# Using AMPL/MINOS

MINOS is an optimization package for linear and nonlinear mathematical programs in continuous variables. This supplement to *AMPL: A Modeling Language for Mathematical Programming* describes the most important features of MINOS for AMPL users. Further information on MINOS is available from:

Stanford Business Software, Inc.
2672 Bayshore Parkway, Suite 304
Mountain View, CA 94043

415-962-8719; fax 415-962-1869

Section §1 below describes the kinds of problems to which MINOS is applicable, and Section §2 explains in general terms how solver-specific directives can be passed from AMPL to MINOS. Sections §3 and §4 then briefly introduce the algorithms that MINOS uses for linear and for nonlinear programming, respectively, and describe the relevant algorithmic directives. Section §5 presents directives for controlling listings from MINOS, and Section §6 discusses options for restarting from a known solution or basis.

## §1 Applicability

MINOS is designed to solve the linear programs described in Chapters 1-8 and 11-12 of *AMPL: A Modeling Language for Mathematical Programming*, as well as the ''smooth'' nonlinear programs described in Chapter 13. Smooth nonlinear functions can be accommodated in both the objective and the constraints; nonlinear equation systems may also be solved by omitting an objective. Nonsmooth nonlinearities are also accepted, but MINOS is not designed for them and in general will not produce reliable results when they are present.

MINOS does not solve integer programs as described in Chapter 15. When presented with an integer program, MINOS ignores the integrality restrictions on the variables, as indicated by a message such as

```
MINOS 5.4: ignoring integrality of 5 variables
```

It then solves the continuous relaxation of the resulting problem, and returns a solution in which some of the integer variables may have fractional values.

MINOS can solve piecewise-linear programs, as described in Chapter 14, provided that they satisfy certain rules that permit them to be transformed to linear programs. Any piecewise-linear term in a minimized objective must be convex, its slopes forming an increasing sequence as in:

```
<<-1,1,3,5; -5,-1,0,1.5,3>> x[j]
```

Any piecewise-linear term in a maximized objective must be concave, its slopes forming a decreasing sequence as in:

```
<<1,3; 1.5,0.5,0.25>> x[j]
```

In the constraints, any piecewise-linear term must be either convex and on the left-hand side of a ≤ constraint (or equivalently, the right-hand side of a ≥ constraint), or else concave and on the left-hand side of a ≥ constraint (or equivalently, the right-hand side of a ≤ constraint). AMPL automatically converts the piecewise-linear program to a linear one, sends the latter to MINOS, and converts the solution back; the conversion has the effect of adding a variable to correspond to each linear piece. Piecewise-linear programs that violate the above rules are converted to integer programs, which are treated as described in the preceding paragraph; MINOS returns a solution to the continuous relaxation of the equivalent integer program, which in general is not optimal for the original piecewise-linear problem.

## §2 Controlling MINOS from AMPL

In many instances, you can successfully apply MINOS by simply specifying a model and data, setting the `solver` option to `minos`, and typing `solve`. For larger linear programs and more difficult nonlinear programs, however, you may need to pass specific directives to MINOS to obtain the desired results.

To give directives to MINOS, you must first assign an appropriate character string to the AMPL option called `minos_options`. When `solve` invokes MINOS, it breaks this string into a series of individual directives. Here is an example:

```
ampl: model diet.mod;
ampl: data diet.dat;

ampl: option solver minos;
ampl: option minos_options 'crash_option=0 \
ampl?    feasibility_tolerance=1.0e-8 scale=no \
ampl?    iterations_limit=100';

ampl: solve;
MINOS 5.4:

crash_option=0
feasibility_tolerance=1.0e-8
scale=no
iterations_limit=100

MINOS 5.4: optimal solution found.
4 iterations, objective 88.2
```

MINOS confirms each directive; it will display an error message if it encounters one that it does not recognize.

All of the directives described below have the form of an identifier, an = sign, and a value; unlike AMPL, MINOS treats upper-case and lower-case letters as being the same. You may store any number of concatenated directives in `minos_options`. The example above shows how to type all the directives in one long string, using the \ character to indicate that the string continues on the next line. Alternatively, you can list several strings, which AMPL will automatically concatenate:

```
ampl: option minos_options 'crash_option=0'
ampl?    ' feasibility_tolerance=1.0e-8 scale=no'
ampl?    ' iterations_limit=100';
```

In this form, you must take care to supply the space that goes between the directives; here we have put it before `feasibility_tolerance` and `iterations_limit`.

