 73 to be substituted in the function:

```
Variable Time;                  // Cooling time in seconds
Variable Temp;                  // Temperature of object
Constant Roomtemp = 73;         // Ambient temperature
Parameter InitTemp;             // Initial temperature
Parameter Coolrate;             // Cooling rate factor
Function Temp = Roomtemp + InitTemp * exp(-Coolrate * Time);
```

CONSTRAIN

CONSTRAIN *parameter=lowvalue,highvalue*; (optional) -- Specifies a lower and upper limit on the range of a parameter value. During the solution process, NLREG may allow a parameter's value to temporarily move in a direction away from its final value. With some functions it may be necessary to constrain the parameter's value so that it does not go negative (e.g., if the function takes the square root of the parameter), or zero (if the parameter is in a denominator).

If a parameter is tightly constrained, NLREG may report "singular convergence" because it is unable to converge to an optimum value of the parameter; however, the estimated values of other parameters may be useful.

If you use a CONSTRAIN statement, t-statistics and confidence intervals for parameter values will *not* be computed. The reason for this is that it is not possible to compute confidence intervals for a parameter whose values may be limited so that they don't reach the true optimal value.

Only a single parameter and its associated limits may be specified on each CONSTRAIN statement, but you may use multiple CONSTRAIN statements. The PARAMETERS statement must precede the CONSTRAIN statement. Use the CONSTANT statement if you wish to define a parameter with a fixed value.

The parameter value is allowed to range from *lowvalue* to *highvalue*. If you want to prevent a parameter value from going to zero, you must specify a value greater than zero for the low value (specifying zero would allow it to reach, but not go below, zero). For example, the following statement constrains the value of *age* to be greater than zero and less than or equal to 100:

```
constrain age = .0001,100;
```

The BADSTEP statement (see below) also can be used to constrain the range of parameter values. Unlike the CONSTRAIN statement which is a declarative statement, the BADSTEP statement is an executable statement that can be controlled by an IF statement, so it can be triggered by non-constant constraints or complex expressions. If a solution can be found using the BADSTEP statement, parameter confidence values are computed. However, the CONSTRAIN statement operates at a deeper level than BADSTEP and generally performs better. Use CONSTRAIN rather than BADSTEP when possible.

Here is an example of a CONSTRAIN statement and the corresponding statement using the BADSTEP statement:

```
CONSTRAIN angle = 0, 1.57;
if (angle < 0 || angle > 1.57) badstep;
```

See the COOLING.NLR, F33EXP.NLR, and POWER.NLR files for examples of the CONSTRAIN statement.

SWEEP

SWEEP *parameter=lowvalue,highvalue,stepsize*; (optional) -- Specifies that the regression analysis is to be performed repeatedly with a set of starting values for the parameter. The first analysis is performed with the parameter having the *lowvalue*; the value of *stepsize* is then added to the parameter's initial value and the analysis is performed again. The process is repeated until the value of the parameter reaches *highvalue*.

Each time the analysis is performed the value of the residual sum of squares is compared with the best previous result. The estimated values of the parameters for the best starting value are saved and used for the final analysis and report.

Only one parameter may be specified on each SWEEP statement, but you may have as many SWEEP statements as there are parameters. The number of regression analyses performed will be equal to the product of the number of parameter values for each SWEEP statement.

The SWEEP statement is useful when you are trying to fit a complicated function that may have "local minimum" values other than the "global minimum". Periodic functions (sin, cos, etc.) are especially troublesome.

Here is an example of a SWEEP statement that tries starting values for the Temperature parameter from 10 to 40 in steps of 0.01. Also, see the SINE.NLR program file for an example of the SWEEP statement.

```
Parameter Temperature;
Sweep Temperature=10,40,0.01;
```

FUNCTION

FUNCTION *depvar* = *function*; (required) -- Specifies the form of the function whose parameters are to be determined. The dependent variable must be the only thing to the left of the equal sign. The expression to the right of the equal sign may contain variables, parameters, constants, operators, and library functions such as sqrt, sin, exp, etc. The VARIABLES and PARAMETERS statements must appear in the program file before the FUNCTION statement. The function may be specified using parameters, input variables, computed variables (declared using the DOUBLE statement), constants, and library functions. You may use more than one FUNCTION statement if you use IF or other conditional statements to select which one will be executed. However, during each execution of your program file one, and only one, FUNCTION statement must be executed. Some example FUNCTION statements are show below:

```
Function  y = p0 + p1*x;

Function  distance = .5 * accel * time^2;

Function  value = price + yrdep*age + miledep*miles;

Function  populatn = base * growrate * exp(time);
```

CORRELATE

CORRELATE [*var1*,*var2*,...]; (optional) -- Causes NLREG to compute and print a correlation matrix. If you do not specify a list of variables the correlation matrix includes all input variables. If you wish to control exactly which variables are included in the matrix, or if you wish to include computed variables (declared with a DOUBLE statement), you may specify a list of variables. The F33.NLR example includes a CORRELATE statement. The following are examples of the CORRELATE statement:

```
correlate;
correlate x1,x2,x3,y;
```

COVARIANCE

COVARIANCE; (optional) -- Causes the variance-covariance matrix for the parameters to be printed.

CONFIDENCE

CONFIDENCE [*percent*]; (optional) -- Specifies that a confidence interval is to be printed for each estimated parameter. The purpose of regression analysis is to determine the best estimate of

parameter values. However, as with most statistical calculations, the values determined are estimates of the true values. The CONFIDENCE statement causes NLREG to print a table showing the range of possible values for each parameter given a specified confidence value. The *percent* parameter specifies the probability that that the actual value of the parameter is within the confidence interval to be computed. For example, the statement

```
Confidence 95;
```

specifies that the confidence interval(s) are to be computed such that there is a 95 percent probability that the actual values of the parameters are within the intervals (or that there is a 5 percent chance that the parameters are outside the intervals). The "*percent*" parameter may range from 50 to 99.999. If the CONFIDENCE statement is used without specifying a percent value, 90 is used by default.

Note: If you use the CONSTRAIN statement to limit the possible range of parameter values, confidence intervals will *not* be computed and displayed. The reason for this is that it is not possible to compute confidence intervals for a parameter whose values may be limited so that they don't reach the true optimal value.

TOLERANCE

TOLERANCE *value*; (optional, default=1E-10) -- Specifies the tolerance factor that is used to determine when the algorithm has converged to a solution. Reducing the tolerance value may produce a slightly more accurate result but will increase the number of iterations and the running time. The tolerance value must be in the range 1E-15 to 1E-1.

ITERATIONS

ITERATIONS *value*; (optional, default=50) -- Specifies the maximum number of iterations that should be attempted by the algorithm. If the solution does not converge to the limit specified by the TOLERANCE statement (or to the default tolerance) before the maximum number of iterations is reached, the process is stopped and the results are printed. Failure to converge before the specified number of iterations could be caused by one of three things:

1. The maximum allowed number of iterations might be too small. Try using an ITERATIONS statement with a larger value.

2. The tolerance factor may be too small. Even a properly converging solution will eventually "level off" or oscillate around a good, but non-zero, sum of squares value. Try using the TOLERANCE statement to increase the tolerance value.

3. The function may not be converging. Try specifying better (or at least different) starting values for the parameters on the PARAMETERS statement. Consider using the SWEEP statement to specify a range of parameter starting values.

OUTPUT

OUTPUT [TO "*file*"] *var1,var2,...*; (optional) -- Specifies that after the analysis is completed, data values are to be written to a file. One record is written for each data observation in the input file. If the "TO *file*" portion of the statement is specified, the output is written to the specified file. If this portion of the statement is omitted, the output values are written to the listing file

along with the results of the analysis. If a file name is specified without an extension, ".OUT" is used by default.

The list of variable names determines which variables are written to the file and the order in which the values appear in each output record. Any variable previously declared with a VARIABLES or DOUBLE statement may be specified. In addition, the following system variable names may appear in the output list:

OBS -- The observation record number, starting at 1 and increasing by 1.

PREDICTED -- The predicted value for the dependent variable for the observation, given the independent variable values and the parameters as calculated by the analysis.

RESIDUAL -- The difference between the actual value of the dependent variable and its predicted value.

EXPRESIDUAL – The expected residual value. This is the same as the expected residual used for X axis values on the NPLOT normal probability plot.

Examples of OUTPUT statements are shown below:

```
output age,miles,value,predicted,residual;
output to "growth.dat" obs,time,populatn,predicted;
```

POUTPUT

POUTPUT "*file*"; (optional) -- The POUTPUT statement specifies that NLREG is to write the final estimated values of the parameters to a file. Each parameter value is written to a separate line of the file. This statement is useful to create a file of estimated parameter values to be fed into another analysis program. This statement can also be used to determine the parameter estimates to more significant digits than displayed in the printed listing because the format used by the POUTPUT statement writes the values with 18 significant digits. The following is an example of a POUTPUT statement:

```
poutput "params.dat";
```

PLOT

PLOT [*options*]; (optional) -- Display a plot of the calculated function and the data observations. Each data point is displayed with a blue 'X'; the function that NLREG fits to the data is superimposed as a yellow curve.

The PLOT statement can display either line plots of variables with one independent variable, or, if you have the Advanced version of NLREG, the PLOT statement can display a 3D surface plot of functions with two independent variables.

In the Standard version of NLREG, the PLOT statement can be used only if the FUNCTION declaration meets the following requirements: (1) there is must only a single independent variable (i.e., variable on the right side of the equal sign); (2) the independent variable must be an input variable (i.e., declared with a VARIABLES statement not a DOUBLE statement). You *may* use symbolic constants declared with the CONSTANT statement in the function. If the function does

not meet these requirements you can produce different types of plots using the SPLOT, RPLOT and NPLOT statements.

If you have the Advanced version of NLREG, you can have two independent variables in the function. In this case, NLREG produces either 3D surface plots or contour plots.

The following options may be specified on the PLOT statement. If more than one option is specified, separate them with commas. For example, to produce a plot with X and Y axis labels use a statement with the following form:

```
PLOT XLABEL="Time",YLABEL="Blood concentration";
```

PLOT statement options:

NOGRID -- suppress the grid lines that are normally displayed with the plot.

NOMARK -- Suppress the display of the 'X' symbols (or circles in the case of 3D surface plots) that normally mark the data points. This causes only the function to be drawn with no data points.

NOFUNCTION -- Suppress the display of the fitted function. Only the input data points will be plotted.

TITLE="*string*" -- specify a title to be displayed with the plot. If no title is specified the title defined by the TITLE statement is used.

NOTITLE -- suppresses the title for the plot that, by default, is the title specified with the TITLE statement.

XVAR=*variable* -- specify the variable to be used for the horizontal (X) dimension of the plot. If you do not specify this option and there is only a single independent variable in the function, it is used by default. This option is primarily used for 3D surface plots where you need to designate which of two variables goes on the X axis.

YVAR=*variable* -- specify the variable to be used for the vertical (Y) dimension. If you do not specify this option and there is only a single independent variable, then the dependent variable of the function (i.e., the one on the left of the equal sign) is used by default. For 3D surface or contour plots, use the YVAR option to specify which of the independent variables is to be used for the Y axis, and use the XVAR option to designate the independent variable to go on the X axis..

ZVAR=*variable* -- specify the variable to be used for the (Z) dimension of a 3D surface or contour plot. If you do not specify this option, then the dependent variable of the function (i.e., the one on the left of the equal sign) is used by default.

XLABEL="*string*" -- specify a label to be printed along the X axis. If you do not use this qualifier, the name of variable whose values determine the X coordinates is used as the default label.

NOXLABEL -- suppress printing any label along the X axis.

YLABEL="*string*" -- specify a label to be printed along the Y axis. If you do not use this qualifier, the name of variable whose values determine the Y coordinates is used as the default label.

NOYLABEL -- suppress printing any label along the Y axis.

ZLABEL="*string*" -- specify a label to be printed along the Z axis for 3D surface plots (Advanced version of NLREG only). If you do not use this qualifier, the name of variable whose values determine the Z coordinates is used as the default label.

NOZLABEL -- suppress printing any label along the Z axis.

DOMAIN=*lowvalue,hivalue* -- specifies the domain of the independent variable over which the plot is to be generated. If no domain is specified, NLREG uses the range of the independent variable for the domain. Note, for 3D surface and contour plots, use the XDOMAIN and YDOMAIN options rather than DOMAIN.

XDOMAIN=*lowvalue,hivalue* – equivalent to the DOMAIN qualifier for functions with a single independent variable. For surface and contour plots, XDOMAIN specifies the domain for the X dimension. If no domain is specified, NLREG uses the range of the independent variable for the domain. Use the YDOMAIN qualifier to specify the domain of the other independent variable.

YDOMAIN=*lowvalue,hivalue* – for 3D surface plots (Advanced version), YLABEL specifies the domain of the second independent variable over which the plot is to be generated. If no domain is specified, NLREG uses the range of the independent variable for the domain. Use the XDOMAIN qualifier to specify the domain of the other independent variable.

