SYSTEMS OPTIMIZATION LABORATORY
DEPARTMENT OF MANAGEMENT SCIENCE AND ENGINEERING
STANFORD UNIVERSITY
STANFORD, CALIFORNIA, USA

MINOS 5.51 USER'S GUIDE

by

Bruce A. Murtagh† and Michael A. Saunders‡

TECHNICAL REPORT SOL 83-20R

December 1983
Revised September 23, 2003

Copyright © 1983–2002 Stanford University

†Graduate School of Management, Macquarie University, Sydney, NSW, Australia (bruce.murtagh@gsm.mq.edu.au).

‡Dept of Management Science and Engineering, Terman Building, Stanford University, Stanford, CA 94305-4026, USA (saunders@stanford.edu).

Research and reproduction of this report were supported by the Department of Energy contract DE-AM03-76SF00326, PA No. DE-AT03-76ER72018; National Science Foundation grants MCS-7926009, ECS-8012974 and CCR-9988205; the Office of Naval Research contracts N00014-75-C-0267 and N00014-02-1-0076; and the Army Research Office contract DAAG29-81-K-0156.

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the above sponsors.

Reproduction in whole or in part is permitted for any purposes of the United States Government.

Contents

Preface to MINOS 5.51 **vii**

Preface to MINOS 5.0 **xv**

1 Introduction **3**
1.1 Linear Programming 4
1.2 Problems with a Nonlinear Objective 5
1.3 Problems with Nonlinear Constraints 7
1.4 Problem Formulation 9
1.5 Restrictions 10
1.6 Storage 10
1.7 Files 11
1.8 Input Data Flow 11
1.9 Multiple SPECS Files 12
1.10 Internal Modifications 13

2 User-written Subroutines **15**
2.1 Subroutine funobj 15
2.2 Subroutine funcon 17
2.3 Constant Jacobian Elements 19
2.4 Subroutine matmod 19
2.5 Warm and Hot Starts 22
2.6 Subroutine matcol 23

3 The SPECS File **25**
3.1 SPECS File Format 26
3.2 Options for the MPS File 27
3.3 Options for Linear Programmming 30
3.4 Options for All Problems 31
3.5 Options for Nonlinear Objectives 35
3.6 Options for All Nonlinear problems 36
3.7 Options for Nonlinear Constraints 41
3.8 Options for Input and Output 45

4 MPS Files **49**
4.1 MPS File Headers 49
4.2 The NAME Header 50
4.3 The ROWS Section 50
4.4 The COLUMNS Section 51
4.5 The RHS Section 52

4.6 The RANGES Section (Optional) 53
4.7 The BOUNDS Section (Optional) 54
4.8 Comments 56
4.9 Restrictions and Extensions in MPS Format 56

5 Basis Files **57**
5.1 NEW and OLD BASIS Files 57
5.2 PUNCH and INSERT Files 59
5.3 DUMP and LOAD Files 61
5.4 Restarting Modified Problems 62

6 Output **65**
6.1 The PRINT file 65
6.1.1 The major iteration log 65
6.1.2 The minor iteration log 67
6.1.3 Crash statistics 70
6.1.4 Basis factorization statistics 70
6.1.5 EXIT conditions 72
6.1.6 Solution output 76
6.2 The SOLUTION file 79
6.3 The SUMMARY file 79

7 Subroutine minoss **81**
7.1 Subroutine minoss 81
7.2 Subroutine mispec 86
7.3 Subroutines miopt, miopti, mioptr 86
7.4 Example Use of minoss 88
7.5 MINOS(IIS): Debugging Infeasible Models 97

8 Library Subroutines **99**
8.1 Subroutine mititle 99
8.2 Subroutine mistart 99
8.3 Subroutine micore 100
8.4 Subroutine minos 101
8.5 Subroutine micmps 101
8.6 Subroutine micjac 102
8.7 Subroutine mirmps 103
8.8 Subroutine miwmps 105

9 Examples **107**
9.1 Linear Programming 108
9.2 Unconstrained Optimization 110
9.3 Linearly Constrained Optimization 111
9.4 Nonlinearly Constrained Optimization 115
9.5 Use of Subroutine `MATMOD` 130
9.6 Things to Remember 132

A System Information **137**
A.1 Distribution Files 137
A.2 Source Files 138
A.3 COMMON Blocks 139
A.4 Subroutine Structure 139
A.5 Matrix Data Structure 140

Preface to MINOS 5.51

This manual is a revision of the 1983 MINOS 5.0 User's Guide. The main changes implemented in MINOS 5.51 are summarized here.

1. MINOS is now callable as a subroutine (see Chapter 7). The stand-alone form of MINOS reads constraint data from an MPS file, whereas subroutine `minoss` has the same information passed to it as parameters. In these notes the term MINOS usually refers to both cases, but occasionally we need to distinguish between them.

2. Upper and lower case may be used in the SPECS file. Numerical values may contain up to 16 characters. For example,

```
Iterations limit            2000
Lower bound          -1.23456E+07
```

3. The default values of some options have changed as follows:

```
Print level                    0
Print frequency              100   (alias Log frequency)
Summary frequency            100
Hessian dimension             50
Superbasics limit             50
Crash option                   3   (new default and new meaning)
Scale option                   2   for LP,   1   for NLP
Factorize frequency          100   for LP,   50  for NLP
LU Factor tolerance        100.0   for LP,   5.0 for NLP
LU Update tolerance         10.0   for LP,   5.0 for NLP
Partial price                 10   for LP,   1   for NLP
Check frequency               60
Penalty parameter            1.0   is equivalent to old default
```

4. `Derivative level 0` requests a function-only search, even if `funobj` and `funcon` compute all gradients. The linesearch calls these routines with `mode` $= 0$, not `mode` $= 2$. An extra call with `mode` $= 2$ is needed after the search, but the net cost may be less if gradients are very expensive (e.g., if the user is estimating them by differences).

5. `funobj` and `funcon` may now return `mode` $= -1$ to mean "My nonlinear function is undefined here". During normal iterations, this signals the linesearch to try again with a shorter steplength.

   Previously, if `funobj` or `funcon` returned `mode` < 0, it meant "Please terminate". To request termination now, set `mode` ≤ -2.

6. `Crash option` 2 and 3 have been altered. The Crash procedure chooses a triangular basis from various rows and columns of $(A \;\; I)$. In some cases it is called more than once as follows:

`Crash option 0`	chooses the all-slack basis $B = I$.
`Crash option 1`	calls Crash once, looking at all rows and columns.
`Crash option 2`	calls Crash twice, looking at linear rows first. Nonlinear rows are treated at the start of Major 2.
`Crash option 3`	(default) calls Crash three times, looking at linear *equality* rows first, then linear inequalities, then nonlinear rows (if any) at the start of Major 2.

7. For problems with many degrees of freedom (lots of superbasic variables), experience suggests the following. Up to a certain point, it is best to provide a full triangular matrix R for the "reduced Hessian approximation" used by the quasi-Newton algorithm. For example,

```
Hessian dimension    1000
Superbasics limit    1000
```

would be suitable for most practical models. However, if the number of superbasic variables does reach 1000, considerable computation is needed to update the 500,000 elements of the dense matrix R.

For more extreme cases it may be better to work with a smaller matrix R:

```
Hessian dimension     100 or 200
Superbasics limit    5000
```

(e.g., for optimization with many variables and few constraints). The number of iterations and function calls will increase substantially. The functions and gradients should therefore be cheap to evaluate.

For general problems with many degrees of freedom, consider LANCELOT. For large problems with bound constraints only, consider LBFGS-B or LANCELOT. Both systems are available via NEOS: `http://www.mcs.anl.gov/home/otc/`

8. `Jacobian = Dense` or `Sparse` is still needed with MPS files, but need not be specified when subroutine `minoss` is used.

9. The `Minor iterations` limit now applies to the *feasible* iterations in each major iteration. Any number of (infeasible) minor iterations are allowed while MINOS iterates towards a "feasible subproblem".

The first major iteration is special—it stops as soon as the original linear constraints are satisfied.

For later major iterations, if the log says `50T` and the `Minor iterations` limit is 40, we know that 10 minor iterations were needed to satisfy the linearized constraints of the subproblem, and a further 40 were spent optimizing the subproblem before it was terminated by the `Minor iterations` limit.

10. `Penalty parameter` 1.0 is now the default, and it is relative to the old default of $100/m_1$, where m_1 is the number of nonlinear constraints. `Penalty parameter` 2.0 means twice the default value. This makes it easier to experiment with.

11. It is possible to turn off all output to the PRINT and SUMMARY files. The Print and Summary options are as follows:

`Print file`	0	No output to PRINT file.
	> 0	Output to specified file.
`Print level`	0	One line per major iteration.
	> 0	Full output as before.
`Print frequency`	0	No minor iteration log.
	i	A minor iteration line every i itns.
`Summary file`	0	No output to SUMMARY file.
	> 0	Output to specified file.
`Summary level`	0	One line per major iteration.
	> 0	More output.
`Summary frequency`	0	No minor iteration log.
	i	A minor iteration line every i itns.

12. Cold, Warm and Hot starts may be used when solving a sequence of problems of the *same size*.

    For stand-alone MINOS, the sequence of problems is defined via the `Cycle` parameters and the user routine `matmod`, which may access the common block

    ```
    logical            gotbas,gotfac,gothes,gotscl
    common    /cycle1/ gotbas,gotfac,gothes,gotscl
    ```

    to say whether or not the existing basis, basic factorization, reduced Hessian, and/or scales should be used to initialize the next solve. If `gotbas` = `.false.`, Crash will be used to choose a starting basis. Otherwise, a basis is assumed to be specified by the array `hs(*)`, and some or all of the other three quantities may be preserved.

    For subroutine `minoss`, these logicals are set if the first parameter `start` is `'Hot xxx'`, where `xxx` is any of the letters `FHS`. See Appendix B.

13. Following the `EXIT` message, some information is output to the PRINT file and the SUMMARY file. Lines of the form

    ```
    Primal inf (scaled)    444    4.6E-07  Dual inf   (scaled)    268    5.2E-06
    Primal infeas          412    2.6E-06  Dual infeas            502    9.3E-07
    Nonlinear constraint violn    2.5E-14
    ```

    show the maximum primal and dual infeasibilities before and after scaling, and the associated variable number. (Variable j is a column x_j for $1 \le j \le n$ and slack s_{j-n} for $n{+}1 \le j \le n{+}m$.)

    Note that "Primal infeasibility" is the amount by which x and s lie outside their bounds. In this example, variable `444` lies furthest outside its bounds *before the solution is unscaled.* More importantly, variable `412` is the most infeasible in the final solution—it lies outside its bounds by `2.6e-6`. If this seems too large, the `Feasibility tolerance` would need to be reduced below the maximum *scaled* infeasibility `4.6e-7` (or the unscaled value `2.6e-6` if scaling was not used).

    Similarly, variable `502` is the one whose reduced gradient has the "wrong sign" by the largest amount. If this seems too large, the `Optimality tolerance` would need to be reduced below `5.2E-06`$*$`norm(pi)`, where the required norm of π is printed three or one lines above (depending on whether scaling was used).

    Where relevant, the `Nonlinear constraint violn` line gives the maximum amount by which any nonlinear constraint value lies outside its bounds in the final unscaled solution.

14. The printed solution and SOLUTION file treat 0.0, 1.0, -1.0 specially. In particular, a dot (`.`) means 0.0, not "Same as the line above"!

15. In the Fortran source code, `integer*2` has been changed to `integer*4` throughout, to allow solution of arbitrarily large problems. This change is reversible. (The variable `nwordh` must be set appropriately in subroutine `m1init`.) If `integer*2` is used, the maximum number of rows is 16383.

16. In source file `mi10*.for`, subroutine `mifile` defines some "hard-wired" file numbers and opens most files by calling `m1open`. Some of the file numbers and `open` statements may need to be altered to suit your system.

17. The first two lines of Old Basis and New Basis files accommodate larger problems than in MINOS 5.1.

New SPECS file keywords

All of the following keywords are new except the first. Crash options 2 and 3 now have a different effect and option 4 is not defined.

`Crash option` i Default = 3

Except on restarts, a Crash procedure is used to select an initial basis from certain rows and columns of the constraint matrix $(A \;\; I)$. The `Crash option` i determines which rows and columns of A are eligible initially, and how many times Crash is called. Columns of I are used to pad the basis where necessary.

$i = 0$ The initial basis contains only slack variables: $B = I$.

1 Crash is called once, looking for a triangular basis in all rows and columns of A.

2 Crash is called twice (if there are nonlinear constraints). The first call looks for a triangular basis in linear rows, and the first major iteration proceeds with simplex iterations until the linear constraints are satisfied. The Jacobian is then evaluated for the second major iteration and Crash is called again to find a triangular basis in the nonlinear rows (retaining the current basis for linear rows).

3 Crash is called up to three times (if there are nonlinear constraints). The first two calls treat *linear equalities* and *linear inequalities* separately. As before, the last call treats nonlinear rows at the start of the second major iteration.

If $i \geq 1$, certain slacks on inequality rows are selected for the basis first. (If $i \geq 2$, numerical values are used to exclude slacks that are close to a bound.) Crash then makes several passes through the columns of A, searching for a basis matrix that is essentially triangular. A column is assigned to "pivot" on a particular row if the column contains a suitably large element in a row that has not yet been assigned. (The pivot elements ultimately form the diagonals of the triangular basis.) For remaining unassigned rows, slack variables are inserted to complete the basis.

`Defaults`

When `minoss` is in use, `call miopt('Defaults')` causes all MINOS options to be set to their default values.

`Expand frequency` i Default = 10000

This option is part of an anti-cycling procedure designed to guarantee progress even on highly degenerate problems [GMSW89].

For linear models, the strategy is to force a positive step at every iteration, at the expense of violating the bounds on the variables by a small amount. Suppose that the `Feasibility tolerance` is δ. Over a period of i iterations, the tolerance actually used by MINOS increases from 0.5δ to δ (in steps of $0.5\delta/i$).

For nonlinear models, the same procedure is used for iterations in which there is only one superbasic variable. (Cycling can occur only when the current solution is at a vertex of the feasible region.) Thus, zero steps are allowed if there is more than one superbasic variable, but otherwise positive steps are enforced.

Increasing i helps reduce the number of slightly infeasible nonbasic basic variables (most of which are eliminated during a resetting procedure). However, it also diminishes the freedom to choose a large pivot element (see `Pivot tolerance`).

`LU density tolerance`	r_1	Default = 0.6
`LU singularity tolerance`	r_2	Default = $\epsilon^{2/3} \approx 10^{-11}$

The density tolerance r_1 is used during LU factorization of the basis matrix. Columns of L and rows of U are formed one at a time, and the remaining rows and columns of the basis are altered appropriately. At any stage, if the density of the remaining matrix exceeds r_1, the Markowitz strategy for choosing pivots is altered to reduce the time spent searching for each remaining pivot. Raising the density tolerance towards 1.0 may give slightly sparser LU factors, with a slight increase in factorization time.

The singularity tolerance r_2 helps guard against ill-conditioned basis matrices. When the basis is refactorized, the diagonal elements of U are tested as follows: if $|U_{jj}| \le r_2$ or $|U_{jj}| < r_2 \max_i |U_{ij}|$, the j-th column of the basis is replaced by the corresponding slack variable. (This is most likely to occur after a restart, or at the start of a major iteration.)

In some cases, the Jacobian matrix may converge to values that make the basis exactly singular. (For example, a whole row of the Jacobian could be zero at an optimal solution.) Before exact singularity occurs, the basis could become very ill-conditioned and the optimization could progress very slowly (if at all). Setting $r_2 =$ `1.0e-5`, say, may help cause a judicious change of basis.

`Minor damping parameter`	r	Default = 2.0

This parameter limits the change in x during a linesearch. It applies to all nonlinear problems, once a "feasible solution" or "feasible subproblem" has been found.

1. A linesearch of the form minimize$_\alpha$ $F(x + \alpha p)$ is performed over the range $0 < \alpha \le \beta$, where β is the step to the nearest upper or lower bound on x. Normally, the first steplength tried is $\alpha_1 = \min(1, \beta)$.

2. In some cases, such as $F(x) = ae^{bx}$ or $F(x) = ax^b$, even a moderate change in the components of x can lead to floating-point overflow. The parameter r is therefore used to define a limit $\bar\beta = r(1 + \|x\|)/\|p\|$, and the first evaluation of $F(x)$ is at the potentially smaller steplength $\alpha_1 = \min(1, \bar\beta, \beta)$.

3. Wherever possible, upper and lower bounds on x should be used to prevent evaluation of nonlinear functions at meaningless points. The `Minor damping parameter` provides an additional safeguard. The default value $r = 2.0$ 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. A "good" starting point may be required. An important application is to the class of nonlinear least-squares problems.

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

`Timing level` i Default = 2

$i = 0$ suppresses timing.
$i = 1$ times input, solve and output.
$i = 2$ times input, solve, output, `funcon` and `funobj`.