If you have specified the directives above, and then want to set `optimality_tolerance` to 1.0e–8 and change `crash_option` to 1, you might think to type:

```
ampl: option minos_options
ampl?    'optimality_tolerance=1.0e-8 crash_option=1';
```

This will replace the previous `minos_options` string, however; the other previously specified directives such as `feasibility_tolerance` and `scale` will revert to their default values. (MINOS supplies a default value for every directive not explicitly specified; defaults are indicated in the discussion below.) To append new directives to `minos_options`, use this form:

```
ampl: option minos_options $minos_options
ampl?     ' optimality_tolerance=1.0e-8 crash_option=1';
```

A $ in front of an option name denotes the current value of that option, so this statement appends more directives to the current directive string. As a result the string contains two directives for `crash_option`, but the new one overrides the earlier one.

## §3 Using MINOS for linear programming

For linear programs, MINOS employs the primal simplex algorithm described in many textbooks. The algorithm maintains a subset of *basic variables* (or, a *basis*) equal in size to the number of constraints. A *basic solution* is obtained by solving for the basic variables, when the remaining nonbasic variables are fixed at appropriate bounds. Each iteration of the algorithm picks a new basic variable from among the nonbasic ones, steps to a new basic solution, and drops some basic variable at a bound.

The coefficients of the variables form a *constraint matrix*, and the coefficients of the basic variables form a nonsingular square submatrix called the *basis matrix*. At each iteration, the simplex algorithm must solve certain linear systems involving the basis matrix. For this purpose MINOS maintains a *factorization* of the basis matrix, which is updated at most iterations, and is occasionally recomputed.

The *sparsity* of a matrix is the proportion of its elements that are not zero. The constraint matrix, basis matrix and factorization are said to be relatively *sparse* or *dense* according to their proportion of nonzeros. Most linear programs of practical interest have many zeros in all the relevant matrices, and the larger ones tend also to be the sparser.

The amount of RAM memory required by MINOS grows with the size of the linear program, which is a function of the numbers of variables and constraints and the sparsity of the coefficient matrix. The factorization of the basis matrix also requires an allocation of memory; the amount is problem-specific, depending on the sparsity of the factorization.

The following MINOS directives apply to the solution of linear programs. The letters $i$ and $r$ denote integer and real values, respectively. The symbol $\varepsilon$ represents the ''machine precision'' — the smallest value such that $1+\varepsilon$ can be distinguished from 1; for many machines it is $2^{-52} \approx 2.22\times10^{-16}$.

**`Crash_option=`** *i* **(default 3)**
**`Crash_tolerance=`** *r* **(default 0.1)**

These directives govern the initial determination of the basic variables, except when the basis is read from a file as described in §6 below.

For $i=0$, the initial basis contains a slack or artificial variable that MINOS automatically associates with each constraint; the basis matrix is an identity matrix. For $i>0$, MINOS uses a heuristic ''crash procedure'' to quickly choose a basis matrix that has fewer artificial variables and is likely to be nonsingular. Since artificial variables are generally not required in an optimal basis, this procedure tends to reduce the number of iterations to optimality.

When $i=1$ or 2 (which are equivalent for linear programs), the crash procedure determines a full initial basis before the first iteration. When $i=3$, the crash procedure initially determines a basis for just the equality constraints, and simplex iterations are performed until those constraints are satisfied (or determined to be infeasible). The crash procedure is then called again to extend the basis to all constraints.

In each column of the coefficient matrix, the crash procedure looks only at coefficients whose magnitude is more than $r$ times the largest magnitude in the column. Thus when $r=0$ it looks at all

coefficients; in such a case it finds a triangular basis matrix that is guaranteed to be nonsingular. As $r$ is increased the risk of singularity is also increased, but fewer artificial variables may be included. The default value of $r = 0.1$ generally works well, but the best value is highly dependent on problem structure and can be determined only through experimentation.

**`Expand_frequency=`** *i* **(default 10000)**

To discourage long series of degenerate iterations, at which no reduction in the objective is achieved, MINOS employs a procedure that lets nonbasic variables move beyond their bounds by a certain tolerance. If `feasibility_tolerance` (see below) is $r$, then the tolerance increases from $r/2$ to $r$ (in steps of $0.5r/i$) in the course of $i$ iterations, after which it is reset and the procedure begins anew. The default setting should be adequate for problems of moderate size.