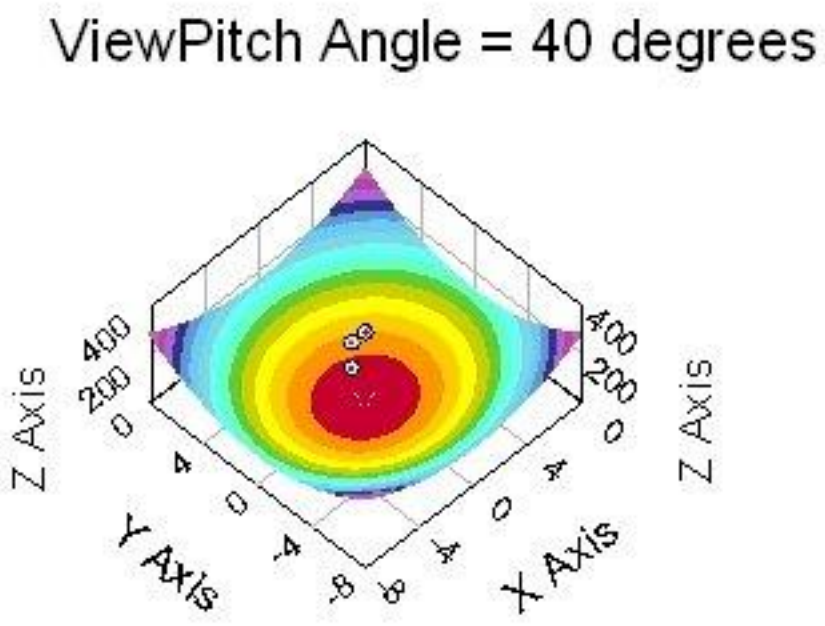
RESIDUAL -- draw vertical lines from each observed data point to the corresponding point on the calculated function line. These lines represent the "residual" value that NLREG is attempting to minimize.

SURFACEPLOT – Display the surface plot of the function when generating a 3D plot. By default, this option is on and does not need to be specified.

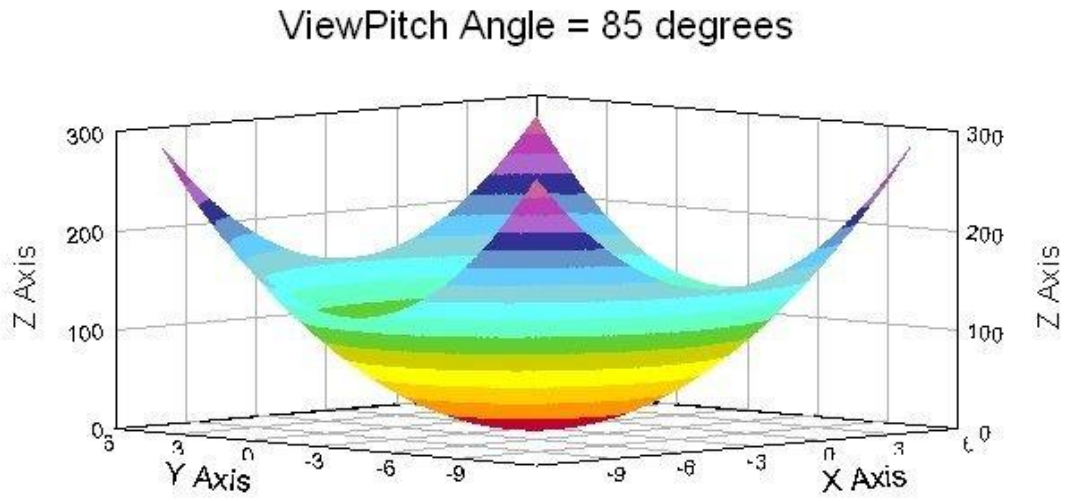
NOSURFACEPLOT -- Suppress the display of the function surface when generating a 3D surface plot. By suppressing the plot of the surface, you can clearly see the actual data points on the 3D grid.

VIEWPITCH=*angle* – when generating a 3D surface plot (Advanced version), VIEWPITCH specifies how many degrees the view is to be rotated downward around the X axis. A value of 0 (zero) would view the plane formed by the X and Y axes (i.e., the view would be from directly above the surface); a value of 90 would give a direct view of the plane formed by the X and Z axes. The default value is 69 degrees.

Here is an example of a plot with a ViewPitch angle of 40 degrees:



Here is the same function plotted with a ViewPitch angle of 85 degrees:

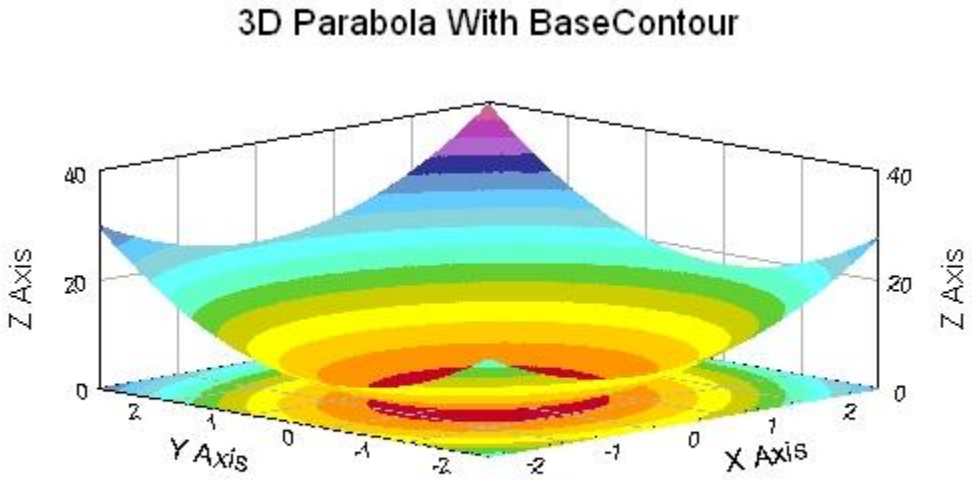


`VIEWYAW=`*angle* – when generating a 3D surface plot (Advanced version), VIEWYAW specifies how many degrees the view is to be rotated counterclockwise (leftward) around the Z axis. A value of 0 (zero) would view the plane formed by the X and Z axes; a value of 90 would give a direct view of the plane formed by the Y and Y axes. The default value is 46 degrees.

VIEWROLL=*angle* – When generating a 3D surface plot (Advanced version), VIEWROLL specifies how many degrees the view is to be rotated clockwise around the Y axis. The default value is 0 degrees.

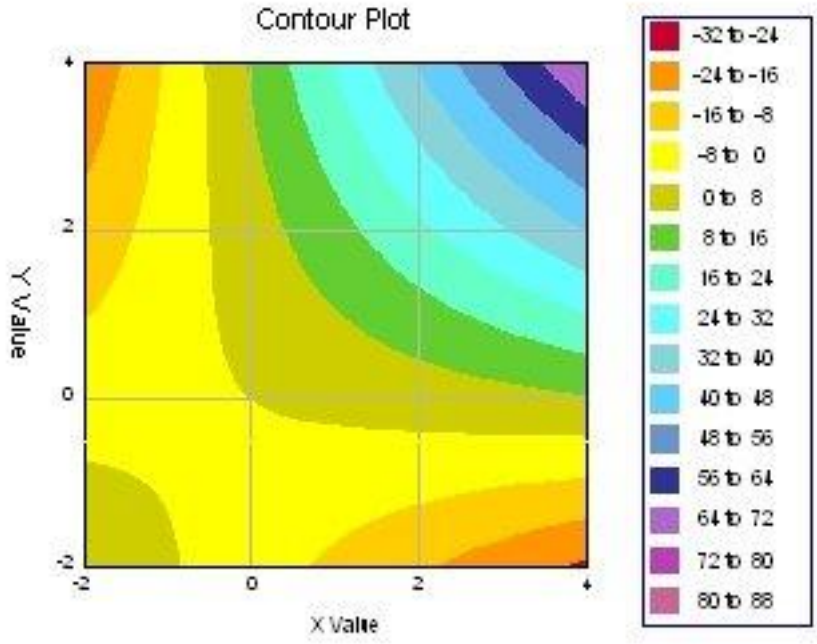
FILLDENSITY=*value* – When generating a 3D surface plot (Advanced version), FILLDENSITY specifies how many lines are to be computed for the surface of the plot. If you see discontinuities or holes in the surface, try increasing the value of FILLDENSITY. The default value is 1.0. If you see any holes in the surface, try increasing FILLDENSITY to 1.2; this will slow down the generation of the plot.

BASECONTOUR – when generating a 3D surface plot (Advanced version), BASECONTOUR tells NLREG to display a contour plot on the plane formed by the X and Y axes below the surface plot. See also the description of the CONTOURPLOT statement which generates just a contour plot without the 3D surface plot above it. Here is an example of a 3D surface plot with BASECONTOUR turned on:



CONTOURPLOT

CONTOURPLOT [*options*]; (optional, available only in the Advanced version) -- Display a contour plot of the calculated function. A contour plot uses colored regions to display the Z value on the X-Y plane. Here is an example of a contour plot:



The CONTOURPLOT statement can only be used if the FUNCTION declaration has two independent variables (i.e., variables on the right of the equal sign).

The following options may be specified on the CONTOURPLOT statement:

`NOGRID` -- suppress the grid lines that are normally displayed with the plot.

`TITLE="`*string*`"` -- specify a title to be displayed with the plot. If no title is specified the title defined by the TITLE statement is used.

`NOTITLE` -- suppresses the title for the plot that, by default, is the title specified with the TITLE statement.

`XVAR=`*variable* -- specify the variable to be used for the horizontal (X) dimension of the plot. If you do not specify this option and there is only a single independent variable in the function, it is used by default.

`YVAR=`*variable* -- specify the variable to be used for the vertical (Y) dimension. If you do not specify this option and there is only a single independent variable, then the dependent variable of the function (i.e., the one on the left of the equal sign) is used by default. For 3D surface or contour plots, use the YVAR option to specify which of the independent variables is to be used for the Y axis.

ZVAR=*variable* -- specify the variable to be used for the (Z) dimension of a 3D surface or contour plot. If you do not specify this option, then the dependent variable of the function (i.e., the one on the left of the equal sign) is used by default.

XLABEL="*string"* -- specify a label to be printed along the X axis. If you do not use this qualifier, the name of variable whose values determine the X coordinates is used as the default label.

NOXLABEL -- suppress printing any label along the X axis.

YLABEL="*string*" -- specify a label to be printed along the Y axis. If you do not use this qualifier, the name of variable whose values determine the Y coordinates is used as the default label.

NOYLABEL -- suppress printing any label along the Y axis.

XDOMAIN=*lowvalue,hivalue* -- specifies the domain for the X variable over which the plot is to be generated. If no domain is specified, NLREG uses the range of the first independent variable for the domain.

YDOMAIN=*lowvalue,hivalue* – specifies the domain of the second (Y) independent variable over which the plot is to be generated. If no domain is specified, NLREG uses the range of the second independent variable for the domain.

FILLDENSITY=*value* – specifies how many lines are to be computed for the surface of the plot. If you see discontinuities or holes in the surface, try increasing the value of FILLDENSITY. The default value is 1.0. If you see any holes in the surface, try increasing FILLDENSITY to 1.2; this will slow down the generation of the plot.

NOLEGEND – tells NLREG not to display the legend box that is usually displayed to the right of the contour plot.

SPLOT

SPLOT [*options*]; (optional) -- Display a scatter line plot of (X,Y) data points. Using the XVAR and YVAR options (see below) you can specify which variable is used for the vertical (Y) dimension and which is used for the horizontal (X) dimension. Any type of variable may be specified including input variables, computed variables (declared with the DOUBLE statement), the dependent variable of the function, and the system variables PREDICTED, RESIDUAL, and OBS.

You may display two scatter plots on the same image; this is useful for comparing computed values with input values. To do this, use the XVAR2 and YVAR2 options to specify the variables for the X and Y dimensions for the second plot. Each data point for the primary plot (specified by XVAR and YVAR) is marked with a blue 'X'. The data points for the second plot (specified by XVAR2 and YVAR2) are marked with yellow triangles. You can use the CONNECT and CONNECT2 options to draw straight-line segments through the points. The NOMARK and NOMARK2 options can be used to suppress the data point markers.

The following options may be specified on the SPLOT statement:

XVAR=*variable* -- specify the variable to be used for the horizontal (X) dimension of the first set of plotted points. This can be any type of variable, input or computed. If you do not specify this option and there is only a single independent variable in the function, it is used by default.

YVAR=*variable* -- specify the variable to be used for the vertical (Y) dimension. This can be any type of variable, input or computed. If you do not specify this option then the dependent variable of the function (i.e., the one on the left of the equal sign) is used by default.

XVAR2=*variable* -- specify the variable to be used for the horizontal (X) dimension of the second set of plotted points. This can be any type of variable. If you specify YVAR2 but not XVAR2, the default is the same variable as specified by XVAR.

