AT&T
Bell Laboratories

Numerical Analysis Manuscript 93–06

Experience with a Primal Presolve Algorithm

*Robert Fourer**
David M. Gay†

April 23, 1993

*Dept. of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL 60201-3119, U.S.A.; `4er@iems.nwu.edu` .

†AT&T Bell Laboratories, Room 2C-463, 600 Mountain Avenue, Murray Hill, New Jersey 07974-0636, U.S.A.; `dmg@research.att.com` .

Experience with a Primal Presolve Algorithm

Robert Fourer

Dept. of Industrial Engineering
and Management Sciences
Northwestern University
Evanston, IL 60201 USA

David M. Gay

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Sometimes an optimization problem can be simplified to a form that is faster to solve. Indeed, sometimes it is convenient to state a problem in a way that admits some obvious simplifications, such as eliminating fixed variables and removing constraints that become redundant after simple bounds on the variables have been updated appropriately. Because of this convenience, the AMPL modeling system includes a ‘‘presolver’’ that attempts to simplify a problem before passing it to a solver. The current AMPL presolver carries out all the primal simplifications described by Brearely et al. in 1975. This paper describes AMPL’s presolver, discusses reconstruction of dual values for eliminated constraints, and presents some computational results.

Introduction

Consider the constrained optimization problem

$$\text{find } x \in \mathbb{R}^n \text{ to minimize } f(x) \tag{1}$$

$$\text{subject to } b \le g(x) \le d \tag{2}$$

$$\text{and } \ell \le x \le u \tag{3}$$

in which $\mathbb{R}^k$ is the set of vectors having k real components and $g \colon \mathbb{R}^n \to \mathbb{R}^m$. This paper is about simplifying the constraints (2) and (3), primarily when the general constraints (2) are linear, i.e., they have the form

$$b \le Ax \le d \tag{4}$$

for some matrix $A \in \mathbb{R}^{m \times n}$. Sometimes the general constraints (2) may imply bounds (3) on the variables or may imply the only values that certain variables can assume. This may let us remove some variables and constraints, a process sometimes called *presolving* the problem.

We are interested in presolving optimization problems expressed in the AMPL modeling language [8]. Indeed, several advantages accrue if the AMPL processor presolves a problem before passing it to a solver. The main advantage is that presolving gives AMPL users flexibility in stating optimization problems. Sometimes it is convenient to have an indexed collection of ‘‘variables’’, some of which have a fixed value, such as an initial inventory. Sometimes it is simplest to specify bounds on a variable when declaring the variable, and other times it is more convenient to state some variable bounds as separate constraints. Another advantage is that presolving may reveal inconsistent constraints, thus providing an early warning about an incorrect problem formulation or data error. Depending on the solver and problem, presolving may make the ‘‘`solve`’’ step faster, because the solver sees a smaller, simpler problem.

Presolve Overview

In their oft-cited paper of 1975, Brearley, Mitra, and Williams [5] discuss presolving linear programming problems (LPs). They recommend recursively

(i) folding singleton rows into bounds on the variables;

(ii) omitting inequalities that will always be slack;

(iii) deducing bounds from constraints that involve several bounded variables; and

(iv) deducing bounds on dual variables.

Because we are concerned in general with nonlinear problems and because we may transmit several objectives to the solver (which might select one of them to optimize or might use all in a multi-criterion optimization algorithm), we do not currently attempt to deduce dual variable bounds (iv). Our current presolve algorithm is thus just a ''primal presolve'' algorithm, a combination of (i), (ii), and (iii) that offers a choice of deduced bounds explained below.

Presolve Details

Most solvers treat bounds on variables separately from more general constraints. AMPL therefore conveys variable bounds separately from general constraints when transmitting a problem to a solver. Suppose a linear constraint is a ''singleton row'', i.e., involves just one variable. If it is an equality constraint, then it *fixes* the variable, i.e., determines its value, and we can remove both the constraint and the variable from the problem. Removing the variable entails updating b and d in (2) or (4), i.e., the left- and right-hand sides of the general constraints. Otherwise, a singleton row implies a lower or upper bound on the variable, and we can remove the constraint after folding the bounds it implies into ℓ and u in (3).

For each constraint, we maintain strengthened versions $\tilde{\ell}$ and $\tilde{u}$ of ℓ and u, and vectors $\tilde{b}$ and $\tilde{d}$ of deduced bounds on the range $\{ g(x) : \tilde{\ell} \le x \le \tilde{u} \}$ of the constraint body $g(x)$. The body (component of g) of each general constraint consists of a sum of terms; for a linear constraint, each term has the form *constant* $\times$ *variable*. Each deduced bound has two components: a bound computed from the finitely bounded terms in the constraint, and a count of unbounded terms; the bound is considered infinite unless the count is zero. Presently we treat all nonlinear terms as having infinite range, i.e., as contributing one to the count of infinities for its components in $\tilde{b}$ and $\tilde{d}$; we clearly have room for improvement here. Each time we sharpen a variable's bounds (or fix the variable), we update $\tilde{b}$ and $\tilde{d}$, possibly reducing some counts of unbounded terms.

Our presolve algorithm consists of two parts: a basic part that carries out the steps (i) and (ii) just discussed, and an extended part that deduces bounds from constraints which involve two or more linear terms. The basic algorithm maintains a stack of constraints to process. The overall algorithm begins by pushing all linear constraints involving at most one term onto the stack. (It is possible for an AMPL model and data to specify empty constraints: constraints whose bodies have no terms. Moreover, a constraint may become empty when the presolve algorithm fixes a variable.) The basic presolve algorithm proceeds by processing the constraint on top of the stack. Constraints having one term either fix the involved variable or imply bounds on it. Constraints whose term count drops below 2 as a result of fixing a variable are pushed onto the stack (unless they are already on it, as determined from a constraint status vector). Feasible empty constraints can simply be removed from the problem. Constraints diagnosed as infeasible after a variable is fixed elicit an error message and are retained. (AMPL denies the first request to solve a problem that the presolve algorithm finds infeasible, but it honors subsequent requests by passing the problem to the designated solver. Most solvers have feasibility tolerances that allow small infeasibilities. AMPL's diagnosis of infeasibility may arise from roundoff error, in which case the solver may report successful solution of the problem.)