**`Factorization_frequency=`** *i* **(default 100)**

The factorization of the basis matrix is updated at most $i$ times before it is recomputed directly. (An update occurs at the end of most iterations, though for some it can be skipped.) Higher values of $i$ may be more efficient for dense problems in which the factorization has two or more times as many nonzeros as the basis matrix, while lower values may be more efficient for very sparse problems. Lower values also require less memory for storing the updates.

**`Feasibility_tolerance=`** *r* **(default 1.0e–6)**

A variable or linear constraint is considered to be *feasible* if it does not lie outside its bounds by more than $r$. If MINOS determines that the sum of all infeasibilities cannot be reduced to zero, it reports ''no feasible solution'' and returns the last solution that it found. The sum of infeasibilities in the reported solution is not necessarily the minimum possible; however, if all the infeasibilities are quite small, it may be appropriate to raise $r$ by a factor of 10 or 100.

**`Iterations_limit=`** *i* **(default 99999999)**

MINOS stops after $i$ iterations, even if it has not yet identified an optimal solution.

**`LU_factor_tolerance=`** $r_1$ **(default 100.0)**
**`LU_update_tolerance=`** $r_2$ **(default 10.0)**

The values of $r_1$ and $r_2$ affect the stability and sparsity of the basis factorization during refactorization and updating, respectively. Smaller values $\geq 1$ favor stability, while larger values favor sparsity; the defaults usually strike a good compromise.

**`LU_singularity_tolerance=`** *r* **(default** $\varepsilon^{2/3} \approx 10^{-11}$**)**

When factoring the basis matrix, MINOS employs $r$ in a test for singularity; any column where singularity is detected is arbitrarily replaced by a slack or artificial column. If the basis becomes nearly singular as the optimum is approached, a larger value of $r < 1$ may give better performance.

**`Optimality_tolerance=`** *r* **(default 1.0e–6)**

The value of $r$ determines the exactness of MINOS's test for optimality. (In technical terms, at an optimum the dual variables and constraints may not violate their bounds by more than $r$ times a measure of the size of the dual solution.) The default setting is usually adequate. A smaller value of $r > 0$ may yield a better objective function value, at the cost of more iterations.

**`Partial_price=`** *i* **(default 10)**

MINOS incorporates a heuristic procedure to search through the nonbasic variables for a good candidate to enter the basis. (In technical terms, MINOS searches for a nonbasic variable that has a sufficiently large reduced cost, where ''sufficiently large'' is determined by a dynamic tolerance.) To save computational expense, only about $1/i$ times the number of variables are searched at an iteration, unless more must be searched to find one good candidate.

As $i$ is increased, iterations tend to take less time, but the simplex algorithm tends to require more iterations; changes in $i$ often have a significant effect on total solution time. The ideal value of $i$ is highly problem-specific, however, and must be determined by experimentation. For linear

programs that have many more variables than constraints, a larger $i$ is usually preferred. A setting of $i = 1$, which causes all nonbasic variables to be searched at each iteration, can be the best choice when the number of variables is small relative to the number of constraints.

**`Pivot_tolerance=`** *r* **(default** $\varepsilon^{2/3} \approx 10^{-11}$**)**

MINOS rejects a nonbasic variable chosen to enter the basis if the resulting basis matrix would be almost singular, according to a test in which the value of $r$ is employed. A larger value of $r$ strengthens the test, promoting stability but possibly increasing the number of rejections.

**`Scale=yes`**
**`Scale=no`**

**`Scale_option=`** *i* **(default 2)**
**`Scale_tolerance=`** *r* **(default 0.9)**

MINOS incorporates a heuristic procedure that scales the constraint matrix in a way that usually brings the elements closer to 1.0, and reduces the ratio between the largest and smallest element. Scaling merely converts the given linear program to an equivalent one, but it can affect the performance of the basis factorization routines, and the choices of nonbasic variables to enter the basis. In some but not all cases, the simplex method solves a scaled problem more efficiently than its unscaled counterpart.

For $i = 0$ no scaling is performed, while for $i = 1$ or $i = 2$ a series of alternating row and column scalings is carried out. For $i = 2$ a certain additional scaling is performed that may be helpful if the right-hand side (constant terms of the constraints) or solution contain large values; it takes into account columns that are fixed or have positive lower bounds or negative upper bounds. `Scale=yes` is the same as the default, while `scale=no` sets $i = 0$.

If $i > 0$, the value of $r$ affects the number of alternate row and column scalings. Raising $r$ to 0.99 (say) may increase the number of scalings and improve the final scaling slightly.