YVAR2=*variable* -- specify the variable to be used for the vertical (Y) dimension of the second set of plotted points. This can be any type of variable. If you specify XVAR2 but not YVAR2, the default is the same variable as specified by YVAR.

CONNECT -- Connect the first set of points (XVAR and YVAR) by straight-line segments. The points are displayed and connected in the same order that they appear in the data file.

CONNECT2 -- Connect the second set of points (XVAR2 and YVAR2) by straight-line segments.

NOMARK -- Suppress the display of the 'X' symbols that normally mark the first set of data points. This can be used with CONNECT to cause only the line to be drawn.

NOMARK2 -- Suppress the display of the triangle symbols that normally mark the second set of data points.

NOGRID -- suppress the grid lines that are normally displayed with the plot.

TITLE="*string*" -- specify a title to be displayed with the plot. If no title is specified the title defined by the TITLE statement is used.

NOTITLE -- suppresses the title for the plot that, by default, is the title specified with the TITLE statement.

XLABEL="*string*" -- specify a label to be printed along the X axis. If you do not use this qualifier, the name of variable whose values determine the X coordinates is used as the default label.

NOXLABEL -- suppress printing any label along the X axis.

YLABEL="*string*" -- specify a label to be printed along the Y axis. If you do not use this qualifier, the name of variable whose values determine the Y coordinates is used as the default label.

NOYLABEL -- suppress printing any label along the Y axis.

DOMAIN=*lowvalue,hivalue* -- specifies the domain over which the plot is to be generated. If no domain is specified, NLREG uses the range of the horizontal variable(s) for the domain.

If there is more than one option, separate them with commas. The following is an example SPLOT statement:

```
splot xvar=time,yvar=sodium,yvar2=potassium,connect,connect2,
      title="Blood concentration over time",
      xlabel="Time (hours)",ylabel="Sodium & Potassium";
```

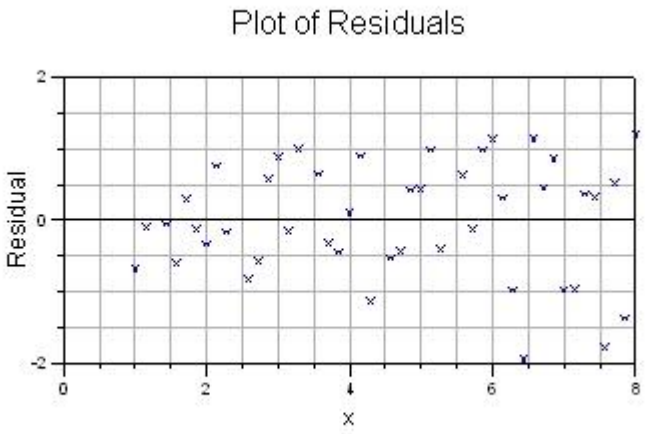
RPLOT

RPLOT [*options*]; (optional) -- Display a plot of the residual values. A "residual value" (or error deviation) is the difference between an actual value of the dependent variable for an observation and the predicted value based on the function fitted by the regression analysis. If the calculated function exactly predicted the actual observation values, all of the residual values would be zero. However, this is usually not the case and the residual values show where, and by how much, the fitted function fails to predict the actual observations.

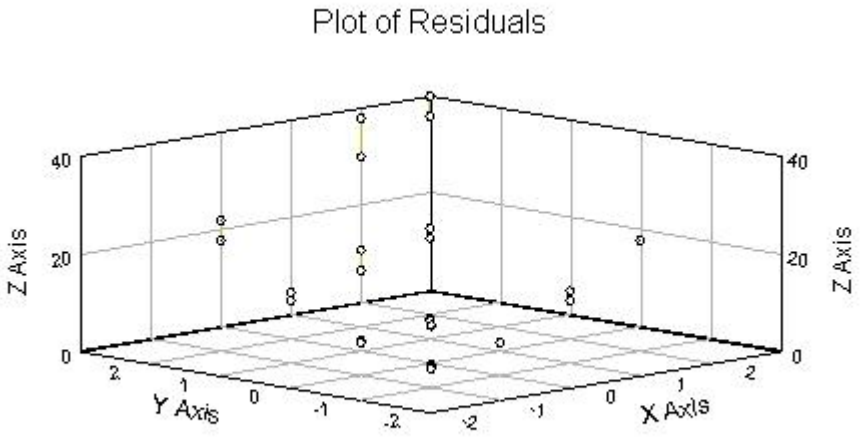
The RPLOT statement causes NLREG to display a plot showing the residual values on the vertical (Y) axis. The variable plotted along the horizontal (X) axis may be specified using the XVAR option (see below). You may specify any variable including the dependent variable and computed variables declared with the DOUBLE statement. If you do not specify a variable and there is a single independent variable in the function it is used. The X-axis label indicates which variable was used.

A residual plot is very useful for determining if the form of the function being fitted is appropriate for the data values. If the residual values are randomly distributed in positive and negative directions then the form (shape) of the fitted function is probably appropriate for the data and the deviations are due to random measurement errors. If, however, the residuals show a systematic pattern such as a periodic cycle, then the function may not be appropriate for the data values. See the discussion of the Durbin-Watson statistic for additional information about auto-correlated residual values. The PLOT, RPLOT, SPLOT, and NPLOT statements may be used in the same program file.

Here is an example of a residual plot (rplot) of a function with one independent variable:



Here is an example of a residual plot for a function with two independent variables (Advanced version only):



The following options may be specified on the RPLOT statement:

XVAR=*variable* -- specify which variable is to be used for the horizontal (X) dimension of the plot. You may specify any variable including independent input variables, the dependent variable of the function (i.e., the one on the left of the equal sign), and computed or transformed variables declared with the DOUBLE statement. If there is only a single independent variable NLREG will use it by default. The label along the X-axis indicates which variable was used.

YVAR=*variable* -- specify the variable to be used for the vertical (Y) dimension. If you do not specify this option and there is only a single independent variable, then the dependent variable of the function (i.e., the one on the left of the equal sign) is used by default. For 3D surface or contour plots, use the YVAR option to specify which of the independent variables is to be used for the Y axis.

ZVAR=*variable* -- specify the variable to be used for the (Z) dimension of a 3D surface or contour plot. If you do not specify this option, then the dependent variable of the function (i.e., the one on the left of the equal sign) is used by default.

NOGRID -- suppress the grid lines that are normally displayed with the plot.

TITLE="*string*" -- specify a title to be displayed with the plot. If this option is not specified, the default title is "Plot of residuals".

NOTITLE -- suppresses the title for the plot which, by default, is "Plot of residuals".

XLABEL="*string*" -- specify a label to be printed along the X axis. If you do not use this qualifier, the name of variable whose values determine the X coordinates is used as the default label.

NOXLABEL -- suppress printing any label along the X axis.

YLABEL="*string*" -- specify a label to be printed along the Y axis. If you do not use this qualifier, the default label is "Residual".

NOYLABEL -- suppress printing any label along the Y axis.

DOMAIN=*lowvalue,hivalue* -- specifies the domain over which the plot is to be generated. If no domain is specified, NLREG uses the range of the X dimension variable.

VIEWPITCH=*angle* – when generating a 3D residual plot (Advanced version), VIEWPITCH specifies how many degrees the view is to be rotated downward around the X axis. A value of 0 (zero) would view the plane formed by the X and Y axes (i.e., the view would be from directly above the surface); a value of 90 would give a direct view of the plane formed by the X and Z axes. The default value is 69 degrees.

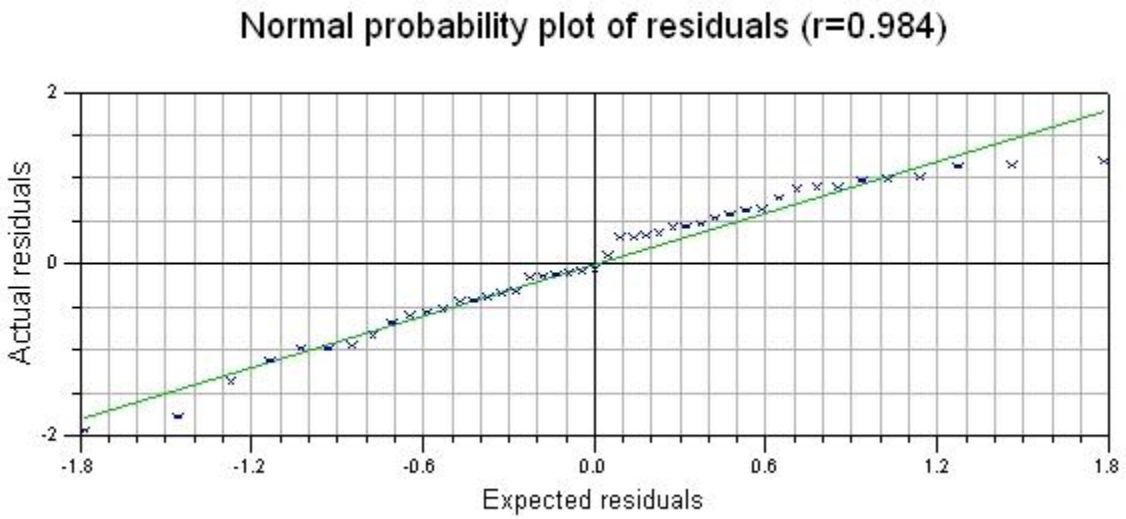
VIEWYAW=*angle* – when generating a 3D residual plot (Advanced version), VIEWYAW specifies how many degrees the view is to be rotated counterclockwise (leftward) around the Z axis. A value of 0 (zero) would view the plane formed by the X and Z axes; a value of 90 would give a direct view of the plane formed by the Y and Y axes. The default value is 46 degrees.

VIEWROLL=*angle* – When generating a 3D residual plot (Advanced version), VIEWROLL specifies how many degrees the view is to be rotated clockwise around the Y axis. The default value is 0 degrees.

If more than one option is specified, separate them with commas.

NPLOT

NPLOT [*options*] (optional) -- Display a normal probability plot of the residual values. In this plot, the actual value of each residual is plotted on the vertical (Y) axis and the expected value of the residual, assuming the residuals are normally distributed, is plotted on the horizontal (X) axis. If the residuals are normally distributed, the resulting plot will be a straight line passing through the origin with a slope of 1 (i.e., the actual value of each residual should equal the expected value from the normal distribution). If the residuals are not normally distributed, the plot will deviate from a straight line. NLREG displays a red line along which the X marks should be displayed if the residuals are normally distributed. Here is an example of a normal probability plot:



When a normal probability plot is requested, NLREG also computes the correlation between the actual residual values and their expected values and displays the correlation coefficient in the title line "(`r`=*n.nnn*)". If the residual values are normally distributed, the correlation should be close to 1.000. A correlation value less than 0.940 suggests that the residuals are not normally distributed.

The PLOT, RPLOT, SPLOT, and NPLOT statements may be used in the same program file.

The following options may be specified on the NPLOT statement:

`GRID` -- display grid lines to make it easier to estimate values.

`TITLE`="*string*" -- specify a title to be displayed with the plot. If no title is specified the default title is "Normal probability plot".

`NOTITLE` -- suppresses the title for the plot.

`XLABEL`="*string*" -- specify a label to be printed along the X axis. If you do not use this qualifier, default label is "Expected residuals".

`NOXLABEL` -- suppress printing any label along the X axis.

YLABEL="*string*" -- specify a label to be printed along the Y axis. If you do not use this qualifier, the default label is "Actual residuals".

NOYLABEL -- suppress printing any label along the Y axis.

If more than one option is specified, separate them with commas.

Assignment Statement

The assignment statement is an executable statement that evaluates an expression and assigns its value to a variable. The syntax for an assignment statement is:

```
variable = expression;    // Assign expression to variable
variable += expression;   // Add expression to variable
variable -= expression;   // Subtract expression from variable
variable *= expression;   // Multiply variable by expression
variable /= expression;   // Divide variable by expression
```

where "*variable*" is a variable that was previously declared using a DOUBLE statement. The variable may be subscripted if it is an array. "*expression*" is a valid arithmetic or logical expression following the rules explained earlier. If the expression involves a relational comparison operator (e.g., <, >, >=, etc.) or a logical operation (`&&`, `||`, `!`), the value 1 is used for true and 0 for false. The expression may contain any type of variable (input, computed, or constant) along with parameters and library functions.

IF Statement

The form of the IF statement is:

`IF (`*expression*`)` *statement1* [`ELSE` *statement2*]

If the expression is true (not zero) *statement1* is executed, if the expression is false (0) and the ELSE clause is specified, *statement2* is executed. The ELSE clause and the second set of controlled statements are optional. You may control groups of statements by enclosing them in braces. The following are examples of valid IF statements:

```
if (x > bigx) bigx = x;

if (x < Pivot) {
    Function Y = B0+B1*(X-Pivot);
} else {
    Function Y = B0+B2*(X-Pivot);
}
```

The PIECE.NLR program file contains an example of an IF statement.

