Chapter 1

An Introduction to the GAMS Modeling System

1.1 Preview

In this chapter we introduce the high-level algebraic modeling language that will be used in the rest of this book to build financial optimization models. The basic elements of the language are given first, together with details on getting started with the language, and the FINLIB library of models are also discussed here.

1.2 Basics of Modeling

Optimization is concerned with the representation and solution of models. Models can be represented in a number of ways, and they can be solved using a number of methods or algorithms. The General Algebraic Modeling System, GAMS, is a system for formulating and solving optimization models. It consists of a language that allows a high-level, algebraic representation of mathematical models, and provides a set of solvers, i.e., numerical algorithms, to solve them.

Why use algebraic modeling? Small models are easy to formulate and solve. They have a simple structure and one can simply edit a file containing the model's coefficients, and then call a standard linear programming solver to solve it. In fact, in the early days of optimization, models were solved using specialized *matrix generators* that provided the necessary input files for solvers. However, as models grow larger and become more complex, they become difficult to manage and to verify using this approach. GAMS was developed in response to the need for powerful and flexible front-end tools to manage large, real-life models. Large collections of data and models are only manageable when they possess structure, and algebra provides this structure in a well-known mathematical notation.

Conceptually, a model consists of two parts: The algebraic structure and the associated data instance. The formal linear programming model

$$\underset{x \in \mathbb{R}^n}{\text{Minimize}} \quad c^{\mathrm{T}} x \tag{1.1}$$

$$\text{subject to} \quad Ax = b, \tag{1.2}$$

$$x \geq 0 \tag{1.3}$$

has the associated data *A, b, c*; see Appendix PFO-A for optimization basics. GAMS provides an algebraic framework for defining and manipulating data as well as building the models that use them. In addition to being concise and easily readable, the GAMS statement of a model is machine-independent and allows interaction with a number of solvers. Hence, it is not dependent on any particular optimizer or computer system.

The GAMS System consists of the GAMS compiler and a number of solvers. The compiler is responsible for user-interaction, by compiling and executing user commands given in a GAMS source file. A source file can implement a simple textbook problem, or it can represent a large-scale system consisting of several interrelated optimization models. The solvers are stand-alone programs that implement optimization algorithms for the different GAMS model types. The GAMS system can be called from the command line (for instance, a Unix prompt), or through the Integrated Development Environment (IDE), a windows-based environment that facilitates modeling by integrating editors with the GAMS system.

Section 1.3 gives an introduction to the GAMS language and Section 1.4 is a guide to quickly become accustomed to using GAMS. Readers who want a quick overview of GAMS modeling may start by reading Section 2.4, which contains a complete example drawn from financial planning.

1.3 The GAMS Language

Optimization models and their associated data are communicated to the GAMS system using a general-purpose language with elements of both ordinary programming languages (data declarations, control structures), and high-level modeling structures such as sets, equations, and models.

GAMS models are typically structured with the following building blocks:

1. *Sets*, which form the basis for indexation and serve as building blocks for data and model definitions.
2. *Data*, which are specified, either through direct statements (perhaps included in external files), or by calculating derived data.
3. *Variables and constraints*, which are used to define models, equations, and an objective.
4. An output section is sometimes used where the final results are calculated and presented.

In the remainder of this section we give an introduction to the elements of the language. In addition to the above-mentioned items, the main topics are *expressions*, which are used in assignment statements and in constraint declarations, and *control structures*, which lend programming language capabilities to GAMS. Readers who want to see a larger, complete modeling exercise may skip ahead to Section 2.4, and then refer back to this section for coverage of advanced features of the language when ready to embark on more substantial models.

1.3.1 Lexical conventions

A GAMS source file is an ordinary text file. The first character position on each line indicates how the line should be interpreted:

`*` (asterisk): A comment line, ignored by GAMS.

`$` (dollar sign): Indicates a compiler directive or option. A list of the most common $-controls is given in Table 1.1. Many more exist than can be covered here; consult the User's Guide (see Notes and References at end of chapter) for complete information.

Any other character indicates a model source line. Customarily such lines start with a space character.

Table 1.1: The most common $-control commands. See Section 1.3.10 for examples of the use of `$SET`, `$IF`, `$LABEL` and `$GOTO` for conditional compilation.

Command	Description
`$[ON|OFF]LISTING`	Controls the echoing of input lines to the listing file
`$[ON|OFF]SYMLIST`	Controls the complete listing of all symbols
`$[ON|OFF]TEXT`	The `$ONTEXT - $OFFTEXT` pair encloses comment lines
`$ABORT`	This option will issue a compilation error and abort the compilation
`$BATINCLUDE`	The `$BATINCLUDE` option inserts a text file plus arguments used inside the include file
`$CALL`	Passes a followed string command to the current operating system command
`$COMMENT`	Changes the start-of-line comment symbol
`$DOLLAR`	Changes the current $ symbol
`$ECHO`	The echo option allows text to be written to a file
`$EOLCOM`	Redefines the end-of-line comment symbol
`$GOTO`	This option will cause GAMS to search for a line starting with `$LABEL` id
`$IF`	The `$IF` dollar control option provides control over conditional processing
`$INCLUDE`	The `$INCLUDE` option inserts a text file
`$INLINECOM`	Redefines the in-line comment symbols
`$LABEL`	This option marks a line to be jumped to by a `$GOTO` statement
`$SET`	Defines control variables
`$STAR`	This option is used to redefine the ****marker
`$STITLE`	This option sets the subtitle in the page header of the listing file
`$TITLE`	This option sets the title in the page header of the listing file

The main lexical elements of a GAMS statement are keywords, identifiers, and operators. The example model shows keywords in all capital letters, and identifiers use a mixture of upper and lower case, but the language is not case sensitive. Identifiers consist of letters, digits, or the underscore character, "_", and must begin with a letter (in early versions of GAMS, identifiers were limited to at most 10 characters).