The values $i = -1$ and -2 are the same as 1 and 2, except the times are not printed at the end. If you are calling subroutine `minoss`, you may print the times in your own format by accessing the following common block:

```
parameter       ( ntime = 5 )
common    /m1tim / tlast(ntime), tsum(ntime), numt(ntime), ltime
```

where

`numt(`k`)`	is the number of times clock k has been turned on.
`tlast(`k`)`	is the time at which clock k was last turned on.
`tsum(`k`)`	is the total time elapsed while clock k was on.
`ltime`	is the `Timing level` i.

For $k = 1$ to 5, clock k times input, solve, output, `funcon` and `funobj` respectively. See subroutines `m1time` and `m1timp` for further details. For `Timing level 2`, MINOS and `minoss` both call `m1time` at the end of a run. This prints the "total time" statistics using a loop of the form

```
do k = 1, ntime
   call m1timp( k, 'Time', tsum(k) )
end do
```

Algorithmic Changes

1. The linesearch takes shorter steps if `funobj` or `funcon` return `mode` $= -1$ (mentioned above).

2. "Basis repair" is sometimes invoked at the start of a major iteration, or following a linesearch failure. A stable, sparse LU factorization of the combined basic/superbasic matrix $\begin{pmatrix} B & S \end{pmatrix}$ is computed by LUSOL in the form

$$P \begin{pmatrix} B^T \\ S^T \end{pmatrix} Q = LU,$$

   where P and Q are permutations and L is well-conditioned. Then P provides a reordering of the columns of $\begin{pmatrix} B & S \end{pmatrix}$ that makes the condition of the new B close to optimal.

   In the major iteration log, `BSswp` gives the number of variables that were swapped between B and S. (Zero means that the current basis was retained. The current reduced Hessian matrix R is then also retained, to help solve the subproblem more quickly.)

3. The triangular reduced-Hessian matrix R is now stored row-wise instead of column-wise in an array `r(*)`, because most updating operations traverse the rows of R. This reduces paging on a machine with virtual memory and improves the use of cache memory when there are many superbasics and `Hessian dimension` is large.

4. Nonlinear objective and constraint functions are not evaluated until the linear constraints have been satisfied (to within the `Feasibility tolerance`). Previously, any nonlinear constraints were evaluated at the starting point regardless of feasibility.

5. Gradient checking now takes place after the linear constraints have been satisfied. Previously, it occurred at the starting point.

MINOS is licensed by Stanford University. Fortran 77 source code for all common machines (mainframes, workstations and PCs) is available from SBSI:

Stanford Business Software, Inc.	http://www.SBSI-SOL-Optimize.com

SBSI handles individual and site licenses, both non-profit and commercial. (SBSI is separate and independent from Stanford University.)

Special commercial licenses, such as those involving the re-sale of MINOS as part of a larger package, are negotiated by OTL, Stanford University:

Office of Technology Licensing	http:// www.stanford.edu/group/OTL/

For many applications involving linear and nonlinear models, we recommend the use of algebraic modeling languages. Two of the most widely used systems are GAMS and AMPL. They provide a convenient interface to MINOS and many other linear, integer and nonlinear optimization systems:

http://www.gams.com	http://www.ampl.com

Bruce Murtagh
Macquarie University

Michael Saunders
Stanford University

September 23, 2003

Preface to MINOS 5.0

Since the middle of 1980, approximately 150 academic and research institutions around the world have installed MINOS/AUGMENTED, the predecessor of the present system. About 30 further installations exist in private industry. With enquiries continuing to arrive almost daily, the need for a combined linear and nonlinear programming system is apparent in both environments. To date, many users have been able to develop substantial nonlinear models and have come to be fairly confident that the `Optimal Solution` message actually means what it says. Certainly, other less joyful exit messages will often have greeted eager eyes. These serve to emphasize that model building remains an art, and that nonlinear programs can be *arbitrarily difficult to solve*. Nevertheless, the success rate has been high, and the positive response from users with diverse applications has inspired us to pursue further development.

MINOS 5.0 is the result of prolonged refinements to the same basic algorithms that were in MINOS/AUGMENTED:

- the simplex method (Dantzig, 1947, 1963),
- a quasi-Newton method (very many authors from Davidon, 1959, onward),
- the reduced-gradient method (Wolfe, 1962), and
- a projected Lagrangian method (Robinson, 1972; Rosen and Kreuser, 1972).

From numerous potential options, it has been possible to develop these particular algorithms into a relatively harmonious whole. The resulting system permits the solution of both small and large problems in the four main areas of smooth optimization:

- linear programming,
- unconstrained optimization,
- linearly constrained optimization, and
- nonlinearly constrained optimization.

In rare cases, the quasi-Newton method may require excessive storage. We have chosen not to provide a nonlinear conjugate-gradient method, or a truncated linear conjugate-gradient method, for this situation. Instead, we retain the quasi-Newton method throughout, restricting it to certain subspaces where necessary. (The strategy for altering the subspaces remains experimental.)

We regret that other obvious algorithms (such as integer programming, piece-wise smooth optimization, the dual simplex method) are still not available. Nor are ranging procedures or parametric algorithms. Sensitivity analysis is still confined to the usual interpretation of Lagrange multipliers.

As before, MINOS 5.0 is a stand-alone system that is intended for use alongside commercial mathematical programming systems whenever such facilities are available. The systems should complement each other.

To users of MINOS/AUGMENTED, the most apparent extensions are a scaling option (for linear constraints and variables only), and the ability to estimate some or all gradients numerically, if they are not computed by the user. On a more mundane level, the names of the user subroutines for computing nonlinearities have been changed from `CALCFG` and `CALCON` to `FUNOBJ` and `FUNCON`, and two new parameters allow access to the workspace used by MINOS.

Internally, one of the major improvements has been the development of a new basis-handling package, which forms the foundation of LUSOL (Gill, *et al.*, 1984), a set of routines for computing and updating a sparse *LU* factorization. This package draws much from the work of Reid (1976, 1982). It replaces the P^4-based procedures in MINOS/AUGMENTED (Saunders, 1976) and is substantially more efficient on problems whose basis matrices are not close to triangular. As before, column updates are performed by the method of Bartels and Golub (1969, 1971), but the implementation is more efficient and there is no severe degradation arising from large numbers of "spikes". We venture to say that LUSOL is the first *truly stable basis package* that has been implemented for production use.

A further vital improvement has been the development of two new linesearch procedures (Gill, *et al.*, 1979) for finding a step length with and without the aid of derivatives. In particular they cater for function values that are somewhat "noisy"—a common practical circumstance.

From a software engineering viewpoint, the source code has been restructured to ease the problems of maintenance and future development. MINOS still stands for *Modular In-core Nonlinear Optimization System*, and we have done our best to respect the implications of the "M". Nevertheless, MINOS 5.0 remains a parameter-driven system. It is a speeding train on a railroad that has parallel tracks and many switches but few closed circuits. Its various modules cannot be called upon in an arbitrary order. In fact, there are 80 parameters that can be set if necessary—these are the switching points along the railroad. Fortunately, only a handful need be set for any particular application. In most cases, the default values are appropriate for large and small problems alike.

For interactive users, a new feature is the SUMMARY file, which provides at the terminal a brief commentary on the progress of a run. Unfortunately, a two-way conversation is not possible. The only *input* engendered by this feature is an occasional dive for the Break key to abort an errant run. While rarely called upon, such a facility can be crucial to the security of one's computer funds.

Throughout the development of MINOS, we have received a great deal of assistance from many kind people. Most especially, our thanks go to Philip Gill, Walter Murray and Margaret Wright, whose knowledge and advice have made much of this work possible. They are largely responsible for the linesearch procedures noted above (which are as vital to nonlinear optimization as basis factors are to linear programming), and they are authorities on all of the algorithms employed within MINOS. Their patience has been called upon continually as other important work at SOL either languished or fell unfairly on their shoulders.

Further to basis factors, we acknowledge the pioneering work of John Reid in implementing the Markowitz-based LU factorization and the Bartels-Golub update. The LUSOL procedures in MINOS 5.0 owe much to the ingenuity embodied in his LA05 package.

Users have naturally provided an essential guiding influence. In some cases they are algorithm developers themselves. At home, we have had constant encouragement from George Dantzig and the benefit of his modeling activity within SOL, notably on the energy-economic model PILOT. We thank him warmly for bringing the Systems Optimization Laboratory into existence. We also thank Patrick McAllister, John Stone and Wesley Winkler for the feedback they have provided by running various versions of MINOS during their work on PILOT. (We note that PILOThas grown to 1500 constraints and 4000 variables, and now has a quadratic objective. From our perspective, it is a nontrivial test problem!) Likewise, Alan Manne has provided encouragement and assistance from the beginning. Two of his nonlinear economic models have been invaluable as test problems (and are included on the MINOS distribution tape). We also thank him and Paul Preckel for the development of procedures for solving sequences of related prob-

lems (Preckel, 1980). The main ingredients of these procedures are now an integral part of MINOS.

From industry, we have received immense benefit from the working relationship between SOL and Robert Burchett of the General Electric Company (Electric Utility Systems Engineering Department) in Schenectady, New York. Many algorithmic and user-oriented details have resulted from his experience and from his interest in the fine points of optimization. Three years ago we did not envisage that problems involving thousands of nonlinear constraints would soon be solved successfully. Rob constantly pushed test versions of MINOS to their limits, and inspired the development of techniques to extend those limits. We thank him for his tireless contributions.

We are also grateful to Zenon Fortuna, Steven Gorelick, Marc Hellman, Thomas McCormick, Larry Nazareth, Scott Rogers, John Rowse and John Tomlin for their helpful suggestions and/or assistance in tracking down bugs. Finally, we thank the staff of the Office of Technology Licensing and the Information Technology Services at Stanford University for undertaking the task of distributing MINOS.

Most of the software development was carried out at the Stanford Linear Accelerator Center with the aid of the Wylbur text editor and the University of Waterloo's WATFIV compiler. This User's Guide was typeset using aTEX*, with editorial assistance from Philip Gill and Margaret Wright.

Bruce Murtagh
University of New South Wales

Michael Saunders
Stanford University

December, 1983

*D. E. Knuth, *TEX and METAFONT, New Directions in Typesetting*, American Mathematical Society and Digital Press, Bedford, Massachusetts (1979).

Chapter 1

September 23, 2003

Introduction

MINOS is a Fortran-based linear and nonlinear programming system, designed to solve large-scale constrained optimization problems of the following form:

$$\underset{x,\,y}{\text{minimize}} \quad F(x) + c^T x + d^T y \tag{1}$$

$$\text{subject to} \quad b_1 \le f(x) + A_1 y \le b_2, \tag{2}$$

$$b_3 \le A_2 x + A_3 y \le b_4, \tag{3}$$

$$l \le (x,\ y) \le u, \tag{4}$$

where the vectors b_i, c, d, l, u and the matrices A_i are constant, $F(x)$ is a nonlinear function of some of the variables, and $f(x)$ is a vector of nonlinear functions. The nonlinearities (if present) may be of a general nature but must be smooth and preferably "almost linear", in the sense that they should not change radically with small changes in the variables. We make the following definitions:

x	the nonlinear variables
y	the linear variables
$(x,\ y)$	the vector $\binom{x}{y}$
(1)	the objective function
(2)	the nonlinear constraints
(3)	the linear constraints
(4)	the bounds on the variables
m	the total number of general constraints in (2) and (3)
n	the total number of variables in x and y
m_1	the number of nonlinear constraints (the dimension of $f(x)$)
n_1	the number of nonlinear variables (the dimension of x)
n_1'	the number of nonlinear objective variables (in $F(x)$)
n_1''	the number of nonlinear Jacobian variables (in $f(x)$)
MINOS	a stand-alone system with its own main program
`minoss`	a callable subroutine.

A large-scale problem is one in which m and n are several hundred or several thousand. MINOS takes advantage of the fact that the constraint matrices A_i and the partial derivatives $\partial f_i(x)/\partial x_j$ are typically sparse (contain many zeros).

The stand-alone form of MINOS reads constraint data from an MPS file similar to that used by commercial mathematical programming systems, whereas subroutine `minoss` (pron. minos-ess) has the same information passed to it as parameters. Usually the term MINOS refers to both cases, but occasionally we need to distinguish between them.

The nonlinear functions $F(x)$ and $f(x)$ are defined by two subroutines, `funobj` and `funcon`. These are linked to MINOS before run-time. Preferably they should provide both the functions and their gradients, but if any gradients are missing, MINOS will estimate them by finite differences.

To bypass the MPS file, one may write a driving program and call `minoss`. For certain problems, both the MPS file and the function subroutines may be bypassed by using modeling languages such as GAMS [BKM88] or AMPL [FGK92]. (These languages allow a concise statement of arbitrarily large models and they are able to access a variety of solvers such as MINOS. In particular, they simplify nonlinear modeling by providing gradients to MINOS automatically.)

The dimensions n_1' and n_1'' allow for the fact that $F(x)$ and $f(x)$ may involve different sets of nonlinear variables "x". The two sets of variables *always overlap*, in the sense that the shorter "x" is always the same as the beginning of the other. This is relevant when it comes to coding subroutines `funobj` and `funcon` to compute $F(x)$ and $f(x)$. If x is the same in both cases, we have $n_1 = n_1' = n_1''$. Otherwise, we define the number of nonlinear variables to be $n_1 = \max(n_1', n_1'')$.

In the following sections we introduce more terminology and give an overview of the MINOS optimization algorithms and the main system features.

1.1 LINEAR PROGRAMMING

When $F(x)$ and $f(x)$ are absent, the problem becomes a *linear program*. Since there is no need to distinguish between linear and nonlinear variables, we use x rather than y. We also convert all general constraints into equalities with the aid of slack variables s, so that the only inequalities are simple bounds on the variables. Thus, we write linear programs in the form

$$\underset{x,\ s}{\text{minimize}} \quad c^Tx \quad \text{subject to} \quad Ax + s = b, \quad l \le (x,\ s) \le u. \tag{5}$$

Problems specified in MPS files and in modeling languages follow the original formulation (1)–(4), but MINOS uses the form (5) internally. Subroutine `minoss` works with (5) directly. When the constraints are linear, the bounds on the slacks are defined so that $b = 0$. When there are nonlinear constraints, some elements of b are nonzero.

In the mathematical programming world, x and s are sometimes called *structural* variables and *logical* variables. Their upper and lower bounds are fundamental to problem formulations and solution algorithms. Some of the components of l may be $-\infty$ and those of u may be $+\infty$. If $l_j = u_j$, a variable is said to be *fixed*, and if its bounds are $-\infty$ and $+\infty$, the variable is called *free*.

Within MINOS, a point (x, s) is said to be *feasible* if the following are true:

- The constraints $Ax + s = b$ are satisfied to within machine precision $\approx 10^{-15}$.
- The bounds $l \le (x,\ s) \le u$ are satisfied to within a *feasibility tolerance* $\delta_{\text{fea}} \approx 10^{-6}$.
- The nonlinear constraints (2) are satisfied to within a *row tolerance* $\delta_{\text{row}} \approx 10^{-6}$.

Tolerances such as δ_{fea} and δ_{row} may be specified by setting `Feasibility tolerance` and `Row tolerance` in the SPECS file (Chapter 3).

MINOS solves linear programs using a reliable implementation of the *primal simplex method* [Dan63], in which the constraints $Ax + s = b$ are partitioned into the form

$$Bx_B + Nx_N = b, \tag{6}$$

where the *basis matrix* B is a square and nonsingular submatrix of $(\,A \ \ I\,)$. The elements of x_B and x_N are called the basic and nonbasic variables respectively. Together, they are a permutation of the vector $(x,\ s)$. Certain *dual variables* π and *reduced costs* d_N are defined by the equations

$$B^T\pi = c_B, \qquad d_N = c_N - N^T\pi, \tag{7}$$

where $(c_B,\ c_N)$ is a permutation of the objective vector $(c,\ 0)$.

At a feasible point, nonbasic variables are typically equal to one of their bounds, and basic variables are somewhere between their bounds. To reach an optimal solution, the simplex method performs a sequence of *iterations* of the following general nature. With guidance from d_N, a nonbasic variable is chosen to move from its current value, and the basic variables are adjusted to satisfy the constraints in (5). Usually one of the basic variables reaches a bound. The basis partition is then redefined with a column of B being replaced by a column of N. When no such interchange can be found to reduce the value of c^Tx, the current solution is optimal.

The simplex method

For convenience, let x denote the variables $(x,\ s)$. The main steps in a simplex iteration are as follows:

Compute dual variables: Solve $B^T\pi = c_B$.

Price: Compute some or all of the reduced costs $d_N = c_N - N^T\pi$ to determine if a favorable nonbasic column a_q exists.

Compute search direction: Solve $Bp_B = \pm a_q$ to determine the basic components of a search direction p along which the objective is improved. (The nonbasic elements of p are $p_N = 0$, except for ± 1 for the element corresponding to a_q.)