WHILE Statement

The WHILE statement loops until the controlling expression becomes false (0) or a BREAK statement is executed within the loop. The form of the WHILE statement is:

```
WHILE (expression) {
    << controlled statements >>
}
```

Each time around the loop the expression is evaluated. If it is true (non-zero) the controlled statements are executed and then the process repeats until the expression becomes false. If a BREAK statement is executed within the loop, execution of the loop terminates and control is transferred to the first statement beyond the end of the loop. If a CONTINUE statement is executed in the loop, control is transferred to the conditional test at the top of the loop. The following is an example of a WHILE statement:

```
while (x < 5) {
    x = x + xmove;
    y = y + ymove;
}
```

DO Statement

The DO statement is very similar to the WHILE statement except the control expression is evaluated at the end of the loop rather than the beginning. This causes the loop always to be executed at least once. The form of the DO statement is:

```
DO {
    << controlled statements >>
} WHILE (expression);
```

For each iteration of the loop the controlled statements are executed and then the conditional expression is evaluated. If it is true (non-zero) control transfers to the first controlled statement at the top of the loop. A BREAK statement may be used to terminate the loop before the conditional expression is evaluated. A CONTINUE statement can be used to cause control to be transferred from within the loop to the point where the conditional expression is evaluated. The following is an example of a DO statement:

```
do {
  x += xstep;
  y += ystep;
} while (x < limit);
```

The BULLET.NLR program contains an example of the DO statement.

FOR Statement

The FOR statement is a looping control statement similar to the WHILE statement; however, the FOR statement also allows you to specify initialization expressions that are executed once at the beginning of the loop, and loop-end expressions that are executed at the end of each loop cycle. The form of the FOR statement is:

`FOR (`*expression1; expression2; expression3*`) statement;`

Execution of a FOR statement proceeds as follows:

1. Evaluate *expression1*. Typically this expression will include assignment operators ("=“) to set initial values for loop variables. If you need more than one initial expression, specify them as a list separated by commas.

2. Evaluate *expression2*. If its value is false (0) terminate the FOR statement and transfer control to the statement that follows the controlled statement. If *expression2* is true, proceed to the next step.

3. Execute the controlled statement. If more than one statement is to be controlled, enclose them with brace characters ("{“ "}”).

4. Evaluate *expression3*. This expression will typically contain operators such as "++”, "+=“, "--”, or "-=″ to modify the value of a loop variable.

5. Transfer control to step 2, where *expression2* is once again evaluated.

The following is an example of a FOR statement:

```
for (time=starttime; time<endtime; time+=timestep) {
    << controlled statements >>
}
```

BREAK Statement

The BREAK statement can be used in FOR, WHILE, and DO loops to terminate the loop and cause control to transfer to the statement beyond the end of the loop. The following is an example of a BREAK statement:

```
time = 0;
x = 0;
while (time < endtime) {
    x += delta * xspeed;
    if (x > 10) break;
}
```

CONTINUE Statement

The CONTINUE statement can be used in FOR, WHILE, and DO loops to terminate the current iteration and begin the next one. When CONTINUE is executed in a WHILE or DO statement, control is transferred to the point in the loop where the loop control expression is evaluated. When CONTINUE is executed in a FOR statement, control is transferred to the bottom of the loop where expression3 is evaluated (which normally augments the values of the loop variables for the next iteration). The form of the CONTINUE statement is:

```
continue;
```

STOP Statement

The STOP statement terminates the calculations for the current iteration. The last value of the independent variable (as specified with a FUNCTION statement) is used as the calculated value of the function. An implicit stop occurs if you "fall through" the last executable statement. The form of the STOP statement is:

```
stop;
```

BADSTEP Statement

The BADSTEP statement terminates the calculations for the current iteration and signals that the function could not be computed. The minimization procedure will then back step and try a different approach toward a solution. The BADSTEP statement is usually used under the control of an IF statement to signal if a potential convergence step is moving into an invalid region.

The BADSTEP statement can be used to put constraints on the possible values of parameters in a similar fashion to the CONSTRAIN statement. The CONSTRAIN statement operates at a deeper level than BADSTEP and usually performs better. However, the CONSTRAIN statement only allows constant values to be specified for the constraints whereas the BADSTEP statement can be controlled by an IF statement with a complex expression.

Parameter confidence information such as t-values are not computed when a CONSTRAIN statement is used, but they are calculated when BADSTEP is used. Use CONSTRAIN rather than BADSTEP when possible. Warning: the use of BADSTEP (or CONSTRAIN) may result in NLREG being unable to converge to a solution.

The form of the BADSTEP statement is:

```
badstep;
```

Here is an example of a BADSTEP statement being controlled by an IF statement:

```
if (Angle > 90 || Distance < 1) badstep;
```

DATACOUNT statement

DATACOUNT *count*; (optional) – Specifies the approximate number of data records that are in the dataset. It is not necessary to use this statement, but if you have a large number of data

observations (i.e., more than 50,000) specifying the approximate count can speed up the analysis. NLREG uses the specified count to allocate the initial data array area. This avoids having the area enlarged and reallocated as the data records are read into memory. Note: if you specify a value, it is better that it be slightly too large than slightly too small. The form of the statement is:

`datacount` *`count;`*

where *count* is an integer value that indicates how many lines are in the data file.

DATASKIP statement

DATASKIP *count*; (optional) -- Specifies how many lines at the front of the data file should be skipped when reading the data records. Normally, NLREG assumes the first record of your data file contains the first data observation for the analysis. However, some data files produced by other programs may begin with one or more other types of lines (titles, comments, etc.) before the first actual data observation. The form of the statement is:

`dataskip` *`count;`*

where *count* is an integer value that indicates how many lines at the front of the data file are to be skipped. For example, the following statement causes 1 line to be skipped:

```
dataskip 1;
```

DATA

DATA ["*file*"]; (required) -- Specifies the name of the file containing the data records, or introduces the data records which follow the statement. If a file name is specified on the DATA statement, the file is opened, its data records are read, and the regression analysis is performed. If a file name is specified without an extension, "`.DAT`" is used by default. Note that if you specify a file name it must be enclosed in quote marks.

If no file name is specified on the DATA statement, the data records must immediately follow the DATA statement in the program file.

If you have a large number of data records, you must use the form of the DATA statement that specifies a file name and place your data records in a separate file. This is necessary because the editor built into NLREG can handle only up to 65,000 characters.

For large files with more than 50,000 data records, you can speed up processing by using the DATACOUNT statement to specify the number of records that NLREG should initially allocate space for.

Each data record must contain at least as many data values as the number of variables specified on the VARIABLES statement(s). The order of the variables as specified on the VARIABLES statement must match the order of the values in each observation. Any data values beyond those required for the specified variables are ignored. Each observation must begin on a new line.

Each set of data values (i.e., a record) is specified as a series of numeric ASCII values terminated by a carriage-return/line-feed. Alternatively, you can use a semicolon to terminate a data record rather than carriage-return/line-feed. The data values within a record must be separated by one or

more spaces and/or a comma. You may place a comment on the end of a data record by beginning the comment with "//". Data values may contain decimal points and may be expressed in exponential notation (i.e., *n.nnnn*E*ppp*).

The DATA statement must be the last statement in the program file. If no file name is specified on the DATA statement, the data records must immediately follow the DATA statement in the program file. The following is an example of a complete program file including data records:

```
Variables age,miles,value;
Parameters base,depage,depmiles;
Function value = base + depage*age + depmiles*miles;
Data;
2  10000  13000
4  42000   9000
1   7000  17000
6  52000   6000
5  48000   8000
```

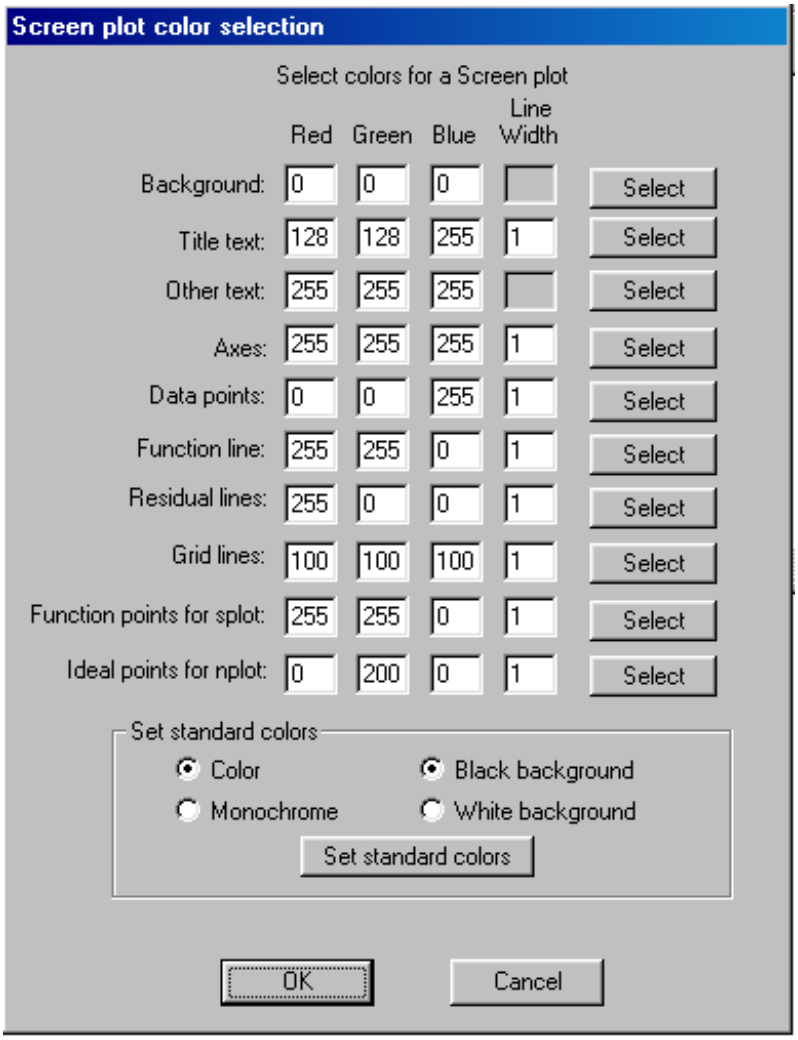
If the data records had been placed in a separate file named CAR.DAT, the statements would be as follows:

```
Variables age,miles,value;
Parameters base,depage,depmiles;
Function value = base + depage*age + depmiles*miles;
Data "car.dat";
```

Setting Colors and Saving Plots

Selecting Colors for Plots

NLREG allows you to select the colors used for items in generated plots. NLREG saves three set of color values, one for plots displayed on the screen ("Screen plot"), one for plots written to your printer ("Printed plot") and one for plots written to .bmp or .jpg image files on disk ("Saved plot"). To select color values, click the "Colors" item on the main menu then chose the color set you want to set from the drop-down menu. At color selection panel will then be displayed:



Type in a numeric value between 0 and 255 for the Red, Green and Blue color components for each item. You can also click the "Select" button for an item and select a color from a color palette. To set the standard color values, click the radio buttons indicating whether you want a color or monochrome plot and chose a black or white background, then click the button labeled "Set standard colors".

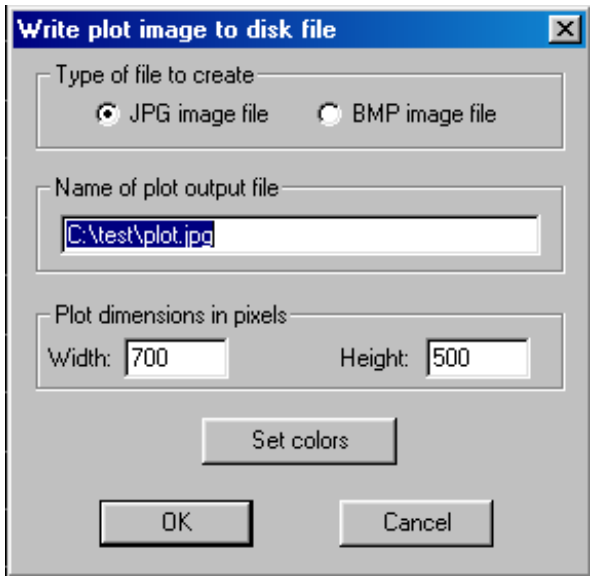
Remember that there are three sets of color values. Setting the colors for screen plots will not affect the colors for printed or saved plots.

Color values you select are stored in the Windows Registry and remembered the next time you run NLREG.

Saving Plots to Disk Files

While viewing a plot screen, you can save the plot image to either a .bmp or .jpg disk file. To save a plot, click either "Save-plot" on the menu or click the save-plot toolbar icon:

A save-plot screen will appear:



Select whether you want the plot stored as a JPG or BMP type file, specify the name of the file and specify the size of the plot image to be stored. You can click the "Set colors" button if you wish to change the colors used for stored plot images.

Understanding the Results

Descriptive Statistics for Variables

NLREG prints a variety of statistics at the end of each analysis. For each variable, NLREG lists the minimum value, the maximum value, the mean value, and the standard deviation. You should confirm that these values are within the ranges you expect.