Once the basic presolve algorithm ends (when the stack is empty), we examine linear constraints having two or more terms. If, say, $d_i < \infty$ and $\tilde{b}_i$ involves one infinity caused by variable x_k, then the ith constraint has the form

$$b_i \le \sum_j A_{i,j} x_j \le d_i$$

and either $A_{i,k} < 0$ and $\tilde{u}_k = +\infty$, or else $A_{i,k} > 0$ and $\tilde{\ell}_k = -\infty$. If $A_{i,k} > 0$, constraint i implies

$$x_k \le (d_i - \sum_{\substack{j \ne k \\ A_{i,j} > 0}} A_{i,j} \tilde{\ell}_j - \sum_{\substack{j \ne k \\ A_{i,j} < 0}} A_{i,j} \tilde{u}_j) \,/\, A_{i,k}; \tag{5}$$

if the right-hand side of (5) is less than $\tilde{u}_k$, then we can reduce $\tilde{u}_k$ accordingly. Similarly, if $A_{i,k} < 0$, constraint i implies

$$x_k \ge (b_i - \sum_{\substack{j \ne k \\ A_{i,j} > 0}} A_{i,j} \tilde{\ell}_j - \sum_{\substack{j \ne k \\ A_{i,j} < 0}} A_{i,j} \tilde{u}_j) \,/\, A_{i,k}; \tag{6}$$

and if the right-hand side of (6) exceeds $\tilde{\ell}_k$, we can accordingly increase $\tilde{\ell}_k$. If $\tilde{d}_i$ involves no infinities, we have a similar opportunity to update bounds on *all* the variables involved in constraint i; moreover, if the updates result in $\tilde{\ell}_k = \tilde{u}_k$, constraint i fixes all its variables. The situation is analogous if $\tilde{b}_i$ involves at most one infinity: we deduce a lower bound on x_k if $A_{i,k} > 0$, and an upper bound if $A_{i,k} < 0$; and we may be able to fix all the hitherto unfixed variables appearing in constraint i.

Sometimes we can improve bounds by allowing a constraint to participate in the above deductions more than once. Indeed, each time we improve a bound on one of the variables involved in constraint i, it is worth considering whether constraint i might imply better bounds on some other variables. This could lead to an infinite sequence of bound improvements. Specifically, if $q + 1$ constraints jointly imply fixed values for $q > 1$ variables, the iteration just described amounts to a Gauss-Seidel (or more general chaotic relaxation) iteration for computing the values of those variables. Consider, for example, the constraints

c1: $x_1 + x_2 \ge 2$

c2: $x_1 - x_2 \le 0$

c3: $0.1 \cdot x_1 + x_2 \le 1.1$

c4: $x_1 \ge 0.$

Constraints c1, c2, and c3 jointly imply that $x_1 = x_2 = 1$. Although the singleton c4 ends up being slack, it is needed to start the process by giving $\tilde{b}_3$ an infinity count of 1: then

c3 implies $x_2 \le 1.1$
c2 implies $x_1 \le 1.1$
c1 implies $x_1 \ge 0.9$
c2 implies $x_2 \ge 0.9$
c3 implies $x_2 \le 1.01$
c2 implies $x_1 \le 1.01$
c1 implies $x_1 \ge 0.99$
c2 implies $x_2 \le 0.99$
c3 implies $x_2 \le 1.001$
etc.

We limit the number of Gauss-Seidel iterations by allowing only a finite number of ''passes''. After the basic presolve algorithm stops, we push onto the constraint stack all constraints having 2 or more remaining terms and at most one infinity in either $\tilde{b}$ or $\tilde{d}$. Then we return to the basic algorithm, augmented by logic for deducing bounds from constraints with two or more terms. During this pass, we push onto a separate stack any linear constraint whose infinity count drops to one or whose infinity count is at most one and one of whose variables has a bound updated. This limits the work of a pass to time proportional to the remaining number of nonzeros in the remaining constraints. At the end of a pass, if the separate stack is not empty and we have not reached the pass limit, we transfer the separate stack to the presolve stack and begin another pass.

Once the iterative part of the presolve algorithm ends, the deduced bounds $\tilde{b}$ and $\tilde{d}$ may imply that some constraints can be discarded without changing the problem's feasible region. If $\tilde{b}_i > b_i$, then we change b_i to $-\infty$. (If the count of infinite terms for $\tilde{b}_i$ is positive, we regard $\tilde{b}_i$ as $-\infty$, and similarly for $\tilde{d}_i$.)

Likewise, if $\tilde{d}_i < d_i$, we change d_i to $+\infty$. These changes may turn constraint i from a range constraint (one with $0 < d_i - b_i < \infty$) into a one-sided constraint or may let us discard the constraint altogether (from the problem presented to the solver).

Degeneracy

Bounds deduced in the extended presolve passes are redundant and thus make the problem more degenerate. Not surprisingly, if AMPL passes the strongest variable bounds it can deduce to a simplex-based solver, the solver often takes more iterations than it takes with variable bounds relaxed to those implied by eliminated constraints. AMPL therefore maintains two sets of variable bounds — the strongest bounds it can deduce, and bounds that it does not know to be redundant with the constraints passed to the solver. By default it passes the latter set, but the ‘‘`var_bounds 2`’’ results reported below correspond to the stronger bounds. Degeneracy is much less of an issue for interior-point than for simplex algorithms, but the effect of changing bound sets is very problem- and algorithm-dependent. Interior-point algorithms sometimes fare worse with tighter bounds because they expend more work per iteration when variables must lie between finite bounds than when variables are bounded only on one side. And despite increased degeneracy, simplex algorithms sometimes run better with the tighter bounds because they choose a different pivot order.

Directed Roundings

A preliminary version of the computational experience reported below revealed a case (*netlib*'s `lp/data/maros`) where AMPL's default presolve settings made it discard constraints that kept the problem from being unbounded. Of course, roundoff error was to blame for this difficulty. When we modified the presolve algorithm to use the directed roundings that are available with IEEE arithmetic [1,2], this difficulty went away. On four other problems from *netlib*'s `lp/data` (*greenbea*, *greenbeb*, *perold*, and *woodw*), AMPL's presolve reported inconsistent constraints before we introduced directed roundings. Because they allow small infeasibilities, the solvers we tried found ‘‘correct’’ solutions to these four problems despite AMPL's diagnoses of infeasibility. Again, these diagnoses went away when we introduced directed roundings. Interestingly enough, on an IBM Risc System 6000, which by default computes $\alpha \times \beta + \gamma$ with just one rounding error (a ‘‘fused multiply-add’’), one of these infeasibility diagnostics returned. Our current policy is to use a compiler option that forbids fused multiply-adds in the presolve algorithm.

Using directed roundings to compute $\tilde{b}$ and $\tilde{d}$ usually only increases by a few percent the time AMPL spends to process a problem. For the larger problems we have examined, the directed roundings seldom add more than 1% to the sum of times for AMPL and the solver.