Find maximum steplength: Find the largest steplength $\alpha_{\max}$ such that $x + \alpha_{\max} p$ continues to satisfy the bounds on the variables. The steplength may be determined by the new nonbasic variable reaching its opposite bound, but normally some basic variable will reach a bound first.

Update: Take the step $\alpha_{\max}$. If this was determined by a basic variable, interchange the corresponding column of B with column a_q from N.

When a starting basis is chosen and the basic variables x_B are first computed, if any components of x_B lie significantly outside their bounds, we say that the current point is *infeasible*. In this case, the simplex method uses a "Phase 1" procedure to reduce the sum of infeasibilities. This is similar to the subsequent "Phase 2" procedure just described.

The feasibility tolerance δ_{fea} is used to determine which Phase is in effect. A similar *optimality tolerance* δ_{opt} is used during pricing to judge whether any reduced costs are significantly large. (This tolerance is scaled by $\|\pi\|$, a measure of the size of the current π.)

If the solution procedures are interrupted, some of the nonbasic variables may lie strictly *between* their bounds: $l_j < x_j < u_j$. In addition, at a "feasible" or "optimal" solution, some of the basic variables may lie slightly outside their bounds: $l_j - \delta_{\text{fea}} \le x_j \le u_j + \delta_{\text{fea}}$. In rare cases, even a few nonbasic variables might lie outside their bounds by as much as δ_{fea}.

MINOS maintains a sparse LU factorization of the basis matrix B, using a Markowitz ordering scheme and Bartels-Golub updates, as implemented in the Fortran package LUSOL [GMSW87]; see [BG69, Bar71, Reid76, Reid82]. The basis factorization is central to the efficient handling of sparse linear and nonlinear constraints.

1.2 PROBLEMS WITH A NONLINEAR OBJECTIVE

When nonlinearities are confined to the term $F(x)$ in the objective function, the problem is a linearly constrained nonlinear program. MINOS solves such problems using a *reduced-gradient* method [Wolf62] combined with a *quasi-Newton* method [Dav59, DS83] that generally leads to superlinear convergence. The implementation follows that described in Murtagh and Saunders [MS78].

As a slight generalization of (6), the constraints $Ax + s = b$ are partitioned into the form

$$Bx_B + Sx_S + Nx_N = b, \tag{8}$$

where x_S is a set of *superbasic variables*. As before, the nonbasic variables are normally equal to one of their bounds, while the basic *and* superbasic variables lie somewhere between their bounds (to within δ_{fea}). Let the number of superbasic variables be n_S, the number of columns in S. At a solution, n_S will be no more than n_1, the number of nonlinear variables, and it is often much smaller than this. In many real-life cases we have found that n_S remains reasonably small, say 200 or less, regardless of the size of the problem. This is one reason why MINOS has proved to be a practical tool.

In the reduced-gradient method, x_S is regarded as a set of "independent variables" that are allowed to move in any desirable direction to reduce the objective function (or the sum of infeasibilities). The basic variables are then adjusted in order to continue satisfying the linear constraints. If it appears that no improvement can be made with the current definition of B, S and N, one of the nonbasic variables is selected to be added to S, and the process is repeated with an increased value of n_S. At all stages, if a basic or superbasic variable encounters one of its bounds, that variable is made nonbasic and the value of n_S is reduced by one.

For linear programs, we may interpret the simplex method as being the same as the reduced-gradient method, with the number of superbasic variables oscillating between 0 and 1. (In general, a step of the simplex method *or* the reduced-gradient method is called a *minor iteration.*)

A certain matrix Z is needed for descriptive purposes. It takes the form

$$Z = \begin{pmatrix} -B^{-1}S \\ I \\ 0 \end{pmatrix}, \tag{9}$$

though it is never computed explicitly. Given LU factors of the basis matrix B, it is possible to compute products of the form Zq and Z^Tg by solving linear equations involving B or B^T. This in turn allows optimization to be performed on the superbasic variables, while the basic variables are adjusted to satisfy the general linear constraints. (In the description below, the reduced-gradient vector satisfies $d_S = Z^Tg$, and the search direction satisfies $p = Zp_S$.)

An important part of MINOS is the quasi-Newton method used to optimize the superbasic variables. This can achieve superlinear convergence during any sequence of iterations for which the B, S, N partition remains constant. It requires a dense upper-triangular matrix R of dimension n_S, which is updated in various ways to approximate the *reduced Hessian*:

$$R^TR \approx Z^THZ, \tag{10}$$

where H is the *Hessian* of the objective function, i.e. the matrix of second derivatives of $F(x)$. As for unconstrained optimization, the storage required for R is sometimes a limiting factor.

The reduced-gradient method

Let g be the gradient of the nonlinear objective (1). The main steps in a reduced-gradient iteration are as follows:

Compute dual variables and reduced gradient: Solve $B^T\pi = g_B$ and compute the reduced-gradient vector $d_S = g_S - S^T\pi$.

Price: If $\|d_S\|$ is sufficiently small, compute some or all of the reduced costs $d_N = g_N - N^T\pi$ to determine if a favorable nonbasic column a_q exists. If so, move that column from N into S, expanding R accordingly.

Compute search direction: Solve $R^TRp_S = -d_S$ and $Bp_B = -Sp_S$ to determine the superbasic and basic components of a search direction p along which the objective is improved. (The nonbasic elements of p are $p_N = 0$.)

Find maximum steplength: Find the largest steplength $\alpha_{\max}$ such that $x + \alpha_{\max}p$ continues to satisfy the bounds on the variables.

Perform linesearch: Find an approximate solution to the one-dimensional problem

$$\underset{\alpha}{\text{minimize}} \quad F(x+\alpha p) \quad \text{subject to} \quad 0 \le \alpha \le \alpha_{\max}.$$

Update (quasi-Newton): Take the step α. Apply a quasi-Newton update to R to account for this step.

Update (basis change): If a superbasic variable reached a bound, move it from S into N. If a basic variable reached a bound, find a suitable superbasic variable to move from S into B, and move the basic variable into N. Update R if necessary.

At an optimum, the reduced gradient d_S should be zero. MINOS terminates when $\|d_S\| \leq \delta_{\text{opt}}\|\pi\|$ and the reduced costs (component of d_N) are all sufficiently positive or negative, as judged by the same quantity $\delta_{\text{opt}}\|\pi\|$.

In the linesearch, $F(x + \alpha p)$ really means the objective function (1) evaluated at the point $(x,\ y,\ s) + \alpha p$. This steplength procedure is another important part of MINOS. Two different procedures are used, depending on whether or not all gradients are known analytically; see [GMSW79, GMW81]. The number of nonlinear function evaluations required may be influenced by setting the `Linesearch tolerance` in the SPECS file.

Normally, the objective function $F(x)$ will never be evaluated at a point x unless that point satisfies the linear constraints and the bounds on the variables. An exception is during a finite-difference check on the calculation of gradient elements. This check is performed at the *starting point* x_0, which takes default values or may be specified. MINOS ensures that the bounds $l \leq x_0 \leq u$ are satisfied, but in general the starting point will not satisfy the general linear constraints. If $F(x_0)$ is undefined, the gradient check should be suppressed (`Verify level -1`), or the starting point should be redefined.

1.3 PROBLEMS WITH NONLINEAR CONSTRAINTS

If any of the constraints are nonlinear, MINOS employs a *projected Lagrangian* algorithm, based on a method due to Robinson [Rob72]; see Murtagh and Saunders [MS82]. This involves a sequence of *major iterations*, each of which requires the solution of a *linearly constrained subproblem*. Each subproblem contains linearized versions of the nonlinear constraints, as well as the original linear constraints and bounds.

At the start of the k-th major iteration, let $(x_k,\ y_k)$ be an estimate of the variables, and let λ_k be an estimate of the Lagrange multipliers (dual variables) associated with the nonlinear constraints. The constraints are linearized by changing $f(x)$ in Equation (2) to its linear approximation:

$$\bar{f}(x, x_k) = f(x_k) + J(x_k)(x - x_k),$$

or more briefly $\bar{f} = f_k + J_k(x - x_k)$, where $J(x_k)$ is the *Jacobian matrix* evaluated at x_k. (The i-th row of the Jacobian is the gradient vector for the i-th nonlinear constraint function.) The subproblem to be solved during the k-th major iteration is then

$$\begin{array}{llcccl} \underset{x,\,y}{\text{minimize}} & \multicolumn{5}{c}{F(x) + c^T x + d^T y - \lambda_k^T f_d + \tfrac{1}{2}\rho_k \|f_d\|^2} \quad (11) \\ \text{subject to} & b_1 & \leq & \bar{f} + A_1 y & \leq & b_2, \quad (12) \\ & b_3 & \leq & A_2 x + A_3 y & \leq & b_4, \quad (13) \\ & l & \leq & (x,\ y) & \leq & u, \quad (14) \end{array}$$

where $f_d = f - \bar{f}$ is the difference between $f(x)$ and its linearization. The objective function (11) is called an *augmented Lagrangian*. The scalar ρ_k is a *penalty parameter*, and the term involving ρ_k is a modified *quadratic penalty function*.

MINOS uses the reduced-gradient method to solve each subproblem. As before, slack variables are introduced and the vectors b_i are incorporated into the bounds on the slacks. The linearized constraints take the form

$$\begin{pmatrix} J_k & A_1 \\ A_2 & A_3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} s_1 \\ s_2 \end{pmatrix} = \begin{pmatrix} J_k x_k - f_k \\ 0 \end{pmatrix}.$$

We refer to this system as $Ax + s = b$ as in the linear case. The Jacobian J_k is treated as a sparse matrix, the same as the matrices A_1, A_2 and A_3. The quantities J_k, b, λ_k and ρ_k change each major iteration.

The projected Lagrangian method

For convenience, suppose that all variables and constraints are nonlinear. The main steps in a major iteration are as follows:

Solve subproblem: Find an approximate solution $(\bar{x}, \bar{\lambda})$ to the kth subproblem (11)–(14).

Compute search direction: Adjust the elements of $\bar{\lambda}$ if necessary (if they have the wrong sign). Define a search direction $(\Delta x,\ \Delta\lambda) = (\bar{x} - x_k,\ \bar{\lambda} - \lambda_k)$.

Find steplength: Choose a steplength σ such that some merit function $M(x, \lambda)$ has a suitable value at the point $(x_k + \sigma\Delta x,\ \lambda_k + \sigma\Delta\lambda)$.

Update: Take the step σ to obtain $(x_{k+1},\ \lambda_{k+1})$. In some cases, adjust ρ_k.

For the first major iteration, the nonlinear constraints are ignored and minor iterations are performed until the original linear constraints are satisfied.

The initial Lagrange multiplier estimate is typically $\lambda_k = 0$ (though it can be provided by the user). If a subproblem terminates early, some elements of the new estimate $\bar{\lambda}$ may be changed to zero.

The penalty parameter initially takes a certain default value $\rho_k = 100.0/m_1$, where m_1 is the number of nonlinear constraints. (A value r times as big is obtained by specifying `Penalty parameter` r.) For later major iterations, ρ_k is reduced in stages when it appears that the sequence $\{x_k, \lambda_k\}$ is converging. In many cases it is safe to specify `Penalty parameter 0.0` at the beginning, particularly if a problem is only mildly nonlinear. This may improve the overall efficiency.

In the output from MINOS, the term `Feasible subproblem` indicates that the *linearized constraints* (12)–(14) have been satisfied. In general, the nonlinear constraints are satisfied only in the limit, so that *feasibility* and *optimality* occur at essentially the same time. The nonlinear constraint violation is printed every major iteration. Even if it is zero early on (say at the initial point), it may increase and perhaps fluctuate before tending to zero. On "well behaved" problems, the constraint violation will decrease quadratically (i.e., very quickly) during the final few major iterations.

For certain rare classes of problem it is safe to request the values $\lambda_k = 0$ and $\rho_k = 0$ for all subproblems by specifying `Lagrangian = No` (in which case the nonlinear constraint functions are evaluated only once per major iteration). However for general problems, convergence is much more likely with the default setting, `Lagrangian = Yes`.

The merit function

Unfortunately, it is not known how to define a merit function $M(x, \lambda)$ that can be *reduced* at every major iteration. As a result, there is no guarantee that the projected Lagrangian method described above will converge from an arbitrary starting point. This has been the principal theoretical gap in MINOS, finally resolved by the PhD research of Michael Friedlander [Fri02]. The main features needed to stabilize MINOS are:

- To relax the linearized constraints via an ℓ_1 penalty function.
- To repeat a major iteration with increased ρ_k (and more relaxed linearized constraints) if the nonlinear constraint violation would increase too much.

In practice, the method of MINOS 5.51 often does converge and a good *rate* of convergence is often achieved in the final major iterations, particularly if the constraint functions are "nearly linear". As a precaution, MINOS prevents radical changes from one major iteration to the next. Where possible, the steplength is chosen to be $\sigma = 1$, so that each new estimate of the solution is $(x_{k+1},\ \lambda_{k+1}) = (\bar{x}, \bar{\lambda})$, the solution of the subproblem. If this point is "too different", a shorter steplength $\sigma < 1$ is chosen.

If the major iterations for a particular model do not appear to be converging, some of the following actions may help:

1. Specify initial activity levels for the nonlinear variables as carefully as possible.
2. Include sensible upper and lower bounds on all variables.
3. Specify a `Major damping parameter` that is lower than the default value. This tends to make σ smaller.
4. Specify a `Penalty parameter` that is higher than the default value. This tends to prevent excessive departures from the constraint linearization.

1.4 PROBLEM FORMULATION

In general, it is worthwhile expending considerable prior analysis to make the constraints completely linear if at all possible. Sometimes a simple transformation will suffice. For example, a pipeline optimization problem has pressure drop constraints of the form

$$\frac{K_1}{d_1^{4.814}} + \frac{K_2}{d_2^{4.814}} + \cdots \le P_T^2 - P_0^2,$$

where d_i are the design variables (pipe diameters) and the other terms are constant. These constraints are highly nonlinear, but by redefining the decision variables to be $x_i = 1/d_i^{4.814}$ we can make the constraints linear. Even if the objective function becomes more nonlinear by such a transformation (and this usually happens), the advantages of having linear constraints greatly outweigh this.

Similarly, it is usually best not to move nonlinearities from the objective function into the constraints. For example, we should *not* replace "minimize $F(x)$" by

$$\text{minimize} \quad z \quad \text{subject to} \quad F(x) - z = 0.$$

In fact, the GAMS system recognizes this situation as long as z is a free variable. When passing the problem to MINOS, GAMS converts that particular constraint into the MINOS objective function "minimize $F(x)$". (AMPL has a more direct method for specifying the objective function.)

Scaling is a very important matter during problem formulation. A general rule is to scale both the data and the variables to be as close to 1.0 as possible. In general we suggest the range 1.0 to 10.0. When conflicts arise, one should sacrifice the objective function in favor of the constraints. Real-world problems tend to have a natural scaling within each constraint, as long as the variables are expressed in consistent physical units. Hence it is often sufficient to apply a scale factor to each row. MINOS has options to scale the rows and columns of the constraint matrix automatically. By default, only the linear rows and columns are scaled, and the procedure is reliable. If you request that the nonlinear constraints and variables be scaled, bear in mind that the scale factors are determined by the initial Jacobian $J(x_0)$, which may differ considerably from $J(x)$ at a solution.

Finally, *upper and lower bounds* on the variables (and on the constraints) are extremely useful for confining the region over which optimization has to be performed. If sensible values are known, they should always be used. They are also important for avoiding singularities in the nonlinear functions. Note that bounds may be violated slightly by as much as the feasibility tolerance δ_{fea}. Hence, if $\sqrt{x_2}$

or $\log x_2$ appear (for example) and if $\delta_{\rm fea} = 10^{-6}$, the lower bound on x_2 would normally have to be at least 10^{-5}. If it is *known* that x_2 will be at least 0.5 (say) at a solution, then its lower bound should be 0.5.

For a detailed discussion of many aspects of numerical optimization, see Gill, Murray and Wright [GMW81]; in particular, see Chapter 8 for much invaluable advice on problem formulation and assessment of results.

1.5 RESTRICTIONS

MINOS is designed to find solutions that are *locally optimal*. The nonlinear functions in a problem must be *smooth* (i.e., their first derivatives must exist), especially near the desired solution. The functions need not be separable.

A certain "feasible" region is defined by the general constraints and the bounds on the variables. If the objective is convex within this region and if the feasible region itself is convex, any optimal solution obtained will be a *global optimum*. Otherwise there may be several local optima, and some of these may not be global. In such cases the chances of finding a global optimum are usually increased by choosing a starting point that is "sufficiently close", but there is no general procedure for determining what "close" means, or for verifying that a given local optimum is indeed global.

