Salford Predictive Modeler®

# Guide to the BASIC Programming Language

*This guide provides an overview of the built-in BASIC programming language available within SPM®.*

Minitab

# BASIC Programming Language

Salford Predictive Modeler® (SPM) contains an integrated implementation of a complete BASIC programming language for transforming variables, creating new variables, filtering cases, and database programming. Because the programming language is directly accessible anywhere in SPM, you can perform a number of database management functions without invoking the data step of another program.

The BASIC transformation language allows you to modify your input files on the fly while you are in an analysis session. Permanent copies of your changed data can be obtained with the RUN command, which does no modeling. BASIC statements are applied to the data as they are read from your dataset and before any modeling takes place, allowing variables created or modified by BASIC to be used in the same manner as unmodified variables on the input dataset.

Although this integrated version of BASIC is much more powerful than the simple variable transformation functions sometimes found in other statistical procedures, it is not meant to be a replacement for more comprehensive data steps found in statistics packages in general use. At present, integrated BASIC does not permit the merging or appending of multiple files, nor does it allow processing across observations. In SPM the programming work space for BASIC is limited and is intended for on-the-fly data modifications of 20 to 40 lines of code. For more complex or extensive data manipulation, we recommend you use your preferred database management software.

The remaining BASIC help topics describe what you can do with BASIC and provide simple examples to get you started. The BASIC help topics provide formal technical definitions of the syntax.

# Getting Started with BASIC Programming Language

Your BASIC program will normally consist of a series of statements that all begin with a “%” sign. (The “%” sign can be omitted inside a "DATA block" described later.) These statements could comprise simple assignment statements that define new variables, conditional statements that delete selected cases, iterative loops that repeatedly execute a block of statements, and complex programs with the flow control provided by GOTO statements and line numbers. Thus, somewhere before a model analysis command such as CART GO, STATS or RUN, you might type:

```
% LET BESTMAN = WINNER

    % IF MONTH=8 THEN LET GAMES = BEGIN
    % ELSE IF MONTH>8 LET GAMES = ENDED
    % LET ABODE = LOG (CABIN)
    % DIM COLORS(10)
    % FOR I= 1 TO 10 STEP 2
    % LET COLORS(I) = Y * I
    % NEXT
    % IF SEX$="MALE" THEN DELETE
```

The % symbol appears only once at the beginning of each line of BASIC code; it should not be repeated anywhere else on the line. You can leave a space after the % symbol or you can start typing immediately; BASIC will accept your code either way.

Our programming language uses standard statements found in many dialects of BASIC.

# BASIC: Overview of BASIC Components

## LET

Assigns a value to a variable. The form of the statement is:

```
% LET variable = expression
```

## IF...THEN

Evaluates a condition, and if it is true, executes the statement following the THEN. The form is:

```
% IF condition THEN statement
```

## ELSE

Can immediately follow an IF...THEN statement to specify a statement to be executed when the preceding IF condition is false. The form is:

```
% IF condition THEN statement
% ELSE statement
```

Alternatively, ELSE may be combined with other IF–THEN statements:

```
% IF condition THEN statement

    % ELSE IF condition THEN statement
    % ELSE IF condition THEN statement
    % ELSE statement
```

## FOR...NEXT

Allows for the execution of the statements between the FOR statement and a subsequent NEXT statement as a block. The form of the simple FOR statement is:

```
% FOR

    % statements
    % NEXT
```

For example, you might execute a block of statements only if a condition is true, as in

```
%IF WINE=COUNTRY THEN FOR

    %LET FIRST=CABERNET
    %LET SECOND=RIESLING
    %NEXT
```

When an index variable is specified on the FOR statement, the statements between the FOR and NEXT statements are looped through repeatedly while the index variable remains between its lower and upper bounds:

```
% FOR [index variable and limits]

    % statements
    % NEXT
```

The index variable and limits form is:

```
%FOR I= start-number TO stop-number [ STEP = stepsize ]
```

where I is an integer index variable that is increased from *start-number* to *stop-number* in increments of *stepsize*. The statements in the block are processed first with I = start-number, then with I = *start-number* + *stepsize*, and repeated until I >=*stop-number*. If STEP=*stepsize* is omitted, the default is to step by 1. Nested FOR–NEXT loops are not allowed.