Primarily for machines that do not offer directed roundings, we have introduced a tolerance τ (option `constraint_drop_tol`, which is 0 by default) and have adjusted AMPL's presolve algorithm so it only changes b_i to $-\infty$ if $\tilde{b}_i - b_i \geq \tau$ and only changes $\tilde{d}_i$ to $+\infty$ if $d_i - \tilde{d}_i \geq \tau$. For example, before we added directed roundings, setting τ to 10^{-13} sufficed to eliminate the trouble with problem *maros*.

Recovering Dual Variables

Suppose x solves (1), (2), and (4). Then there exist dual variables y for (1), (2), and (4) that satisfy

$$(c - A^{\mathrm{T}} y)_j \begin{cases} \geq 0 \text{ if } x_j = \ell_j \\ \leq 0 \text{ if } x_j = u_j \\ = 0 \text{ if } \ell_j < x_j < u_j \end{cases} \tag{7a}$$

with

$$y_i \begin{cases} \geq 0 \text{ if } (Ax)_i = b_i \\ \leq 0 \text{ if } (Ax)_i = d_i \\ = 0 \text{ if } b_i < (Ax)_i < d_i \end{cases} \tag{7b}$$

where $c = \nabla f(x)$ is the gradient of the objective function *f*. When it invokes a solver, AMPL expects the solver to return dual values for the constraints it sees. To compute dual variables for constraints eliminated by presolving, it is necessary to record which eliminated constraints were responsible for the bounds conveyed to the solver. We then examine the eliminated constraints in the reverse order of their elimination. Constraints *i* that did not imply any of the bounds conveyed to the solver get $y_i = 0$. Constraints *i* that implied a single bound must have had one remaining nonzero coefficient $A_{i,j}$, and we choose y_i to satisfy component *j* of (7a) and *i* of (7b); this has no effect on the other components of (7) for variables and constraints not yet fixed or removed when constraint *i* was eliminated.

The only other case is a constraint *i* that, together with several then-current variable bounds, fixed several variables, say x_j for $j \in J$. The use described above of a stack in the presolve algorithm ensures that the variable bounds $\tilde{\ell}$ and $\tilde{u}$ that were current when constraint *i* was processed satisfied $\tilde{\ell}_j < \tilde{u}_j$ for all $j \in J$. Thus if $J^+ = \{j \in J\colon A_{i,j} > 0\}$ and $J^- = \{j \in J\colon A_{i,j} < 0\}$, then $J = J^+ \cup J^-$ and exactly one of

$$\sum_{j \in J^+} A_{i,j} \tilde{u}_j + \sum_{j \in J^-} A_{i,j} \tilde{\ell}_j = \tilde{b}_i \tag{8}$$

or

$$\sum_{j \in J^+} A_{i,j} \tilde{\ell}_j + \sum_{j \in J^-} A_{i,j} \tilde{u}_j = \tilde{d}_i \tag{9}$$

holds. In either case, there is a whole ray of *y* values that will satisfy the relevant components of (7). Let $\sigma = 1$ if (8) holds and -1 if (9) holds. Then all sufficiently large choices of σy_i can satisfy the components of (7a) corresponding to *J* and component *i* of (7b). AMPL chooses y_i to make one of these conditions hold with equality.

Computational Experience

As one example of the effects of AMPL's presolve, Table 1 shows resulting problem sizes and times for some of the problems considered in our first AMPL paper, [7]. Here and below, *presolve 0* means even the basic presolve algorithm was omitted; *presolve 1* means just the basic algorithm was used; *presolve 10* means 9 passes of the extended presolve algorithm were allowed. The *var_bounds 2* lines are for the alternate stronger bounds computed for *presolve 10*. The final column shows ''solve'' times (exclusive of the relatively small problem input and solution output times) for MINOS 5.4 running on an SGI Indigo with 50 MHz clock (R4000 processor with R4010 floating-point chip). This small sample of results illustrates how problem-dependent the effects of presolving are; as subsequent graphs show, these effects also depend on the solver used.

The figures below show solve-time ratios (defined below) for several solvers on the LP test problems in the `lp/data` directory [9] of *netlib* [6]. These problems are expressed in ''MPS format'' (which is described, e.g., in chapter 9 of [12]). We used an *awk* script, `m2a`, to turn the MPS format into data for a suitable AMPL model, `mps1.mod`, both of which are available from *netlib*'s `ampl/models` directory; the appendix gives problem sizes resulting from the three *presolve* settings. The time needed to present the problem to the solvers was generally small compared with the time the solvers needed to find a solution, particularly for the larger problems. For instance, Table 2 shows times (seconds of user plus system time under default conditions, corresponding to the *presolve 10* time results) for the major AMPL and solver steps to solve problem *pilot*. In Table 2, ''input'' time consists mostly of reading the data for *pilot*, ''genmod'' time is everything else before presolving, and ''output'' time is for writing a binary file that encodes the problem. Table 2 also gives times for several solvers: `alpo` is one of Vanderbei's interior-point codes [15, 17], and `loqo` is another [16, 18]; `cplex` is CPLEX [4] version 2.0; `minos` is MINOS [13, 14] version 5.4; and `osl` uses the default simplex algorithm of OSL [3, 10] version 1.2.

In the figures that follow, we have sorted the problems in order of increasing solve time by CPLEX with all defaults (again corresponding to the *presolve 10* results). Table 3 shows the sort order we used.

The figures that follow show solve-time ratios. Again using an SGI Indigo with 50 MHz clock, we measured the time each solver needed to solve each problem (after it had been read into memory and before

Problem	option	rows	cols	nonzeros	iters	seconds
cms	presolve 0	2521	24277	142893	6681	324.83
	presolve 1	1681	24277	142053	12285	517.97
	presolve 10	1681	24277	142053	12285	517.44
	var_bounds 2	"	"	"	10041	433.23
dist08	presolve 0	789	2728	8085	298	2.72
	presolve 1	448	2451	7252	350	2.05
	presolve 10	393	2123	6268	303	1.68
	var_bounds 2	"	"	"	335	1.70
dist13	presolve 0	1264	2262	6629	280	3.39
	presolve 1	447	1563	4510	325	1.58
	presolve 10	377	1363	3910	245	1.06
	var_bounds 2	"	"	"	249	1.08
git2	presolve 0	410	1089	3756	383	1.35
	presolve 1	376	1089	3746	359	1.26
	presolve 10	286	1089	3051	324	0.97
	var_bounds 2	"	"	"	402	1.09
git3	presolve 0	1330	12745	47980	3881	60.80
	presolve 1	1330	12745	47980	3381	60.60
	presolve 10	1239	12745	46441	3634	53.60
	var_bounds 2	"	"	"	6014	86.28
prod08	presolve 0	469	560	1807	321	1.44
	presolve 1	417	551	1716	278	1.17
	presolve 10	417	551	1716	278	1.17
	var_bounds 2	"	"	"	273	1.13
prod13	presolve 0	729	885	2887	463	3.11
	presolve 1	647	871	2736	539	3.20
	presolve 10	647	871	2736	539	3.17
	var_bounds 2	"	"	"	466	2.83

Table 1. *Sample presolve results.*
Times are `minos` solve seconds on a 50MHz SGI Indigo.

the solution was written back) with several presolve variants: none (*presolve 0*), basic presolve (*presolve 1*), and 9 extended presolve passes with either conservative (*presolve 10*) or aggressive (*var_bounds 2*) variable bounds. We divided the latter three times by the first to obtain the solve-time ratios presented in the figures, denoted ‘‘1’’, ‘‘+’’ and ‘‘∗’’, respectively. Table 4 shows the means and standard deviations of these ratios, excluding any whose time for *presolve 10* was less than 0.2 seconds.