Integer restrictions cannot be imposed directly. If a variable x_j is required to be 0 or 1, a common ploy is to include a quadratic term $x_j(1-x_j)$ in the objective function. MINOS will indeed terminate with $x_j = 0$ or 1, but inevitably the final solution will just be a local optimum. (Note that the quadratic is negative definite. MINOS will find a global minimum for quadratic functions that are positive definite or positive semidefinite, assuming the constraints are linear.)

1.6 STORAGE

MINOS uses one large array of main storage for most of its workspace. The implementation places no fixed limit on the size of a problem or on its shape (many constraints and relatively few variables, or *vice versa*). In general, the limiting factor will be the amount of main storage available on a particular machine, and the computation time required. On personal computers with 640K of main memory, problem size is limited to perhaps 300 constraints and 500 variables for linear models, and somewhat less if there are many nonlinear variables. Machines with virtual memory can process several thousand constraints and many thousands of variables. If there are nonlinearities, we assume that they are not excessively expensive, since they may need to be evaluated thousands of times.

Some detailed knowledge of a particular model will usually indicate whether the solution procedure is likely to be efficient. An important quantity is m, the total number of general constraints in (2) and (3). The workspace required by MINOS is roughly m kilobytes (mK), plus about 300K for the program itself.

Another important quantity is n, the total number of variables in x and y. The previous comments assume that n is not much larger than m, the number of constraints. A typical ratio is $n/m = 2$ or 3.

If there are many nonlinear variables (i.e., if n_1 is large), much depends on whether the objective function or constraints are highly nonlinear or not. The crucial item is n_S, the number of superbasic variables at an optimum. We know that n_S is zero for purely linear problems, and that n_S need never be larger than $n_1 + 1$. In practice, n_S is often very much less than this upper limit. For example, consider a quadratic programming problem with objective function $c^Tx + \frac{1}{2}x^TQx$. In many cases the linear term c^Tx is dominant and MINOS will solve the problem almost as efficiently as if Q were zero. (The number of superbasics will remain small throughout.) On the other hand, if the quadratic term is dominant and involves a large number of variables, the final number of superbasics may be very large.

The same is true for more general nonlinearities.

In the quasi-Newton algorithm, the dense triangular matrix R has dimension n_S and requires about $0.004 n_S^2$K of storage. If it seems likely that n_S will be very large, some aggregation or reformulation of the problem should be considered.

1.7 FILES

MINOS operates primarily within main memory, and is well suited to a virtual storage environment. Certain disk files are accessed as follows:

Input file	*Record Length* (characters)
Specs file	72
Read file	not used
MPS file	61
Basis files	80

Output file	*Record Length* (characters)
Summary file	78
Print file	129
Solution file	111
Basis files	80

Fixed-length, blocked records may be used in all cases, and the files are always accessed sequentially. The logical record length must be at least that shown. For efficiency, the physical block size should be several hundred characters in most cases.

Unit numbers for the Specs, Read, Summary and Print files are defined at compile time; typically they will be 4, 5, 6 and 9, but they may depend on the installation. The remaining unit numbers are specified at run time in the Specs file.

Unit numbers for the Read, Print and Summary files are stored in the following Fortran common block:

```
common    /m1file/ iread,iprint,isumm
```

It may be convenient to reference these in the user subroutines `funobj`, `funcon` and `matmod`, or in a program that calls subroutine `minoss`.

The READ file is not used explicitly by MINOS, but its unit number is used to test if a file should be rewound. (Thus, input files are subject to a Fortran `rewind` as long as they are not the same as the Read file.) The Print file is used frequently. Other output files are rewound if they are not the same as the Print file.

The following table summarizes when certain files are needed:

	Specs	MPS
MINOS	Yes	Yes
`minoss`	Optional	No
GAMS	Optional	No
AMPL	No	No

1.8 INPUT DATA FLOW

Some or all of the following items are supplied by the user:

- Subroutine `funobj`
- Subroutine `funcon`

- Subroutine `matmod`
- A Specs file
- An MPS file
- A Basis file
- Data read by `funcon` on its first entry
- Data read by `funobj` on its first entry
- Data read by `funcon` on its last entry
- Data read by `funobj` on its last entry

The order of the files and data is important if all are stored in the same input stream.

Subroutines `funobj` and `funcon` define the nonlinear objective and constraint functions respectively (if any); they are not needed for linear programs.

Subroutine `matmod` is occasionally needed for applications involving a sequence of closely related problems.

The Specs file defines various run-time parameters such as the `Iteration limit`. Its file number is defined at compile time.

The MPS file specifies names for the constraints and variables, and defines all the linear constraints and bounds. The data format is similar to that used in commercial mathematical programming systems (hence the name). The format has been generalized slightly for nonlinear problems.

If desired, a Basis file may be loaded at the beginning of a run. This will normally have been saved at the end of an earlier run. Three kinds of basis file are available; they are used to restart the solution of a problem that was interrupted, or to provide a good starting point for some slightly modified problem.

1.9 MULTIPLE SPECS FILES

One or more problems may be processed during a run. The parameters for a particular problem are delimited by `Begin` and `End` in the Specs file. While scanning for the keyword `Begin`, MINOS recognizes the keywords `Skip` and `Endrun`. Thus in the example

```
Begin Case 1
    .
    .
End Case 1
Skip Case 2
    .
    .
End Case 2
Begin Case 3
    .
    .
End Case 3
Endrun
Begin Case 4
    .
    .
End Case 4
```

only the first and third problem will be processed.

1.10 INTERNAL MODIFICATIONS

A sequence of closely related problems may be specified within a single SPECS file, via the `Cycle` parameter; for example,

```
Begin Cycling example
   .
   .
   Cycle limit        10
End example
```

indicates that up to 10 problems are to be processed. This is intended for cases where the solution of one problem P_k is needed to *define* the next problem P_{k+1}.

The actual method for defining the next problem in a cycle depends on the application. Sometimes it can be done by changing the output from the function subroutines `funobj` and/or `funcon`. Alternatively, the user may provide a third subroutine `matmod` to perform some modifications to the problem data. If the `Cycle limit` is 2 or more, `matmod` is called before the problem is solved (cycle 0), and also after each solve (cycle 1, 2, 3, ...).

If necessary, the number of linear variables can be *increased* when a problem P_{k+1} is defined. We think of this as *adding new columns to* P_k. The new columns are not included in the MPS file, and their sparsity pattern need not be known until P_k has been solved. Instead, an appropriate number of `Phantom columns` and `Phantom elements` are defined in the SPECS file (to reserve a pool of storage), and the user's subroutine `matmod` generates each new column by calling the MINOS subroutine `matcol`.

Chapter 2

September 23, 2003

User-written Subroutines

To solve linear programs, you must give MINOS the constant data defining A, c, etc. For nonlinear problems, you must also provide some Fortran code to compute your nonlinear functions. Nonlinearities in the objective are defined by subroutine `funobj`. Those in the constraints are defined separately by subroutine `funcon`. On every entry (except perhaps the last), these subroutines must return appropriate function values `f`. Wherever possible, they should also return all gradient elements in the array `g(*)`. This provides maximum reliability and corresponds to the default setting, `Derivative level 3` in the SPECS file.

In practice it is often convenient not to code gradients. MINOS is able to estimate gradients by finite differences, by making a call to `funobj` or `funcon` for each nonlinear variable x_j whose partial derivatives need to be estimated. However, this reduces the reliability of the optimization algorithms, and can be very expensive if there are many nonlinear variables.

As a compromise, MINOS allows you to code *as many gradients as you like*. This option is implemented as follows: just before a function routine is called, each element of the gradient array `g(*)` is initialized to a specific value (-111111.0). On exit, any element retaining that value must be estimated by finite differences.

Some rules of thumb follow for writing `funobj` and `funcon`:

1. For maximum reliability, use `Derivative level 3` and code all gradients analytically.

2. While developing the function routines, use the `Verify` parameter to check the computation of any gradient elements that are supposedly known.

3. If not all gradients are known, compute as many of them as you can. (It often happens that some of them are constant or even zero.) It may then be convenient to compute the known gradients every time the function routines are called, even though they will be ignored if `mode` $= 0$.

4. If the known gradients are expensive to compute, set `Derivative level 0` and use the parameter `mode` to avoid computing on certain entries. `Derivative level 0` requests a function-only linesearch, even if you compute all gradients. During the linesearch, `funobj` and `funcon` are called with `mode` $= 0$, not `mode` $= 2$. An extra call with `mode` $= 2$ is needed after the search, but the net cost may be less if gradients are very expensive.

   This approach is recommended if you happen to be estimating derivatives by your own finite-difference scheme. (It is much easier to let MINOS do the differencing—one column at a time—but if you have nonlinear constraints, by using the sparsity structure of the Jacobian you may be able to make `funcon` estimate several columns of the Jacobian at once.)

2.1 SUBROUTINE FUNOBJ

This subroutine is provided by the user to calculate the objective function $F(x)$ and as much of its gradient $g(x)$ as possible. It is not needed if the objective is linear and entered as a row of A.

Specification:

```
subroutine funobj( mode, n, x, f, g, nstate, nprob, z, nwcore )
```

```
implicit           double precision (a-h,o-z)
integer            mode, n, nstate, nprob, nwcore
double precision   f, x(n), g(n), z(nwcore)
```

Note: The `double precision` declarations should be `real` on machines for which single-precision floating-point is adequate; e.g., Cray and Convex systems. The `implicit` statement is then not necessary. In general, it is safer to declare all variables explicitly and to specify `implicit none` if your Fortran compiler allows it.

On entry:

`mode` says whether or not to compute gradients. It can be ignored if `Derivative level` is 1 or 3. In this case, `mode` will always have the value 2 and you must compute all elements of `g(*)`.

Otherwise, when `Derivative level` is 0 or 2, MINOS will call `funobj` sometimes with `mode` = 2 and sometimes with `mode` = 0. You may test `mode` to decide what to do:

> If `mode` = 2, compute `f` and as many elements of `g(*)` as possible.
>
> If `mode` = 0, compute `f` but set `g(*)` only if you wish. (On exit, the contents of `g(*)` will be ignored.)

`n` is n_1', the number of variables involved in $F(x)$. They must be the first n_1' variables in the problem. For stand-alone MINOS, n_1' is defined by `Nonlinear variables` or `Nonlinear objective variables` in the SPECS file. For `minoss`, n_1' is the parameter `nnobj`.

`x(*)` contains the current values of the nonlinear objective variables, x.

`nstate` indicates the first and last entries to `funobj`.

If `nstate` = 0, there is nothing special about the current call to `funobj`.

If `nstate` = 1, MINOS is calling your subroutine for the first time. Some data may need to be input or computed and saved in local or `common` storage. Note that if there are nonlinear constraints, the first call to `funcon` will occur *before* the first call to `funobj`.

If `nstate` $\geq$ 2, MINOS is calling your subroutine for the *last* time. You may wish to perform some additional computation on the final solution. (If `Cycle limit` is specified, this call occurs at the end of each cycle.) Note again that if there are nonlinear constraints, the last call to `funcon` will occur *before* the last call to `funobj`.

In general, the last call is made with `nstate` = 2 + `ierr`, where `ierr` indicates the status of the final solution. In particular, if `nstate` = 2, the current `x` is optimal; if `nstate` = 3, the problem appears to be infeasible; if `nstate` = 4, the problem appears to be unbounded; and if `nstate` = 5, the iterations limit was reached. In some cases, the solution may be *nearly* optimal if `nstate` = 11; this value occurs if the linesearch procedure was unable to find an improved point.

If the nonlinear functions are expensive to evaluate, it may be desirable to do nothing on the last call, by including the following statement at the start of the subroutine: `if (nstate .ge.  2) return`.

`nprob` is an integer that you can set to the value i by saying `Problem number` i. The default value is `nprob` = 0.

`z(*)` is the primary work array used by MINOS and `minoss`. In certain applications it may be desirable to access parts of this array, using various `common` blocks to pinpoint the required locations; see §§A.5 and **??**. Otherwise, `z(*)` and `nwcore` can be ignored.

`nwcore` is the dimension of `z(*)`. It allows compilers to check that you don't access elements outside `z(*)`.

On exit:

`mode` is an error indicator. Normally `mode` should not be altered, but if for some reason you wish to terminate solution of the current problem, set `mode` ≤ -2.

You may set `mode` $= -1$ to mean "My nonlinear function is undefined here". During normal iterations, this signals the linesearch to try again with a shorter steplength.

`f` is the computed value of the objective function $F(x)$.

`g(*)` is the computed gradient vector $g(x)$. In general, `g(`j`)` should be set to the partial derivative $\partial F/\partial x_j$ for as many j as possible (except perhaps if `mode` $= 0$; see above).

2.2 SUBROUTINE FUNCON

This subroutine is provided by the user to compute the nonlinear constraint functions $f(x)$ and as many of their gradients as possible. It is not needed if the constraints are entirely linear. Note that the gradients of the vector $f(x)$ define the Jacobian matrix $J(x)$. The j-th column of $J(x)$ is the vector $\partial f/\partial x_j$. The i-th row of $J(x)$ is the gradient vector $\partial f^i/\partial x$.

Subroutine `funcon` may be coded in two different ways, depending on the method used for storing the Jacobian, as specified in the SPECS file. The default method `Jacobian = Dense` is the simplest. It is suitable for moderate-sized problems.

`Jacobian = Dense`

Specification:

```
      subroutine funcon( mode, m, n, njac, x, f, g,
     $                   nstate, nprob, z, nwcore )

      implicit           double precision (a-h,o-z)
      integer            mode, m, n, njac, nstate, nprob, nwcore
      double precision   x(n), f(m), g(m,n), z(nwcore)
```

Note: As in `funobj`, `double precision` should be `real` on some machines.

On entry:

`mode` says whether or not to compute gradients. It can be ignored if `Derivative level` is 2 or 3. In this case, `mode` will always have the value 2 and you must compute all elements of `g(*,*)` (except perhaps if there are some constant Jacobian elements; see next section).

Otherwise, when `Derivative level` is 0 or 1, you may test `mode` to decide what to do:

If `mode` = 2, compute `f(*)` and as much of `g(*,*)` as possible.

If `mode` = 0, compute `f` but set `g(*,*)` only if you wish. (On exit, the contents of `g(*,*)` will be ignored.)

`m` is m_1, the number of nonlinear constraints (not counting the objective function). These must be the first m_1 constraints in the problem. Any linear constraints or linear objective rows must come *after* the nonlinear rows. For stand-alone MINOS, m_1 is defined by `Nonlinear constraints` in the SPECS file. For `minoss`, m_1 is the parameter `nncon`.

`n` is n_1'', the number of variables involved in $f(x)$. These must be the first n_1'' variables in the problem. For stand-alone MINOS, n_1'' is defined by `Nonlinear variables` or `Nonlinear Jacobian variables` in the SPECS file. For `minoss`, n_1'' is the parameter `nnjac`.

`njac` is the value `m*n`.

`x(*)` contains the current values of the nonlinear Jacobian variables, x.

On exit:

`mode` is an error indicator. Normally `mode` should not be altered, but if for some reason you wish to terminate solution of the current problem, set `mode` ≤ -2.

You may set `mode` $= -1$ to mean "My nonlinear function is undefined here". During normal iterations, this signals the linesearch to try again with a shorter steplength.

`f(*)` contains the computed values of the functions in the constraint vector $f(x)$.

`g(*,*)` is the computed Jacobian matrix $J(x)$. The vector $\partial f/\partial x_j$ should be stored in the j-th column of the 2-dimensional array `g(*,*)` (except perhaps if `mode` = 0; see above). Equivalently, the gradient of the i-th constraint should be stored in the i-th row of `g(*,*)`. Thus,

$$\mathtt{g}(i,j) = \partial f^i/\partial x_j, \qquad \mathtt{g}(*,j) = \partial f/\partial x_j, \qquad \mathtt{g}(i,*) = \partial f^i/\partial x.$$

The other parameters are the same as for subroutine `funobj`.

`Jacobian = Sparse`

Specification:

```
    subroutine funcon( mode, m, n, njac, x, f, g,
   $                   nstate, nprob, z, nwcore )

    implicit           double precision (a-h,o-z)
    integer            mode, m, n, njac, nstate, nprob, nwcore
    double precision   x(n), f(m), g(njac), z(nwcore)
```

This is the same as for `Jacobian = Dense`, except for the declaration of `g(njac)`.

On entry:

`njac` is the number of nonzero elements in the Jacobian matrix $J(x)$. For stand-alone MINOS, this is exactly the number of entries in the MPS file that referred to nonlinear rows and nonlinear Jacobian columns (the first `m` rows in the ROWS section and the first `n` columns in the COLUMNS section).

Usually `njac` will be less than `m*n`. The actual value of `njac` may not be of any use when coding `funcon`, but any expression involving `g(`l`)` should have the subscript l between `1` and `njac`.

On exit:

`g(*)` is the computed elements of the Jacobian matrix (except perhaps if `mode` = 0; see previous page).