## DIM

Creates an array of subscripted variables. For example, a set of five scores could be set up with:

```
% DIM SCORE(5)
```

This creates the variables SCORE(1), SCORE(2), –, SCORE(5).

The size of the array must be specified with a literal integer up to a maximum size of 99; variable names may not be used. You can use more than one DIM statement, but be careful not to create so many large arrays that you exceed the maximum number of variables allowed (currently 32000).

## DELETE

Deletes the current case from the data set.

## Operators

The table below lists the operators that can be used in BASIC statement expressions. Operators are evaluated in the order they are listed in each row with one exception: a minus sign before a number (making it a negative number) is evaluated after exponentiation and before multiplication or division. The "<>" is the "not equal" operator.

| | | | | | | |
|---|---|---|---|---|---|---|
| Numeric Operators | ( ) | ^ | * | / | + | - |
| Relational Operators | < | <= | <> | = | => | > |
| Logical Operators | AND | OR | NOT | | | |

## BASIC Special Variables

BASIC has five built-in variables available for every data set. You can use these variables in BASIC statements and create new variables from them. You may not redefine them or change their values directly.

| Variable | Definition | Values |
|---|---|---|
| CASE | observation number | 1 to maximum observation number |
| BOF | logical variable for beginning of file | 1 for first record in file, 0 otherwise |
| EOF | logical variable for end of file | 1 for last record in file, 0 otherwise |

## BASIC Mathematical Functions

Integrated BASIC also has a number of mathematical and statistical functions. The statistical functions can take several variables as arguments and automatically adjust for missing values. Only numeric variables may be used as arguments. The general form of the function is:

```
FUNCTION(variable, variable, ….)
```

Integrated BASIC also includes a collection of probability functions that can be used to determine probabilities and confidence level critical values, and to generate random numbers.

### Multiple-Argument Functions

| Function | Definition | Example |
|---|---|---|
| AVG | arithmetic mean | %LET XMEAN=AVG(X1,X2,X3) |
| MAX | maximum | %LET BEST=MAX(Y1,Y2,Y3,Y4,Y5) |
| MIN | minimum | %LET MINCOST=MIN(PRICE1,OLDPRICE) |
| MIS | number of missing values | |
| STD | standard deviation | |
| SUM | summation | |

**Single-Argument Functions**

| Function | Definition | Example |
|---|---|---|
| ABS | absolute value | %ABSVAL=ABS(X) |
| ACS | arc cosine | |
| ASN | arc sine | |
| ATH | arc hyperbolic tangent | |
| ATN | arc tangent | |
| COS | cosine | |
| EXP | exponential | |
| LOG | natural logarithm | %LET LOGXY=LOG(X+Y) |
| SIN | sine | |
| SQR | square root | %LET PRICESR=SQR(PRICE) |
| TAN | tangent | |

The following shows the distributions and any parameters that are needed to obtain values for either the random draw, the cumulative distribution, the density function, or the inverse density function. Every function name is composed of three letters:

Key-Letter:
This first letter identifies the distribution.

Distribution-Type Letters:
RN (random number), CF (cumulative),
DF (density), IF (inverse).

## BASIC Probability Functions

CART BASIC also includes a collection of probability functions that can be used to determine probabilities and confidence level critical values, and to generate random numbers.

The following table shows the distributions and any parameters that are needed to obtain values for the random draw, the cumulative distribution, the density function, or the inverse density function. Every function name is composed of two parts:

The "Key" (first) letter identifies the distribution ϕ.