The GAMS examples that follow use comments of the standard kind ("*" in position 1), but in addition assume that text that starts with the sequence "#" to the end of the current line is regarded as a comment. Hence, the command

```
$EOLCOM #
```

is assumed to be in effect in all examples. C++ and Java programmers might prefer to use the more familiar

```
$EOLCOM //
```

1.3.2 Sets

The primary tool for structuring large-scale models is the *set*. In any nontrivial model we will need to use data, variables, and constraints that are indexed, and sets form the basis for such indexing.

The simplest set declarations have the form:

```
SET Time                   /  2002 * 2006 /;  ALIAS (Time,t);
SET Bonds "Bonds universe" /GOVT_1 * GOVT_4/; ALIAS(Bonds,i,j);
```

These lines declare the two sets `Time` and `Bonds`. The set `Time` contains the elements from 2002 through 2006 (the asterisk indicates filling out the intervening elements; one could have written this: `SET Time /2002, 2003, 2004, 2005, 2006/;`). Similarly, the `Bonds` set contains bonds named `GOVT_1` through `GOVT_4`.

The ALIAS statement is a convenient way to declare indices to be used in connection with sets. The code above binds the name `t` to the set `Time`, and the names `i` and `j` to the set `Bonds`. These names can henceforth be used as indices into their associated sets (and only those sets).

The text "Bonds universe" in the `Bonds` declaration is an explanatory text that GAMS outputs in the listing whenever it lists the `Bonds` set, as a help in documentation. Such texts can occur in all declarations and can be a great help when reading GAMS listings. They need not be enclosed in quotation marks, but if they aren't then they cannot contain certain characters, which can lead to quite subtle syntax errors.

Indices and indexation

Most GAMS modeling elements (data, variables, etc.) can be indexed, with up to 10 indices. For instance, a two-dimensional parameter `F` can be defined over the sets declared above as (more on data declarations in Section 2.2.1):

```
PARAMETER F(Bonds, Time);  # -- or:
PARAMETER F(i, t);         # same thing
```

These two declarations have identical meaning, given the `ALIAS` declarations of `i` and `t`, and specify that `F` takes two indices belonging to (aliased to) the sets `Bonds` and `Time`, respectively.

Indices are used, for instance in expressions, to pick out individual elements of indexed objects. If `F(t,i)` is a bond's cashflows, then the following calculates each bond's total cashflows:

```
PARAMETER Total_CF(i);
Total_CF(i) = SUM(t, F(t,i));
```

and stores it into the declared one-dimensional parameter. Notice that this assignment is automatically performed for each value `i` in `Bonds`. There is more information on the summation operation in Section 1.3.3.

Leads and lags are indices that are shifted by some constant, as in `F(i,t+1)` or `F(i,t-1)`, respectively. The lead or lag need not be 1, but may be any integral expression whose value is known at compile time (endogenous). Leads and lags may only be used on static sets (not dynamic sets; see below).

There are no "index errors" in GAMS: If a lead or lagged index reaches beyond the underlying set, the result is 0; `F(i,t+1)` is 0 when `t` is the set's last element.

It is sometimes convenient to treat sets as being *circular*, so that leads beyond the end "wrap around" to the beginning and vice versa. This is indicated by using the `++` or `\verb` operator: `F(i++1,t)` references the next bond (from `i`), cyclically.

Constant set elements used as indices have to be specified in quotes:

```
F("GOVT_1", "2002") = 20000;
```

The `ORD` and `CARD` set functions

The `ORD` function takes as an argument the name of a set (or an index aliased to a set) and returns the ordinate value of the index relative to the set. The `CARD` function takes a set name (or index) and returns the cardinality of the set, i.e., the number of elements in the set:

```
P(i) $ (ORD(i) < CARD(i)) = 7;  # All elements except the last
                                # set equal to 7
P(i) $ (ORD(i) = CARD(i)) = 3;  # and the last one set to 3
```

`ORD` is defined only on *static, one-dimensional* sets. They are not defined on constant set elements: `ORD("GOVT_4")` is illegal.

Subsets and multidimensional sets

A set can be declared as a subset of another set. For instance,

```
SET Callable(Bonds) /GOVT_1, GOVT_3/;
```

specifies a subset, `Callable`, of the `Bonds` set and declares bonds `GOVT_1` and `GOVT_3` as being callable bonds. Subsets can be multidimensional:

```
SET Matur(i,t) / GOVT_1.2003, GOVT_2.2004,
                 GOVT_3.2005, GOVT_4.2006 /;
```

might specify the maturity years of the bonds as a subset of the cartesian product `Bonds` × `Time`. Note the dot-notation: `GOVT_1.2003` specifies that the element `(GOVT_1, 2003)` belongs to `Matur`.

Dynamic sets

The sets seen so far were all *static*. Their elements were explicitly listed as part of their declaration. GAMS also allows *dynamic* sets, which are calculated during execution of the model allowing them to change dynamically depending on some model characteristics. Dynamic sets are always subsets of another set (or other sets) and do not have the `/ ... /` part in their declaration. Consider:

```
SET Time           /  2002 * 2006 /;  ALIAS (Time,t);
PARAMETER tau(t);
tau(t) = ORD(t) - 1;              # Relative time: 0, 1, 2, ...

PARAMETER L(t), PV;

SET Sub(t); ALIAS (Sub, s);       # Dynamic subset of Time

Sub(t) = YES $ (tau(t) >= 1);     # All but the first year
L(t) $ (tau(t) < tau("2006")) = 7;# OK even with constant index
PV = SUM(s, tau(s) * L(s));       # OK even with dynamic set index
```

This fragment defines the set `Sub` as a dynamic subset of `Time`, and initializes it to contain all but the first element. The expression `YES $ (tau(t) >= 1)` means to include element `t` in `Sub` if the condition following the $-operator is satisfied (*conditional expressions* are covered in detail in Section 1.3.3).