Parameter Estimates

For each parameter, NLREG displays the initial parameter estimate (which you specified on the PARAMETER statement, or 1 by default), the final (maximum likelihood) estimate, the standard error of the estimated parameter value, the "t" statistic comparing the estimated parameter value with zero, and the significance of the t statistic. Nine significant digits are displayed for the parameter estimates. If you need to determine the parameters to greater precision, use the POUTPUT statement.

The final estimate parameter values are the results of the analysis. By substituting these values in the equation you specified to be fitted to the data, you will have a function that can be used to predict the value of the dependent variable based on a set of values for the independent variables. For example, if the equation being fitted is

```
y = p0 + p1*x
```

and the final estimates are 1.5 for p0 and 3 for p1, then the equation

```
y = 1.5 + 3*x
```

is the best equation of this form that will predict the value of y based on the value of x.

t Statistic

The "t" statistic is computed by dividing the estimated value of the parameter by its standard error. This statistic is a measure of the likelihood that the actual value of the parameter is *not* zero. The larger the absolute value of t, the less likely that the actual value of the parameter could be zero.

Prob(t)

The "Prob(t)" value is the probability of obtaining the estimated value of the parameter if the actual parameter value is zero. The smaller the value of Prob(t), the more significant the parameter and the less likely that the actual parameter value is zero. For example, assume the estimated value of a parameter is 1.0 and its standard error is 0.7. Then the t value would be 1.43 (1.0/0.7). If the computed Prob(t) value was 0.05 then this indicates that there is only a 0.05 (5%) chance that the actual value of the parameter could be zero. If Prob(t) was 0.001 this indicates there is only 1 chance in 1000 that the parameter could be zero. If Prob(t) was 0.92 this indicates that there is a 92% probability that the actual value of the parameter could be zero; this implies

that the term of the regression equation containing the parameter can be eliminated without significantly affecting the accuracy of the regression.

One thing that can cause Prob(t) to be 1.00 (or near 1.00) is having redundant parameters. If at the end of an analysis several parameters have Prob(t) values of 1.00, check the function carefully to see if one or more of the parameters can be removed. Also try using a DOUBLE statement to set one or more of the parameters to a reasonable fixed value; if the other parameters suddenly become significant (i.e., Prob(t) much less than 1.00) then the parameters are mutually dependent and one or more should be removed.

The t statistic probability is computed using a two-sided test. The CONFIDENCE statement can be used to cause NLREG to print confidence intervals for parameter values. The SQUARE.NLR example regression includes an extraneous parameter (p0) whose estimated value is much smaller than its standard error; the Prob(t) value is 0.99982 indicating that there is a high probability that the value is zero.

Final Sum of Squared Deviations

In addition to the variable and parameter values, NLREG displays several statistics that indicate how well the equation fits the data. The "Final sum of squared deviations" is the sum of the squared differences between the actual value of the dependent variable for each observation and the value predicted by the function, using the final parameter estimates.

Average and Maximum Deviation

The "Average deviation" is the average over all observations of the absolute value of the difference between the actual value of the dependent variable and its predicted value.

The "Maximum deviation for any observation" is the maximum difference (ignoring sign) between the actual and predicted value of the dependent variable for any observation.

Proportion of Variance Explained

The "Proportion of variance explained (R2)" indicates how much better the function predicts the dependent variable than just using the mean value of the dependent variable. This is also known as the "coefficient of multiple determination." It is computed as follows: Suppose that we did not fit an equation to the data and ignored all information about the independent variables in each observation. Then, the best prediction for the dependent variable value for any observation would be the mean value of the dependent variable over all observations. The "variance" is the sum of the squared differences between the mean value and the value of the dependent variable for each observation. Now, if we use our fitted function to predict the value of the dependent variable, rather than using the mean value, a second kind of variance can be computed by taking the sum of the squared difference between the value of the dependent variable predicted by the function and the actual value. Hopefully, the variance computed by using the values predicted by the function is better (i.e., a smaller value) than the variance computed using the mean value. The "Proportion of variance explained" is computed as 1 – (variance using predicted value / variance using mean). If the function perfectly predicts the observed data, the value of this statistic will be 1.00 (100%). If the function does no better a job of predicting the dependent variable than using the mean, the value will be 0.00.

Adjusted Coefficient of Multiple Determination

The "adjusted coefficient of multiple determination (Ra^2)" is an R^2 statistic adjusted for the number of parameters in the equation and the number of data observations. It is a more conservative estimate of the percent of variance explained, especially when the sample size is small compared to the number of parameters. It is computed using the formula:

$$Ra^2 = 1 - \frac{n-1}{n-p} \times \left(1 - R^2\right)$$

where n is the number of observations, p is the number of parameters, and R^2 is the unadjusted coefficient of multiple determination.

Standard Error

The standard error statistic is computed as follows:

1. Compute the sum of the squared residuals, SSE.
2. Compute the number of degrees of freedom from:
   NDF = (number of observation - number of parameters computed)
3. Compute the mean squared error from:
   MSE = SSE / NDF
4. Compute the standard error from:
   Standard Error = sqrt(MSE)

Durbin-Watson Statistic

The "Durbin-Watson test for auto-correlation" is a statistic that indicates the likelihood that the deviation (error) values for the regression have a first-order autoregression component. The regression models assume that the error deviations are uncorrelated.

In business and economics, many regression applications involve time series data. If a non-periodic function, such as a straight line, is fitted to periodic data, the deviations have a periodic form and are positively correlated over time; these deviations are said to be "auto-correlated" or "serially correlated." Auto-correlated deviations may also indicate that the form (shape) of the function being fitted is inappropriate for the data values (e.g., a linear equation fitted to quadratic data).

If the deviations are auto-correlated, there may be a number of consequences for the computed results: 1) The estimated regression coefficients no longer have the minimum variance property; 2) the mean square error (MSE) may seriously underestimate the variance of the error terms; 3) the computed standard error of the estimated parameter values may underestimate the true standard error, in which case the t values and confidence intervals may be incorrect. Note that if an appropriate periodic function is fitted to periodic data, the deviations from the regression will be uncorrelated because the cycle of the data values is accounted for by the fitted function.

Small values of the Durbin-Watson statistic indicate the presence of auto-correlation. Consult significance tables in a good statistics book for exact interpretations; however, a value less than 0.80 usually indicates that auto-correlation is likely. If the Durbin-Watson statistic indicates that the residual values are auto-correlated, it is recommended that you use the RPLOT and/or NPLOT statements to display a plot of the residual values.

If the data has a regular, periodic component you can try including a sin term in your function. The TREND.NLR example fits a function with a sin term to data that has a linear growth with a superimposed sin component. With the sin term the function has a residual value of 29.39 and a Durbin-Watson value of 2.001; without the sin term (i.e., fitting only a linear function) the residual value is 119.16 and the Durbin-Watson value is 0.624 indicating strong auto-correlation. The general form of a sin term is

```
amplitude * sin(2*pi*(x-phase)/period)
```

where *amplitude* is a parameter that determines the magnitude of the sin component, *period* determines the period of the oscillation, and *phase* determines the phase relative to the starting value. If you know the period (e.g., 12 for monthly data with an annual cycle) you should specify it rather than having NLREG attempt to determine it.

If an NPLOT statement is used to produce a normal probability plot of the residuals, the correlation between the residuals and their expected values (assuming they are normally distributed) is printed in the listing. If the residuals are normally distributed, the correlation should be close to 1.00. A correlation less than 0.94 suggests that the residuals are not normally distributed.

Analysis of Variance Table

An "Analysis of Variance" table provides statistics about the overall significance of the model being fitted.

F Value and Prob(F)

The "F value" and "Prob(F)" statistics test the overall significance of the regression model. Specifically, they test the null hypothesis that *all* of the regression coefficients are equal to zero. This tests the full model against a model with no variables and with the estimate of the dependent variable being the mean of the values of the dependent variable. The F value is the ratio of the mean regression sum of squares divided by the mean error sum of squares. Its value will range from zero to an arbitrarily large number.

The value of Prob(F) is the probability that the null hypothesis for the full model is true (i.e., that all of the regression coefficients are zero). For example, if Prob(F) has a value of 0.01000 then there is 1 chance in 100 that all of the regression parameters are zero. This low a value would imply that at least some of the regression parameters are nonzero and that the regression equation does have some validity in fitting the data (i.e., the independent variables are not purely random with respect to the dependent variable).

Correlation Matrix

The CORRELATE statement can be used to cause NLREG to print a correlation matrix. A "correlation coefficient" is a value that indicates whether there is a linear relationship between two variables. The absolute value of the correlation coefficient will be in the range 0 to 1. A value of 0 indicates that there is no relationship whereas a value of 1 indicates that there is a perfect correlation and the two variables vary together. The sign of the correlation coefficient will be negative if there is an inverse relationship between the variables (i.e., as one increases the other decreases).

For example, consider a study measuring the height and weight of a group of individuals. The correlation coefficient between height and weight will likely have a positive value somewhat less than one because tall people tend to weigh more than short people. A study comparing number of cigarettes smoked with age at death will probably have a negative correlation value.

A correlation matrix shows the correlation between each pair of variables. The diagonal of the matrix has values of 1.00 because a variable always has a perfect correlation with itself. The matrix is symmetric about the diagonal because X correlated with Y is the same as Y correlated with X.

Problems occur in regression analysis when a function is specified that has multiple independent variables that are highly correlated. The common interpretation of the computed regression parameters as measuring the change in the expected value of the dependent variable when the corresponding independent variable is varied while all other independent variables are held constant is not fully applicable when a high degree of correlation exists. This is due to the fact that with highly correlated independent variables it is difficult to attribute changes in the dependent variable to one of the independent variables rather than another. The following are effects of fitting a function with high correlated independent variables:

1. Large changes in the estimated regression parameters may occur when a variable is added or deleted, or when an observation is added or deleted.

2. Individual tests on the regression parameters may show the parameters to be non-significant.

3. Regression parameters may have the opposite algebraic sign than expected from theoretical or practical considerations.

4. The confidence intervals for important regression parameters may be much wider than would otherwise be the case. The solution to these problems may be to select the most significant of the correlated variables and use only it in the function.

Note: the correlation coefficients indicate the degree of *linear* association between variables. Variables may be highly related in a nonlinear fashion and still have a correlation coefficient near 0.

Theory of Operation

Minimization Algorithm

NLREG uses a model/trust-region technique along with an adaptive choice of the model Hessian. The algorithm is essentially a combination of Gauss-Newton and Levenberg-Marquardt methods; however, the adaptive algorithm often works much better than either of these methods alone.

The basis for the minimization technique used by NLREG is to compute the sum of the squared residuals for one set of parameter values and then slightly alter each parameter value and recompute the sum of squared residuals to see how the parameter value change affects the sum of the squared residuals. By dividing the difference between the original and new sum of squared residual values by the amount the parameter was altered, NLREG is able to determine the approximate partial derivative with respect to the parameter. This partial derivative is used by NLREG to decide how to alter the value of the parameter for the next iteration.

If the function being modeled is well behaved, and the starting value for the parameter is not too far from the optimum value, the procedure will eventually converge to the best estimate for the parameter. This procedure is carried out simultaneously for all parameters and is, in fact, a minimization problem in *n*-dimensional space, where '*n*' is the number of parameters.

For a much more detailed explanation of the regression algorithm used by NLREG see *ACM Transactions on Mathematical Software* 7,3 (Sept. 1981) "Dennis, J.E., Gay, D.M., and Welsch, R.E. -- An adaptive nonlinear least-squares algorithm."

Convergence Criterion

NLREG has several convergence criteria that stop the iterative minimization procedure. The TOLERANCE statement can be used to alter the convergence tolerance value.

Two internal variables are used to determine when convergence has occurred. RFCTOL has a default value of 1E-10 and can be altered by use of the TOLERANCE statement. AFCTOL has a default value of 1E-20 and is only altered by the TOLERANCE statement if the value specified is less than the default value. In the discussion which follows the "function value" is half the sum of the squared residuals computed using the current parameter estimates.

"Relative function convergence" is reported if the predicted maximum possible function reduction is at most RFCTOL*ABS(F0) where F0 is the function value at the start of the current iteration, and if the last step attempted achieved no more than twice the predicted function decrease.

"Absolute function convergence" is reported if the function value is less than AFCTOL.

Hints for NLREG Use

Convergence Failures

One of the potential problems that confronts any nonlinear minimization procedure is non-convergence. Non-convergence is usually not a problem for regressions using a linear model, but becomes a more serious consideration when using complicated nonlinear functions; increasing the number of parameters aggravates the problem.

Non-convergence can occur in two ways: the solution may diverge or it may converge to the wrong solution -- a local minimum rather than the global minimum. Periodic functions, such as sin, and cos, are particularly prone to convergence problems. For example, consider a nonlinear regression performed with the function:

```
y = offset + amplitude * sin(frequency * x)
```

where *x* and *y* are variables, and *offset*, *amplitude*, and *frequency* are the parameters whose values are to be determined. If the starting value for *frequency* is not reasonably close to the correct value, the solution may converge to a harmonic (multiple) or sub-harmonic (fundamental) value of the frequency. A program file named SINE.NLR is supplied with the statements and data to perform this analysis.

