

Lucent Technologies
Bell Labs Innovations



Symbolic-Algebraic Computations in a Modeling Language for Mathematical Programming

David M. Gay
Bell Labs, Murray Hill, NJ
dmg@bell-labs.com

Technical Report 00-3-03
Computing Sciences Research Center
Bell Laboratories
Murray Hill, NJ 07974

July 21, 2000

This paper was written for the proceedings of a seminar on “Symbolic-algebraic Methods and Verification Methods — Theory and Applications”, held 21–26 November 1999 at Schloss Dagstuhl, Germany.

Symbolic-Algebraic Computations in a Modeling Language for Mathematical Programming

David M. Gay

1 Introduction

AMPL is a language and environment for expressing and manipulating *mathematical programming* problems, i.e., minimizing or maximizing an algebraic objective function subject to algebraic constraints. The AMPL processor simplifies problems, as discussed in more detail below, but calls on separate *solvers* to actually solve problems. Solvers obtain information about the problems they solve, including first and, for some solvers, second derivatives, from the AMPL/solver interface library.

This paper gives an overview of symbolic and algebraic computations within the AMPL processor and its associated solver interface library. The next section gives a more detailed overview of AMPL. Section 3 discusses communications with solvers. Correctly rounded decimal-to-binary and binary-to-decimal conversions reduce one possible source of confusion and are discussed in section 4. An overview of AMPL's problem simplifications appears in section 5. Directed roundings help these simplifications, as described in section 6; a short discussion of the inconvenience this currently entails appears in section 7. Finally, section 8 provides concluding remarks and section 9 gives references.

2 AMPL overview

AMPL is a language and computing environment designed to simplify the tasks of stating, solving, and generally manipulating mathematical programming problems, such as the problem of finding $x \in \mathbb{R}^n$ to

$$\begin{aligned} &\text{minimize } f(x) \\ &\text{subject to } \ell \leq x \leq u \end{aligned} \tag{1}$$

where $f: \mathbb{R}^n \rightarrow \mathbb{R}$ and $c: \mathbb{R}^n \rightarrow \mathbb{R}^m$ are algebraically defined functions.

AMPL began when Bob Fourer spent a sabbatical at Bell Labs. He had written in Fourer (1983) about the need for modeling languages in the context of mathematical programming, and his sabbatical came at a time of interest in “little languages” at Bell Labs — see Bentley (1986). AMPL permits stating problem (1) in a notation close to conventional mathematical notation that can be typed on an ordinary keyboard. In the spirit of a little language, AMPL does not solve problems itself, but rather translates them to a form that is easily manipulated with the help of the AMPL/solver interface library: see Gay (1997). AMPL can invoke various *solvers* that use the interface library to obtain information about (1), such as objective function values $f(x)$, constraint bodies $c(x)$, the constraint bounds ℓ and u , gradients $\nabla f(x)$, Lagrangian Hessians

$$W(x, \lambda) = \nabla^2 f(x) - \sum_{i=1}^m \lambda_i \nabla^2 c_i(x),$$

etc. After computing a solution, a solver uses the interface library to return solution information, such as “optimal” primal (x) and dual (λ) variable values, to the AMPL processor. A growing AMPL command language facilitates inspecting solutions and other problem data and modifying problems, perhaps to solve a sequence of problems.

One powerful feature of the AMPL modeling language is its ability to state problems involving sets of entities without knowledge of the specific values of the sets. This permits separating a *model*, i.e., a class of optimization problems parameterized by some fundamental sets and “parameters”, from the data needed to specify a particular problem instance. Models usually involve both fundamental and derived sets and parameters; the derived entities are computed from fundamental entities or previously computed derived ones. The AMPL command language permits changing the values of fundamental entities, in which case derived entities are recomputed as necessary.

Though it started as a little language, by the early 1990s AMPL had grown sufficiently that we felt it reasonable to write the book of Fourer et al. (1993) about it. AMPL continues to grow. Extensions since the AMPL book appeared are described in the AMPL web site

<http://www.ampl.com/ampl/>

which contains much more information about AMPL, including pointers to various papers about it.