The `ORD` function is not defined on dynamic sets, but notice that a parameter such as `tau`, which is declared on the static set `Time`, can be used with both constant arguments, `tau("2006")` and with a dynamic set index, `tau(Sub)` or `tau(s)`, as long as all arguments belong to `Time`. This is a very useful technique to extend `ORD`-like mappings from dynamic, even multi-dimensional, sets to numbers. `CARD`, however, is defined on dynamic (even multidimensional) sets.

1.3.3 Expressions, functions, and operators

Data manipulations and constraint definitions require the use of *expressions* and *assignments*. GAMS provides a rich set of operators and built-in functions for forming expressions. Expressions are built up from numerical values, that is, *floating point constants, scalars*, and *parameter* and *table* elements. Numerical values may be joined using *operators*; a list of operators is given in Table 1.2. GAMS also defines a number of *functions*; see Table 1.3. In addition, GAMS has a number of calendar (date/time) functions; see Table 1.4.

Table 1.2: Operators in GAMS expressions, grouped by priority. All operators accept and return numerical values. The logical operators interpret non-zero as true, and they return 0 or 1 for **false** or **true**.

Operator	**Description**
`$`	Conditional operator
`**`	Exponentiation
`*,/`	Multiplication and division
`+,-`	Addition and subtraction
`LT, <`	Less than
`GT, >`	Greater than
`EQ, =`	Equals
`LE, < =`	Less than or equal to
`GE, > =`	Greater than or equal to
`NE, <>`	Not equal to
`NOT`	Logical Not
`AND`	Logical And
`OR`	Logical Or
`XOR`	Logical Exclusive Or

Consider this simple example where some statistics of stock returns are calculated:

```
SET Time  /t1 * t5/;      ALIAS(Time, t);
SET Stock /stk1 * stk3/;  ALIAS(Stock, i, j);

TABLE Return(t, i) "Return of stock i in time period t,
   in percent"
            stk1    stk2    stk3
     t1       22      -8       4
     t2        3      20      12
     t3        3     -10       2
     t4      -30       8       1
     t5        3      -2       0 ;

PARAMETER MeanRet(i);  # Each stock's average return over time
MeanRet(i) = SUM(t, Return(t,i)) / CARD(t);

PARAMETER VarCov(i,j); # Variance-Covariance matrix of returns
VarCov(i,j) = SUM(t, (Return(t,i) - MeanRet(i)) *
                     (Return(t,j) - MeanRet(j))) / (CARD(t)-1);

DISPLAY MeanRet, VarCov;
```

Table 1.3: Functions in GAMS. Notes: (1) non-differentiable and results in discontinuous nonlinear programming model if used on (endogenous) variables; (2) not continuous and illegal when used on (endogenous) variables; (3) pseudo-random, may not occur in equation definitions; (4) takes one or more set indices as their first parameter and perform an indexed operation; (5) takes a set or index argument.

Function	Description	Note
`ORD`	Ordinate value of index	5
`CARD`	Cardinality of set	5
`SUM`	Summation over set	4
`PROD`	Product over set	4
`SMIN`	Minimum over set	4
`SMAX`	Maximum over set	4
`ERRORF(X)`	Integral of std. normal from $-\infty$ to x	1
`EXP(X)`	Exponential, e^x	
`LOG(X)`	Natural log (for $x > 0$)	
`LOG10(X)`	Base-10 log (for $x > 0$)	
`NORMAL(X,Y)`	Normal distribution; mean x, std.dev y	3
`UNIFORM(X,Y)`	Uniform distribution in $[x, y]$	3
`ABS(X)`	Absolute value	1
`CEIL(X)`	Smallest integer $\geq x$	2
`FLOOR(X)`	Largest integer $\leq x$	2
`MAPVAL(X)`	Mapping function (see User's Guide)	2
`MAX(X,Y, ... )`	Maximum of arguments	1
`MIN(X,Y, ... )`	Minimum of arguments	1
`MOD(X,Y)`	Remainder (modulo)	2
`POWER(X,Y)`	Power; y must be an integer	
`ROUND(X)`	Rounding to nearest integer	2
`ROUND(X,Y)`	Rounding to y decimal places	2
`SIGN(X)`	-1, 0 or 1 depending on the sign of x	2
`SQR(X)`	Square of x	
`SQRT(X)`	Square root	
`TRUNC(X)`	Rounding towards 0	2
`ARCTAN(X)`	Arcus tangent, result in radians	
`COS(X)`	Cos, x in radians	
`SIN(X)`	Sin, x in radians	

The first assignment statement assigns to `MeanRet(i)` the average return of the three stocks over the five time periods. GAMS automatically performs the assignment for each value of the index `i`; notice the use of the `SUM` function to perform the summation over the `t` index, and the use of the `CARD` function to average.

The $-operator and conditional expressions

The $-operator is somewhat unusual and deserves special attention. It is a binary operator, and the meaning of the expression `(exp) $ (cond)` is as follows.

Table 1.4: Date and Time functions in GAMS. Notes: (1) JDATE converts a date (given as year, month, day) into the day number, where "Day 1" is January 1, 1900; (2) JTIME converts a time (given as hour [0, . . ., 23], minute [0, . . ., 59], second [0, . . ., 59]) into a fraction of a day; (3) these routines convert a day number into year, month, day, day of week, and check for leap years; (4) JSTART and JNOW return, in addition to date information, information on the time of day, taking no parameters; (5) these routines convert the result of JTIME, JSTART or JNOW back into time of day.