AMPL times		Solver	Read	Solve
input	4.31	`alpo`	0.51	414.61
genmod	2.45	`cplex`	0.63	374.36
presolve	1.68	`loqo`	0.41	409.2
output	0.41	`loqo`	0.41	409.2
Total	8.85	`minos`	0.41	949.32

Table 2. *Indigo seconds for ‘‘pilot’’.*

Seq	Name	Seq	Name	Seq	Name	Seq	Name	Seq	Name
1	afiro	20	share1b	39	ship12s	58	wood1p	77	pilot.we
2	recipe	21	agg3	40	finnis	59	czprob	78	pilotnov
3	sc50a	22	scorpion	41	tuff	60	sctap3	79	bnl2
4	kb2	23	standata	42	scagr25	61	scsd8	80	80bau3b
5	beaconfd	24	sc205	43	shell	62	degen2	81	fit2d
6	sc50b	25	ship04s	44	grow7	63	scfxm3	82	truss
7	stocfor1	26	sctap1	45	etamacro	64	maros	83	greenbeb
8	agg	27	forplan	46	gfrd-pnc	65	d6cube	84	pilot.ja
9	adlittle	28	brandy	47	fffff800	66	fit1p	85	greenbea
10	sc105	29	israel	48	boeing1	67	grow15	86	degen3
11	vtp.base	30	seba	49	ship08l	68	woodw	87	d2q06c
12	scagr7	31	standmps	50	fit1d	69	cycle	88	fit2p
13	blend	32	ship04l	51	sctap2	70	pilot4	89	pilot
14	share2b	33	scfxm1	52	scfxm2	71	bnl1	90	stocfor3
15	bore3d	34	ship08s	53	scrs8	72	grow22	91	pilot87
16	boeing2	35	e226	54	sierra	73	stocfor2	92	dfl001
17	agg2	36	bandm	55	stair	74	nesm		
18	lotfi	37	scsd6	56	ship12l	75	perold		
19	scsd1	38	capri	57	ganges	76	25fv47		

Table 3. *Ordering of* `lp/data` *problems in subsequent figures.*

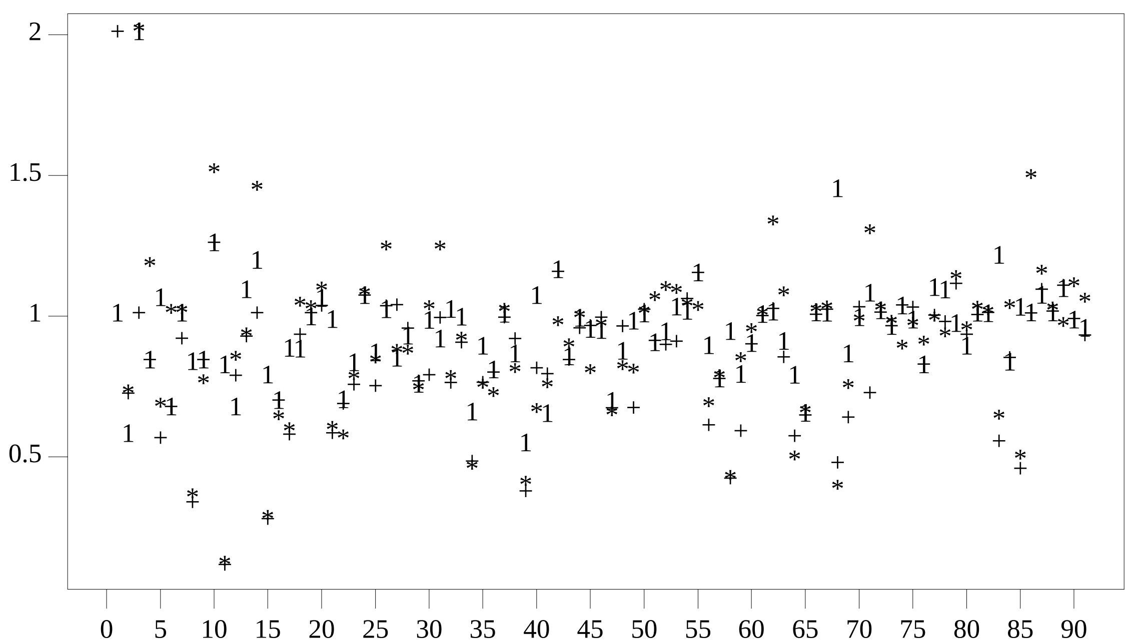

Figure 1. *Time ratios for* `minos`.
1 = *presolve 1*; + = *presolve 10*; ∗ = *var_bounds 2*.