3 Communication with solvers

Solvers run as separate processes, possibly on remote machines. AMPL communicates with solvers via files; it encodes problem descriptions in “.nl” files, which include sparse-matrix representations of the nonzero structure and linear parts of constraints and objectives and expression graphs for nonlinear expressions. The AMPL/solver interface library offers several .nl file readers that read .nl files and may prepare for nonlinear function and derivative evaluations. Expression graphs are encoded in a Polish prefix form, but this detail is invisible outside of the .nl file readers.

The interface library’s computations of first derivatives (gradients) proceed by backwards automatic differentiation, as Gay (1991) describes. Preparations for Hessian computations are more elaborate; they involve several “tree walks”, i.e., passes over the expression graphs, in part to identify and exploit partially separable structure: objective functions (and constraint bodies) often have the form

$$f(x) = \sum_i f_i(A_i x), \quad (2)$$

in which each A_i matrix has just a few rows and represents a linear change of variables. As Griewank and Toint (1981, 1984) have pointed out, partially separable structure is well worth exploiting when one computes or approximates the Hessian matrix (of second partial derivatives) $\nabla^2 f$, which has the form

$$\nabla^2 f(x) = \sum_i A_i^T \nabla^2 f_i A_i. \quad (3)$$

Indeed, computing or approximating each term in (3) sometimes leads to substantially faster Hessian computations than would otherwise occur. Gay (1996) gives many more details about how the AMPL/solver interface library finds and exploits partially separable structure, including a more elaborate form, “group partial separability”, that Conn et al. (1992) exploit in their solver LANCELOT:

$$f(x) = \sum_i f_i \left(\sum_{j \in S_i} \phi_{i,j}(A_{i,j}x) \right),$$

in which $\phi_{i,j}$ is a function of one variable.

Some solvers deal only with linear and quadratic objectives. The interface library provides a special reader for such solvers. It does a tree walk that extracts the (constant) Hessian of a quadratic function and complains if the function is nonlinear but not quadratic.

4 Binary \leftrightarrow decimal conversions

By default, AMPL writes binary `.nl` files, but it can also write equivalent ASCII (text) files. Binary files are faster to read and write, but ASCII files are more portable: they can be written by one kind of computer and read by another. Both kinds begin with 10 lines of text that provide problem statistics and a code that indicates the format in which the rest of the file is written.

Most current computers use binary “IEEE arithmetic”, or at least the representation for floating-point numbers described in the IEEE (1985) arithmetic standard. The AMPL/solver interface library will automatically swap bytes if necessary so that a binary `.nl` file written on a machine that uses big-endian IEEE arithmetic can be read on a machine that uses little-endian IEEE arithmetic and vice versa.

To remove one source of confusion and inaccuracy and make binary and ASCII `.nl` files completely interchangeable, AMPL and its solver interface library use correctly rounded binary \leftrightarrow decimal conversions, which is now possible on all machines where AMPL has run other than old Cray machines. Details are described in Gay (1990).

Part of the reason for mentioning binary \leftrightarrow decimal conversions here is to point out a recent extension to Gay (1990) that carries out correctly rounded conversions for other arithmetics with properties similar to binary IEEE arithmetic. This includes correct directed roundings and rounding of a decimal string to a floating-point interval of width at most one unit in the last place, both of which are obviously useful for rigorous interval computations. There is no paper yet about this work, but the source files are available as

```
ftp://netlib.bell-labs.com/netlib/fp/gdtoa.tgz
```

which includes a README file for documentation.

5 Presolve

The AMPL processor simplifies problems in some ways before sending them to solvers, a process called “presolving”. The original motivation was just to permit flexibility in stating bounds on variables: a solver should see the same problem

independently of whether bounds on a variable are stated in the variable's declaration or in explicit constraints. But we have incorporated all of the "primal" simplifications described by Brearley et al. (1975), since this sometimes permits diagnosing infeasibility without invoking a solver and it sometimes permits transmitting a significantly smaller problem to the solver. Moreover, while some solvers have their own presolver (generally also based on Brearley et al. (1975)), others do not, and these other solvers sometimes solve problems significantly faster after AMPL's presolve phase has acted.