Function	Description	Note
JDATE(Y,M,D)	Day number	1
JTIME(H,M,S)	Time of day as fraction	2
GYEAR(J)	Year	3
GMONTH(J)	Month	3
GDAY(J)	Day of month	3
GDOW(J)	Day of week	3
GLEAP(J)	1 if leap year, 0 otherwise	3
JSTART	Start of current GAMS job	4
JNOW	Date and time when called	4
GHOUR(J)	Hour	5
GMINUTE(J)	Minute	5
GSECOND(J)	Second	5

```
(exp) $ (cond)      # GAMS conditional expression
(cond)? (exp) : 0   # Same construct in C/C++/Java
```

First, evaluate the expression cond. If its value is non-zero, then the complete expression has the value of exp, otherwise it has the value 0 and exp is not evaluated. For example, the annual cashflows of a set of bonds given their coupon rates and maturity years can be calculated by:

```
PARAMETER F(i,t);
F(t,i) =       1      $ (tau(t)  = Maturity(i))
          + coupon(i) $ (tau(t) <= Maturity(i) and tau(t) > 0);
```

where tau(t) maps elements of the time set to calendar years; a bond's cashflow is composed of its price (negative at purchase), its principal payment (1, at maturity), and coupon payments; see Figures 2.1 and 2.2 for the complete model.

An expression such as tau(t) = 0 is called a *conditional expression*. GAMS has very simple rules for forming and using conditional expressions: any non-zero numerical value is interpreted as "true" when used in a conditional expression, and zero is interpreted as "false." Consistent with this, the relational operators (= , <, <= , >, >= , <>) and the logical operators AND, OR, and NOT return the numerical values 1 for "true" and 0 for "false." It is advisable to use parentheses around the operands of the $-operator; this aids readability and avoids any confusion about the priority of $ relative to other operators.

Notice also the use of an indexed set or parameter as a condition. Given the declarations

```
SET Matur(i,t);
PARAMETER F(t,i);
```

the expressions `Matur(i,t)` and `F(t,i)` are legal $-conditions, testing whether the pair `(i,t)` belongs to `Matur`, or whether `F(t,i)` is non-zero, respectively.

The $-operator also has another important function as a control structure: any time GAMS performs an indexed operation, a $-condition can be used to specify a subset of the indices over which the operation should be performed. Details are given in the relevant sections under equation definitions (Section 1.3.6), assignment statements (Section 1.3.4), and the SUM, PROD, SMIN and SMAX functions below.

Special functions: SUM, PROD, SMIN, SMAX

The functions `SUM`, `PROD`, `SMAX`, and `SMIN` have two arguments where the first must be a set (or index) expression. They form the sum, product, element-wise maximum, and element-wise minimum over the second argument values:

```
SUM(i, x(i));                  # sum of x's in a constraint
SMIN( (i,j), A(i,j));          # multidimensional indexation
```

Note the use of parentheses to sum over multiple indices.

The index set over which the operation is applied can be qualified using the $-operator, as in:

```
PROD(i $ Callable(i), x(i));   # conditional indexation
PROD(Callable(i), x(i));       # shorthand; same as above

* Find maximum below-diagonal element of A:
SMAX( (i,j) $ (ORD(i) > ORD(j)), A(i,j));
```

The operation will be performed only over indices that satisfy the condition. When the condition is simply an indexed set or parameter, as in `Callable(i)`, the conditional summation can be abbreviated as shown in the second line above, which aids readability.

Applying one of these functions over an empty set results in the operations's *neutral element*: 0 for `SUM`, 1 for `PROD`, `-INF` for `SMAX`, and `INF` for `SMIN`. Comparing index values directly is not possible:

```
SMAX( (i,j) $ (i > j), A(i,j));           # Illegal!
SMAX( (i,j) $ (ORD(i) > ORD(j)), A(i,j)); # OK!

L(t) $ (t < "2006")           = 7;        # Illegal!
L(t) $ (tau(t) < tau("2006")) = 7;        # OK!
```

It is necessary to use some function that converts from set elements to numerical values, such as `ORD` or some parameter indexed by the set (as `tau` in the example above). For the special case of testing for equality, however, GAMS has a function `SAMEAS`:

```
SUM( (i,j) $ SAMEAS(i,j), A(i,j)); # Sum of A's diagonal elements
```

SAMEAS compares the names of the set elements referred to by the indices (not their ordinal values), and is legal even if the two indices belong to different sets.

Extended arithmetic: INF, EPS, NA, UNDF

GAMS defines certain special values that can be used in and result from expressions, but which are not numbers. The most common one is INF, which represents the (extended real) number ∞. For instance, a free VARIABLE has lower and upper bounds -INF and INF.

Another common special value is EPS, which is returned by solvers as the marginal value of degenerate variables or constraints (that is, non-basic variables or binding constraints with zero marginals). Knowing this makes it easy to pick out the final basis from a linear programming solution as those variables and (slack/artificial variables corresponding to) equations having "." as the marginal value as opposed to EPS or a non-zero value.

Finally, the last two special values are NA (used to represent missing, or "Not Available," data) and UNDF, for undefined (usually erroneous) results.

The rules for arithmetic using these values are well defined in GAMS but rather complicated; see the User's Guide or the library model crazy.gms for details.

1.3.4 Assignment statements

The assignment statement is used, as in any other language, to assign values to parameters (SCALARs, PARAMETERs, and TABLEs). The parameter on the left-hand side always has all indices specified, and GAMS automatically performs the assignment for each index value. Some examples:

```
left_over = 1-tax_rate;
Total_CF(i) = SUM(t, F(t,i));
P(i) $ (ORD(i) = CARD(i)) = 3;
```

The first line is a simple assignment of a single value. The second is performed for all indices i, the third only for those values of i that satisfy the condition. Note that in a conditional assignment that the condition is placed *before* the = sign.