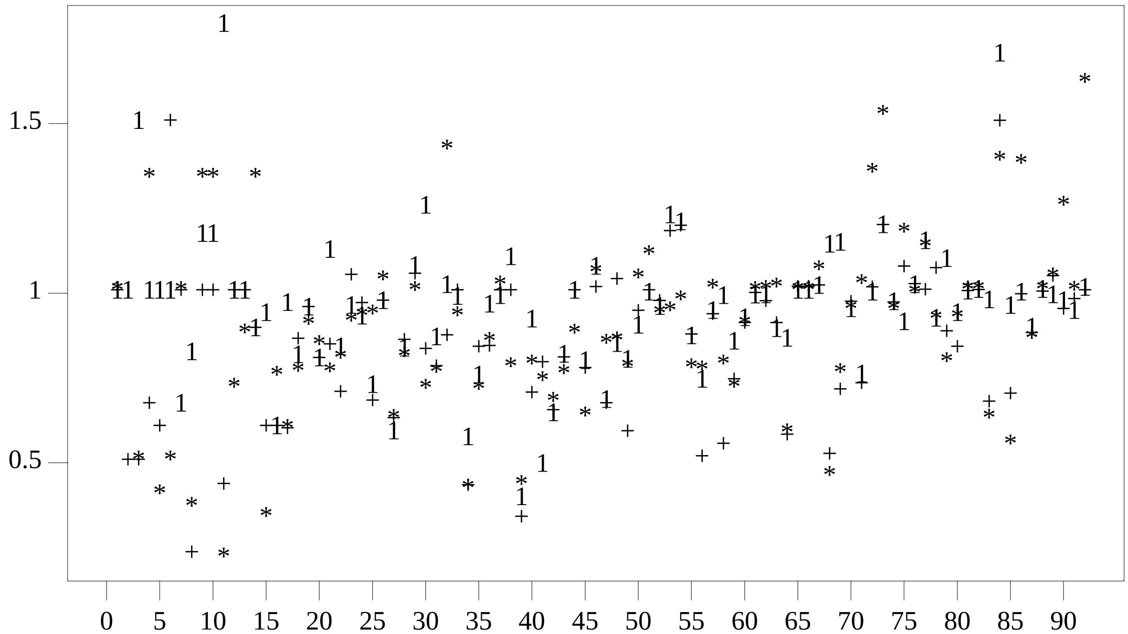

Figure 2. *Time ratios for* `cplex`.
1 = *presolve 1*; + = *presolve 10*; ∗ = *var_bounds 2*.

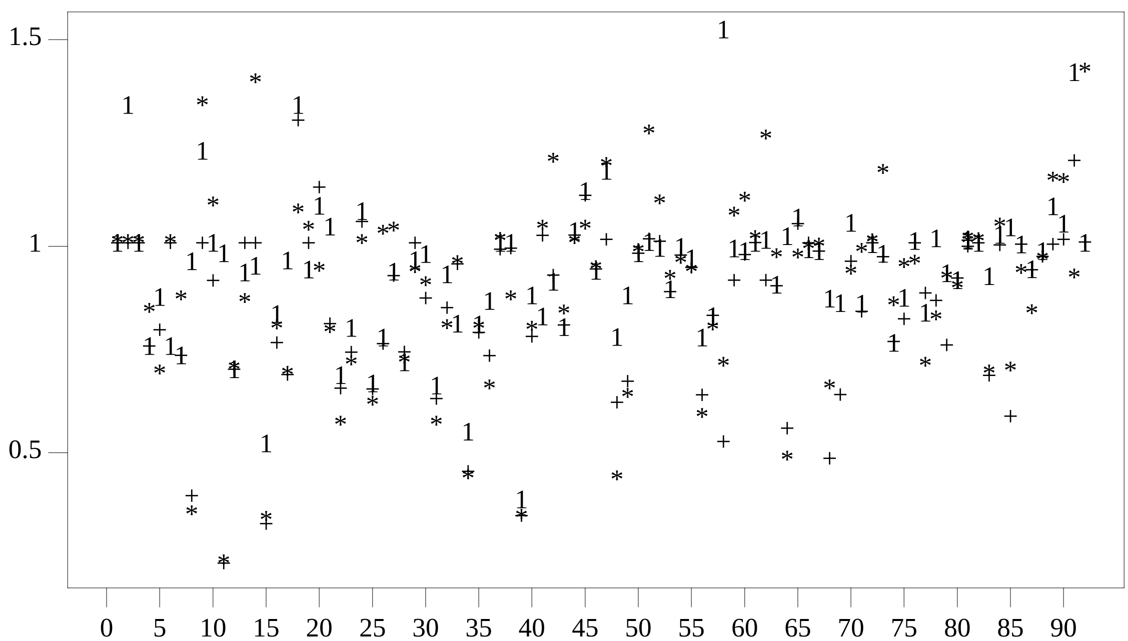

Figure 3. *Time ratios for* `osl`.
1 = *presolve 1*; + = *presolve 10*; ∗ = *var_bounds 2*.

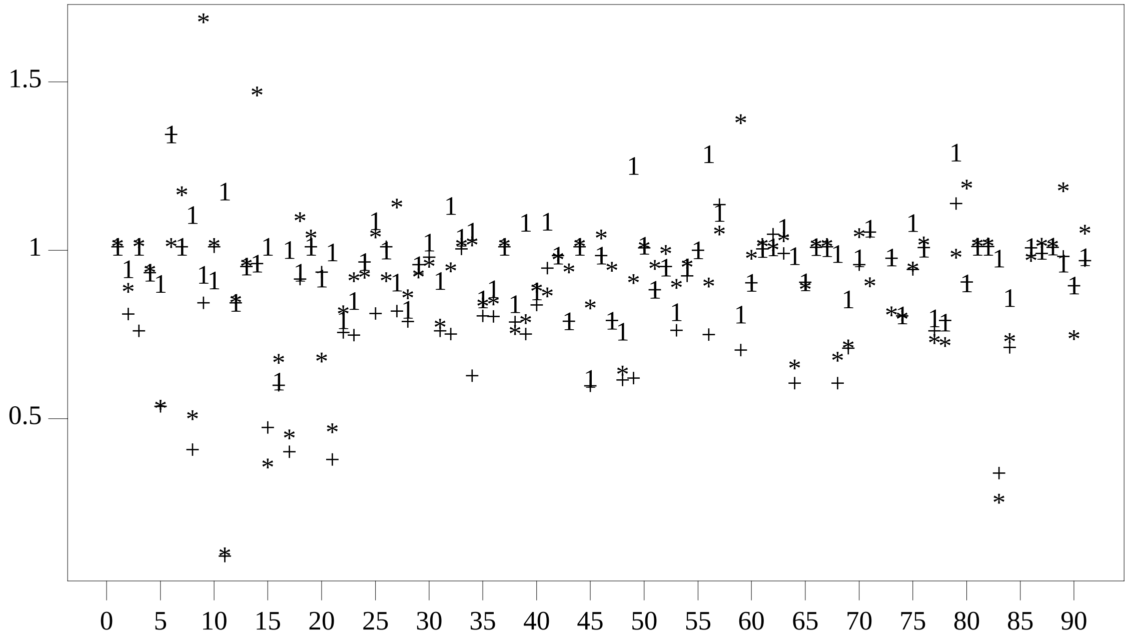

Figure 4. *Time ratios for* `alpo`.
1 = *presolve 1*; + = *presolve 10*; ∗ = *var_bounds 2*.