Fourer and Gay (1994) describe much of AMPL's presolve algorithm, and Ferris et al. (1999) describe extensions to it for complementarity constraints. For this paper, a short overview will suffice. The simplification method of Brearley et al. (1975) applies to linear constraints and objectives; it proceeds to recursively

1. fold singleton rows into variable bounds;
2. omit slack inequalities;
3. deduce bounds from other rows;
4. deduce bounds on dual variables.

AMPL currently omits step 4, since in general there can be several objectives (e.g., for a multi-objective solver) and some solvers can be asked to reverse the sense of optimization, maximizing an objective declared as intended to be minimized and vice versa.

AMPL's presolver currently treats nonlinearities quite primitively, assuming simply that a nonlinear expression can produce values in all of $(-\infty, +\infty)$; this is clearly an area in which there is plenty of opportunity to do better. But even now it is possible for a variable that appears in a nonlinear expression to be "fixed", i.e., have its value determined, by other constraints, in which case AMPL replaces the nonlinear variable by a constant, which may turn previously nonlinear constraints into linear ones that can now participate in presolve simplifications.

In slightly more detail, AMPL maintains a stack of constraints to process, which permits folding simple bound constraints into variable bounds in linear time. For linear constraints involving more than one one variable, AMPL makes several passes, which amount to Gauß-Seidel iterations. For example, consider the constraints

$$\begin{aligned}x + y &\geq 2 \\x - y &\leq 0 \\0.1 \cdot x + y &\leq 1.1 \\x &\geq 0\end{aligned}\tag{4}$$

System (4) has a single feasible point, $(1, 1)$, which the Gauß-Seidel iterations only approximate; AMPL's default 9 passes deduce the bounds

$$\begin{aligned}0.99999 &\leq x \leq 1.00001, \\0.99999 &\leq y \leq 1.00001.\end{aligned}$$

Such bounds, while an overestimate, sometimes permit deducing that other inequalities can never be tight — or are inconsistent.

Some solvers use an active-set strategy: they maintain a “basis” that involves inequality constraints currently holding as equalities. Degeneracy is said to occur when the choice of basis is not unique; solvers sometimes must carry out extra iterations when degeneracy occurs. Conveying the tightest deduced bounds to solvers can introduce extra degeneracy, so by default AMPL does not convey the tightest variable bounds it has deduced when those bounds are implied by other constraints that the solver sees. Solvers that do not use a basis, such as interior-point solvers, may not be bothered by degeneracy, and it is possible to tell AMPL to send the tightest deduced bounds to such solvers.

Table 1 illustrates several of the above points. It shows results for two variants of a small shipping problem, one (`git2`) with variable bounds stated in separate constraints, the other (`git3`) with variable bounds stated in variable declarations, and with various settings of AMPL’s `presolve` and `var_bounds` options in the columns labeled *ps* and *vb*, respectively: *ps* = 0 omits all presolve simplifications; this is the only *ps* setting under which `git2` and `git3` differ in the solver’s eyes. For *ps* = 1, presolve deductions involving two or more nonzeros per constraint are omitted, whereas for *ps* = 10 (the default, permitting 9 Gauß-Seidel iterations), they are allowed; only one iteration is required for this particular problem, but the stronger deductions do permit reducing the problem size. Under the (default) setting of *vb* = 1, variable bounds implied by constraints that the solver sees are not transmitted to the solver, whereas they are transmitted for *vb* = 2. The columns labeled *m*, *n*, and *nz* give the number of constraints, variables, and constraint nonzeros in the problem seen by the solver; in this particular example, the number of variables does not change, but it does in other examples. The time column shows execution times for MINOS 5.5, a nonlinear solver by Murtagh and Saunders (1982) that uses an active-set strategy and solves linear problems (such as the `git` problems) as a special case. MINOS benefits from AMPL’s presolve phase, as it does not have its own presolver, and it is affected by the `var_bounds` setting. The times are CPU seconds on a machine with a 466 MHz DEC Alpha 21164A processor, running Red Hat Linux 6.0 (with compilation by the `egcs-2.91.66` C compiler after conversion of the MINOS source from Fortran to C by the Fortran 77 to C converter *f2c* of Feldman et al. (1990)).