Assignments performed over indices are performed in "parallel," in the sense that the right-hand side is first calculated for each index value, and only then is the left-hand side simultaneously updated. See in Section 1.3.9 (the LOOP statement) how this behavior can be circumvented if desired.

As a complete example, consider the following calculation of a lower-triangular matrix to hold covariance information:

```
SET Lower(i,j);
Lower(i,j) = YES $ (ORD(i) > ORD(j));

PARAMETER VarCov2(i,j);  # Lower-triangular Var-Covar matrix
VarCov2(i,i) = SUM(t, SQR(Return(t,i) - MeanRet(i))) / (CARD(t)-1);
```

```
VarCov2(Lower(i,j)) =
       2 * SUM(t, (Return(t,i) - MeanRet(i)) *
                  (Return(t,j) - MeanRet(j))) / (CARD(t)-1);
```

In this example we want to calculate variance-covariance information into a matrix efficiently, by only storing elements in the lower triangular half of the `VarCov2` matrix, defined by the dynamically calculated subset `SET Lower`. The assignment to `VarCov2(i,i))` calculates the diagonal, and the assignment to `VarCov2(Lower(i,j))` is shorthand for:

```
VarCov2(i,j) $ Lower(i,j) =  ...
```

so that only below-diagonal elements are assigned. Altogether, this calculation is almost twice as fast as calculating `VarCov` shown on page 7, yet used in a typical variance expression such as

```
SUM( (i,j),  x(i) * VarCov2(i,j) * x(j) )
```

it is equivalent (but again about twice as fast to evaluate).

1.3.5 Variable declarations

Variable declarations are used to declare the variables used in a model. Variables can be *continuous* or *discrete* or some mixture of the two. Continuous variables are allowed to take on a range of variables between some (possibly infinite) lower and upper bounds, while discrete variables must take on an integer value between some finite bounds. The different declaration possibilities are shown in Table 1.5. Variables can have up to 10 indices.

Variable attributes

After declaration of a variable it is always possible to change its bounds:

```
POSITIVE VARIABLES y(i,j);
y.LO(i,j) = -4;
y.UP(i,j) = 10;
y.FX("2","3") = y.LO("2","3") + 18;
```

Table 1.5: The different kinds of variables and their declaration. The default bounds can be reset through the `LO` and `UP` (or `FX`) attributes.

Keyword	**Type**	**Default Lower Bound**	**Default Upper Bound**
`FREE` (Default)	Continuous	`-INF`	`INF`
`POSITIVE`	Continuous	0	`INF`
`NEGATIVE`	Continuous	`-INF`	0
`BINARY`	Discrete	0	1
`INTEGER`	Discrete	0	100

Here, a two-dimensional array of variables is declared as non-negative (default bounds 0 and ∞), but then the bounds are reset to -4 and 10, by setting `LO` and `UP` attributes. Finally, `y("2","3")` is *fixed* at a specific value. Assigning a value to the `FX` attribute is equivalent to setting the variable's lower and upper bounds to the same value. Fixing a variable does not remove it from the model; see the `HOLDFIXED` attribute on page 15 for how to do this. Variables also have two other attributes which are set by solvers: `L` is the "level" value (for instance the optimal value after the problem is solved), and `M` is the "marginal," or reduced cost. These can both be initialized by the user, which is useful in nonlinear programming to provide a starting point for the solver.

Variables also have a scaling attribute, `SCALE`; see the User's Guide for details.

1.3.6 Constraints: Equation declarations

Equations are used to declare and define model constraints:

```
EQUATIONS constr(i), objective;
constr(i) .. SUM(j, A(i,j) * x(j)) =L= b(i);
objective .. z =E= SUM(j, c(j) * x(j));
```

Here, we declare a set of constraints `constr(i)`, and an individual constraint, `objective`. They are then defined (indicated by the ".." symbol). Each of the constraints `constr(i)` is a less-or-equal inequality constraint, as indicated by `=L=`. The `objective` constraint is an equality indicated by `=E=`. Greater-or-equal constraints are specified using `=G=`. A fourth relation, `=N=`, indicates that the constraint is present but non-binding; no use has yet been found for it. The expressions used to define constraints are covered in Section 1.3.3.

Endogenous variables in constraints

Constraint expressions are the only places where endogenous variables (GAMS variables), like `x(j)`, can be used without their attributes (`L`, `M`, `UP`, `LO`, etc.). Note a few cautions regarding endogenous variables in constraints (an "endogenous expression" is an expression that is or contains an endogenous variable):

- nonlinear GAMS functions or operators, when used on endogenous expressions, lead to nonlinear (`NLP` or `DNLP`) models; see Section 1.3.8 for details on model types.
- non-continuous GAMS functions may not be used on endogenous expressions.
- the pseudo-random number generator functions `UNIFORM` and `NORMAL` may not be used at all in constraints.
- endogenous expressions cannot be used in the conditional part of a $-condition.

A $-condition may be placed before the `..` symbol in a constraint definition. The constraint is only then generated and included in the model if the condition is satisfied:

```
constr(i) $ (ORD(i) > 3) .. SUM(j, A(i,j) * x(j)) =L= b(i);
objective $ 0 .. z =E= SUM(j, c(j) * x(j));  # constraint excluded
```

Equation attributes

Constraints have the attributes LO, UP, L and M. To understand these it is useful to consider, for instance, a less-equal constraint to be written as:

$$lhs\texttt{=L=}rhs$$

where *lhs* consists of all variable terms of the constraint and *rhs* consists of all constant terms. Then the level attribute .L is the value (after a solve) of the constraint's left-hand side, and the bounds attributes LO, UP are the bounds on it; for a less-equal constraint the left-hand side has bounds -INF and *rhs*. The M attribute is the constraint's dual price.