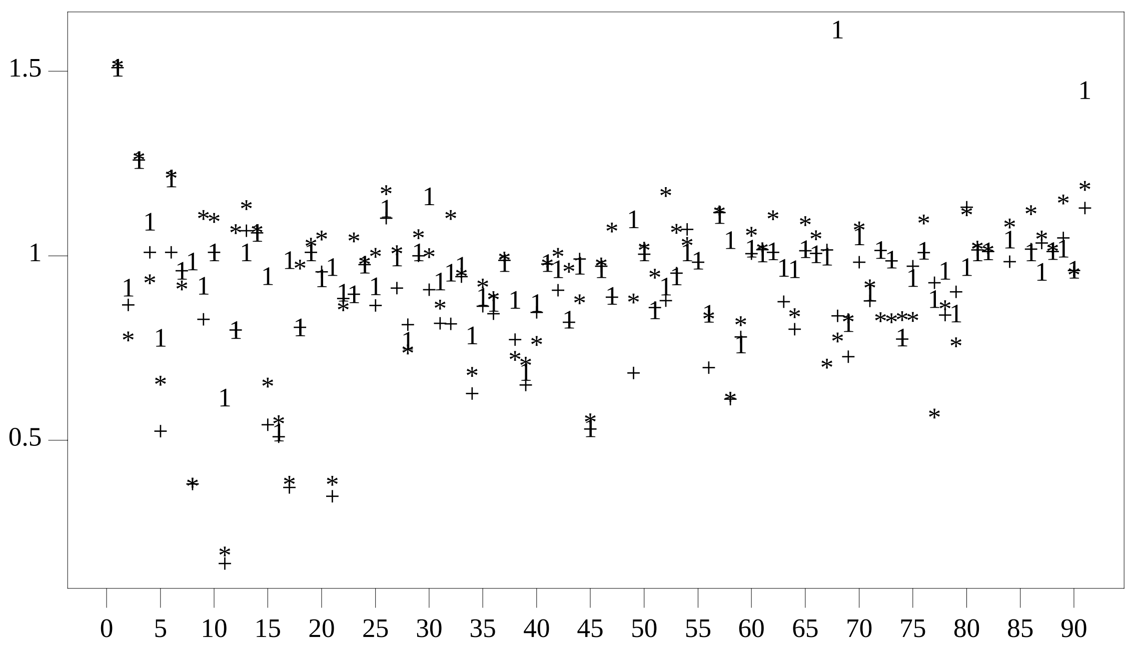

Figure 5. *Time ratios for* `loqo`.
1 = *presolve 1*; + = *presolve 10*; ∗ = *var_bounds 2*.

Solver	*presolve 1* mean	*presolve 1* dev.	*presolve 10* mean	*presolve 10* dev.	*var_bounds 2* mean	*var_bounds 2* dev.
`alpo`	0.95	0.12	0.83	0.18	0.87	0.19
`cplex`	0.94	0.18	0.88	0.19	0.91	0.23
`loqo`	0.95	0.15	0.87	0.18	0.90	0.19
`minos`	0.93	0.14	0.85	0.19	0.88	0.22
`osl`	0.93	0.17	0.85	0.19	0.87	0.22

Table 4. *Time ratio statistics.*

OSL has its own presolve algorithm that does some of the same things as AMPL's presolve algorithm, and that can also be asked to eliminate equality constraints involving two variables (*simplify 1*) or equality constraints of the form

$$x_j = \sum_{k \in S} x_k$$

with $j \notin S$ and $x_i \geq 0$, $i \in \{j\} \cup S$ (*simplify 2*). Figures 6 and 7 show time ratios for `osl` with *simplify 1* and *simplify 2*, respectively. The numerator is the time for `osl` with OSL's presolve divided by the time for `osl` with no presolving (by either AMPL or OSL); ''0'', ''1'', ''+'' and ''∗'' signify numerator runs with AMPL settings *presolve 0*, *presolve 1*, *presolve 10* and *var_bounds 2* (with *presolve 10*). Table 5 gives summary statistics for the `osl` runs; *simplify –1* is for runs with `osl`'s presolver turned off. Table 5 and Figures 6 and 7 omit ratios for runs where the *presolve 10* times were less than 0.2 seconds.

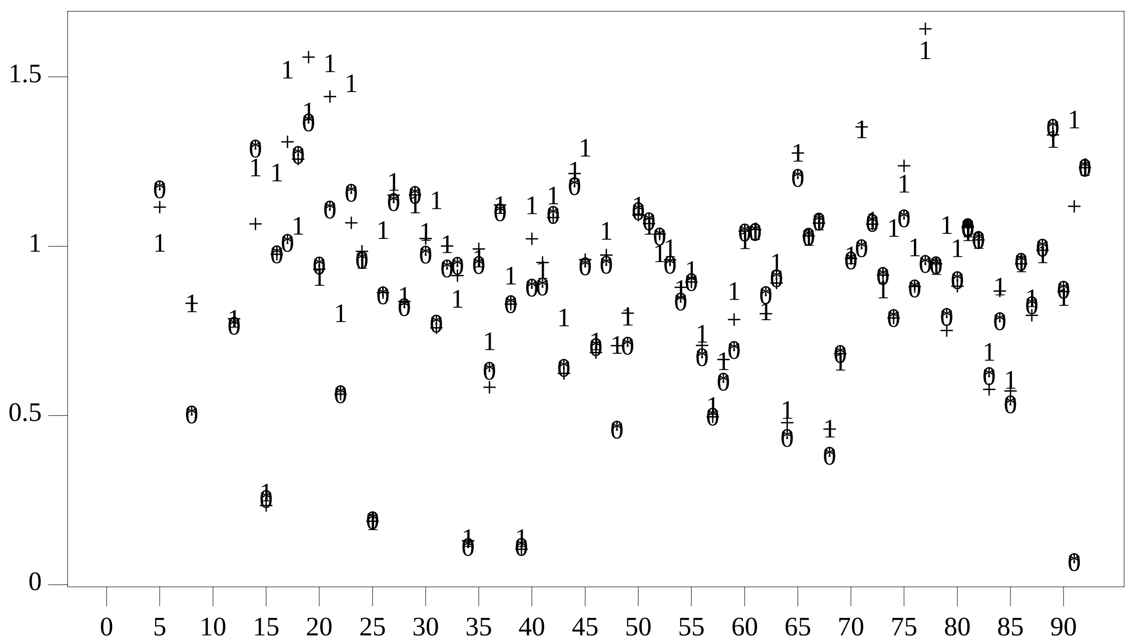

Figure 6. *Time ratios for* `osl` *with simplify 1.*
0 = *presolve 0*; 1 = *presolve 1*;
+ = *presolve 10*; ∗ = *var_bounds 2*.