If NLREG stops with the message "False convergence" this indicates that the iterations appear to be converging to a noncritical point. This may mean that the convergence tolerances are too small for the accuracy to which the function and gradient are being computed, that there is an error in computing the gradient, or that the function or gradient is discontinuous near x.

False convergence can occur if the function is not continuous (smooth) and the computed value changes in "jumps" when small increments are made to the values of the parameters being optimized. The BULLET.NLR example discusses this in more detail.

The SWEEP statement can be very useful in cases like the sine example. In the SINE.NLR example analysis, the actual value of the frequency is 3; the function converges to the correct solution if the starting value is in the range 2.6 to 3.3. However, this example is quite insensitive to the starting value of the amplitude parameter. With an actual value of 2, the correct solution is found with starting values from 1 through 10000. Similarly, the offset parameter, which had an actual value of 10, was successfully determined with starting values ranging from 1 to over 50000.

Another example that is sensitive to a parameter starting value is POWER.NLR, which attempts to determine the values of the parameters p0, p1, and p2 for the function

```
y = p0 + p1*x^p2
```

(where "x^p2" means x raised to the p2 power). The actual value of p2 in the example data is 2; the solution converges correctly if the starting value of p2 is in the range 1.8 to 3.8. As with the other example, the solution is relatively insensitive to the starting values of p0 and p1.

Singular Matrix Problems

Another possible problem is that the analysis may stop with the message "Singular convergence. Mutually dependent parameters?". This is usually due to one of two things: (1) a redundant parameter that is co-dependent with another parameter, or (2) a situation where the value of one parameter "blocks" the effect of other parameters. As an example of a redundant parameter, consider the function

```
y = p0 + p1*p2*x
```

This is a simple linear equation except there are two parameters, *p1*, and *p2*, which are both factors to the variable *x*. It should be clear that there is no unique solution to this problem since any value of *p1* is possible if the right value of *p2* is chosen. Similarly, the function

```
y = p0 + p1 + p2*x
```

has no unique solution since either *p0* or *p1* is redundant. Similarly, in the equation

```
y = p0 + p1*exp(x+p2)
```

either *p1* or *p2* is redundant.

The second type of singular matrix problem can be illustrated by the function

```
y = p0 + p1*x^p2
```

If, during the solution process, *p1* takes on the value 0, then varying the value of *p2* has no effect on the equation, and NLREG cannot figure out which way to change the value of *p2* to move toward convergence. The solution to this problem is to assign a starting value that is not zero to *p1*, and use the CONSTRAIN statement to force *p1* to remain non-zero.

Performance Issues

NLREG is carefully programmed and compiled with an optimizing compiler for maximum performance. However, NLREG is a real "number cruncher," and the nonlinear regression algorithm is mathematically very elaborate. During each iteration, NLREG computes gradients, Jacobians, Hessians, and eigenvalues, and performs QR and Cholesky matrix decompositions. All calculations are carried out using double precision (64 bit) floating point.

Very long running times can result if you use the SWEEP statement with many starting values. The problem is compounded if you have multiple SWEEP statements. If you use the SWEEP statement to try a large number of starting parameter values, you can save time by using the ITERATIONS statement to specify a small number of iterations (such as 5) during the initial attempt to find a solution. Once a feasible set of starting parameter values has been determined, remove the SWEEP statement, specify the starting values on the PARAMETERS statement, increase the number of iterations, and rerun the analysis to get the final result.

Program Limits

The following is a summary of the NLREG program limitations:

Maximum number of variables = 500 Standard version; 2000 Advanced version.

Maximum number of parameters = 5 Standard version; 2000 Advanced version.
Maximum number of work variables = 5 Standard version; 1000 Advanced version.
Maximum length of variable or parameter names = 30
Maximum length of a program statement = 20000
Maximum length of data records = 500 Standard version; unlimited Advanced version.

The maximum number of data observations that the Advanced version of NLREG can handle is limited only by the available memory space provided by Windows. However, if you have a large number of data records, you must use the form of the DATA statement that specifies a file name and place your data records in a separate file. This is necessary because the editor built into NLREG can only handle up to 65,000 characters.

Example Analyses

A number of example regression program files are provided with your NLREG distribution. All of the example program files have the extension ".NLR". Some of the important ones are described below, others contain comment lines that explain what they do.

LINEAR.NLR -- Simple linear regression with plotted function and data.

QUAD.NLR -- Fit a quadratic equation. Plot the function and the data.

ASYMPTOT.NLR -- Fit an asymptotic function Y = 12 - 10/X.

AIDS.NLR -- A logistic curve is a growth curve used to model functions which increase gradually at first, more rapidly in the middle growth period, and slowly at the end, leveling off at a maximum value after some period of time. This type of curve is frequently used to model biological growth patterns where there is an initial exponential growth period followed by a leveling off as more of the population is infected or as the food supply or some other factor limits further growth. The form of the symmetric logistic growth function is:

```
y = k / (1 + exp(a + b*x))
```

where *k*, *a*, and *b* are parameters that shape and scale the function. The value of *b* is negative.

The AIDS.NLR example fits a logistic curve to the number of new cases of AIDS reported in the United States during the period 1981 through 1992. The computed function fits the data remarkably well showing that the AIDS infection rate is following a classic logistic curve and should level off at about 47,500 new cases per year (in the United States). The DOMAIN option on the PLOT statement causes NLREG to extrapolate the plot of the function through 1995. Note, because of recent advances in drugs that greatly slow the progression of AIDS, the morbidity curve does not follow its natural form through recent years.

BARO5.NLR -- The BARO5 example fits a 5-parameter logistic (sigmoid) function to data relating blood pressure to heart rate.

F33.NLR -- Multivariate linear regression (multiple regression). Calculate the value of a used Beech F33 Bonanza airplane using a linear model based on its age, the number of hours on its airframe, and the number of hours on its engine. The t value and Prob(t) indicate that the number of hours on the engine (*Engdep* parameter) is not significant to the regression model; the other parameters are significant but airframe hours is less significant than the base price and age of the plane.

F33YEAR.NLR -- Similar to F33.NLR except the price of the Bonanza is calculated based on a linear function of only the age.

F33EXP.NLR -- Similar to F33YEAR.NLR except a negative exponential function is used rather than a linear function. Compare the fit of this model with that of the F33YEAR.NLR example.

SINE.NLR -- Fit an equation involving a sin function. The SWEEP statement is used to find a starting point that will converge.

TREND.NLR -- Fit a function that has a linear growth term and a periodic component involving a sin term.

PENDULUM.NLR -- The periodic motion of a pendulum damped by friction can be described by: X = A*exp(-alpha*Time)*cos(w*(Time-Phase))+beta. This example fits this function to the actual measured position of a pendulum at various times.

SQUARE.NLR -- Fit a sine series to a square wave. Note in this example that the *p0* parameter, which represents the constant term of the equation, has an estimated value of 9.22715E-006 (very nearly zero) and a standard error of 0.0398754. This yields a t value of nearly zero and Prob(t) of 0.99982 which means that there is a 99.982% chance that the actual value of *p0* may be zero (it is in fact zero). This illustrates how you can use the t value and Prob(t) to identify extraneous parameters.

COOLING.NLR -- Fit an equation involving an exponential function. If a heated object is allowed to cool, the rate of cooling at any instant is proportional to the difference between the object's temperature and the ambient (room) temperature. In other words, an object cools faster at first, while it is hot, and the rate of cooling slows down as the temperature of the object approaches the ambient temperature. The function that relates the object's temperature to time is:

```
Temperature = Roomtemp+InitTemp*exp(-Coolrate*Time)
```

Where *InitTemp* is the number of degrees above room temperature at time 0, and *Coolrate* is a factor that depends on the mass of the object, how well it is insulated, etc. The exp function is the value of *e* (2.7182818...) raised to a power. The COOLING.NLR example determines the parameters *InitTemp* and *Coolrate* to fit an equation of this form to some data the author collected.

BOIL.NLR -- The boiling point of water decreases as the pressure in the vessel containing the water decreases. "Clapeyron's equation" shows that the boiling point is related to pressure according to the following function:

```
Temperature = b / log(Pressure/a) - 459.7
```

Where *Temperatue* is in degrees Fahrenheit (the 459.7 constant converts degrees Fahrenheit to degrees Rankine -- relative to absolute zero), *Pressure* is the pressure in the vessel in pounds per square inch, and *a* and *b* are parameters whose values are to be determined. The data for this example was collected by the author's son for a science project.

MAGNET.NLR -- Fit a function involving an arc tangent and a variable to the third power. This is an interesting physics problem. If a magnet is placed due east of a compass, the deflection of the compass needle from north is approximately equal to the arc tangent of

the ratio of the strength of the magnet's field relative to the earth's magnetic field. The strength of the magnet's field at the compass is inversely proportional to the cube of the distance from the magnet to the compass. Thus, the function relating these terms is

```
Deflection = atand(Strength / Distance ^ 3)
```

The atand() function computes the arc tangent of a value in degrees. In the example, Deflection and Distance are the variables, and the value of the Strength parameter is determined.

DIODE.NLR -- The current through a diode increases sharply as the voltage across the diode is increased. An equation that approximates the current flow as a function of the voltage is:

```
I = exp(b*(V-c))
```

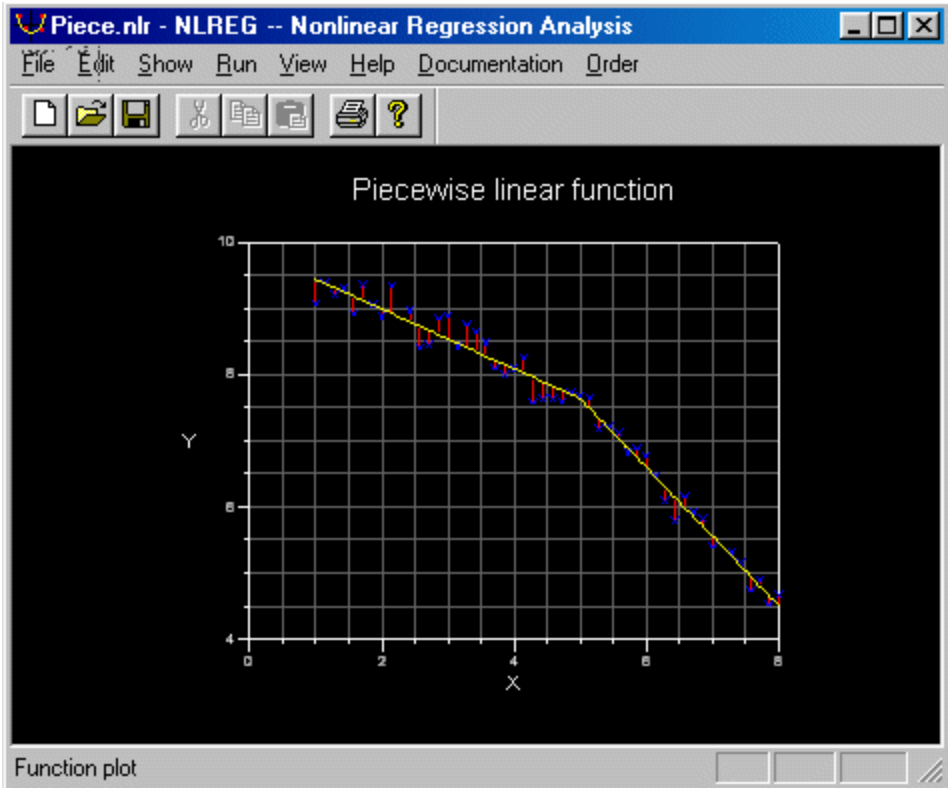
where *I* is the current, *V* is the voltage, and *b*, and *c* are parameters that are to be estimated by the nonlinear regression.

AVLTIME.NLR -- An AVL tree is a balanced binary tree used to store information in a computer's memory. Because the entries in an AVL tree are kept in sorted order, and the tree is kept in a balanced form, it is possible to rapidly find any entry in the tree. The time required to create an AVL tree with *N* entries is approximately equal to:

```
Time = a + b*N*log2(N)
```

where *a* is a constant term equal to the overhead involved in starting and completing a tree creation, and *b* is a growth coefficient that depends on the speed of the computer. The log2(*N*) function is the log base 2 of *N* (the number of entries). The AVLTIME.NLR example fits an equation to a data set that relates the time in seconds required to create an AVL tree with the number of entries in the tree.