**`Weight_on_linear_objective=`** *r* **(default 0.0)**

When $r = 0$, MINOS carries out the simplex algorithm in two phases. Phase 1 minimizes the ‘‘sum of infeasibilities’’, which has to be zero in any feasible solution; then phase 2 minimizes or maximizes the ‘‘true objective’’ supplied by the linear program.

When $r > 0$, MINOS instead minimizes the objective defined by

$$\sigma\ r\ (\text{true objective}) + (\text{sum of infeasibilities})$$

where $\sigma$ is +1 for a minimization problem or –1 for a maximization. If a feasible minimum is found, then it is also optimal to the original linear program. Otherwise $r$ is reduced by a factor of 10, and another pass is begun; $r$ is set to zero after the fifth reduction, or whenever a feasible solution is encountered.

The effect of this directive is problem-dependent, and must be determined by experimentation. A good ‘‘intermediate’’ value of $r$ may yield an optimal solution in fewer iterations than the two-phase approach.

## §4 Using MINOS for nonlinear programming

For mathematical programs that are nonlinear in the objective but linear in the constraints, MINOS employs a reduced gradient approach, which can be viewed as a generalization of the simplex algorithm. In addition to the basic variables, the algorithm maintains a subset of *superbasic* variables that also may vary between their bounds. An iteration attempts to reduce the objective within the subspace of basic and superbasic variables, employing a quasi-Newton algorithm — adapted from unconstrained nonlinear optimization — to select a *search direction* and *step length.* When no further progress can be made with the current collection of basic and superbasic variables, a new superbasic variable is chosen from among the nonbasic ones; when a basic or superbasic variables encounters a bound as a result of a step, it is made nonbasic.

To deal with nonlinear constraints, MINOS further generalizes its algorithm by means of a projected Lagrangian approach. At each *major iteration*, a linear approximation to the nonlinear constraints is constructed around the current solution, and the objective is modified by adding two terms — the *Lagrangian* and the *penalty term* — which compensate (in a rough sense) for the inaccuracy of the linear approximation. The resulting *subproblem* is then solved by a series of *minor iterations* of the reduced gradient algorithm described in the previous paragraph. The optimum of this subproblem becomes the current solution for the next major iteration.

AMPL sends MINOS the information it needs to compute the values and partial derivatives of all nonlinear terms in the objective and constraints. The method of ''automatic differentiation'' is employed to compute the derivatives efficiently and exactly (subject to the limitations of computer arithmetic).

MINOS stores the linear and nonlinear parts of a mathematical program separately. The representation of the linear part is essentially the same as for a linear program, while the representation of the nonlinear part requires an amount of RAM memory depending on the complexity of the nonlinear terms. A factorization of the basis matrix is maintained as for linear programs, and has the same memory requirements. The quasi-Newton algorithm also maintains a factorization of a dense matrix that occupies memory proportional to the square of the number of superbasic variables. This number is small is many applications, but it is highly problem-dependent; in general terms, a problem that is ''more nonlinear'' will tend to have more superbasics.

The following MINOS directives are useful in solving nonlinear mathematical programs. Since the reduced gradient method incorporates many of the ideas from the simplex method, many of the directives for linear programming carry over in a natural way; others address the complications introduced by a nonlinear objective and constraints.

**`Completion=partial`** **(default)**
**`Completion=full`**

When there are nonlinear constraints, this directive determines whether subproblems should be solved to moderate accuracy (partial completion), or to full accuracy (full completion). Partial completion may reduce the work during early major iterations; an automatic switch to full completion occurs when several numerical tests indicate that the sequence of major iterations is converging. Full completion from the outset may lead to fewer major iterations, but may result in more minor iterations.

**`Crash_option=`** *i* **(default 3)**
**`Crash_tolerance=`** *r* **(default 0.1)**

`Crash_option` values 0 and 1, and the `crash_tolerance`, work the same as for linear programs. A `crash_option` of 2 first finds a basis for the linear constraints; after the first major iteration has found a solution that satisfies the linear constraints, the crash heuristic is called again to extend the basis to the nonlinear constraints. A value of 3 is similar except that linear equalities and linear inequalities are processed by separate passes of the crash routine, in the same way as for linear programs.

**`Expand_frequency=`** *i* **(default 10000)**

This directive works the same as for linear programming, but takes effect only when there is currently just one superbasic variable.