Remaining letters define function: RN (random number), CF (cumulative), DF (density), IF (inverse).

| Distribution | Key-Letter | Random Draw (RN) | Cumulative (C) Density (D) Inverse (I) | Comments (ϕ is the probability for inverse density function) |
|---|---|---|---|---|
| Beta | B | BRN | BCF(β,p,q)<br>BDF(β,p,q)<br>BIF(ϕ,p,q) | β = beta value<br>p,q = beta parameters |
| Binomial | N | NRN(n,p) | NCF(x,n,p)<br>NDF(x,n,p)<br>NIF(ϕ,n,p) | n = number of trials<br>p = prob of success in trial<br>x = binomial count |
| Chi-square | X | XRN(df) | XCF($\chi^2$,df)<br>XDF($\chi^2$,df)<br>XIF(ϕ,df) | $\chi^2$ = chi-squared valued<br>f = degrees of freedom |
| Exponential | E | ERN | ECF(x)<br>EDF(x)<br>EIF(ϕ) | x = exponential value |
| F | F | FRN(df1,df2) | FCF(F,df1,df2)<br>FDF(F,df1,df2)<br>FIF(ϕ,df1,df2) | df1, df2 = degrees of freedom<br>F = F-value |
| Gamma | G | GRN(p) | GCF(γ,p)<br>GDF(γ,p)<br>GIF(ϕ,p) | p = shape parameter<br>γ = gamma value |
| Logistic | L | LRN | LCF(x)<br>LDF(x)<br>LIF(ϕ) | x = logistic value |
| Normal (Standard) | Z | ZRN | ZCF(z)<br>ZDF(z)<br>ZIF(ϕ) | z = normal z-score |
| Poisson | P | PRN(p) | PCF(x,p)<br>PDF(x,p)<br>PIF(ϕ,p) | p = Poisson parameter<br>x = Poisson value |
| Studentized | S | SRN(k,df) | SCF(s,k,df)<br>SDF(s,k,df)<br>SIF(ϕ,k,df) | k = parameter<br>f = degrees of freedom |

| | | | | |
|---|---|---|---|---|
| t | T | TRN(df) | TCF(t,df)<br>TDF(t,df)<br>TIF(ϕ,df) | df = degrees of freedom<br>t = t-statistic |
| Uniform | U | URN | UCF(x)<br>UDF(x)<br>UIF(ϕ) | x = uniform value |
| Weibull | W | WRN(p,q) | WCF(x,p,q)<br>WDF(x,p,q)<br>WIF(ϕ,p,q) | p = scale parameter<br>q = shape parameter |

These functions are invoked with either 0, 1, or 2 arguments as indicated in the table above, and return a single number, which is either a random draw, a cumulative probability, a probability density, or a critical value for the distribution.

We illustrate the use of these functions with the chi-square distribution. To generate 10 random draws from a chi-square distribution with 35 degrees of freedom for each case in your data set:

```
% DIM CHISQ(10)

    % FOR I= 1 TO 10
    % LET CHISQ(I)=XRN(35)
    % NEXT
```

To evaluate the probability that a chi-square variable with 20 degrees of freedom exceeds 27.5:

```
%LET CHITAIL = 1 - XCF(27.5, 20)
```

The chi-square density for the same chi-square value is obtained with:

```
%LET CHIDEN = XDF(27.5, 20)
```

Finally, the 5% point of the chi-squared distribution with 20 degrees of freedom is calculated with:

```
%LET CHICRIT = XIF(.95, 20)
```

## Missing Values

The system missing value is stored internally as the largest negative number allowed. Missing values in BASIC programs and printed output are represented with a period or dot ("."), and missing values can be generated and their values tested using standard expressions.

Thus, you might type:

```
%IF NOSE=LONG THEN LET ANSWER=.

    %IF STATUS=. THEN DELETE
```

Missing values are propagated so that most expressions involving variables that have missing values will themselves yield missing values.

One important fact to note: because the missing value is technically a very large negative number, the expression X < 0 will evaluate as true if X is missing.