For stand-alone MINOS, these elements must be stored column-wise into `g(*)` in exactly the same positions as implied by the MPS file, ignoring elements in linear rows. For `minoss`, `g(*)` must be stored in the positions implied by parameters `a(*)` and `ha(*)` (ignoring elements in linear rows). There is no internal check for consistency (except indirectly via the `Verify` parameter), so great care is essential.

The other parameters are the same as for `Jacobian = Dense`.

2.3 CONSTANT JACOBIAN ELEMENTS

Suppose that `Derivative level` is 2 or 3. This usually means that `funcon` will compute all constraint gradients (so MINOS does not have to estimate them by finite differences). However, any *constant* elements of `g(*)` need not be set by `funcon`. Instead, you may think of them as being *initialized* in the MPS file (for stand-alone MINOS) or in the `minoss` parameter `a(*)`. An element of `g(*)` that is not computed in `funcon` will retain its initial value.

This feature is useful when `Jacobian = Dense` and many Jacobian elements are identically zero. Such elements need not be specified in the MPS file, nor set in `funcon` (but for `minoss` they must be initialized at zero in `a(*)`).

Note that constant *nonzero* elements do affect `f(*)`. If J_{ij} is constant, the array element `g(`i`,`j`)` need not be set in `funcon`, but the value `g(`i`,`j`)*x(`j`)` must be added to `f(`i`)`.

When `Jacobian = Sparse`, constant Jacobian elements are normally nonzero. If the correct initial value is supplied, the corresponding element `g(`l`)` need not be reassigned in `funcon`, but a term of the form `g(`l`)*x(`j`)` must be added to one of the elements of `f(*)`. (This feature allows a matrix generator to output constant data to the MPS file. Similarly, a program calling `minoss` can set constant data within the parameter `a(*)`. Subroutine `funcon` does not need to know that data at compile time, but can *use* it at run time to compute the elements of `f(*)`.)

Remember, if `Derivative level` is 0 or 1 then unassigned elements of `g(*)` are *not* treated as constant; they are estimated by finite differences, at significant expense.

2.4 SUBROUTINE MATMOD

This subroutine allows you to define a sequence of related problems and have them solved one by one. It is used in conjunction with the `Cycle` options (and possibly the `Phantom` options; see subroutine `matcol`).

If the `Cycle limit` is 1 (the default), `matmod` is never called. If it is 2 or more, `matmod` is called before the original problem is solved (cycle 0), and also after each problem in the sequence is solved (cycle 1, 2, 3, ...).

Within `matmod` you may change the problem data in any desired way; for example, you might alter some bounds on the variables or revise some of the constraint coefficients. You could also communicate with the function routines `funobj` and `funcon` to alter their behavior (typically by altering variables in one of your own `common` blocks). Finally, `matmod` may specify whether a "warm" or "hot" start should be used when MINOS starts solving the new problem.

In general, `matmod` is intended for use with stand-alone MINOS (since `minoss` can be called repeatedly with arbitrary problem changes between calls).

Specification:

```
      subroutine matmod( ncycle, nprob, finish,
     $                   m, n, nb, ne, nka, ns, nscl, nname,
     $                   a, ha, ka, bl, bu,
     $                   ascale, hs, name1, name2,
     $                   x, pi, rc, z, nwcore )

      implicit           double precision (a-h,o-z)
      integer            ncycle, nprob,
     $                   m, n, nb, ne, nka, ns, nscl, nname, nwcore
      logical            finish
      integer*4          ha(ne), hs(nb)
      integer            ka(nka), name1(nname), name2(nname)
      double precision   a(ne), ascale(nscl), bl(nb), bu(nb),
     $                   x(nb), pi(m), rc(nb), z(nwcore)
```

Note: The `integer*4` declaration normally means the same as `integer`, but to conserve storage in some installations, all `integer*4`s in the MINOS source code may have been changed to `integer*2`. If so, the same applies here and in subroutine `matcol` below.

On entry:

`ncycle` says how many problems have been solved.

If `ncycle` = 0, this is the first time `matmod` has been called. MINOS has read the MPS file. If a Basis file was specified, it has been read and `hs` is defined. (Otherwise, Crash has not yet been called and `hs` does not define a basis.) The problem has not yet been scaled or solved. This entry allows `matmod` to initialize problem-dependent quantities. If you do not wish to do anything before solving the first problem, include the following statement at the beginning of `matmod`:
`if (ncycle .eq.  0) return`.

Otherwise, `ncycle` gives the number of the cycle that has just terminated. It is the number of problems that have been solved.

`nprob` is 0 by default. It can be set to the value i by saying `Problem number` i in the Specs file.

`finish` is `.false.`

`m` is m, the number of rows in the constraint matrix.

`n` is n, the number of variables, excluding slacks.

`nb` is $nb = n + m$, the number of variables, including slacks. (It is the length of many arrays including `bl` and `bu`. The name is short for Number of Bounds.)

`ne` is the number of elements in the constraint matrix (used only to dimension `a(*)` and `ha(*)`).

`nka` is $n + 1$ (used to dimension `ka`).

`ns` is the number of superbasic variables.

`nscl` says if the problems are being scaled prior to each solve. If `nscl` = 1, scaling has not been specified. (There is only one element in the array `ascale` and it is undefined.) Otherwise, `nscl` = `nb` and `ascale` contains the scales used for the cycle that just finished (assuming `ncycle` > 0). However, the problem itself has been unscaled.

`nname` is normally the same as `nb`, assuming MINOS read an MPS file. If `matmod` is for some reason being used with `minoss`, `nname` is the same as the `minoss` parameter: it may be `nb` or 1, depending on whether names exist.

`a(*)`, `ha(*)`, `ka(*)` contain the current constraint matrix. See §A.5.

`bl(*)`, `bu(*)` are the lower and upper bounds on all column and slack variables (x, s), in that order.

`ascale(*)` contains scale factors for columns and rows, in that order. They are defined if `ncycle` > 0 and scaling has been requested (`nscl` > 1).

`hs(*)` is the state vector for all variables. See §A.5.

`name1(*)`, `name2(*)` contain the first and second halves of the names of the columns and rows, in that order, in `a4` format. For example, if the 20th variable were named `'Capital '`, we would have `name1(20)` = `'Capi'` and `name2(20)` = `'tal '`. Sometimes it may be useful to determine the index of a column or row by searching these arrays.

`x(*)` contains the values of all variables (x, s).

`pi(*)` contains the values of the dual variables π. The first m_1 components are the current estimates of λ, the Lagrange multipliers for the nonlinear constraints. In some cases, good initial values for λ can assist convergence of the projected Lagrangian algorithm. They may be provided to MINOS by the MPS file, but it may be more convenient to define them in `matmod` on the first entry, when `ncycle` = 0. (Subroutine `minoss` allows them to be passed in directly.)

`z(*)` is the primary work array used by MINOS. As in `funobj` or `funcon`, it may be desirable to access parts of this array, using various `common` blocks to pinpoint the required locations.

`nwcore` is the dimension of `z(*)`.

On exit:

`finish` should be set to `.true.` if you wish the cycles to be terminated; e.g., if some convergence criterion has been satisfied. The `Cycle tolerance` may be useful for specifying a numerical value at run-time. This is stored in the variable `cnvtol` in the common block

```
common    /cyclcm/ cnvtol,jnew,materr,maxcy,nephnt,nphant,nprint
```

`a(*)` contains the constraint matrix, perhaps with some elements modified. In general, coefficients cannot be added or deleted; instead, existing entries may be changed to or from zero.

With care, a coefficient could be moved to some other row in the same column, by altering `ha(*)` appropriately. With even greater care, a coefficient could be moved from one column to another, by altering both `ha(*)` and `ka(*)`.

Additional entries may be created at the *end* of `a(*)`, `ha(*)` and `ka(*)`; see subroutine `matcol`.

If scaling has been requested, a problem is unscaled at the end of each cycle, and rescaled at the beginning of the next. Subroutine `matmod` may therefore treat `a`, `bl`, `bu`, `x` and `pi` as being in original units.

2.5 WARM AND HOT STARTS

At the end of each cycle, MINOS performs certain functions to initiate the solution of the next problem. The options are:

- retain the current basis B, or perform Crash;
- retain the current basis factorization $B = LU$, or refactorize;
- retain the current projected Hessian matrix $R^T R$, or reset it;
- retain existing scales, or compute new ones from the current constraint matrix A and/or the Jacobian;
- retain the current Lagrange multiplier estimates λ for nonlinear constraints, or set them to specified values.

The default action is to retain current values. This may not be desirable in some cases; for example, if `a(*)` is changed, the basis factorization may be rendered incorrect. If necessary, `matmod` should include the declarations

```
logical            gotbas,gotfac,gothes,gotscl
common   /cycle1/ gotbas,gotfac,gothes,gotscl
```

and should change the appropriate logical variables from true to false.

For example, if `gotscl` remains true, MINOS will use the scale factors from the previous solve (assuming scaling has been requested). If the matrix coefficients in `a(*)` have changed significantly, it may be better to set `gotscl` = `.false.` and let the scaling procedure be invoked again.

Similarly, if `gotbas` remains true, a basis is assumed to be specified by the array `hs(*)`. It should provide a good starting point if the problem has not been dramatically altered. Otherwise, setting `gotbas` = `.false.` means that you want Crash to choose a starting basis (a cold start). (The arrays `hs(*)` and `xn(*)` still come into play, in the same way as they do on entry to subroutine `minoss`).

For problems with nonlinear constraints, the current basis factorization will not be retained even if `gotfac` remains true. When the constraints are linear, the main thing to remember is that if you change `a(*)` or `ha(*)` (even just linear objective coefficients), it is essential to set `gotfac` = `.false.`

For any nonlinear problems, if `gotbas` remains true then it is probably sensible to leave `gothes` true, unless for some reason the number of superbasic entries in `hs(*)` is altered.

For problems with nonlinear constraints, the multiplier estimates λ_k may be changed by resetting the appropriate elements of the array `pi(*)`.

For subroutine `minoss`, the logical variables in the above common block are set via the first parameter, `start`, which may be `'Cold'`, `'Warm'`, `'Hot'` or `'Hot xxx'`, where `xxx` is any number of the letters `FHS` to say "Keep the Factors, Hessian and/or Scales" as appropriate. See subroutine `minoss`.

2.6 SUBROUTINE MATCOL

If `Phantom columns` c and `Phantom elements` e are defined in the SPECS file (along with `Cycle limit` k), the MINOS subroutine `matcol` may be called by `matmod` up to c times during cycles 2 through k. The aim is to turn at most c "Phantom columns" into normal columns containing a total of at most e nonzero elements. `matmod` must provide an array `col(*)` and a zero tolerance `ztol` for each call. The significant elements of `col` will be packed into the matrix data structure, to form a new column. The associated variable will be given the default `Lower bound` and `Upper bound`, and a scale factor of 1.0.

Specification:

```
    subroutine matcol( m, n, nb, ne, nka,
   $                   a, ha, ka, bl, bu, col, ztol )

    implicit           double precision (a-h,o-z)
    integer            m, n, nb, ne, nka
    integer*4          ha(ne)
    integer            ka(nka)
    double precision   a(ne), bl(nb), bu(nb), col(m), ztol
```

On entry:

`m` is the length of the array `col(*)`. Usually this will be m, the number of rows in the constraint matrix. In general, it may be anywhere in the range $1 \le \mathtt{m} \le m$, if the new column is known to be zero beyond position `m`.

`n` is the total number of columns in the problem, including all Phantom columns. This number is the same for all calls to `matcol`.

`col(*)` is a the dense vector that is to be packed to become a new matrix column.

`ztol` is a "zero tolerance" for deleting negligible elements from `col` when it is packed into `a(*)` and `ha(*)`. On most machines, a reasonable value is `ztol = 1.0e-8`.

On exit:

`a(*)`, `ha(*)`, `ka(*)`, `bl(*)`, `bu(*)` will have been suitably modified.

The other parameters come directly from `matmod`.

In general it is advisable to make `matmod` test for errors when calling `matcol`. At present, this is done by declaring the common block

```
      common    /cyclcm/ cnvtol,jnew,materr,maxcy,nephnt,nphant,nprint
```

within `matmod`. If `materr` $= 0$ after a call to `matcol`, a new column was successfully created, and it is now column number `jnew`. For further details, see the `Cycle` options in §?? and the example in §??.

The SPECS File

The performance of MINOS is controlled by many parameters or "options", each of which has a default value. A SPECS file allows you to specify some of the options for a particular problem. Note that *most options should be left at their default value.* If experimentation is necessary, we recommend changing just one option at a time.

The following examples illustrate the format. In all cases, the first group of options is needed by stand-alone MINOS (for reading the MPS file) but not by `minoss`.

Linear Programs. To solve LP problems of any dimension up to some known limits, use a SPECS file of following form:

```
Begin LP example
   MPS file                 10
   Rows                   2400   * These 3 values aren't exact.
   Columns               10000   * They are suitable for the
   Elements              71000   * first 60 Netlib test problems.

   Iteration limit       40000
   Solution                 No
End LP specs
```

Problems with a Nonlinear Objective. Here, we have to give the number of variables in the objective, and an over-estimate of the final number of superbasics.

```
Begin Nonlinear Objective example
   MPS file                 10   * These 5 lines are not needed
   Rows                   1000   * if you are calling minoss.
   Columns                2000   *
   Elements               7000   *
   Nonlinear variables     150   * This value must be exact.

   Scale                    No   * if constraints are well scaled.
   Verify gradients              * until they all "seem OK".
   Superbasics limit       100   * or bigger if necessary (but less than no. nonlinearities).
   Iterations             5000
End
```

Problems with Nonlinear Constraints. The nonlinear dimensions must be given exactly. Here, the objective and constraints have the same number of nonlinear variables.

```
Begin Nonlinear Constraint example
   MPS file                 10
   Rows                   1000
   Columns                2000
```

```
   Elements              7000
   Nonlinear constraints  100   * These 2 values
   Nonlinear variables    300   * must be exact.

   Scale                   No   * at least initially.
   Verify gradients             * until they "seem OK".
   Superbasics            100
End
```

Mildly Nonlinear Constraints. Models that are very nearly linear may optimize more efficiently if some of the "cautious" defaults are relaxed. (Here, the objective and constraints are nonlinear in different, perhaps overlapping, sets of variables.)

```
Begin Easy Nonlinear Constraint example
   ...
   ...
   Nonlinear constraints         100   * These 3 values
   Nonlinear Jacobian  variables  200   * must be exact.
   Nonlinear objective variables  300   *

   Scale                   No
   Superbasics            100
   Minor iterations       100
   Penalty parameter      0.1   * or perhaps 0.0
End
```

Highly Nonlinear Constraints. Conversely, there is no guarantee that the major iterations will converge. Raising the penalty parameter and reducing the damping parameter from their default values may help.

```
Begin Difficult Nonlinear Constraint example
   ...
   ...
   Nonlinear constraints         100
   Nonlinear Jacobian  variables  200
   Nonlinear objective variables    0   * The objective is linear.

   Superbasics             100
   Major iterations        200
   Iterations             5000
   Completion             Full
   Penalty parameter       5.0
   Major damping parameter  0.1
   Row tolerance           1.0e-5   * for nonlinear rows.
   Optimality tolerance    1.0e-5   * to stop a little early.
End
```

3.1 SPECS FILE FORMAT

The first and last lines of a SPECS file contain `Begin` and `End` as shown above. Most of the first line is echoed to the SUMMARY file. Comments may appear on any lines following a `*`, or beyond column

72. Comments and blank lines are ignored. Otherwise, each line specifies a single option using the following items:

1. A *keyword*, such as `LU`.
2. A *phrase*, such as `factor tolerance`. This is zero or more words, none of which begins with a digit, `+` or `-`.
3. A *number*, such as `10.0` (only for some options). This is an integer or real value with up to 16 contiguous characters in Fortran's `I`, `F`, `E` or `D` formats.

The items may appear anywhere before column 72, in upper or lower case. Some of the keywords have synonyms, and abbreviations are allowed if there is no ambiguity.

Stand-alone MINOS requires a SPECS file. It is input from a predetermined unit (typically unit 4). For subroutine `minoss`, the driving program must first call `mispec` (§7.2) to input a SPECS file from a specified unit.

The following sections describe all options that may appear in the SPECS file, and give their default values. The number ϵ denotes machine precision (typically 10^{-15} or 10^{-16}). The options are grouped in the following order:

3.2 MPS File
3.3 Linear Programmming
3.4 All Problems
3.5 Nonlinear Objectives
3.6 All Nonlinear Problems
3.7 Nonlinear Constraints
3.8 Input and Output

3.2 OPTIONS FOR THE MPS FILE

The following options apply to stand-alone MINOS, for reading constraint data from an MPS file. They are not needed by subroutine `minoss`.

`Aij tolerance` t Default = `1.0e-10`

During input of the MPS file, matrix elements a_{ij} in the COLUMNS section are ignored if $|a_{ij}| < t$. If `Cycle limit > 1` and a_{ij} is to be changed from zero to a value greater than t during a later cycle, set $t = 0.0$ to retain all entries in the MPS file.