**`Factorization_frequency=`** *i* **(default 100)**

The basis is refactorized every *i* minor iterations, as in linear programming. If there are nonlinear constraints, the computation of a new linear approximation necessitates a refactorization at the start of each major iteration; thus the `minor_iterations` directive normally takes precedence over this one.

**`Feasibility_tolerance=`** *r* **(default 1.0e–6)**

Essentially the same as for linear programming.

MINOS allows nonlinear functions to be evaluated at points that satisfy the linear constraints and bounds to within the tolerance $r$. Thus every attempt should be made to bound the variables strictly away from regions where nonlinear functions are undefined. As an example, if the function `sqrt(X)` appears, it is essential to enforce a positive lower bound on `X`; for $r$ at its default value of 1.0e–6, the constraint `X>=1.0e-5` might be appropriate. (Note that `X>=0` does not suffice.)

When there are nonlinear constraints, this tolerance applies to each linear approximation. Feasibility with respect to the nonlinear constraints themselves is governed by the setting of the `row_tolerance` directive.

**`Hessian_dimension=`** $r$ **(default 50 or superbasics_limit)**

Memory proportional to $r^2$ is set aside for the factorization of the dense matrix required by the quasi-Newton method, limiting the size of the matrix to $r \times r$. To promote rapid convergence as the optimum is approached, $r$ should be greater than the expected number of superbasic variables whenever available memory permits. It need never be larger than 1 + the number of variables appearing in nonlinear terms, and for many problems it can be much less.

**`Iterations_limit=`** $i$ **(default 99999999)**

MINOS stops after a total of $i$ minor iterations, even if an optimum has not been indicated.

**`Lagrangian=yes`** **(default)**
**`Lagrangian=no`**

This directive determines the form of the objective function used for the linearized subproblems. The default value `yes` is highly recommended. The `penalty_parameter` directive is then also relevant.

If `no` is specified, the nonlinear constraint functions will be evaluated only twice per major iteration. Hence this option may be useful if the nonlinear constraints are very expensive to evaluate. However, in general there is a great risk that convergence may not be achieved.

**`Linesearch_tolerance=`** $r$ **(default 0.1)**

After the quasi-Newton method has determined a search direction at a minor iteration, $r$ controls the accuracy with which the objective function is minimized in that direction. The default value $r = 0.1$ requests a moderately accurate search, which should be satisfactory in most cases.

If the nonlinear functions are cheap to evaluate, a more accurate search may be appropriate; try $r = 0.01$ or $r = 0.001$. The number of iterations should decrease, and this will reduce total run time if there are many constraints.

If the nonlinear functions are expensive to evaluate, a less accurate search may be appropriate; try $r = 0.5$ or perhaps $r = 0.9$. The number of iterations will probably increase, but the total number of function evaluations may decrease enough to compensate.

**`LU_factor_tolerance=`** $r_1$ **(default 100.0)**
**`LU_update_tolerance=`** $r_2$ **(default 10.0)**

Same as for linear programming.

**`LU_singularity_tolerance=`** $r_1$ **(default** $\varepsilon^{2/3} \approx 10^{-11}$**)**
**`LU_swap_tolerance=`** $r_2$ **(default** $\varepsilon^{1/4} \approx 10^{-4}$**)**

When the problem is nonlinear, these directives influence tests for singularity after refactorization of the basis matrix. The `LU_singularity_tolerance` works the same as for linear programming. The `LU_swap_tolerance` is employed by a test to determine when a basic variable should be swapped with a superbasic variable to discourage singularity. A smaller value of $r_2$ tends to discourage this swapping.

**`Major_damping_parameter=`** $r$ **(default 2.0)**

This directive may assist convergence on problems that have highly nonlinear constraints. It is intended to prevent large relative changes between solutions (both the primal variables and the dual multipliers in the Lagrangian term) at successive subproblems; for example, the default value 2.0

prevents the relative change from exceeding 200 percent. This is accomplished by picking the actual new solution to lie somewhere on the line connecting the previous solution to the proposed new solution determined by the algorithm.

This is a very crude control. If the sequence of major iterations does not appear to be converging, first re-run the problem with a higher `penalty_parameter` (say 2, 4 or 10). If the subproblem solutions continue to change violently, try reducing $r$ to a value such as 0.2 or 0.1.

**`Major_iterations=`** *i* **(default 50)**

MINOS stops after $i$ major iterations, even if an optimum has not been indicated.