Table 1. Illustration of `presolve` settings (see text).

Problem	<i>ps</i>	<i>vb</i>	<i>m</i>	<i>n</i>	<i>nz</i>	iters	time
<code>git2</code>	0	1	1299	1089	4645	341	1.03
<code>git3</code>	0	1	410	1089	3756	383	0.39
<code>git3</code>	1	1	376	1089	3746	359	0.36
<code>git3</code>	10	1	286	1089	3051	324	0.30
<code>git3</code>	10	2	286	1089	3051	402	0.32

6 Directed rounding benefits

Rounding errors could lead to incorrect deductions in AMPL’s presolve algorithm. Assume the *i*-th constraint has the form

$$b_i \leq f_i(x) \leq d_i. \tag{5}$$

AMPL’s presolve algorithm deduces bounds $\tilde{b}_i \leq f_i(x) \leq \tilde{d}_i$ on $f_i(x)$ from bounds

on x ; if the arithmetic were exact, then $\tilde{b}_i \geq b_i$ would imply that the lower inequality in (5) could be discarded without changing the set of feasible x values, and similarly $\tilde{d}_i \leq d_i$ would imply that the upper inequality in (5) could be discarded. Initially we attempted to cope with rounding errors by introducing a tolerance τ (option `constraint_drop_tol`, which is 0 by default) and requiring

$$\tilde{b}_i - b_i \geq \tau$$

or

$$d_i - \tilde{d}_i \geq \tau$$

before discarding the lower or upper inequality in (5). For example, on the *maros* test problem of *netlib*'s `lp/data` collection of Gay (1985), under binary IEEE arithmetic, $\tau = 10^{-13}$ suffices, whereas the default $\tau = 0$ leads to incorrect deductions. (Problems in *netlib*'s `lp/data` collection can be fed to AMPL with the help of a model, `mps.mod`, and *awk* script, `m2a`, that are available from *netlib*.)

Rather than requiring users to guess suitable values for τ , it is safer to use directed roundings in computing deduced bounds \tilde{b}_i and \tilde{d}_i . On machines with IEEE arithmetic, AMPL has been using directed roundings in this context since the early 1990s. These roundings sometimes eliminate incorrect warnings about infeasibility.

Directed roundings can also affect the number of Gauß-Seidel iterations in AMPL's presolve algorithm. Table 2 shows the numbers of such iterations required without (column "near") and with directed roundings on the problems in *netlib*'s `lp/data` directory where directed roundings matter here.

Table 2. Presolve iterations with IEEE nearest and directed rounding.

problem	Rounding	
	near	directed
80bau3b	10	9
blend	6	5
czprob	10	5
israel	6	5
kb2	5	4

Table 3 compares the performance of MINOS, running on the previously described machine, without and with directed roundings in AMPL's presolve algorithm on the problems in *netlib*'s `lp/data` directory where this rounding makes a difference. It is disappointing that the directed roundings often lead MINOS to take more time (CPU seconds) and iterations.

7 Directed rounding frustrations

The directed roundings discussed in the previous section are merely a special case of interval computations; directed roundings are obviously important for interval computations in general. Although machines that with IEEE arithmetic must provide directed roundings, the means of accessing them remain system dependent and sometimes even require use of assembly code. The recently updated C standard does

Table 3. MINOS times and iterations affected by presolve rounding on *netlib*'s lp/data problems.

problem	rounding	rows	cols	nonzeros	iters	time
greenbea	near	1962	4153	24480	15072	103
	dir.	2014	4270	24145	15342	116
greenbeb	near	1966	4151	24474	7942	54
	dir.	2033	4311	25393	8862	62
maros	near	691	1112	7554	1235	2.6*
	dir.	697	1127	7717	1504	3.3
perold	near	600	1276	5678	3395	11.1
	dir.	597	1269	5637	3388	8.5
* Unbounded						

finally provide facilities for controlling rounding direction, but (as of this writing) these facilities are not yet widely available. Similarly, a forthcoming update to the Fortran standard will probably include control of the rounding direction (where possible), but portable specification of the rounding direction so far remains elusive.