This tolerance does not apply to Jacobian elements (i.e., those belonging to nonlinear constraints and Jacobian variables: $i \le m_1$ and $j \le n_1''$).

`Bounds` a Default = blank

This specifies the 8-character name of a Bounds set to be selected from the BOUNDS section of the MPS file. If a is blank, MINOS selects the first set (if any). If you don't want the first set to be used, specify a dummy name such as `Bounds = NONE`.

`Columns` n Default = $3 *$ `Rows`

This must specify an *over-estimate* of the number of columns in the constraint matrix (excluding slack variables, but including any `Phantom columns`). If n proves to be too small, MINOS issues a warning and continues reading the MPS file to determine the true value. If the SPECS file and MPS file are on different units, the MPS file is re-read. Otherwise, the problem is terminated.

`Elements` e Default = $5 *$ `Columns`

This must specify an *over-estimate* of the number of nonzero elements a_{ij} in the constraint matrix, including all entries in a `Dense` or `Sparse` Jacobian, and all nonzeros in the matrices A_1, A_2, A_3. (It should also include the number of `Phantom elements`, if any.)

`Coefficients` is a valid alternative keyword. If e proves to be too small, MINOS continues in the manner described under `Columns`.

`Error message limit` k Default = 10

This is the maximum number of error messages to be printed for each type of error occurring when the MPS file is read. The default value is reasonable for early runs on a particular problem. If the same MPS file is used repeatedly, k can be reduced to suppress warning of non-fatal errors.

`Jacobian` `Dense` Default
`Jacobian` `Sparse`

If there are nonlinear constraints, this determines the manner in which the constraint gradients are evaluated and stored. It affects the MPS file and subroutine `funcon`.

The `Dense` option is convenient if there are not many nonlinear constraints or variables. It requires storage for three dense matrices of order $m_1 \times n_1$. The MPS file may then contain any number of Jacobian entries. Usually this means no entries at all.

For efficiency, the `Sparse` option is preferable in all nontrivial cases. (*Beware—it must be specifically requested.*) The MPS file must then specify the position of all Jacobian elements (that are not identically zero), and subroutine `funcon` must store the elements of the Jacobian array `g` in exactly the same order.

In both cases, if `Derivative level = 2` or `3` the MPS file may specify Jacobian elements that are constant for all values of the nonlinear variables. The corresponding elements of `g` need not be reset in `funcon`.

`List limit` k Default = 0

This limits the number of lines of the MPS file to be listed on the Print file during input. All comments and headers are listed (`NAME`, `ROWS`, `COLUMNS`, etc.), along with their position in the file.

`Lower bound` l Default = 0.0

This specifies a default lower bound for all variables (excluding slacks). Individual variables may have their lower bound altered by a Bounds set in the MPS file.

`Lower bound = 1.0e-5` (say) is a useful method for bounding all variables away from singularities at zero. Explicit bounds may also be necessary in the MPS file.

If all or most variables are to be *free*, use `Lower bound = -1.0e+20` to specify "minus infinity". The default upper bound is already `1.0e+20`, which is treated as "plus infinity".

`MPS file` k Default = Specs file

This is the Fortran file number containing the required MPS file. For nontrivial problems it is usually best to store the MPS file separately from the Specs file. If the `Rows`, `Columns` or `Elements` estimates prove to be too low, MINOS determines the correct values and re-reads the MPS file.

The default value is the same as for the Specs file, since it may be convenient to keep the Specs and MPS files together. The value is system-dependent. It is typically set to `ispecs = 4` in subroutine `minos1`.

`Nonlinear constraints`	m_1	Default = 0
`Nonlinear variables`	n_1	Default = 0
`Nonlinear objective variables`	n_1'	Default = 0
`Nonlinear Jacobian variables`	n_1''	Default = 0

These keywords are needed if a problem has a nonlinear objective or nonlinear constraints or both. They define the parameters `m` and `n` in subroutines `funobj` and `funcon`. For example, `m` in `funcon` takes the value m_1, if $m_1 > 0$.

`Nonlinear variables` may be used if only the objective is nonlinear, or if the objective function and the constraints involve the *same* set of nonlinear variables x. (It sets `n` $= n_1$ in `funobj` and `funcon`.)

Otherwise, specify n_1' and n_1'' separately. If $m_1 = 0$, the value $n_1'' = 0$ is assumed, regardless of n_1 or n_1'.

Remember that the nonlinear constraints and variables must always be the first ones in the problem. If both the objective and constraints are nonlinear, it is usually best to place Jacobian variables before objective variables, so that $n_1'' \le n_1'$. This affects the way the function subroutines should be coded, and the order in which variables should be placed in the COLUMNS section of the MPS file.

For example, a problem of the form "min x^2 subject to $6x + y^2 + z^2 = 8$" would be better written as "min z^2 subject to $x^2 + y^2 + 6z = 8$", so that $n_1' = 3$ and $n_1'' = 2$.

`Objective`	a	Default = blank

This specifies the 8-character name of a type `N` row to be selected from the ROWS section of the MPS file. For linear programs, that row defines the objective function $c^T x$. More generally, it defines a linear function $c^T x + d^T y$ to be added to the nonlinear objective $F(x)$.

If a is blank, MINOS selects the first `N` row (if any). If you don't want any `N` rows to be used, specify a dummy name such as `Objective = NONE`. If the objective is defined entirely by subroutine `funobj`, it may be helpful to specify `Objective = funobj`. (However, don't expect a different name to invoke a different subroutine!)

Note: Objective rows must be listed *after* nonlinear constraint rows in the ROWS section of the MPS file.

`Phantom columns`	c	Default = 0
`Phantom elements`	e	Default = 0

These specify that some extra columns and matrix elements may be generated beyond those contained in the MPS file. The `Columns` and `Elements` keywords must be large enough to include c and e respectively.

The `Cycle` keywords are also relevant (§3.4). Your subroutine `matmod` must generate each new column by calling the MINOS subroutine `matcol`.

`Ranges`	a	Default = blank

This specifies the 8-character name of a Ranges set to be selected from the RANGES section of the MPS file. If a is blank, MINOS selects the first set (if any). If you don't want the first set to be used, specify a dummy name such as `Ranges = NONE`.

`RHS`	a	Default = blank

This specifies the 8-character name of a right-hand side to be selected from the RHS section of the MPS file. If a is blank, MINOS selects the first RHS (if any). If you want the right-hand side to be zero, specify a dummy name such as `RHS = NONE` or `RHS = zero`.

`Rows` m Default = 100

This must specify an *over-estimate* of the number of rows in the ROWS section of the MPS file. It includes the number of nonlinear constraints and the number of general linear constraints. If m proves to be too small, MINOS continues in the manner described under `Columns`.

`Upper bound` u Default = `1.0e+20`

This specifies a default upper bound for all variables (excluding slacks). Individual variables may have their upper bound altered by a Bounds set in the MPS file.

3.3 OPTIONS FOR LINEAR PROGRAMMMING

The following options apply specifically to linear programs.

`Crash option` i Default = 3
`Crash tolerance` t Default = 0.1

Except on restarts, a Crash procedure is used to select an initial basis from certain rows and columns of the constraint matrix $(\,A\;\;I\,)$. The `Crash option` i determines which rows and columns of A are eligible initially, and how many times Crash is called. Columns of I are used to pad the basis where necessary.

$i = 0$ The initial basis contains only slack variables: $B = I$.

1 Crash is called once, looking for a triangular basis in all rows and columns of A.

2 Crash is called twice (if there are nonlinear constraints). The first call looks for a triangular basis in linear rows, and the first major iteration proceeds with simplex iterations until the linear constraints are satisfied. The Jacobian is then evaluated for the second major iteration and Crash is called again to find a triangular basis in the nonlinear rows (retaining the current basis for linear rows).

3 Crash is called up to three times (if there are nonlinear constraints). The first two calls treat *linear equalities* and *linear inequalities* separately. As before, the last call treats nonlinear rows at the start of the second major iteration.

If $i \geq 1$, certain slacks on inequality rows are selected for the basis first. (If $i \geq 2$, numerical values are used to exclude slacks that are close to a bound.) Crash then makes several passes through the columns of A, searching for a basis matrix that is essentially triangular. A column is assigned to "pivot" on a particular row if the column contains a suitably large element in a row that has not yet been assigned. (The pivot elements ultimately form the diagonals of the triangular basis.) For remaining unassigned rows, slack variables are inserted to complete the basis.

The `Crash tolerance` allows Crash to ignore certain "small" nonzeros in each column of A. If $a_{\max}$ is the largest element in column j, other nonzeros a_{ij} in the column are ignored if $|a_{ij}| \leq a_{\max} \times t$. (To be meaningful, t should be in the range $0 \leq t < 1$.)

When $t > 0.0$, the bases obtained by Crash may not be strictly triangular, but are likely to be nonsingular and almost triangular. The intention is to choose a basis containing more columns of A and fewer (arbitrary) slacks. A feasible solution may be reached sooner on some problems.

For example, suppose the first m columns of A are the matrix shown under `LU factor tolerance`; i.e., a tridiagonal matrix with entries -1, 4, -1. To help Crash choose all m columns for the initial basis, we would specify `Crash tolerance` t for some value of $t > 1/4$.

3.4 OPTIONS FOR ALL PROBLEMS

The following options have the same purpose for all problems, whether they linear or nonlinear.

`Check frequency` k Default = 60

Every k-th minor iteration after the most recent basis factorization, a numerical test is made to see if the current solution x satisfies the general linear constraints (including linearized nonlinear constraints, if any). The constraints are of the form $Ax + s = b$, where s is the set of slack variables. To perform the numerical test, the residual vector $r = b - Ax - s$ is computed. If the largest component of r is judged to be too large, the current basis is refactorized and the basic variables are recomputed to satisfy the general constraints more accurately.

`Check frequency 1` is useful for debugging purposes, but otherwise this option should not be needed.

`Cycle limit` l Default = 1
`Cycle print` p Default = 1
`Cycle tolerance` t Default = 0.0
`Phantom columns` c Default = 0
`Phantom elements` e Default = 0

`Debug level` l Default = 0

This causes various amounts of information to be output to the Print file. Most debug levels are not helpful to normal users, but they are listed here for completeness.

$l = 0$ No debug output.

$l = 2$ (or more) Output from `m5setx` showing the maximum residual after a row check.

$l = 40$ Output from `lu8rpc` (which updates the LU factors of the basis matrix), showing the position of the last nonzero in the transformed incoming column.

$l = 50$ Output from `lu1mar` (which computes the LU factors each refactorization), showing each pivot row and column and the dimensions of the dense matrix involved in the associated elimination.

$l = 100$ Output from `m2bfac` and `m5log` listing the basic and superbasic variables and their values at every iteration.

`Defaults`

This causes all MINOS options to be set to their default values. When `minoss` is in use, `call miopt('Defaults')` causes all MINOS options to be set to their default values.

`Expand frequency` k Default = 10000

This option is part of an anti-cycling procedure designed to guarantee progress even on highly degenerate problems. See [GMSW89].

"Cycling" can occur only if zero steplengths are allowed. Here, the strategy is to force a positive step at every iteration, at the expense of violating the bounds on the variables by a small amount.

Suppose that the `Feasibility tolerance` is δ. Over a period of k iterations, the tolerance actually used by MINOS increases from 0.5δ to δ (in steps of $0.5\delta/k$).

Every k iterations, or when feasibility and optimality are first detected, a resetting procedure eliminates any infeasible nonbasic variables. Some additional iterations may be needed to restore feasibility and optimality. Increasing k reduces that likelihood, but it gives less freedom to choose large pivot elements during basis changes. (See `Pivot tolerance`.)

`Factorization frequency` k Default = 100 (LP) or 50 (NLP)

With linear programs, most iterations cause a basis change, in which one column of the basis matrix B is replaced by another. The LU factors of B must be updated accordingly. At most k updates are performed before the current B is factorized directly.

Each update tends to add nonzeros to the LU factors. Since the updating method is stable, k mainly affects the efficiency of minor iterations, rather than stability.

High values of k (such as 100 or 200) may be more efficient on "dense" problems, when the factors of B tend to have two or three times as many nonzeros as B itself. Lower values of k may be more efficient on problems that are very sparse.

`Feasibility tolerance` t Default = `1.0e-6`

This sets the feasibility tolerance $\delta_{\rm fea} = t$ (see §3.3). A variable or constraint is considered *feasible* if it does not lie outside its bounds by more than $\delta_{\rm fea}$.

MINOS first attempts to satisfy the linear constraints and bounds. If the sum of infeasibilities cannot be reduced to zero, the problem is declared infeasible. Let *sinf* be the corresponding sum of infeasibilities. If *sinf* is quite small, it may be appropriate to raise t by a factor of 10 or 100. Otherwise, some error in the data should be suspected. If *sinf* is not small, there may be other points that have a significantly smaller sum of infeasibilities. MINOS does not attempt to find a solution that minimizes the sum.

For `Scale option` 1 or 2, feasibility is defined in terms of the *scaled* problem (since it is then more likely to be meaningful). The final unscaled solution can therefore be infeasible by an unpredictable amount, depending on the size of the scale factors. Precautions are taken so that in a "feasible solution" the original variables will never be infeasible by more than 0.1. Values that large are very unlikely.

`Iterations limit` k Default = $3m$

MINOS stops after k iterations even if the simplex method has not yet reached a solution. If $k = 0$, no iterations are performed, but the starting point is tested for both feasibility and optimality.

`LU factor tolerance` t_1 Default = 100.0 (LP) or 5.0 (NLP)
`LU update tolerance` t_2 Default = 10.0 (LP) or 5.0 (NLP)

These tolerances affect the stability and sparsity of the basis factorization $B = LU$ during refactorization and updating, respectively. They must satisfy t_1, $t_2 \ge 1.0$. The matrix L is a product of matrices of the form

$$\begin{pmatrix} 1 & \\ \mu & 1 \end{pmatrix},$$

where the multipliers μ satisfy $|\mu| \le t_i$. Values near 1.0 favor stability, while larger values favor sparsity. The default values usually strike a good compromise. For large and relatively dense problems, $t_1 = 10.0$ or 5.0 (say) may give a useful improvement in stability without impairing sparsity to a serious degree.

For certain very regular structures (e.g., band matrices) it may be necessary to reduce t_1 and/or t_2 in order to achieve stability. For example, if the columns of A include a submatrix of the form

$$\begin{pmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{pmatrix},$$

one should set both t_1 and t_2 to values in the range $1.0 \le t_i < 4.0$.

`LU density tolerance` t_3 Default $= 0.6$
`LU singularity tolerance` t_4 Default $= \epsilon^{2/3} \approx 10^{-11}$

The density tolerance t_3 is used during LU factorization of the basis matrix. Columns of L and rows of U are formed one at a time, and the remaining rows and columns of the basis are altered appropriately. At any stage, if the density of the remaining matrix exceeds t_3, the Markowitz strategy for choosing pivots is altered to reduce the time spent searching for each remaining pivot. Raising the density tolerance towards 1.0 may give slightly sparser LU factors, with a slight increase in factorization time.

The singularity tolerance t_4 helps guard against ill-conditioned basis matrices. When the basis is refactorized, the diagonal elements of U are tested as follows: if $|U_{jj}| \le t_4$ or $|U_{jj}| < t_4 \max_i |U_{ij}|$, the j-th column of the basis is replaced by the corresponding slack variable. (This is most likely to occur after a restart, or at the start of a major iteration.)

In some cases, the Jacobian matrix may converge to values that make the basis exactly singular. (For example, a whole row of the Jacobian could be zero at an optimal solution.) Before exact singularity occurs, the basis could become very ill-conditioned and the optimization could progress very slowly (if at all). Setting $t_4 =$ `1.0e-5`, say, may help cause a judicious change of basis.

`Maximize`
`Minimize` Default

This specifies the required direction of optimization.

`Multiple price` k Default $= 1$

It is not normal to set $k > 1$ for linear programs, as it causes MINOS to use the reduced-gradient method rather than the simplex method. The number of iterations, and the total work, are likely to increase.

The reduced-gradient iterations do *not* correspond to the very efficient multiple pricing "minor iterations" carried out by certain commercial linear programming systems. Such systems require storage for k dense vectors of dimension m, so that k is usually limited to 5 or 6. In MINOS, the total storage requirements increase only slightly with k. (The `Superbasics limit` must be at least k.)

`Optimality tolerance` t Default $=$ `1.0e-6`

This is used to judge the size of the reduced gradients $d_j = g_j - \pi^T a_j$, where g_j is the gradient of the objective function corresponding to the j-th variable, a_j is the associated column of the constraint matrix (or Jacobian), and π is the set of dual variables.