**`Minor_damping_parameter=`** *r* **(default=2.0)**

This directive limits the change in the variables during the determination of the step length in a minor iteration. The default value should not affect progress on well behaved problems, but setting $r = 0.1$ or 0.01 may be helpful when rapidly varying functions are present. (In the case of functions that increase or decrease rapidly to infinite values, upper and lower bounds should also be supplied wherever possible to avoid evaluation of the function at points where overflow will occur.)

In cases where several local optima exist, specifying a small value for $r$ may help locate an optimum near the starting point.

**`Minor_iterations=`** *i* **(default 40)**

At a major iteration, MINOS allows any number of minor iterations for the purpose of finding a point that is feasible in the linearized constraints. Once such a feasible point has been found, at most $i$ additional minor iterations are carried out. A moderate value (say, $30 \le i \le 100$) prevents excessive effort being expended at early major iterations, but allows later subproblems to be solved to completion.

The first major iteration is special: it terminates as soon as the linear constraints and bounds are satisfied.

**`Multiple_price=`** *i* **(default 1)**

This directive specifies the number of nonbasic variables that are chosen to become superbasic at one time. Normally the default of $i = 1$ yields good results, but it may help to set $i > 1$ if the number of superbasic variables at the starting point is much less than the number expected at the optimum.

For linear programs, $i > 1$ will cause the reduced-gradient algorithm to be applied in place of the simplex algorithm, but generally with inferior results. This differs from the ‘‘multiple pricing’’ designed to accelerate the choice of an entering variable in other large-scale simplex implementations.

**`Optimality_tolerance=`** *r* **(default 1.0e-6)**

Same as for linear programming.

**`Partial_price=`** *i* **(default 1)**

Same as for linear programming, except for the default value.

**`Penalty_parameter=`** *r* **(default 1.0)**

When there are nonlinear constraints, $r$ determines the magnitude of the penalty term that is appended to the objective. For early runs on a problem with unknown characteristics, the default value should be acceptable. If the problem is highly nonlinear and the major iterations do not converge, a larger value such as 2 or 5 may help; but if $r$ is too large, the rate of convergence may be unnecessarily slow. On the other hand, if the functions are nearly linear or a good starting point is known, it will often be safe to specify $r = 0.0$, although in general a positive $r$ is necessary to ensure convergence.

**`Pivot_tolerance=`** *r* **(default** $\varepsilon^{2/3} \approx 10^{-11}$**)**

Same as for linear programming.

**Radius_of_convergence=** *r* **(default 0.01)**

When there are nonlinear constraints, MINOS employs a heuristic test to determine when the solution may have entered a region of fast convergence near the optimum. At this stage the penalty parameter is reduced and full completion is chosen (see directives `completion` and `penalty_parameter`). A smaller value of *r* delays entry into this stage.

**Row_tolerance=** *r* **(default 1.0e-6)**

This directive specifies how accurately the nonlinear constraints should be satisfied at an optimal solution, relative to the magnitudes of the variables. A larger value may be appropriate if the problem functions involve data known to be of low accuracy.

**Scale=yes**
**Scale=no**

**Scale_option=** *i* **(default 1)**
**Scale_tolerance=** *r* **(default 0.9)**

Mostly the same as for linear programming. The default of $i = 1$ causes only the linear variables and constraints to be scaled, and should generally be tried first. When $i = 2$ all variables are scaled; the result depends on the initial linear approximation, and should therefore be used only if a good starting point is provided, and the problem is not highly nonlinear.

**Subspace_tolerance=** *r* **(default 0.5)**

This directive controls the extent to which optimization is confined to the current subset of basic and superbasic variables, before one or more nonbasic variables are added to the superbasic set. A smaller value of *r* tends to prolong optimization over the current subset, possibly increasing the number of iterations but reducing the number of basis changes.

A large value such as $r = 0.9$ may sometimes lead to improved overall efficiency, if the number of superbasic variables has to increase substantially between the starting point and an optimal solution.

**Superbasics_limit=** *i* **(default 50 or hessian_dimension)**

The number of superbasic variables is limited to *i*. (See also the comments on the `hessian_dimension` directive.)

**Unbounded_objective_value=** $r_1$ **(default 1.0e+20)**
**Unbounded_step_size=** $r_2$ **(default 1.0e+10)**

A nonlinear program is reported as unbounded if, in the quasi-Newton algorithm carrying out a minor iteration, the function value exceeds $r_1$ in magnitude or the length of the step exceeds $r_2$. Unboundedness may occur even though there exists a finite local optimum of interest; it is best prevented by bounding the variables to keep them within a meaningful region.