1.3.7 Model declarations

Model declarations serve to collect the constraints and variables that are part of the model, and to name the model.

```
MODEL Dedication /cfm, constr/;
```

Between the slashes are listed the names (without indices) of any constraints that should be part of the model Dedication. If all the constraints defined in the source file up to this point are part of the model, one can write:

```
MODEL Dedication /ALL/;
```

Model attributes

Models have "attributes" which are used to communicate information to and from a solver. Some are set by the user and correspond to setting the corresponding value using an OPTION statement; see Table 1.6. For instance, Dedication.RESLIM = 200; allows the solver to spend at most 200 seconds solving the Dedication model.

Others are set as a result of executing a SOLVE statement and they can be used to test the result of solving a model and hence decide on further actions to take:

Table 1.6: The most important OPTIONs. The argument N indicates a non-negative integer.

Keyword	Description
DECIMALS = N	Prints numerical values with N decimals
ITERLIM	Maximum number of solver iterations (default 1000)
LIMCOL = N	Lists N equations for each equation block (default 3)
LIMROW = N	Lists N variables for each variable block (default 3)
OPTCA, OPTCR	Sets optimality tolerance for MIP (see Page 15)
RESLIM	Maximum number of solver CPU seconds (default 1000)
SOLPRINT = ON/OFF	Lists the solution after each solve statement
SYSOUT = ON/OFF	Includes solver output files into listing

```
IF (Dedication.MODELSTAT = 1,  # Optimal! Solve another one:
  Solve Model2 MINIMIZING z USING lp;
ELSE
  DISPLAY "Could not solve Dedication";
)
```

The most important values of MODELSTAT are 1: optimal, 2: locally optimal, 3: unbounded, and 4: infeasible;

Substituting fixed variables

The variable suffix FX will "fix" a variable, i.e., set its upper and lower bounds to the same value, but the variable is still present in the problem even though it has only a single feasible value. The MODEL attribute HOLDFIXED, when set to 1:

```
MODEL m /all/;
m.HOLDFIXED = 1;
```

will cause the values of all fixed variables to be substituted for the value throughout in the model. This can greatly reduce the complexity of a model, for instance converting a nonlinear model to a linear one. The only piece of information lost is the variable's dual information (marginal).

1.3.8 The SOLVE statement and model types

The SOLVE statement has the general form:

```
SOLVE model_name MINIMIZING obj_var USING model_type;
```

where model_name is the model to be solved, obj_var is the variable whose value should be minimized (one can also ask for MAXIMIZING the value), and model_type indicates the type of model to be solved; see Table 1.7. GAMS will select a default solver that is capable of solving the indicated model type, or a desired solver can be specified:

```
OPTION LP = BDMLP;
```

causes GAMS to use BDMLP to solve LP models.

The variable obj_var appearing in the SOLVE statement should be continuous without bounds.

Model types

GAMS recognizes several model types, as listed in Table 1.7. The most important are:

- LP: If the model contains only linear constraints and continuous variables, it's an LP. LP's are generally very easy to solve, except when extremely large.
- MIP: If the model contains linear constraints but discrete (integer or binary) variables, then it's a MIP model. These can be very time-consuming to solve. Be aware that

Table 1.7: GAMS Model types.

Keyword	Description	Variable and constraint typologies
LP	Linear Program	Linear
MIP	Mixed-integer Program	Linear, discrete
RMIP	Relaxed MIP	As MIP; solved as an LP
NLP	Nonlinear Program	Linear, nonlinear
DNLP	Discontinuous NLP	Linear, nonlinear, non-differentiable constraints
MINLP	Mixed-Integer NLP	Linear, discrete, nonlinear
RMINLP	Relaxed MINLP	As MINLP; solved as a NLP
MCP	Mixed Complementarity Program	Complementarity constraints
CNS	Constrained Nonlinear System	LP or NLP without objective function

INTEGER VARIABLES have implicit upper bounds of 100, so it is usually a good idea to set relevant upper bounds explicitly. Also, by default, a solution that is probably within 10 % of the optimum may be returned – to force the solver to go for an optimal one use OPTION OPTCR = 0. Also, the default iteration and resource limits of 1000 iterations and 1000 CPU seconds are, in some cases, not sufficient for the convergence of the solver. Use, for instance, OPTION ITERLIM = 999999999, RESLIM = 1200; to allow 20 minutes but virtually unlimited iterations for the solution.

- RMIP: To solve a MIP model while ignoring the integrality constraints, use RMIP. This is useful for model debugging.
- NLP: If your model contains nonlinear constraints and continuous variables, it's an NLP. These can be easy or difficult depending (mostly) upon whether the constraint set is convex, and the objective function convex (for minimization) or concave (for maximization). The best result possible for an NLP is "locally optimal"; the solver has no way to guarantee that a locally optimal solution is also globally optimal.
- MINLP: May contain nonlinear expressions and discrete variables.
- DNLP: May contain nonlinear constraints that are not differentiable, hence very unreliable to solve. One should usually try to reformulate such models, or "smooth" any "kinks."

1.3.9 Control structures

The control structures consist of the IF, WHILE, FOR, LOOP statements. They are used to control the execution of statements, depending on a condition. Control statements may not contain declarations.