PIECE.NLR -- Piecewise linear function. Fit a function consisting of two linear pieces that bend at X=5. When X is less than 5, the slope of the function is *B1*. When X is greater than or equal to 5, the slope is *B2*. *B0* is the Y value of the function at X=5 (i.e., at the pivot point). The IF statement is used to control which function model is used depending on whether the value of the dependent variable is greater than or less than the pivot point. Here is a plot of the piecewise regression function fitted to the data:


Piece.nlr - NLREG -- Nonlinear Regression Analysis
File Edit Show Run View Help Documentation Order
Piecewise linear function
10
8
6
4
Y
0
2
4
6
8
X
Function plot

Special Applications

Fitting the Integral of a Function

In some nonlinear regression applications, it may be necessary to fit data to the integral of a function rather than to the function itself. For example, the dependent variable might correspond to the area under a Gaussian distribution between lower and upper X values, so the function being fitted is the integral of the Gaussian distribution rather than the Gaussian distribution itself.

To handle these situations, NLREG provides an **Integral** built-in, library function that computes the numerical integral of a function or expression using Romberg's method. Parameters whose values are being computed by NLREG may be used in expressions for the lower and upper limits of the integration interval, and they may be used in the expression that is being integrated.

The form of the Integral function is:

Integral(*variable*, *LowLimit*, *HighLimit*, *expression*[, *tolerance*])

The Integral function computes $\int_{variable=LowLimit}^{variable=HighLimit} expression$

Variable is the name of a work variable that has been declared earlier in the program using a DOUBLE statement. The expression is integrated over this variable. *Variable* must consist only of a single variable name – not an expression or constant – and the variable must be a work variable, not a parameter or the dependent or independent variable.

LowLimit is the lower limit of the integration range, and *HighLimit* is the upper limit of the integration range. Full expressions may be used to specify the low and high limits. These expressions may include operators, functions, parameters whose values are being calculated and dependent variables.

Expression is the expression or function whose integral is to be computed. It may contain parameters and dependent variables. Usually it will contain the variable declared by the first argument (*variable*).

Tolerance is an optional argument that may be omitted; it specifies the accuracy to which the integral will be computed. If you omit the *tolerance* parameter, a default value of 1E-8 is used. If you specify the *tolerance* parameter, its value must be in the range 1E-3 to 1E-14. As you decrease the tolerance value (i.e., approach 1E-14), the accuracy of the integral increases, and the computation time also increases – sometimes dramatically.

Here is an example NLREG program that fits the area under a Gaussian (normal distribution) curve to a set of data values. The independent variable, *x*, is the upper range of the region whose area corresponds to the value of the dependent variable, *y*. That is, the value of *y* corresponds to the area under the Gaussian from 0 up to *x*. Two parameters are calculated: *mean* – the mean value of the Gaussian distribution, and *StdDev* – the standard deviation of the distribution. The **npd** built-in, library function (see separate description) is used to compute the value of the Gaussian with a specified mean and standard deviation at a specified x value. The three

parameters for npd are the x coordinate along the curve, the mean of the curve (i.e., center point of the distribution) and the standard deviation.

```
Variables x, y;
Parameters Mean, StdDev;
Double t;
Function y = Integral(t, 0, x, npd(t,Mean,StdDev));
Data;
```

In this example, the '*t'* work variable ranges from 0 to x as the integral is computed; so the value of npd(t,Mean,StdDev) is the height of the Gaussian for each value of *t* as *t* is swept across the interval (0,x); the Integral function computes the area under the curve across that interval. Symbolically, this corresponds to:

$$y = \int_{t=0}^{t=x} \text{npd}(t, Mean, StdDev)$$

Omitted Dependent Variable

There is a class of nonlinear regression problems that can be best expressed by omitting the dependent variable (i.e., the variable on the left of the equal sign). To understand what this means, first consider the normal regression case with a dependent variable. For each observation the function is evaluated and the computed value is subtracted from the corresponding value of the dependent variable for that observation. This residual value is then squared and added to the other squared residual values. The goal is to minimize the total sum of squared residuals. In the case where the dependent variable is omitted, the function is computed for each observation and the value of the function is squared (i.e., it is treated as the residual) and added to the other squared values. The goal is to minimize the sum of the squared values of the function. Thus, for a perfect fit the computed value of the function for every observation would be zero.

To perform this type of analysis, omit the dependent variable and equal sign from the left side of the function specification.

As an example of this type of analysis consider the problem of fitting a circle to a set of points that form a roughly circular pattern (i.e., a "circular regression"). Our goal is to determine the center point of the circle (*Xc,Yc*) and the radius (*R*) which will make the circle best fit the points so that the sum of the squared distances between the points and the perimeter of the circle is minimized (the points are as close to the perimeter of the circle as possible).

For this problem, we have three parameters whose values are to be determined: *Xc*, *Yc*, and *R*. There will be one data observation for each point to which the circle is being fitted. For each point there are two variables, *Xp* and *Yp*, the X and Y coordinates of the point's position.

Since our goal is to minimize the sum of the squared distances from the points to the perimeter of the circle, we need a function that will compute this distance for each point. If the center of the circle is at (*Xc,Yc*) and the position of a point is (*Xp,Yp*) then, from the theorem of Pythagoras, we know the distance from the center to the point is

```
sqrt((Xp-Xc)^2 + (Yp-Yc)^2)
```

But we are interested in the distance from the perimeter to the point. Since the radius of the circle is *R*, the distance from the perimeter to the point (along a straight line from the center to the point) is

```
sqrt((Xp-Xc)^2 + (Yp-Yc)^2) - R
```

That is, the distance from the perimeter to the point is equal to the distance from the center to the point less the distance from the center to the perimeter (the radius). The distance will be positive or negative depending on whether the point is outside or inside the circle but this does not matter since the value is squared as part of the minimization process.

The NLREG statements for this analysis are as follows:

```
Variables  Xp,Yp;
Parameters Xc,Yc,R;
Function   sqrt((Xp-Xc)^2 + (Yp-Yc)^2) - R;
```

Note that there is no dependent variable or equal sign to the left of the function. NLREG will determine the values of the parameters *Yc*, *Yc*, and *R* such that the sum of the squared values of the function (i.e., the sum of the squared distances) is minimized. The CIRCLE.NLR file contains a full example of this analysis.

As a second example similar to the first one, consider a town that is trying to decide where to place a fire station. The location should be central such that the sum of the squared distances from the station to each house is minimized. NLREG can be used to determine the coordinates of the station (*Xc,Yc*) given a set of coordinates for each house location (*Xh,Yh*) by using a slightly simpler function than the first example:

```
Function  sqrt((Xh-Xc)^2 + (Yh-Yc)^2);
```

Another example is **orthogonal linear regression**. In ordinary linear regression, the goal is to minimize the sum of the squared *vertical* distances between the *y* data values and the corresponding *y* values on the fitted line. In orthogonal linear regression the goal is to minimize the *orthogonal* (perpendicular) distances from the data points to the fitted line.

The slope-intercept equation for a line is:

Y = m*X + b

where *m* is the slope and *b* is the intercept.

A line perpendicular to this line will have –(1/*m*) slope, so the equation will be:

Y' = -X/m + b'

If this line passes through some data point (*X0,Y0*), its equation will be:

Y' = -X/m + (X0/m + Y0)

The perpendicular line will intersect the fitted line at a point (X*i,Yi*) where *Xi* and *Yi* are defined by:

Xi = (X0 + m*Y0 – m*b) / (m^2 + 1)
Yi = m*Xi + b

So the orthogonal distance from (*X0,Y0*) to the fitted line is the distance between (*X0,Y0*) and (*Xi,Yi*) which is computed as:

distance = sqrt((X0-Xi)^2 + (Y0-Yi)^2)

So the goal of the NLREG program is to minimize the sum of these orthogonal distances. Here is a NLREG program that does this:

```
Title "Fit a line to data points minimizing orthogonal distances";
Variables X0, Y0;
Parameters m, b;
Double Xi, Yi, distance;
Xi = (X0 + m*Y0 - m*b) / (m^2 + 1);
Yi = m*Xi + b;
distance = sqrt((X0-Xi)^2 + (Y0-Yi)^2);
Function distance;
Data;
```

Root Finding and Expression Minimization

Although it is designed for nonlinear regression analysis, NLREG can also be used to find the root (zero point) or minimum absolute value of a nonlinear expression. To use NLREG in this fashion follow these steps:

- Do not use any VARIABLE statements.
- Use PARAMETER statements to specify the names and optional starting values for the parameters whose values are to be determined as the roots or minimum value of the expression.
- Use the FUNCTION statement to specify the expression whose roots or minimum value is to be found; do NOT specify a dependent variable and equal sign -- specify only the expression that is to be minimized.
- Do not include any data records after the DATA statement; it simply signals the end of the program file and causes the analysis to begin.

The following is an example program file to find the root of the expression sin(x)-log(x):

```
Parameter x;
Function sin(x) - log(x);
Data;
```

Notice that the "variable" in the expression, X, is not declared to be a variable but rather a parameter. This example is included in the file MINSL.NLR that you can run.

For this type of analysis, NLREG determines the values of the parameters that minimize the absolute value of the expression. If the expression has a zero value (i.e., a root), that value is found since that is the smallest possible absolute value. If the expression does not have a zero point, NLREG determines the values of the parameters that produce the smallest absolute value of the expression. For example, the expression 2*X^2-3*X+10 does not have a root but reaches a

minimum value of 8.875 when X is 0.75. The MINPAROB.NLR program file contains this example.

There are a number of cautions that you should keep in mind when using NLREG to find roots or minimum values:

- NLREG will find only one root or minimum value per analysis. For example, the expression $9-X^2$ has two roots: -3 and +3. NLREG will find one of the roots; which one it finds depends on the starting value specified for X.
- NLREG will find only real roots, not complex.
- If the expression contains a local minimum, NLREG may find it rather than the global minimum or root. Of course, if you are looking for a local minimum in a certain region this could be considered a feature. For example, the expression $0.5*X^3+5*(X-2)^2+15$ has a local minimum at X=1.61 and a root at X=-13.38. If the starting value of X is less than -8.3 the root is found; if the starting value is greater than -8.3, the local minimum is found. The SWEEP statement can be used to try multiple starting values when searching for a global minimum.

Function Minimization Examples

MINFALL.NLR -- The time taken for an object to slide down a frictionless guide from position (0,*h*) to another position (*d*,0) (i.e., falling through a distance *h* while moving horizontally a distance *d*) depends on the path that the object takes as it follows the guide. It turns out that the path that minimizes the descent time is not a straight line from (0,*h*) to (*d*,0) but rather a curve called a brachistochrone with a steeper slope near the beginning, that gives the object a chance to accelerate quickly, and then a shallower slope further on.

Finding the shape of this curve is a classic problem in the branch of mathematics called the Calculus of Variations. The MINFALL example solves a simpler case of this problem: the object slides along a straight guide from (0,1000) to an intermediate position (*px*,*py*), and then along another straight guide from (*px*,*py*) to (1000,0). What point, (*px*,*py*), minimizes the descent time?

Note concerning the answer: The fall time for the object if it follows a straight guide from (0,1000) to (1000,0) is 2.0203 seconds; the fall time if it follows the two straight segments found by MINFALL is 1.8748; the fall time if it follows the ideal curved brachistochrone is 1.8590. The speed of the object at the end of the fall is the same regardless of the path taken due to conservation of energy.

MINFUEL.NLR -- A lunar lander is hovering above the surface of the moon looking for a suitable landing site. Available fuel is critical and the desired site is 200 meters away. How long should the horizontal thruster be fired to start and stop the motion over the ground? The vertical thruster must be used continuously to keep the lander from being pulled to the surface. If too little horizontal thrust is used, the spacecraft will move slowly and much fuel will be consumed by the vertical thruster counterbalancing the downward gravitational pull while hovering over the surface. On the other hand, if the horizontal thruster is fired for a long time, the spacecraft will move quickly (minimizing the hovering time) but excessive fuel will be used during the horizontal acceleration and deceleration. MINFUEL.NLR determines how long the thruster should be fired during

the start and stop accelerations such that the total fuel consumption (start thrust + stop thrust + hover) is minimized.

BULLET.NLR -- A bullet is fired from a gun: what angle of elevation of the gun will result in the bullet traveling the maximum distance before hitting the ground? If you ignore the effects of air resistance, the bullet travels in a parabolic arc and it is easy to compute that the optimum firing angle is 45 degrees. However, if you consider the effects of air resistance this becomes a much more difficult problem -- one for which there is no simple analytical solution. The BULLET.NLR example uses NLREG to compute the firing angle that optimizes the horizontal distance traveled, taking into consideration the effects of air resistance. Since there is no analytical function to compute the distance as a function of the angle, the NLREG procedure uses an iterative procedure to simulate the flight of the bullet and determine the horizontal distance traveled for a specified angle. This procedure uses the "finite differences" method (also called "Euler's method") to compute the effects of gravitational acceleration and air resistance forces on the bullet over short steps of time. For each angle, the procedure iterates until the bullet hits the ground and then determines the horizontal distance traveled at the point of impact.