By construction, the reduced gradients for basic variables are always zero. Optimality is declared if the reduced gradients for nonbasic variables at their lower or upper bounds satisfy $d_j/\|\pi\| \geq -t$ or $d_j/\|\pi\| \leq t$ respectively, and if $|d_j|/\|\pi\| \leq t$ for superbasic variables.

In those tests, $\|\pi\|$ is a measure of the size of the dual variables. It is included to make the tests independent of a large scale factor on the objective function. The quantity actually used is defined by $\sigma = \sum_{i=1}^{m} |\pi_i|$, $\|\pi\| = \max\{\sigma/\sqrt{m}, 1.0\}$, so that only scale factors larger than 1.0 are allowed for. If the objective is scaled down to be very *small*, the optimality test reduces to comparing d_j against t.

`Partial price`	p	Default = 10 (LP) or 1 (NLP)

This parameter is recommended for large problems that have significantly more variables than constraints. It reduces the work required for each "pricing" operation (when a nonbasic variable is selected to become basic or superbasic).

When $p = 1$, all columns of the constraint matrix $(\,A \;\; I\,)$ are searched. Otherwise, A and I are partitioned to give p roughly equal segments A_j, I_j ($j = 1$ to p). If the previous pricing search was successful on A_j, I_j, the next search begins on the segments A_{j+1}, I_{j+1}. (Subscripts are modulo p.)

If a reduced gradient is found that is larger than some dynamic tolerance, the variable with the largest such reduced gradient (of appropriate sign) is selected to become superbasic. (Several may be selected if multiple pricing has been specified.) If nothing is found, the search continues on the next segments A_{j+2}, I_{j+2}, and so on.

`Partial price` t (or $t/2$ or $t/3$) may be appropriate for time-stage models having t time periods.

`Pivot tolerance`	t	Default = $\epsilon^{2/3} \approx 10^{-11}$

Broadly speaking, the pivot tolerance is used to prevent columns entering the basis if they would cause the basis to become almost singular. When x changes to $x + \alpha p$ for some search direction p, a "ratio test" is used to determine which component of x reaches an upper or lower bound first. The corresponding element of p is called the pivot element.

For linear problems, elements of p are ignored (and therefore cannot be pivot elements) if they are smaller than the pivot tolerance t. For nonlinear problems, elements smaller than $t\|p\|$ are ignored.

It is common for two or more variables to reach a bound at essentially the same time. In such cases, the `Feasibility tolerance` provides some freedom to maximize the pivot element and thereby improve numerical stability. Excessively small Feasibility tolerances should therefore not be specified.

To a lesser extent, the `Expand frequency` also provides some freedom to maximize the pivot element. Excessively *large* Expand frequencies should therefore not be specified.

`Scale option`	l	Default = 2 (LP) or 1 (NLP)
`Scale`	`Yes`	
`Scale`	`No`	
`Scale linear variables`		Same as `Scale option 1`
`Scale nonlinear variables`		Same as `Scale option 2`
`Scale all variables`		Same as `Scale option 2`
`Scale tolerance`	t	Default = 0.9
`Scale, Print`		
`Scale, Print, Tolerance`	t	

Three scale options are available as follows:

$l = 0$ No scaling. If storage is at a premium, this option saves $m + n$ words of workspace.

$l = 1$ Linear constraints and variables are scaled by an iterative procedure that attempts to make the matrix coefficients as close as possible to 1.0 (see Fourer, 1982). This sometimes improves the performance of the solution procedures.

$l = 2$ All constraints and variables are scaled by the iterative procedure. Also, a certain additional scaling is performed that may be helpful if the right-hand side b or the solution x is large. This takes into account columns of $(\,A \quad I\,)$ that are fixed or have positive lower bounds or negative upper bounds.

`Scale Yes` sets the default scaling. (*Caution*: If all variables are nonlinear, `Scale Yes` unexpectedly does *nothing*, because there are no linear variables to scale.) `Scale No` suppresses scaling (equivalent to `Scale option 0`).

If nonlinear constraints are present, `Scale option 1` or `0` should generally be tried at first. `Scale option 2` gives scales that depend on the initial Jacobian, and should therefore be used only if (a) a good starting point is provided, and (b) the problem is not highly nonlinear.

`Scale, Print` causes the row-scales $r(i)$ and column-scales $c(j)$ to be printed. The scaled matrix coefficients are $\bar{a}_{ij} = a_{ij}c(j)/r(i)$, and the scaled bounds on the variables and slacks are $\bar{l}_j = l_j/c(j)$, $\bar{u}_j = u_j/c(j)$, where $c(j) \equiv r(j-n)$ if $j > n$.

All forms except `Scale option` may specify a tolerance t, where $0 < t < 1$ (for example: `Scale, Print, Tolerance = 0.99`). This affects how many passes might be needed through the constraint matrix. On each pass, the scaling procedure computes the ratio of the largest and smallest nonzero coefficients in each column:

$$\rho_j = \max_i |a_{ij}| / \min_i |a_{ij}| \qquad (a_{ij} \neq 0).$$

If $\max_j \rho_j$ is less than t times its previous value, another scaling pass is performed to adjust the row and column scales. Raising t from 0.9 to 0.99 (say) usually increases the number of scaling passes through A. At most 10 passes are made.

If a `Scale option` has not already been specified, `Scale, Print` or `Scale tolerance` both set the default scaling.

`Weight on linear objective` w Default = 0.0

This keyword invokes the so-called *composite objective* technique, if the first solution obtained is infeasible, and if the objective function contains linear terms. While trying to reduce the sum of infeasibilities, the method also attempts to optimize the linear objective. At each infeasible iteration, the objective function is defined to be

$$\underset{x}{\text{minimize}} \quad \sigma w(c^T x) + (\text{sum of infeasibilities}),$$

where $\sigma = 1$ for minimization, $\sigma = -1$ for maximization, and c is the linear objective. If an "optimal" solution is reached while still infeasible, w is reduced by a factor of 10. This helps to allow for the possibility that the initial w is too large. It also provides dynamic allowance for the fact that the sum of infeasibilities is tending towards zero.

The effect of w is disabled after 5 such reductions, or if a feasible solution is obtained.

The `Weight` option is intended mainly for linear programs. It is unlikely to be helpful on nonlinear problems.

3.5 OPTIONS FOR NONLINEAR OBJECTIVES

The following options apply to nonlinear programs whose constraints are linear.

`Crash option` l Default = 3
`Crash tolerance` t Default = 0.1

These options are the same as for linear programs.

3.6 OPTIONS FOR ALL NONLINEAR PROBLEMS

`Expand frequency` k Default = 10000

This option is used the same as for linear programs, but takes effect only when there is just one superbasic variable. (Cycling can occur only when the current solution is at a vertex of the feasible region. Thus, zero steps are allowed if there is more than one superbasic variable, but otherwise positive steps are enforced.) Increasing k helps reduce the number of slightly infeasible nonbasic basic variables (most of which are eliminated during a resetting procedure). However, it also diminishes the freedom to choose a large pivot element (see `Pivot tolerance`).

`Feasibility tolerance` t Default = `1.0e-6`

When the constraints are linear, a *feasible solution* is one in which all variables, including slacks, satisfy their upper and lower bounds to within the absolute tolerance t. (Since slacks are included, this means that the general linear constraints are also satisfied to within t.)

When nonlinear constraints are present, a *feasible subproblem* is one in which the linear constraints and bounds, as well as the current linearization of the nonlinear constraints, are satisfied to within the tolerance t.

MINOS first attempts to satisfy the linear constraints and bounds. If the sum of infeasibilities cannot be reduced to zero, the problem is declared infeasible.

Normally, the nonlinear functions $F(x)$ and $f(x)$ are evaluated only at points x that satisfy the linear constraints and bounds. If the functions are undefined in certain regions, every attempt should be made to eliminate these regions from the problem. For example, for a function $F(x) = \sqrt{x_1} + \log x_2$, it would be essential to place lower bounds on both variables. If `Feasibility tolerance` $= 10^{-6}$, the bounds $x_1 \geq 10^{-5}$ and $x_2 \geq 10^{-4}$ might be appropriate. (The log singularity is more serious; in general, keep variables as far away from singularities as possible.)

An exception is during optional gradient checking (see `Verify`), which occurs before any optimization takes place. The points at which the functions are evaluated satisfy the bounds but not necessarily the general constraints. If this causes difficulty, gradient checking should be suppressed by setting `Verify level -1`.

If a subproblem is infeasible, the bounds on the linearized constraints are relaxed in several stages until the subproblem appears feasible. (The true bounds are restored for the next subproblem.) This approach sometimes allows the optimization to proceed successfully. In general, infeasible subproblems are a symptom of difficulty and it may be necessary to increase the `Penalty parameter` or alter the starting point.

Note: Feasibility with respect to the nonlinear constraints is measured against the `Row tolerance`, not the `Feasibility tolerance`.

`Hessian dimension` r Default = 50

This specifies that an $r \times r$ triangular matrix R is to be available for use by the quasi-Newton algorithm (to approximate the reduced Hessian matrix according to $Z^THZ \approx R^TR$). Suppose there are s superbasic variables at a particular iteration. *Whenever possible, r should be greater than s.*

If $r \geq s$, the first s columns of R are used to approximate the reduced Hessian in the normal manner. If there are no further changes to the set of superbasic variables, the rate of convergence is usually superlinear. If $r < s$, a matrix of the form

$$R = \begin{pmatrix} R_r & 0 \\ & D \end{pmatrix}$$

is used to approximate the reduced Hessian, where R_r is an $r \times r$ upper triangular matrix and D is a *diagonal* matrix of order $s - r$. The rate of convergence is no longer superlinear (and may be arbitrarily slow).

The storage required is of order $\frac{1}{2}r^2$, which is substantial if r is as large as 1000 (say). In general, r should be a slight over-estimate of the final number of superbasic variables, whenever storage permits. It need not be larger than $n_1 + 1$, where n_1 is the number of nonlinear variables. For many problems it can be much smaller than n_1.

`Iterations limit` k Default $= 3m + 10n_1$

If the constraints are linear, this is the maximum number of iterations allowed for the simplex method or the reduced-gradient method. Otherwise, it is the maximum number of *minor* iterations, summed over all major iterations.

If $k = 0$, no minor iterations are performed, but the starting point is tested for both feasibility and optimality.

`Linesearch tolerance` t Default $= 0.1$

For nonlinear problems, this controls the accuracy with which a steplength α is located during one-dimensional searches of the form

$$\underset{\alpha}{\text{minimize}} \quad F(x + \alpha p) \quad \text{subject to} \quad 0 < \alpha \leq \beta.$$

A linesearch occurs on most minor iterations for which x is feasible. (If the constraints are nonlinear, the function being minimized is the augmented Lagrangian in equation (5).)

The value of t must satisfy $0.0 \leq t < 1.0$. The default value $t = 0.1$ requests a moderately accurate search, and should be satisfactory in most cases. If the nonlinear functions are cheap to evaluate, a more accurate search may be appropriate; try $t = 0.01$ or $t = 0.001$. The number of iterations should decrease, and this will reduce total run time if there are many linear or nonlinear constraints. If the nonlinear functions are *expensive* to evaluate, a less accurate search may be appropriate; try $t = 0.5$ or perhaps $t = 0.9$. (The number of iterations will probably increase, but the total number of function evaluations may decrease enough to compensate.)

`LU singularity tolerance` t_3 Default $= \epsilon^{2/3} \approx 10^{-11}$
`LU swap tolerance` t_4 Default $= \epsilon^{1/4} \approx 10^{-4}$

The singularity tolerance t_3 helps guard against ill-conditioned basis matrices. When the basis is refactorized, the diagonal elements of U are tested as follows: if $|U_{jj}| \leq t_3$ or $|U_{jj}| < t_3 \max_i |U_{ij}|$, the j-th column of the basis is replaced by the corresponding slack variable. (This is most likely to occur after a restart, or at the start of a major iteration.)

In some cases, the Jacobian matrix may converge to values that make the basis exactly singular. (For example, a whole row of the Jacobian could be zero at an optimal solution.) Before exact singularity occurs, the basis could become very ill-conditioned and the optimization could progress very slowly (if at all). Setting $t_3 =$ `1.0e-5`, say, may help cause a judicious change of basis.

The `LU swap tolerance` is somewhat similar but can take effect more easily. It is again used only after a basis factorization, and normally just at the start of a major iteration. If a diagonal of U seems to be rather small (as measured by t_4) relative to the biggest diagonal of U, a basis change is made in which the basic variable associated with the small diagonal of U is swapped with a carefully chosen superbasic variable (if there are any). The number of superbasic variables stays the same. A message is printed to advise that a swap has occurred.

In practice this tends to help problems whose basis is becoming ill-conditioned. If the number of swaps becomes excessive, set `LU swap tolerance 1.0e-6`, say, or perhaps smaller.

`Minor damping parameter` d Default = 2.0

This parameter limits the change in x during a linesearch. It applies to all nonlinear problems, once a "feasible solution" or "feasible subproblem" has been found.

A linesearch of the form $\text{minimize}_\alpha F(x + \alpha p)$ is performed over the range $0 < \alpha \le \beta$, where β is the step to the nearest upper or lower bound on x. Normally, the first steplength tried is $\alpha_1 = \min(1, \beta)$, but in some cases, such as $F(x) = ae^{bx}$ or $F(x) = ax^b$, even a moderate change in the components of x can lead to floating-point overflow.

The parameter d is therefore used to define a limit $\bar{\beta} = d(1 + \|x\|)/\|p\|$, and the first evaluation of $F(x)$ is at the potentially smaller steplength $\alpha_1 = \min(1, \bar{\beta}, \beta)$.

Wherever possible, upper and lower bounds on x should be used to prevent evaluation of nonlinear functions at meaningless points. The `Minor damping parameter` provides an additional safeguard. The default value $d = 2.0$ should not affect progress on well behaved problems, but setting $d = 0.1$ or 0.01 may be helpful when rapidly varying functions are present. A "good" starting point may be required. An important application is to the class of nonlinear least-squares problems.

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

`Multiple price` k Default = 1

"Pricing" refers to a scan of the current nonbasic variables to determine if any should be changed from their current value (by allowing them to become superbasic or basic).

If multiple pricing is in effect, the k best nonbasic variables are selected for admission to the superbasic set. ("Best" means the variables with largest reduced gradients of appropriate sign.) If partial pricing is also in effect, the k best variables are selected from the current partition of A and I.

On large nonlinear problems it may help to set $k > 1$ if there are not many superbasic variables at the starting point but many at the optimal solution.

`Optimality tolerance` t Default = `1.0e-6`

`Partial price` p Default = 10 (LP) or 1 (NLP)

This parameter may be useful for large problems that have significantly more variables than constraints. Larger values reduce the work required for each "pricing" operation (when a nonbasic variable is selected to become basic or superbasic).

`Scale option`	l	Default = 2 (LP) or 1 (NLP)
`Scale`	`Yes`	
`Scale`	`No`	
`Scale linear variables`		Same as `Scale option 1`
`Scale nonlinear variables`		Same as `Scale option 2`
`Scale all variables`		Same as `Scale option 2`
`Scale tolerance`	t	Default = 0.9
`Scale, Print`		
`Scale, Print, Tolerance`	t	

Three scale options are available as follows:

- $l = 0$ No scaling. If storage is at a premium, this option saves $m + n$ words of workspace.
- $l = 1$ If some of the variables are linear, the constraints and linear variables are scaled by an iterative procedure that attempts to make the matrix coefficients as close as possible to 1.0.
- $l = 2$ The constraints and variables are scaled by the iterative procedure. Also, a certain additional scaling is performed that may be helpful if the right-hand side b or the solution x is large. This takes into account columns of $(\,A \quad I\,)$ that are fixed or have positive lower bounds or negative upper bounds.

`Scale option 1` is the default for nonlinear problems. (Only linear variables are scaled.)

`Scale Yes` sets the default. (*Caution*: If all variables are nonlinear, `Scale Yes` unexpectedly does *nothing*, because there are no linear variables to scale.) `Scale No` suppresses scaling (equivalent to `Scale option 0`).

The `Scale tolerance` and `Scale, Print` options are the same as for linear programs.

`Subspace tolerance`	t	Default = 0.5

This controls the extent to which optimization is confined to the current set of basic and superbasic variables (Phase 4 iterations), before one or more nonbasic variables are added to the superbasic set (Phase 3). The value specified must satisfy $0 < t \le 1$.

When a nonbasic variable x_j is made superbasic, the norm of the reduced-gradient vector (for all superbasics) is recorded. Let this be $\|Z^T g_0\|$. (In fact, the norm is $|d_j|$, the size of the reduced gradient for the new superbasic variable x_j.)

Subsequent Phase 4 iterations continue at least until the norm of the reduced-gradient vector satisfies $\|Z^T g\| \le t \times \|Z^T g_0\|$. ($\|Z^T g\|$ is the size of the largest reduced-gradient among the superbasic variables.)