The IF statement

The IF statement has the following forms:

```
IF (condition, stat-seq)
IF (condition, stat-seq ELSE stat-seq)
IF (condition, stat-seq ELSEIF condition, stat-seq ...
   ELSE stat-seq)
```

where condition is a conditional expression (1.3.3) and stat-seq is a semicolon-separated list of executable statements. Some examples follow:

```
IF (i < 0,
  DISPLAY "i is negative"
);

IF (i < 0,
  DISPLAY "i is negative"
ELSEIF i = 0,
  DISPLAY "i is zero"
ELSE
  DISPLAY "i is positive"
);
```

The IF ... ELSEIF ... variant allows an arbitrary number of ELSEIF parts, and the ELSE part is optional.

Iterative control structures

These statements allow repeated execution of groups of statements until some condition is satisfied (WHILE), or under control of either a scalar parameter (FOR) or a set index (LOOP). Their syntax is:

```
WHILE(condition, stat-seq)
FOR (parm = val1 TO/DOWNTO val2 BY val3, stat-seq)
LOOP(set-index, stat-seq)
```

where condition is a conditional expression (1.3.3) and stat-seq is a semicolon-separated list of executable statements. The FOR loop iterates a parameter parm through a range of values, as for instance:

```
PARAMETER p;
FOR (p = 10 TO 20 , stat-seq)          # p = 10, 11, 12, ..., 20
FOR (p = 20 DOWNTO 10 , stat-seq)      # p = 20, 19, 18, ..., 10
FOR (p = 0 TO 1 BY 0.1, stat-seq)      # p = 0, 0.1, 0.2, ..., 1
```

A powerful use of the iterative statements is to solve sequences of related models by having a SOLVE statement in the iteration. To use this facility it is necessary to know that

every time a SOLVE statement is executed, the model is *regenerated* from scratch: GAMS runs through the equation definitions using the latest values of all sets and parameters they reference. By changing these as part of the iteration one can generate a different model in each run through the loop.

As an example, Figure 1.1 shows how to solve a sequence of related models under control of a FOR statement. The model is the Klee-Minty model (see, e.g., Nash and Sofer [1996] for a discussion on the Klee-Minty problem), which is interesting because it may require 2^m iterations with a naive implementation of the simplex algorithm, where m, the number of variables and constraints, is a parameter:

$$\underset{z\in\Re, x\in\Re^m}{\text{Maximize}} \quad z = \sum_{j=1}^{m} 10^{m-j} x_j \tag{1.4}$$

$$\text{subject to} \quad 2\sum_{j=1}^{i-1} 10^{i-j} x_j + x_i \leq 100^{i-1}, i = 1, \ldots, m \tag{1.5}$$

$$x \geq 0 \tag{1.6}$$

As another example, Figure 1.2 shows the use of a LOOP to control the iterations, and a dynamic set s is used to control generation of the model. A solution report is built along the way.

```
SET base /1 * 10000/; ALIAS (base, j);   # Allow at most m = 10000
SET i(base);                             # A dynamic subset of base
PARAMETER numval(j);
numval(j) = ORD(j);          # Map set elements to numerical values

PARAMETER m;

EQUATIONS obj, constr(base);     # Declare constr over the base set

VARIABLE z; POSITIVE VARIABLES x(j);

obj       .. z =E= SUM(i, POWER(10, m-numval(i)) * x(i));
constr(i) .. 2 * SUM(j$(numval(j) < numval(i)),
                     POWER(10, numval(i)-numval(j)) * x(j)) + x(i)
              =L= POWER(100, numval(i)-1);

MODEL KleeMinty /all/;

FOR (m = 1 TO 4,            # Solve a sequence of KleeMinty models
  i(j) = YES $ (numval(j) <= m);     # i contains  /1, 2, ..., m/;
  SOLVE KleeMinty MAXIMIZING z USING lp;
  DISPLAY m, z.L;
);
```

Figure 1.1: GAMS model for solving the Klee-Minty problem for $m = 1, 2, 3, 4$.

```
SET control /1*1000/; ALIAS(control,c);
SET s(control);                              # a dynamic subset
* Tiny model to be solved:
VARIABLE x, z;
EQUATION eqn;
PARAMETER parm(c) /1 = 10, 2 = 50, 3 = 80/; # some data

eqn(s) .. z = E = SQR(x) + parm(s); # depends on contents of s
x.LO = 0.1; x.UP = 100;

MODEL testmodel /all/;

PARAMETER solution(control, *);
PARAMETER converged; converged = 0;

LOOP(c $ (NOT converged),      # c is used only within the loop
  s(control) = YES $ (ORD(control) = ORD(c)); # controls eqn(s)

  SOLVE testmodel MAXIMIZING z USING NLP;

  solution(c, "parm") = parm(c);
  solution(c, "x") = x.L;
  solution(c, "obj") = z.L;
  converged = ...;  # 1 when converged and want to terminate
);
DISPLAY solution;
```

Figure 1.2: Using a `LOOP` iterative statement to control a `SOLVE` statement and to create an iteration-by-iteration solution report. Notice the way the three sets/indices (`control`, `s`, and `c`) are declared and used: `control` is the base set, allowing up to 1000 iterations; `c` is the loop control index; and `s` is a dynamic subset of the control set, which (in this example) contains the the current loop index as its only element, and which in turn controls the generation of equation `eqn`. The loop can be terminated at any time by setting `converged` to 1.