The situation is even worse in the currently popular Java world. Although Java makes a big fuss about using part of the IEEE arithmetic standard, it makes no provision for directed roundings. To get them, one must resort to using the Java Native Interface to call functions written in another language.

8 Conclusion

AMPL permits separating a model, i.e., a symbolic representation of a class of problems, from the data required to specify a particular problem instance. Once AMPL has a problem instance, it can make simplifications before transmitting the problem to a solver; these simplifications sometimes benefit from use of directed roundings. Expression graphs sent to the solver can be manipulated by the AMPL/solver interface library to arrange for efficient gradient and Hessian computations. The net effect is that hidden symbolic and algebraic manipulations play a significant role in making life easier for AMPL users, who usually want to concentrate on formulating and using their intended mathematical programming problems, rather than worrying about arcane technical details.

Acknowledgement. I thank Bob Fourer for helpful comments on the manuscript.

9 References

IEEE Standard for Binary Floating-Point Arithmetic, Institute of Electrical and Electronics Engineers, New York, NY, 1985. ANSI/IEEE Std 754-1985.

Bentley, J. L. (Aug. 1986), "Little Languages," *Communications of the ACM* **29** #8: 711–721.

Conn, A. R.; Gould, N. I. M.; and Toint, Ph. L., *LANCELOT, a Fortran Package for Large-Scale Nonlinear Optimization (Release A)*, Springer-Verlag, 1992. Springer Series in Computational Mathematics 17.

Feldman, S. I.; Gay, D. M.; Maimone, M. W.; and Schryer, N. L., "A Fortran-to-C Converter," Computing Science Technical Report No. 149 (1990), Bell Laboratories, Murray Hill, NJ.

Ferris, Michael C.; Fourer, Robert; and Gay, David M. (1999), "Expressing Complementarity Problems in an Algebraic Modeling Language and Communicating Them to Solvers," *SIAM Journal on Optimization* **9** #4: 991–1009.

Fourer, R. (1983), "Modeling Languages Versus Matrix Generators for Linear Programming," *ACM Trans. Math. Software* **9** #2: 143–183.

Fourer, Robert; Gay, David M.; and Kernighan, Brian W., *AMPL: A Modeling Language for Mathematical Programming*, Duxbury Press/Wadsworth, 1993. ISBN: 0-89426-232-7.

Gay, D. M. (1985), "Electronic Mail Distribution of Linear Programming Test Problems," *COAL Newsletter* #13: 10–12.

Gay, D. M., "Correctly Rounded Binary-Decimal and Decimal-Binary Conversions," Numerical Analysis Manuscript 90-10 (11274-901130-10TMS) (1990), Bell Laboratories, Murray Hill, NJ.

Gay, David M., "Automatic Differentiation of Nonlinear AMPL Models," pp. 61–73 in *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, ed. A. Griewank and G. F. Corliss, SIAM (1991).

Gay, D. M., "More AD of Nonlinear AMPL Models: Computing Hessian Information and Exploiting Partial Separability," in *Computational Differentiation: Applications, Techniques, and Tools*, ed. George F. Corliss, SIAM (1996).

Gay, David M., "Hooking Your Solver to AMPL," Technical Report 97-4-06 (April, 1997), Computing Sciences Research Center, Bell Laboratories. See <http://www.ampl.com/ampl/REFS/hooks2.ps.gz>.

Griewank, A. and Toint, Ph. L., "On the Unconstrained Optimization of Partially Separable Functions," pp. 301–312 in *Nonlinear Optimization 1981*, ed. M. J. D. Powell, Academic Press (1982).

Griewank, A. and Toint, Ph. L. (1984), "On the Existence of Convex Decompositions of Partially Separable Functions," *Math. Programming* **28**: 25–49.

Murtagh, B. A. and Saunders, M. A. (1982), "A Projected Lagrangian Algorithm and its Implementation for Sparse Nonlinear Constraints," *Math. Programming Study* **16**: 84–117.