If you study the example, you will see that the force due to air resistance is proportional to the square of the speed. In this example, the initial muzzle velocity was chosen to be 300 meters/second. The air resistance coefficient was selected so that the bullet would have a terminal velocity (if dropped from a very high height through the air) of about 90 meters/second. If you set the air resistance coefficient to 0 (which has the effect of ignoring air resistance) you will find that the computed optimum angle is about 45 degrees . But with air resistance, the optimum angle is about 32 degrees.

This type of optimization, using a function that is computed through simulated steps, can be tricky to get to converge. The problem is that the function is not continuous (smooth) but rather is computed using finite steps. A small change in the angle can cause the bullet to hit the ground during different steps of the simulation producing "jumps" in the distance computed as a function of the angle. This causes problems for NLREG's optimization procedure that is using the change in distance to guide its choice of trial angles while searching for convergence. If these jumps are severe, NLREG will stop with the message "False convergence". In order for the bullet example to converge correctly it is necessary to use very small step sizes and to perform an interpolation on the point of impact using the last position above ground and the final position which may be below ground.

If you have a slow computer the example will take a long time to run since it must simulate the flight of the bullet through small time steps while searching for the optimum angle.

Command-line Version of NLREG

NLREGCA is the command-line version of the NLREG nonlinear regression program.

Installation: NLREGCA.EXE will be installed into the same directory where the interactive version of NLREG is installed (normally, c:\program files\NLREG). You should either copy NLREGCA.EXE into a directory that is part of your PATH search list, or modify your PATH to include the NLREG installation directory (or you could explicitly run it from its directory).

Use: Create a NLREG program file with the same commands that you would use for the interactive version of NLREG (see the NLREG reference manual). It is best to give the program file the extension ".nlr". The following commands may *not* be used with the command-line version: PLOT, SPLOT, RPLOT, NPLOT.

Use the following statement to execute the program file:

NLREGCA *program_file* [*options*]

Where *program_file* is the name of the program file. If you do not specify a file extension, ".nlr" is assumed by default.

The following options may be specified on the command line after the name of the program file:

/list *file_name* – Create a listing file with the specified name. If this option is not specified, NLREGCA will create a listing file that has the same name as the program file but with the extension ".lst".

/poutput *file_name* – Write the computed values of the parameters to a file whose name is specified. This option has the same effect as using the POUTPUT statement within the program file. The computed value of each parameter is written on a separate line to the output file.

Here is an example command line that runs a program file named "aids.nlr", writes the listing to "aids.lst" and writes the parameters to "aids.par":

NLREGCA aids /list aids.lst /poutput aids.par

DLL and COM Version of NLREG

DLL (Dynamic Link Library) and COM object versions of NLREG are available as separate products. They can be called from applications to allow NLREG to be used as a nonlinear regression "engine" that operates behind the scenes.

The COM object version is best for use with applications written in Visual Basic and ASP. The traditional (non-COM) .dll is easier to call from C++ programs because it uses natural data types rather than variants.

NLREG.dll has entry points 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.

For information about NLREG.dll, please see the web page: http://www.nlreg.com or contact the author. Quantity discounts are available where NLREG.dll is being distributed with an application.

Here is a complete example Visual Basic program that calls the NLREG DLL COM Library:

```
Private Sub Command1_Click()
'
'  Reference the NLREG COM library
'
Dim nlreg As NLREGCOMLib.nlreg
Set nlreg = New NLREGCOMLib.nlreg
'
'  Misc. data declarations.
'
Dim ProgramSource As String
Dim Report As String
Dim VarName As String
Dim Status As Long
Dim NumVar As Long
Dim Xindex, Yindex As Long
Dim Aindex, Bindex As Long
Dim Avalue, Bvalue As Double
Dim Index As Long
Dim PredictedY, Rsquared As Double
'
'  Initialize the NLREG library.
'
nlreg.Initialize (0)
```

```
'
'  Compile the NLREG program.
'  Note: If the program is in a file, use CompileFile().
'
ProgramSource = "Variables x,y; Parameters a,b; Function y=a+b*x;
data;"
Status = nlreg.Compile(ProgramSource)
If (Status <> 1) Then
    ' Some sort of compile error occurred.
    Report = nlreg.GetAnalysisReport(0)
    Stop
End If
'
'  Check information about the variables.
'
NumVar = nlreg.NumInputVariables()
For Index = 0 To NumVar - 1 Step 1
    VarName = nlreg.InputVariableName(Index)
Next
'
'  Get index numbers for the input variables.
'
Xindex = nlreg.InputVariableFind("x")
Yindex = nlreg.InputVariableFind("y")
'
'  Provide a set of x,y data values.
'
Status = nlreg.SetDataRows(5)
Status = nlreg.SetDataValue(0, Xindex, 1#)
Status = nlreg.SetDataValue(0, Yindex, 3#)
Status = nlreg.SetDataValue(1, Xindex, 2#)
Status = nlreg.SetDataValue(1, Yindex, 5#)
Status = nlreg.SetDataValue(2, Xindex, 3#)
Status = nlreg.SetDataValue(2, Yindex, 7.1)
Status = nlreg.SetDataValue(3, Xindex, 4#)
Status = nlreg.SetDataValue(3, Yindex, 8.5)
Status = nlreg.SetDataValue(4, Xindex, 5#)
Status = nlreg.SetDataValue(4, Yindex, 11.2)
'
'  Perform the regression analysis.
'
Status = nlreg.Compute()
If (Status < 0) Then
    ' Some sort of error occurred during the analysis.
    Report = nlreg.GetAnalysisReport(0)
    Stop
End If
'
'  Get the report from the analysis.
'
Report = nlreg.GetAnalysisReport(0)
```

```
'
'  Get the computed values of the 'a' and 'b' parameters.
'
Aindex = nlreg.ParameterFind("a")
Bindex = nlreg.ParameterFind("b")
Avalue = nlreg.ParameterValue(Aindex, 0)
Bvalue = nlreg.ParameterValue(Bindex, 0)
'
'  Use the fitted function to predict some y values given x values.
'
Status = nlreg.SetEvaluationValue(Xindex, 5#)
PredictedY = nlreg.EvaluateFunction(0)
Status = nlreg.SetEvaluationValue(Xindex, 6.5)
PredictedY = nlreg.EvaluateFunction(0)
'
'  Get the proportion of variance explained (R^2) for the model.
'
Rsquared = nlreg.GetStatistic(6)
'
'  End of program
'
End Sub
```

Acknowledgement and Use of NLREG

Acknowledgement

The nonlinear regression algorithm used by NLREG was published in *ACM Transactions on Mathematical Software* 7,3 (Sept. 1981) "Dennis, J.E., Gay, D.M., and Welsch, R.E. -- *An Adaptive Nonlinear Least-Squares Algorithm.*"

Use and Distribution of NLREG

There are two versions of the NLREG program: demonstration and registered. You are welcome to make copies of the demonstration version of NLREG and pass them on to friends or post this program on bulletin boards or distribute it via disk catalog services, CD ROMS, or other means provided the entire NLREG distribution is included in its original, unmodified form.

As a demonstration product, you are granted a no-cost, trial period of 30 days during which you may evaluate NLREG. If you find NLREG to be useful, educational, and/or entertaining, and continue to use it beyond the 30 day trial period, you are required to compensate the author by purchasing the program.

In return for registering, you will be authorized to continue using NLREG beyond the trial period on a single computer system. Contact the author for information about licensing NLREG on multiple computer systems. Your registration fee will be refunded if you encounter a serious bug that cannot be corrected.

The registered version of NLREG may *not* be redistributed or used on more than one computer system.

Copyright Notice

Both the NLREG program and documentation are copyright © 1991-2004 by Phillip H. Sherrod. You are not authorized to modify the program or documentation. "NLREG" is a trademark of Phillip H. Sherrod.

Web page

Up-to-date information about NLREG can be found on the web page: http://www.nlreg.com

Disclaimer

This software and documentation are provided on an "as is" basis. This program may contain "bugs" and inaccuracies, and its results should not be assumed to be correct unless they are verified by independent means. Phillip H. Sherrod disclaims all warranties relating to this software, whether expressed or implied, including but not limited to any implied warranties of merchantability or fitness for a particular purpose. Neither Phillip H. Sherrod nor anyone else who has been involved in the creation, production, or delivery of this software shall be liable for any indirect, consequential, or incidental damages arising out of the use or inability to use such

software, even if Phillip H. Sherrod has been advised of the possibility of such damages or claims. The person using the software bears all risk as to the quality and performance of the software.

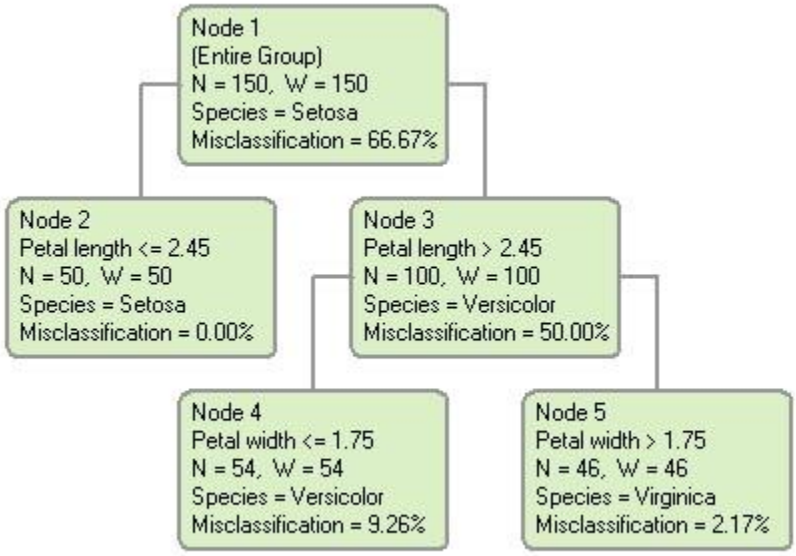
This agreement shall be governed by the laws of the State of Tennessee and shall inure to the benefit of Phillip H. Sherrod and any successors, administrators, heirs and assigns. Any action or proceeding brought by either party against the other arising out of or related to this agreement shall be brought only in a state or federal court of competent jurisdiction located in Williamson County, Tennessee. The parties hereby consent to in personam jurisdiction of said courts.

DTREG Decision Tree Generation Program

If you are trying to model data that contains categorical variables such as sex, race, marital status, etc. you may want to consider creating a decision tree to model the data rather than trying to fit a mathematical function to it.

The DTREG program, by the same author as NLREG, analyzes data and generates a decision tree to model the data. For information about DTREG, please see **http://www.dtreg.com**.

Here is an example of a decision tree generated by DTREG:



The DTREG program analyzes ("mines") a set of data values and generates a decision tree that can be used to predict the value of a "**target variable**" based on the values of a set of "**predictor variables**". Like a real tree, a decision tree has a "root", "branches" and "leaves". A prediction is made by entering the tree at the root and following the branches left or right based on values of the predictor variables until a leaf is reached. Each leaf shows the most likely value for the target variable given the set of predictor values that led to the leaf. (See the example tree above.)

Decision trees have a number of advantages over competing procedures:

- Decision trees are easy to build. Just feed a dataset into DTREG, and it will do all the work of building a decision tree and pruning it to the optimal size.
- Decision trees are easy to understand. Unlike nonlinear regression models, or, worse, neural networks, decision trees provide a clear, logical representation of the data model. They can be understood and used by people who do not have to be mathematicians and statisticians.
- Decision trees handle both continuous and categorical variables. Categorical variables such as gender, race, religion, marital status and geographic region are difficult to model using numerically-oriented techniques such as regression and neural networks. In

contrast, categorical variables are handled easily by decision trees. DTREG uses advanced techniques that enable it to build classification trees even when there are a large number of predictor categories.

- Decision trees can perform classification as well as regression. The predicted value from a decision tree is not simply a numerical value but can be a predicted category such as male/female, malignant/benign, frequent buyer/occasional buyer, etc.
- DTREG accepts text data as well as numeric data. If you have categorical variables with data values such as "Male", "Female", "Married", "Tennessee", "Protestant", etc., there is no need to code them as numeric values.
- Decision trees automatically handle interactions between variables. For example, men in the North may have different characteristics than men in the South; likewise, there may be differences between southern women and northern women. So it is necessary to consider both gender and region when making a prediction. Decision trees automatically deal with these interactions by partitioning the cases and then analyzing each group.
- Decision trees identify important variables. By examining which variables are used to split nodes near the top of the tree, you can quickly determine the most important variables. DTREG carries this further by analyzing all the splits generated by each variable and the selection of surrogate splitters. A table ranking overall variable importance is included in the analysis report. The determination of variable importance is particularly useful in studies where many variables are available (for example, demographic data).
- Decision trees handle missing data values well. DTREG uses a sophisticated technique involving "surrogate splitters" to handle cases with missing values. This allows cases with some available values and some missing values to be utilized to the maximum extent when building models. It also enables DTREG to predict values for cases with missing data.
- Decision trees do not require the specification of the form of a function to be fitted to the data as is required by nonlinear regression.

Index

A

B

C

D

E

F

G

H

I

J

S

T

U

V

W

X

Y

Z