Defeating parallel assignment

An assignment statement such as

```
MeanRet(i) = SUM(t, Return(t,i)) / CARD(t);
```

is executed "in parallel": GAMS performs the assignment for each value of the controlling index `i` "at once." But sometimes the parallel assignment feature gets in the way. Consider calculating the Fibonacci numbers from 1 to 100. One might be tempted to do this:

```
SET i /1*100/; ALIAS(i,j);
PARAMETER Fibonacci(i);
```

```
Fibonacci("1") = 1;
Fibonacci("2") = 2;
Fibonacci(i) = Fibonacci(i-1) + Fibonacci(i-2); # Bad idea!
```

The last assignment would not work because of the "parallel assignment" feature of GAMS: Most of the values referenced on the right-hand side are equal to zero (i.e., `Fibonacci("3")` through `Fibonacci("100")`), and the values assigned to `Fibonacci("1")` through `Fibonacci("2")` use undefined indices, -1 and 0. The way to implement this kind of *recurrence relation* is to force GAMS to execute the assignment element-by-element in a specified order, rather than in parallel. The `LOOP` statement does this:

```
Fibonacci("1") = 1;
Fibonacci("2") = 2;
LOOP(i $ (ORD(i) > 2),
  Fibonacci(i) = Fibonacci(i-1) + Fibonacci(i-2); # OK!
);
```

Note the use of the $-operator to limit the `LOOP` statement to values of `i` greater than 2.

1.3.10 Conditional compilation

Some of the GAMS $-control commands (see Table 1.1) are particularly useful for conditional compilation, that is, including or excluding parts of the GAMS source code depending on some condition. We give here an example:

```
$SET switch 2        # Select this case among several

 DISPLAY "beginning...";

$IF NOT "%switch%" == "1" $GOTO case2
    DISPLAY "case 1";
$   GOTO continue

$LABEL case2
$IF NOT "%switch%" == "2" $GOTO case3
    DISPLAY "case 2";
$   GOTO continue

$LABEL case3
$IF NOT "%switch%" == "3" $GOTO error
    DISPLAY "case 3";
$   GOTO continue

$LABEL error
    ABORT "switch has an illegal value";

$LABEL continue
    DISPLAY "Carrying on...";
```

1.4 Getting Started

The GAMS system can be executed in two modes. On computers running Windows, through an Integrated Development Environment graphical interface that facilitates managing the files involved in a GAMS project. On computers running Unix, through a simpler command-line interface where the GAMS system is called from a command-line window. Both modes of execution are described below.

1.4.1 The Integrated Development Environment

The GAMS Integrated Development Environment (IDE) provides an environment for managing GAMS modeling projects that facilitates the process of editing input files, executing GAMS, and viewing output files. It is a graphical environment that is available on Windows systems, and is expected to be available on Unix systems as well.

The GAMS IDE is project-oriented. This means that all the files associated with a model, or a set of models, are collected in a *project file*. Even if your GAMS "project" consists of only a single GAMS input file and its output file, there are advantages to organizing these files in a project.

Creating a new GAMS project

To create a new GAMS project, first close any open files in the GAMS IDE ("File – Close" for each one). Then open the "File – Project – New Project" window and navigate to the directory where you want your project to reside (you may already have your GAMS source file there). Under "File name," enter the name of your project. To add existing source files to your project, use "File – Open" and navigate to your source files (usually .gms and .inc files), adding them to the project one by one (or add multiple files at once, using the standard key sequences). To create a new source file in your project, use "File – New"; then immediately after "File – Save as" to give the new file a name.

Opening an existing GAMS project

Use "File – Project", to check if the project is already listed in the window, or click "Open project" and navigate to it.

Executing GAMS models

To execute a GAMS source (.gms) file, make sure it is the active file in the IDE ("has focus"). Then use the Run entry from the File menu, press F9 or click on the Run icon at the top of the main window. Next to the Run icon is an entry field to specify additional parameters for the GAMS run. Additional parameters have the same effect as if they were specified from the commandline.

While GAMS is compiling and executing the model it displays a log window showing what is going on. If the run takes a while, you may check the Update entry at the bottom to make sure the log window is updated every time GAMS or one of the solvers outputs a line. After execution, the listing file is made the active file, and can be examined for the solution or any error messages.

1.4.2 Command line interaction

The simplest and most general way to use the GAMS system is from the command line through text files. An input file containing the model's source code, and having a name with the extension `.gms`, is created using an ordinary text editor, such as vi, emacs, or Notepad (if using a word processor you should "save as type .TXT"). This file is then submitted to the GAMS system by issuing the command:

```
gams dedicate
```

from the Unix prompt. GAMS will look for and compile the file `dedicate.gms`, and generate an output file, `dedicate.lst`, containing a listing of the input file and the solutions of any models solved. This file is then examined using a text editor. It is often convenient to have several open windows: one in which to edit the input file, one to call GAMS, and one to look at the output.

1.4.3 The model library

The fastest way to build a new model, or to learn the language, is to study existing models that address a related problem. The GAMS system includes a large library of models that demonstrate applications drawn from engineering, finance, and economics. The library is an excellent resource for learning GAMS, or for learning about modeling in a particular problem area. The FINLIB library, which is documented in this book, contains several of the financial optimization models discussed in the companion volume *Practical Financial Optimization*.

From the GAMS IDE, "File – Model Library – Open GAMS Model Library" gives access to more than 100 models in the standard library and to the FINLIB library. Clicking on one of the models will add it to your current project and you can now modify it as you want (you will not be allowed to modify the original file).

From the command line,

```
gamslib index
```

copies a file containing a list of the library models into the current directory; the commands

```
gamslib 1
gamslib dedicate
```

both copy model number 1, dedicate.gms, into the current directory.

Notes and References

The GAMS manual, *GAMS: A User's Guide* and other documents can be downloaded from `http://www.gams.com`. A demonstration version of the GAMS system can be obtained from `http://www.gams.com/download`.

At the time of writing it is also possible to download the GAMS Integrated Developer Environment (for Windows), containing a sample of solvers and the GAMS main compiler. This system runs in "demo mode," allowing the solution of small to medium-sized models. To obtain licenses for solving larger models, for other solvers, or for systems for other machines, contact `support@gams.com`.