## §5 Output from MINOS

When invoked by `solve`, MINOS normally returns just a few lines to your screen to summarize its performance. The directives described below let you choose more output, either to the screen or to a file. This may be useful for monitoring the progress of a long run, or for comparing in detail the effects of selected algorithmic directives.

Because MINOS is a Fortran program that adheres to certain Fortran conventions, the treatment of file specifications is somewhat different from that of most MS-DOS or UNIX programs. As indicated below, MINOS directives refer to files by file unit numbers, and an additional directive is needed to associate a file number with a file name of your choosing.

**Summary_file=** *f* **(default 0)**
**Summary_level=** $i_1$ **(default 0)**
**Summary_frequency=** $i_2$ **(default 100)**
*f = filename*

These directives control a summary listing of MINOS's progress. The default setting $f = 0$ suppresses the listing; any value $f$ = 1, 2, 3, 4 or 6 through 99 causes the listing to be written to a file named `fort.`*f* in the current directory, or to the named file if the directive $f =$ *filename* is present. Thus to send the summary listing to `fort.9` in the current directory, you would say

```
ampl: option minos_options 'summary_file=9';
```

or to send the listing to `steel3.sum`, you could use

```
ampl: option minos_options 'summary_file=9 9=steel3.sum';
```

A value of $f = 5$ is not allowed. A value of $f = 6$ causes the listing to be written to the ''standard output'', which in interactive operation is your screen unless you have redirected it with a command of the form `solve >`*filename*. For mathematical programs that take a long time to solve, `summary_file=6` is recommended to help you track the algorithm's progress.

For linear programs, MINOS writes a ''log line'' to the summary listing every $i_2$ iterations, as well as some statistics on the entire run at the end. For example:

```
ampl: model steelT3.mod; data steelT3.dat;
ampl: option minos_options 'summary_file=6 summary_frequency=5';
ampl: solve;
MINOS 5.4:
summary_file=6
summary_frequency=5

    Itn       dj  ninf      sinf       objective
      1 -9.5E-01     1 3.429E+00 -9.99999859E+01
 Itn  2 -- feasible solution.  Objective =   2.499999647E+02
      5 -5.1E+01     0 0.000E+00  1.14300000E+05
     10 -4.9E+01     0 0.000E+00  3.04350000E+05
     15 -4.0E+01     0 0.000E+00  4.05892857E+05
     20  1.4E+01     0 0.000E+00  5.11953143E+05
     25 -5.5E-01     0 0.000E+00  5.14521714E+05

 EXIT -- optimal solution found

 No. of iterations               25     Objective value   5.1452171425E+05
 No. of degenerate steps          1     Percentage                   4.00
 Norm of x                 5.153E+03     Norm of pi               2.391E+02
 Norm of x  (unscaled)     8.159E+04     Norm of pi (unscaled)    3.375E+03
```

The items in the log line are the iteration number, the reduced cost of the variable chosen to enter the basis, the number and sum of infeasibilities in the current solution, and the value of the objective.

When there are nonlinearities in the objective but linear constraints, the reduced cost in the log lines is replaced by the more general reduced gradient, and two items are added at the end: the cumulative number of evaluations of the objective function, and the number of superbasic variables.

If there are nonlinearities in the constraints, MINOS writes a log line at each major iteration, and if $i_1 > 0$ also a log line after every $i_2$ minor iterations.

**Print_file=** *f* **(default 0)**
**Print_level=** $i_1$ **(default 0)**
**Print_frequency=** $i_2$ **(default 100)**
$f =$ *filename*

These directives are essentially the same as the preceding ones, except that they produce more extensive listings. Details are provided in the MINOS *User's Guide.*

**timing=** *i* **(default 0)**

When this directive is changed to 1 from its default value of 0, a summary of processing times is displayed:

```
MINOS times:
read:       0.15
solve:      2.77       excluding minos setup: 2.42
write:      0.00
total:      2.92
```

`read` is the time that MINOS takes to read the problem from a file that has been written by AMPL. `solve` is the time that MINOS spends setting up and solving the problem; the time for solving alone is also given. `write` is the time that MINOS takes to write the solution to a file for AMPL to read.

## §6 Restarting from an initial guess

MINOS can often make good use of an initial guess at a solution to a mathematical program. The optimal solution to one linear program may provide a particularly good guess for a similar one to be solved next. For nonlinear programs, a good initial guess may be crucial to the success of the algorithm, and different guesses may lead to different locally optimal solutions being identified.