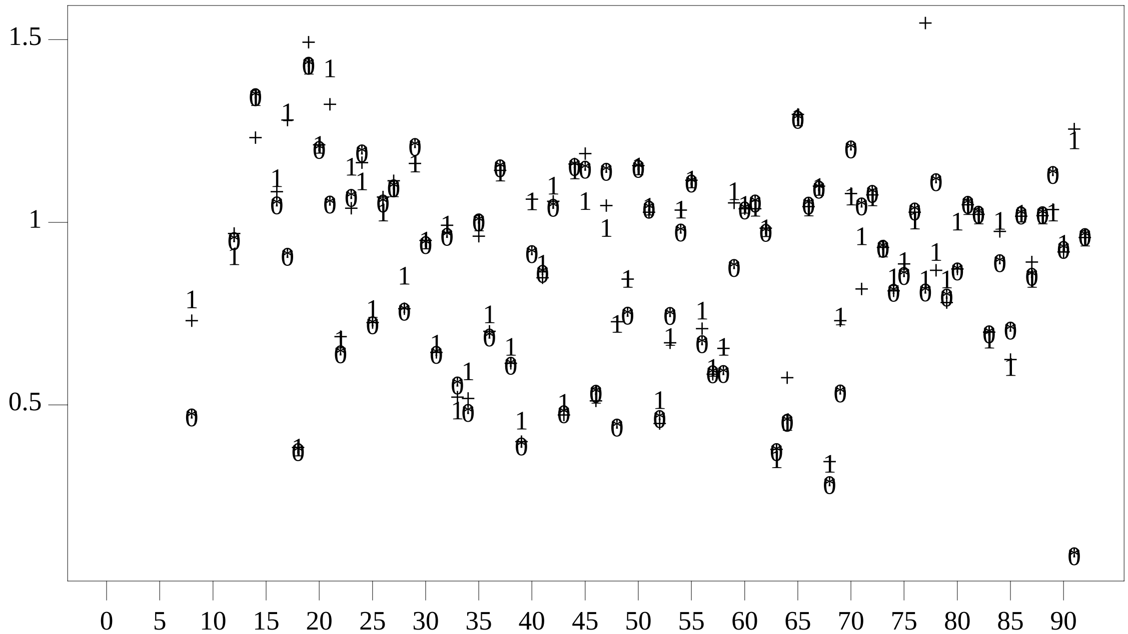

Figure 7. *Time ratios for* `osl` *with simplify 2*.
0 = *presolve 0*; 1 = *presolve 1*;
+ = *presolve 10*; ∗ = *var_bounds 2*.

	simplify –1		*simplify 1*		*simplify 2*	
	mean	dev.	mean	dev.	mean	dev.
`presolve 0`	1.00	0.00	0.95	0.29	0.90	0.25
`presolve 1`	0.93	0.17	0.90	0.29	0.90	0.26
`presolve 10`	0.85	0.19	0.85	0.27	0.86	0.27
`var_bounds 2`	0.87	0.22	0.90	0.35	0.90	0.30

Table 5. *Time ratio statistics for* `osl`.

Discussion

The results illustrated in the figures and summarized in Table 4 appear to be consistent with results reported by Lustig, Marsten, and Shanno in Figures 2 and 3 of [11]. All these results confirm that presolving can save time.

The summary statistics in Table 5 suggest that it can often be worthwhile for AMPL to carry out its presolve algorithm even when sending a problem to a solver that has its own presolve algorithm.

Though it is not obvious from Table 5, Figures 3, 6 and 7 reveal that OSL's *simplify 1* and *simplify 2* strategies are well worth using on some problems. Adding these strategies to AMPL's presolve algorithm would probably be worthwhile.

REFERENCES

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

[2] *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, Institute of Electrical and Electronics Engineers, New York, NY, 1987. ANSI/IEEE Std 854-1987.

[3] ‘‘Optimization Subroutine Library Guide and Reference, Release 2,’’ SC23-0519-03 (1992), IBM Corp..

[4] *Using the CPLEX Callable Library and CPLEX Mixed Integer Library,* CPLEX Optimization, Inc., 1992.

[5] A. L. Brearley, G. Mitra, and H. P. Williams, ‘‘Analysis of Mathematical Programming Problems Prior to Applying the Simplex Method,’’ *Math. Programming* **8** (1975), pp. 54–83.

[6] J. J. Dongarra and E. Grosse, ‘‘Distribution of Mathematical Software by Electronic Mail,’’ *Communications of the ACM* **30** #5 (May 1987), pp. 403–407.

[7] R. Fourer, D. M. Gay, and B. W. Kernighan, ‘‘A Modeling Language for Mathematical Programming,’’ *Management Science* **36** #5 (1990), pp. 519–554.

[8] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming,* The Scientific Press, 1993.

[9] D. M. Gay, ‘‘Electronic Mail Distribution of Linear Programming Test Problems,’’ *COAL Newsletter* #13 (1985), pp. 10–12.

[10] M. S. Hung, W. O. Rom, and A. D. Waren, *Optimization with OSL,* The Scientific Press, 1993.

[11] I. J. Lustig, R. E. Marsten, and D. F. Shanno, ‘‘Computational Experience with a Primal-Dual Interior Point Method for Linear Programming,’’ *Linear Algebra and Its Applications* **152** (1991), pp. 191–222.

[12] B. A. Murtagh, in *Advanced Linear Programming: Computation and Practice,* McGraw-Hill, New York (1981).

[13] B. A. Murtagh and M. A. Saunders, ‘‘A Projected Lagrangian Algorithm and its Implementation for Sparse Nonlinear Constraints,’’ *Math. Programming Study* **16** (1982), pp. 84–117.

[14] B. A. Murtagh and M. A. Saunders, ‘‘MINOS 5.1 User’s Guide,’’ Technical Report SOL 83-20R (1987), Systems Optimization Laboratory, Stanford University, Stanford, CA.

[15] R. J. Vanderbei, ‘‘A Brief Description of ALPO,’’ *OR Letters* **10** (1991), pp. 531–534.

[16] R. J. Vanderbei, ‘‘LOQO Users Manual,’’ Report SOR 92-5 (1992), Princeton University.

[17] R. J. Vanderbei, ‘‘ALPO: Another Linear Program Optimizer,’’ *ORSA J. Computing (to appear)* (1993).

[18] R. J. Vanderbei and T. J. Carpenter, ‘‘Symmetric Indefinite Systems for Interior-Point Methods,’’ *Math. Programming (to appear)* (1993).

Appendix: Problem Sizes for *lp/data* Problems

The tables that follow show the problem sizes for *presolve 0*, *presolve 1*, and *presolve 10* on successive lines. The Seq column gives the sequence numbers from Table 3.