BASIC statements included in your command stream are executed when a "Hot Command" such as CART GO, STATS, SCORE GO, or RUN is encountered; thus, they are processed before any model estimation or scoring is attempted. This means that any new variables created in BASIC are available for use in MODEL and KEEP statements, and any cases that are deleted via BASIC will not be used in the analysis.

## More Examples

It is easy to create new variables or change old variables using BASIC. The simplest statements create a new variable from other variables already in the data set. For example:

```
% LET PROFIT=PRICE *QUANTITY2* LOG(SQFTRENT), 5*SQR(QUANTITY)
```

BASIC allows for easy construction of Boolean variables, which take a value of 1 if true and 0 if false. In the following statement, the variable XYZ would have a value of 1 if any condition on the right-hand side is true, and 0 otherwise.

```
% LET XYZ = X1<.5 OR X2>17 OR X3=6
```

Suppose your data set contains variables for gender and age, and you want to create a categorical variable with levels for male-senior, female-senior, male-non-senior, and female-non-senior. You might type:

```
% IF GENDER$ = "" OR AGE = . THEN LET NEWVAR = .

    % ELSE IF GENDER$ = "Male" AND AGE < 65 THEN LET NEWVAR=1
    % ELSE IF GENDER$ = "Male" AND AGE >= 65 THEN LET NEWVAR=2
    % ELSE IF GENDER$ = "Female" AND AGE < 65 THEN LET NEWVAR=3
    % ELSE LET NEWVAR = 4
```

If the measurement of several variables changed in the middle of the data period, conversions can be easily made with the following:

```
% IF YEAR  > 1986 OR MEASTYPE$ = "OLD" THEN FOR

    % LET TEMP = (OLDTEMP - 32) / 1.80
    % LET DIST = OLDDIST / .621
    % NEXT
    % ELSE FOR
    % LET TEMP = OLDTEMP
    % LET DIST = OLDDIST
    % NEXT
```

If you would like to create powers of a variable (square, cube, etc.) as independent variables in a polynomial regression, you could type something like:

```
% DIM AGEPWR(5)

    % FOR I = 1 TO 5
    % LET AGEPWR(I) =  AGE^I
    % NEXT
```

## Filtering the Data Set or Splitting the Data Set

Integrated BASIC can be used for flexibly filtering observations. To remove observations with SSN missing, try:

```
% IF SSN= . THEN DELETE
```

To delete the first 10 observations, type:

```
% IF CASE <= 10 THEN DELETE
```

Because you can construct complex Boolean expressions with BASIC, using programming logic combined with the DELETE statement gives you far more control than is available with the simple SELECT statement. For example:

```
% IF AGE>50 OR INCOME<15000 OR (REGION=9 AND GOLF=.) THEN DELETE
```

It is often useful to draw a random sample from a data set to fit a problem into memory or to speed up a preliminary analysis. By using the uniform random number generator in BASIC, this is easily accomplished with a one-line statement:

```
% IF URN < .5 THEN DELETE
```

The data set can be divided into an analysis portion and a separate test portion distinguished by the variable TEST:

```
% LET TEST =  URN < .4
```

This sets TEST equal to 1 in approximately 40% of all cases and 0 in all other cases. The following draws a stratified random sample taking 10% of the first stratum and 50% of all other strata:

```
% IF DEPVAR = 1 AND URN < .1 THEN DELETE
    % ELSE IF DEPVAR<>1 AND URN < .5 THEN DELETE
```

# Randomly Partitioning Data

SPM modeling techniques make extensive use of test and holdout samples, when they are available. The learn sample consists of those records used to build the model. The test sample is typically used to evaluate the model after it has been built, and may also be used to determine a best "pruning" of the model if there are several to choose from. The holdout sample is completely independent, it has no effect on how the model is built or pruned and is used exclusively to measure how the model performs predictively.