There are two aspects to the initial guess for MINOS: the initial basis, and the initial solution.

### The initial basis

Several directives tell MINOS where to read or write a record of the currently basic and superbasic variables, and of the bounds at which the nonbasic variables are fixed. These directives make use of Fortran unit numbers like those described for the summary and print files above.

**`New_basis_file=`** $f_1$ **(default 0)**
**`Backup_basis_file=`** $f_2$ **(default 0)**
**`Save_frequency=`** $i$ **(default 100)**

$f_1$**=** *filename*
$f_2$**=** *filename*

For $f_1$ = 1, 2, 3, 4 or 7 through 99, a record of the basis is written every $i$ iterations and at the completion of the algorithm, either to a file named `fort.`$f_1$ in the current directory, or to the named file if the directive $f_1$= *filename* is present. In particular, upon successful completion the file contains a record of the optimal basis. (Values $f_1$ = 5 and $f_1$ = 6 should not be used.)

The value of $f_2$, which should be different from $f_1$, is interpreted in the same way. If both $f_1$ and $f_2$ are specified as different from zero, then two records of the basis are written; this is to protect against a crash or other failure that may corrupt one of them.

**`Old_basis_file=`** $f$ **(default 0)**
$f$ = *filename*

If $f$ has its default value of 0, MINOS determines an initial basis by use of a heuristic algorithm (see the `crash_option` directive in §3 above). Otherwise, an initial basis is read from a file. For $f$ = 1, 2, 3, 4 or 7 through 99, a previously written record of the basis is read from either the file named `fort.`$f$ in the current directory, or the named file if the directive $f$ = *filename* is present. The numbers of variables and constraints recorded in the file must match the numbers of variables and constraints in AMPL's currently active problem. (Values $f_1$ = 5 and $f_1$ = 6 should not be used.)

This feature can be useful for quickly solving a series of linear programs that differ only in the data. Before the first linear program is solved, `new_basis_file` is set to a positive integer so that a record of the optimal basis will be saved:

```
ampl: model steelT3.mod; data steelT3.dat;
ampl: option minos_options 'new_basis_file=77';
ampl: solve;
MINOS 5.4:
new_basis_file=77
MINOS 5.4: optimal solution found.
24 iterations, objective 514521.7143
```

Then a data value is changed, and `old_basis_file` is set to the same integer so that the previously optimal basis is used as a start:

```
ampl: let avail[1] := 32;
ampl: option minos_options 'old_basis_file=77';
ampl: solve;
MINOS 5.4:
old_basis_file=77
MINOS 5.4: optimal solution found.
1 iterations, objective 493648.7143
```

Only 1 iteration is necessary to step to an adjacent basis that is optimal for the modified linear program. This approach works as long as the change to the data does not affect the numbers of variables and constraints. (These numbers can be reduced by simplifications carried out in AMPL's presolve phase, even when it would seem that a change in the data should have no effect. If you receive an unexpected MINOS message that ''input basis had wrong dimensions,'' use the AMPL command `option show_stats 1` to see what presolve has done, or use `option presolve 0` to turn presolve off.)

Another use for this feature is to restart MINOS, possibly with some directives changed, after it has reached an iteration limit short of optimality. See the discussions of `iterations_limit`, `minor_iterations` and `major_iterations` in previous sections.

### *The initial solution*

AMPL passes to MINOS an initial value for each variable. As explained in the AMPL book, an initial value may be specified as part of a variable's declaration in the model (Section 8.1), or as part of the model's data (Section 9.3); the `let` command can also change the values of variables (Section 10.8). If a variable is not assigned an initial value in any of these ways, then it has an initial value of zero.

If a variable has an initial value that is outside its bounds, MINOS moves the initial value to the nearest bound. Then MINOS fixes all initially nonbasic and superbasic variables at their initial values, and solves for the basic variables. As a result, MINOS's initial values for the basic variables are generally different from the values supplied by AMPL.

AMPL also passes to MINOS an initial dual value associated with each constraint. Initial dual values are specified much like the initial (primal) values for variables, except that they are given in a constraint's declaration or by use of a constraint's name in a data statement or `let` command. MINOS uses the initial dual values for nonlinear constraints to initialize the corresponding dual multipliers in the Lagrangian term of the objective. Other initial dual values are ignored.

### *Acknowledgement*

Much of this material is based on text by Philip E. Gill, Walter Murray, Bruce A. Murtagh and Michael A. Saunders.