Seq	Name	Rows	Cols	Nonzeros	Seq	Name	Rows	Cols	Nonzeros
76	25fv47	821	1571	10400	15	bore3d	233	315	1429
		777	1546	10247			189	273	1272
		777	1546	10247			138	189	726
80	80bau3b	2262	9799	21002	28	brandy	220	249	2148
		2026	9266	20046			124	208	1906
		2021	9247	19979			123	205	1882
9	adlittle	56	97	383	38	capri	271	353	1767
		53	96	374			255	321	1590
		53	96	374			249	321	1545
1	afiro	27	32	83	69	cycle	1903	2857	20720
		25	32	81			1605	2750	17139
		23	32	77			1528	2530	15426
8	agg	488	163	2410	59	czprob	929	3523	10669
		432	163	2304			737	3104	9417
		174	112	898			689	2770	8337
17	agg2	516	302	4284	87	d2q06c	2171	5167	32417
		481	301	4238			2098	5157	32321
		317	301	2814			2097	5157	32319
21	agg3	516	302	4300	65	d6cube	415	6184	37704
		481	301	4254			403	6183	37696
		322	301	2856			403	6183	37696
36	bandm	305	472	2494	62	degen2	444	534	3978
		258	425	2034			444	534	3978
		246	401	1927			442	534	3974
5	beaconfd	173	262	3375	86	degen3	1503	1818	24646
		136	229	3058			1503	1818	24646
		96	175	1995			1503	1818	24646
13	blend	74	83	491	92	dfl001	6071	12230	35632
		72	83	489			6071	12230	35632
		71	83	487			6071	12230	35632
71	bnl1	643	1175	5121	35	e226	223	282	2578
		572	1169	5049			164	271	2432
		558	1113	4818			161	260	2306
79	bnl2	2324	3489	13999	45	etamacro	400	688	2409
		2123	3455	13671			334	542	1868
		2110	3432	13557			333	542	1852
48	boeing1	351	384	3485	47	fffff800	524	854	6227
		304	380	2789			476	817	6042
		292	373	2309			475	817	6038
16	boeing2	166	143	1196	40	finnis	497	614	2310
		125	143	801			419	549	1957
		125	143	801			397	543	1904

Seq	Name	Rows	Cols	Nonzeros	Seq	Name	Rows	Cols	Nonzeros
50	fit1d	24	1026	13404	74	nesm	662	2923	13288
		24	1026	13404			646	2740	13054
		24	1026	13404			646	2740	13054
66	fit1p	627	1677	9868	75	perold	625	1376	6018
		627	1677	9868			620	1308	5819
		627	1677	9868			597	1269	5630
81	fit2d	25	10500	129018	89	pilot	1441	3652	43167
		25	10500	129018			1428	3447	41059
		25	10500	129018			1391	3397	40805
88	fit2p	3000	13525	50284	84	pilot.ja	940	1988	14698
		3000	13525	50284			881	1673	11643
		3000	13525	50284			825	1591	11319
27	forplan	161	421	4563	77	pilot.we	722	2789	9126
		134	418	4493			722	2711	8862
		131	415	4396			704	2680	8734
57	ganges	1309	1681	6912	70	pilot4	410	1000	5141
		1125	1497	6544			402	962	5025
		1124	1497	6532			393	951	4961
46	gfrd-pnc	616	1092	2377	91	pilot87	2030	4883	73152
		590	1066	2325			2010	4658	70639
		590	1066	2325			2003	4646	70595
85	greenbea	2392	5405	30877	78	pilotnov	975	2172	13057
		2315	5229	30144			886	1939	11988
		1967	4156	24106			871	1919	11880
83	greenbeb	2392	5405	30877	2	recipe	91	180	663
		2313	5215	30074			83	151	622
		1976	4167	24206			75	137	596
67	grow15	300	645	5620	10	sc105	105	103	280
		300	645	5620			104	103	280
		300	645	5620			104	103	280
72	grow22	440	946	8252	24	sc205	205	203	551
		440	946	8252			203	202	550
		440	946	8252			203	202	550
44	grow7	140	301	2612	3	sc50a	50	48	130
		140	301	2612			49	48	130
		140	301	2612			49	48	130
29	israel	174	142	2269	6	sc50b	50	48	118
		163	142	2258			48	48	118
		163	142	2258			48	48	118
4	kb2	43	41	286	42	scagr25	471	500	1554
		43	41	286			347	499	1423
		43	41	286			347	499	1423
18	lotfi	153	308	1078	12	scagr7	129	140	420
		134	300	1017			95	139	379
		134	300	1017			95	139	379
64	maros	846	1443	9614	33	scfxm1	330	457	2589
		803	1391	9437			287	448	2515
		694	1112	7237			281	439	2476

Seq	Name	Rows	Cols	Nonzeros	Seq	Name	Rows	Cols	Nonzeros
52	scfxm2	660	914	5183	49	ship08l	778	4283	12802
		574	896	5035			680	4259	12676
		562	878	4957			520	3149	9346
63	scfxm3	990	1371	7777	34	ship08s	778	2387	7114
		861	1344	7555			408	2091	6172
		843	1317	7438			326	1632	4795
22	scorpion	388	358	1426	56	ship12l	1151	5427	16170
		297	335	1254			833	5223	15504
		292	331	1227			687	4224	12507
53	scrs8	490	1169	3182	39	ship12s	1151	2763	8178
		452	1137	3042			461	2187	6396
		450	1134	3031			417	1996	5823
19	scsd1	77	760	2388	54	sierra	1227	2036	7302
		77	760	2388			1212	2016	7242
		77	760	2388			1135	2016	7088
37	scsd6	147	1350	4316	55	stair	356	467	3856
		147	1350	4316			356	385	3666
		147	1350	4316			356	385	3666
61	scsd8	397	2750	8584	23	standata	359	1075	3031
		397	2750	8584			311	1046	2889
		397	2750	8584			301	1038	2843
26	sctap1	300	480	1692	31	standmps	467	1075	3679
		284	480	1638			419	1046	3537
		284	480	1638			403	1038	3275
51	sctap2	1090	1880	6714	7	stocfor1	117	111	447
		1033	1880	6489			98	100	398
		1033	1880	6489			98	100	398
60	sctap3	1480	2480	8874	73	stocfor2	2157	2031	8343
		1408	2480	8595			2129	2015	8255
		1408	2480	8595			2129	2015	8255
30	seba	515	1028	4352	90	stocfor3	16675	15695	64875
		515	1028	4352			16617	15663	64567
		450	898	4114			16617	15663	64567
20	share1b	117	225	1151	82	truss	1000	8806	27836
		110	220	1118			1000	8806	27836
		110	220	1118			1000	8806	27836
14	share2b	96	79	694	41	tuff	333	587	4520
		93	79	691			292	582	4514
		93	79	691			286	563	4324
43	shell	536	1775	3556	11	vtp.base	198	203	908
		487	1476	2958			165	182	764
		487	1476	2958			54	118	339
32	ship04l	402	2118	6332	58	wood1p	244	2594	70215
		348	2114	6292			243	2594	70214
		317	1915	5695			171	1802	48578
25	ship04s	402	1458	4352	68	woodw	1098	8405	37474
		260	1366	4048			1097	8405	37473
		241	1291	3823			736	5549	24114