A common technique for establishing these samples is simply to take a random percentage of the available data, say 20% (which is the default that TreeNet® uses, by the way), and set this aside as a test sample. This can be done easily, as in this example which partitions the data with 25% test and 20% holdout (with the remaining 55% as learn):

```
PARTITION TEST=0.25, HOLDOUT=0.20
```

Perhaps you instead want only 30% of females but only 10% of men to be placed in the test sample. The following example would implement this:

```
%let test = 0
%if gender$ = "Male" and urn > 0.90 then let test=1
%else if gender$ = "Female" and urn > 0.70 then let test = 1
partition sepvar = test
```

Suppose you wanted to establish a 25% test and 20% holdout partitioning but ensure that, in the course of building a series of models, records assigned to the test sample in the first model were guaranteed to be assigned to the test sample in all the models, and similarly for the holdout sample. An easy way to accomplish this is by creating a "learn/test partitioning variable" and adding it permanently to your dataset. The following example takes a uniform random draw between 0 and 1, assigns the top 25% to the test sample (lth=1), the next 20% to the holdout sample (lth=-1) and the remaining 55% to the learn sample (lth=0):

```
use "original_data.csv"
%let lth = urn
%if lth > 0.75 then let lth = 1
%else if lth > 0.55 and lth <= 0.75 then let lth = -1
%else let lth = 0
save "partitioned_data.csv"
run
```

By using the partitioned version of your dataset, you can then build a series of models across which the test and holdout samples are consistently defined (if that is important to your analysis):

```
use "partitioned_data.csv"
partition sepvar = lth
model target
cart go
treenet go
rf go
gps go
mars go
```

# DATA Blocks

A DATA block is a block of statements appearing between a DATA command and a DATA END command. These statements are treated as BASIC statements, even though they do not start with "%." Here is an example:

```
DATA

    let ranbeta1=brn(.25,.75)
    let ranbeta2=brn(.75,.25)
    let ranbin1=nrn(100,.25)
    let ranbin2=nrn(500,.75)
    let ranchi1=xrn(1)
    let ranchi2=xrn(2)
    DATA END
```

# Advanced Programming Features

Integrated BASIC also allows statements to have line numbers that facilitate the use of flow control with GOTO statements. Line numbers must be integers less than 32000, and we recommend that if you use any line numbers at all, all your BASIC statements should be numbered. BASIC will execute the numbered statements in the order of the line numbers, regardless of the order in which the statements are typed, and unnumbered BASIC statements are executed before numbered statements.

Here is an example of using the GOTO:

```
%10 IF PARTY=GOP THEN GOTO 96

    %20 LET NEWDEM=1
    %30 LET VEEP$="GORE"
    %40 GOTO 99
    %96 LET VEEP$="KEMP"
    %99 LET CAMPAIGN=1
```

## BASIC Programming Language Commands

The following pages contain a summary of the BASIC programming language commands. They include syntax usage and examples.

## DELETE Statement

*Purpose*

Drops the current case from the data set.

*Syntax*

```
% DELETE
% IF condition THEN DELETE
```

*Examples*

To keep a random sample of 75% of a data set for analysis:

```
% IF URN < .25 THEN DELETE
```

## DIM Statement

*Purpose*

Creates an array of subscripted variables.

*Syntax*

```
% DIM var(n)
```

where n is a literal integer. Variables of the array are then referenced by variable name and subscript, such as var(1), var(2), etc.

In an expression, the subscript can be another variable, allowing these array variables to be used in FOR…NEXT loop processing. See the section on the FOR…NEXT statement for more information.

*Examples*

```
% DIM QUARTER(4)
% DIM MONTH(12)
% DIM REGION(9)
```

## ELSE Statement

*Purpose*

Follows an IF...THEN to specify statements to be executed when the condition following a preceding IF is false.

*Syntax*

The simplest form is:

```
% IF condition THEN statement1
% ELSE statement2
```

The statement2 can be another IF…THEN condition, thus allowing IF…THEN statements to be linked into more complicated structures. For more information see the section for IF…THEN.

*Examples*

```
% 5 IF TRUE=1 THEN GOTO 20
% 10 ELSE GOTO 30
% IF AGE <=2 THEN LET AGEDES$ = "baby"
% ELSE IF AGE <= 18 THEN LET AGEDES$ = "child"
% ELSE IF AGE < 65 THEN LET AGEDES$ = "adult"
% ELSE LET AGEDES$ = "senior"
```

## FOR...NEXT Statement

*Purpose*

Allows the processing of steps between the FOR statement and an associated NEXT statement as a block. When an optional index variable is specified, the statements are looped through repetitively while the value of the index variable is in a specified range.

*Syntax*

The form is:

```
% FOR [index variable and limits]
% statements
% NEXT
```

The index variable and limits is optional, but if used, it is of the form

```
x = y TO z [STEP=s]
```

where x is an index variable that is increased from y to z in increments of s. The statements are processed first with x = y, then with x = y + s, and so on until x= z. If STEP=s is omitted, the default is to step by 1.

*Remarks*

Nested FOR…NEXT loops are not allowed and a GOTO which is external to the loop may not refer to a line within the FOR…NEXT loop. However, GOTOs may be used to leave a FOR...NEXT loop or to jump from one line in the loop to another within the same loop.

*Examples*

To have an IF…THEN statement execute more than one statement if it is true:

```
% IF X<15 THEN FOR
% LET Y=X+4
% LET Z=X-2
% NEXT
```

## GOTO Statement

*Purpose*

Jumps to a specified numbered line in the BASIC program.

*Syntax*

The form for the statement is:

```
% GOTO ##
```

where ## is a line number within the BASIC program.

*Remarks*

This is often used with an IF…THEN statement to allow certain statements to be executed only if a condition is met.

If line numbers are used in a BASIC program, all lines of the program should have a line number. Line numbers must be positive integers less than 32000.

*Examples*

```
% 10 GOTO 20
% 20 STOP
% 10 IF X=. THEN GOTO 40
% 20 LET Z=X*2
% 30 GOTO 50
% 40 LET Z=0
% 5O STOP
```

## IF. . . THEN Statement

*Purpose*

Evaluates a condition and, if it is true, executes the statement following the THEN.

*Syntax*

```
% IF condition THEN statement
```

An IF…THEN may be combined with an ELSE statement in two ways. First, the ELSE may be simply used to provide an alternative statement when the condition is not true:

```
% IF condition THEN statement1
% ELSE statement2
```

Second, the ELSE may be combined with an IF…THEN to link conditions:

```
% IF condition THEN statement
% ELSE IF condition2 THEN statement2
```

To allow multiple statements to be conditionally executed, combine the IF…THEN with a FOR...NEXT:

```
% IF condition THEN FOR
% statement
% statement
% NEXT
```

*Examples*

To remove outlier cases from the data set:

```
% IF ZCF(ABS((z-zmean)/zstd))>.95 THEN DELETE
```

## LET Statement

*Purpose*

Assign a value to a variable.

*Syntax*

The form of the statement is:

```
% LET variable =  expression
```

The expression can be any mathematical expression, or a logical Boolean expression. If the expression is Boolean, then the variable defined will take a value of 1 if the expression is true or 0 if it is false. The expression may also contain logical operators such as AND, OR and NOT.

*Examples*

```
% LET AGEMONTH = YEAR - BYEAR + 12*(MONTH , BMONTH)
% LET SUCCESS =(MYSPEED = MAXSPEED)
% LET COMPLETE = (OVER = 1 OR  END=1)
```

## STOP Statement

*Purpose*

Stops the processing of the BASIC program on the current observation. The observation is kept but any BASIC statements following the STOP are not executed.

*Syntax*

The form of the statement is:

```
% STOP
```

*Examples*

```
%10 IF X = 10 THEN GOTO 40
%20 ELSE STOP
%40 LET X = 15
```