A smaller value of t is likely to increase the total number of iterations, but may reduce the number of basis changes. A larger value such as $t = 0.9$ may sometimes lead to improved overall efficiency, if the number of superbasic variables is substantially larger at the optimal solution than at the starting point.

Other convergence tests on the change in the function being minimized and the change in the variables may prolong Phase 4 iterations. This helps to make the overall performance insensitive to larger values of t.

`Superbasics limit`	s	Default = 50

This places a limit on the storage allocated for superbasic variables. Ideally, s should be set slightly larger than the "number of degrees of freedom" expected at an optimal solution.

For nonlinear problems, the number of degrees of freedom is often called the "number of independent variables". Normally, s need not be greater than $n_1 + 1$, where n_1 is the number of nonlinear variables. For many problems, s may be considerably smaller than n_1. This saves storage if n_1 is very large.

If `Hessian dimension` r is specified, the default value of s is the same number (and conversely). This is a safeguard to ensure superlinear convergence wherever possible. Otherwise, the default for both r and s is 50.

`Unbounded objective value`	r_1	Default = `1.0e+20`
`Unbounded step size`	r_2	Default = `1.0e+10`

These parameters are intended to detect unboundedness in nonlinear problems. During a linesearch of the form $\min_\alpha F(x + \alpha p)$, if $|F|$ exceeds r_1 or if α exceeds r_2, iterations are terminated with the exit message `Problem is unbounded (or badly scaled)`.

If singularities are present, unboundedness in $F(x)$ may be manifested by a floating-point overflow (during the evaluation of $F(x + \alpha p)$), before the test against r_1 can be made.

Unboundedness in x is best avoided by placing finite upper and lower bounds on the variables. See also the `Minor damping parameter`.

`Verify level`	l	Default = 0
`Verify objective gradients`		Same as `Verify level 1`
`Verify constraint gradients`		Same as `Verify level 2`
`Verify`		Same as `Verify level 3`
`Verify gradients`		Same as `Verify level 3`
`Verify`	`Yes`	Same as `Verify level 3`
`Verify`	`No`	Same as `Verify level 0`

These options refer to a check on the gradients computed by your nonlinear function routines `funobj` and `funcon` at the starting point (the initial value of the nonlinear variables `x(*)`). Values output in the gradient array `g(*)` are compared with estimates obtained by finite differences.

$l = 0$ Only a "cheap" test is performed, requiring three evaluations of the nonlinear objective (if any) and two evaluations of the nonlinear constraints.

$l = 1$ A more reliable check is made on each component of the objective gradient.

$l = 2$ A check is made on each column of the Jacobian matrix associated with the nonlinear constraints.

$l = 3$ A detailed check is made on both the objective and the Jacobian.

$l = -1$ No checking is performed. This may be necessary if the functions are undefined at the starting point.

`Verify level 3` is recommended for a new function routine, particularly if the "cheap" test indicates error. Missing gradients are not checked (so there is no overhead). If there are many nonlinear variables, the `Start` and `Stop` keywords may be used to limit the check to a subset.

As noted, gradient verification occurs at the starting point, before a problem is scaled, and before the first basis is factorized. The bounds on x will be satisfied, but the general linear constraints may not. If the nonlinear objective or constraint functions are undefined, you could initially specify

```
Objective = NONE
Nonlinear objective variables   0
Major iterations                1
New basis file                 11  (say)
Verify level                   -1
```

to obtain a point that satisfies the linear constraints, and then restart with the correct linear and nonlinear objective, along with

```
Old basis file                 11
Verify level                    3
```

3.7 OPTIONS FOR NONLINEAR CONSTRAINTS

The following options apply to problems with nonlinear constraints.

`Completion`	`Partial`	Default
`Completion`	`Full`	

When there are nonlinear constraints, this determines whether subproblems should be solved to moderate accuracy (partial completion), or to full accuracy (full completion). MINOS implements the option by using two sets of convergence tolerances for the subproblems.

Use of partial completion may reduce the work during early major iterations, unless the `Minor iterations` limit is active. The optimal set of basic and superbasic variables will probably be determined for any given subproblem, but the reduced gradient may be larger than it would have been with full completion.

An automatic switch to full completion occurs when it appears that the sequence of major iterations is converging. The switch is made when the nonlinear constraint error is reduced below 100*(`Row tolerance`), the relative change in λ_k is 0.1 or less, and the previous subproblem was solved to optimality.

Full completion tends to give better Lagrange-multiplier estimates. It may lead to fewer major iterations, but may result in more minor iterations.

`Crash option`	l	Default = 3
`Crash tolerance`	t	Default = 0.1

Let A refer to the linearized constraint matrix.

$l = 0$ The initial basis contains only slack variables: $B = I$.

$l = 1$ A is evaluated at the starting point. Crash is called once, looking for a triangular basis in all rows and columns of A.

$l = 2$ A is evaluated only after the linear constraints are satisfied. Crash is called twice. The first call looks for a triangular basis in linear rows. The first major iteration proceeds with simplex-type iterations until the linear constraints are satisfied. A is then evaluated for the second major iteration and Crash is called again to find a triangular basis in the nonlinear rows (retaining the current basis for linear rows).

$l = 3$ Crash is called three times, treating *linear equalities* and *linear inequalities* separately, with simplex-type iterations in between. As before, the last call treats nonlinear rows at the start of the second major iteration.

`Feasibility tolerance` t Default = `1.0e-6`

A "feasible subproblem" is one in which the linear constraints and bounds, as well as the current linearization of the nonlinear constraints, are satisfied to within t.

Note that feasibility with respect to the nonlinear constraints is determined by the `Row tolerance` (not the `Feasibility tolerance`).

MINOS first attempts to satisfy the linear constraints and bounds. If the sum of infeasibilities cannot be reduced to zero, the problem is declared infeasible.

If `Scale option` = 1 or 2, feasibility is defined in terms of the scaled problem (since it is then more likely to be meaningful).

Normally, the nonlinear functions $F(x)$ and $f(x)$ are evaluated only at points x that satisfy the linear constraints and bounds. If the functions are undefined in certain regions, every attempt should be made to eliminate these regions from the problem. For example, for a function $F(x) = \sqrt{x_1} + \log x_2$, it would be essential to place lower bounds on both variables. If `Feasibility tolerance` $= 10^{-6}$, the bounds $x_1 \geq 10^{-5}$ and $x_2 \geq 10^{-4}$ might be appropriate. (The log singularity is more serious; in general, keep variables as far away from singularities as possible.)

An exception is during optional gradient checking (see `Verify`), which occurs before any optimization takes place. The points at which the functions are evaluated satisfy the bounds but not necessarily the general constraints. If this causes difficulty, gradient checking should be suppressed by setting `Verify level -1`.

If a subproblem is infeasible, the bounds on the linearized constraints are relaxed in several stages until the subproblem appears feasible. (The true bounds are restored for the next subproblem.) This approach sometimes allows the optimization to proceed successfully. In general, infeasible subproblems are a symptom of difficulty and it may be necessary to increase the `Penalty parameter` or alter the starting point.

`Jacobian` `Dense` Default
`Jacobian` `Sparse`

This is most important for stand-alone MINOS (and for subroutine `mirmps`) to ensure that the MPS file is interpreted correctly. See `Jacobian` in §3.2. For `minoss`, setting the correct value saves either time or storage.

`Lagrangian` `Yes` Default
`Lagrangian` `No`

This determines the form of the objective function used for the linearized subproblems. The default value `Yes` is highly recommended. The `Penalty parameter` value is then also relevant.

If `No` is specified, the nonlinear constraint functions are 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.

`Major damping parameter` d Default = 2.0

This parameter may assist convergence on problems that have highly nonlinear constraints. It is intended to prevent large relative changes between subproblem solutions (x_k, λ_k) and (x_{k+1}, λ_{k+1}). For example, the default value 2.0 prevents the relative change in either x_k or λ_k from exceeding 200 per cent. It will not be active on well behaved problems. (If all components of x_k or λ_k are small, the norms of those vectors will not be allowed to increase beyond about 2.0.)

The parameter is used to interpolate between the solutions at the beginning and end of each major iteration. Thus, x_{k+1} and λ_{k+1} are changed to

$$x_k + \sigma(x_{k+1} - x_k) \quad \text{and} \quad \lambda_k + \sigma(\lambda_{k+1} - \lambda_k)$$

for some steplength $\sigma < 1$. In the case of nonlinear equations (where the number of constraints is the same as the number of variables) this gives a *damped Newton method*.

This is a very crude control. If the sequence of major iterations does not appear to be converging, one should first re-run the problem with a higher `Penalty parameter` (say 2, 4 or 10). (Skip this re-run in the case of a square system of nonlinear equations: there are no degrees of freedom and the `Penalty parameter` value has essentially no effect.)

If the subproblem solutions continue to change violently, try reducing d to 0.2 or 0.1 (say).

`Major iterations` k Default = 50

This is the maximum number of major iterations allowed. It is intended to guard against an excessive number of linearizations of the nonlinear constraints, since in some cases the sequence of major iterations may not converge.

The progress of the major iterations can be best monitored using `Print level 0` (the default).

`Minor iterations` k Default = 40

This is the maximum number of minor iterations allowed during a major iteration, *after* the linearized constraints for that subproblem have been satisfied. (An arbitrary number of minor iterations may be needed to find a feasible point for each subproblem.) The `Iterations limit` provides an independent limit on the total minor iterations (across all subproblems).

A moderate value (e.g., $30 \le k \le 200$) prevents excessive effort being expended on 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 (if possible), ignoring the nonlinear constraints.

In general it is unsafe to specify a value as small as $k = 1$ or 2. (Even when an optimal solution has been reached, a few minor iterations may be needed for the corresponding subproblem to be recognized as optimal.)

`Optimality tolerance` t Default = `1.0e-6`

`Penalty parameter` r Default = 1.0

This specifies that the initial value of ρ_k in the augmented Lagrangian (5) should be r times a certain default value $100/m_1$, where m_1 is the number of nonlinear constraints. It is used when `Lagrangian = Yes` (the default setting).

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. In general, a positive r may be necessary to ensure convergence, *even for convex programs.*

On the other hand, if r is too large, the rate of convergence may be unnecessarily slow. If the functions are not highly nonlinear or a good starting point is known, it is often safe to specify `Penalty parameter 0.0`.

If several related problems are to be solved, the following strategy for setting the `Penalty parameter` may be useful:

1. Initially, use a moderate value for r (such as the default) and a reasonably low `Iterations` and/or `Major iterations` limit.
2. If successive major iterations appear to be terminate with radically different solutions, try increasing the penalty parameter. (See also the `Major damping parameter`.)

3. If there appears to be little progress between major iterations, it may help to reduce the penalty parameter.

`Radius of convergence` r Default = 0.01

This determines when the penalty parameter ρ_k is reduced (if initialized to a positive value). Both the nonlinear constraint violation (see *rowerr* below) and the relative change in consecutive Lagrange multipler estimates must be less than r at the start of a major iteration before ρ_k is reduced or set to zero.

A few major iterations later, full completion is requested if not already set, and the remaining sequence of major iterations should converge quadratically to an optimum.

`Row tolerance` t Default = `1.0e-6`

This specifies how accurately the nonlinear constraints should be satisfied at a solution. The default value is usually small enough, since model data is often specified to about that accuracy.

Let *viol* be the maximum violation of the nonlinear constraints (2), and let $rowerr = viol/(1 + xnorm)$, where *xnorm* is a measure of the size of the current solution (x, y). The solution is regarded as acceptably feasible if $rowerr \le t$.

If the problem functions involve data that is known to be of low accuracy, a larger `Row tolerance` may be appropriate. On the other hand, nonlinear constraints are often satisfied with rapidly increasing accuracy during the last few major iterations. It is common for the final solution to satisfy $rowerr = O(\epsilon)$.

`Scale option` l Default = 2 (LP) or 1 (NLP)
`Scale` `Yes`
`Scale` `No`

`Scale linear variables` Same as `Scale option 1`
`Scale nonlinear variables` Same as `Scale option 2`
`Scale all variables` Same as `Scale option 2`

`Scale tolerance` r Default = 0.9
`Scale, Print`
`Scale, Print, Tolerance` r

Three scale options are available as follows:

$l = 0$ No scaling. If storage is at a premium, this option saves $m + n$ words of workspace.

$l = 1$ Linear constraints and variables are scaled by an iterative procedure that attempts to make the matrix coefficients as close as possible to 1.0.

$l = 2$ All constraints and variables are scaled by the iterative procedure. Also, a certain additional scaling is performed that may be helpful if the right-hand side b or the solution x is large. This takes into account columns of $(\,A \quad I\,)$ that are fixed or have positive lower bounds or negative upper bounds.

`Scale option 1` is the default for nonlinear problems. (Only linear variables are scaled.)

`Scale Yes` sets the default scaling. *Caution*: If all variables are nonlinear, `Scale Yes` unexpectedly does nothing, because there are no linear variables to scale.) `Scale No` suppresses scaling (equivalent to `Scale option 0`).

With nonlinear constraints, `Scale option 1` or `0` should generally be tried first. `Scale option 2` gives scales that depend on the initial Jacobian, and should therefore be used only if (a) a good starting point is provided, and (b) the problem is not highly nonlinear.

`Verify level`	l	Default = 0
`Verify objective gradients`		Same as `Verify level 1`
`Verify constraint gradients`		Same as `Verify level 2`
`Verify`		Same as `Verify level 3`
`Verify gradients`		Same as `Verify level 3`
`Verify`	`Yes`	Same as `Verify level 3`
`Verify`	`No`	Same as `Verify level 0`

This option refers to a finite-difference check on the gradients (first derivatives) of each nonlinear function. It occurs before a problem is scaled, and before the first basis is factorized. (Hence, the variables may not yet satisfy the general linear constraints.)

- $l = 0$ Only a "cheap" test is performed, requiring three evaluations of the nonlinear objective (if any) and two evaluations of the nonlinear constraints.
- $l = 1$ A more reliable check is made on each component of the objective gradient.
- $l = 2$ A check is made on each column of the Jacobian matrix associated with the nonlinear constraints.
- $l = 3$ A detailed check is made on both the objective and the Jacobian.
- $l = -1$ No checking is performed. This may be necessary if the functions are undefined at the starting point.

3.8 OPTIONS FOR INPUT AND OUTPUT

The following options specify various files to be used, and the amount of printed output required.

`Print file`	f	Default = 9 (typically)
`Print level`	l	Default = 0
`Print frequency`	k	Default = 100

The PRINT file provides more complete information than the SUMMARY file. It includes a listing of the main options, statistics about the problem, scaling information, the iteration log, the exit condition, and (optionally) the final solution. It also includes error messages.

The default Print file is defined in your system-specific documentation. If f is specified, it must be a valid Fortran unit number in the range $0 \le f \le 99$. It should be different from the SUMMARY file. `Print file 0` suppresses output to the Print file. (This may be appropriate for repetitive optimizations.)

For problems with linear constraints, `Print level 0` gives most of the normal output. `Print level 1` produces statistics for the basis matrix B and its LU factors each time the basis is factorized. `Print frequency` k produces one line of the iteration log every k minor iterations. `Print frequency 1` causes every minor iteration to be logged. `Print frequency 0` is shorthand for $k = 99999$.

For problems with nonlinear constraints, `Print level 0` produces just one line of output per major iteration. This provides a short summary of the progress of the optimization. The `Print`

`frequency` is ignored. If `Print level` > 0, certain quantities are printed at the start of each major iteration, and minor iterations are logged according to the `Print frequency`.

In the latter case, the value of l is best thought of as a binary number of the form

```
Print level   JFLXB
```

where each letter stands for a digit that is either 0 or 1. The quantities referred to are:

`B` Basis statistics, as mentioned above.

`X` x_k, the nonlinear variables involved in the objective function or the constraints.

`L` λ_k, the Lagrange-multiplier estimates for the nonlinear constraints. (Suppressed if `Lagrangian = No`, since then $\lambda_k = 0$.)

`F` $f(x_k)$, the values of the nonlinear constraint functions.

`J` $J(x_k)$, the Jacobian matrix.

To obtain output of any item, set the corresponding digit to 1, otherwise to 0. For example, `Print level 10` sets `X` $= 1$ and the other digits equal to zero. The nonlinear variables will be printed each major iteration.

If `J` $= 1$, the Jacobian is output column-wise. Column j is preceded by the value of the corresponding variable x_j and a key to indicate whether the variable is basic, superbasic or nonbasic. (Hence if `J` $= 1$, there is no reason to specify `X` $= 1$ unless the objective contains more nonlinear variables than the Jacobian.) A typical line of output is

```
  3  1.250000D+01 BS       1  1.00000D+00       4  2.00000D+00
```

which means that x_3 is basic at value 12.5, and the third column of the Jacobian has elements of 1.0 and 2.0 in rows 1 and 4.

`Solution`	`Yes`	Default
`Solution`	`No`	
`Solution file`	f	

The `Yes` and `No` options control whether the final solution is output to the Print file. Numerical values are printed in `f16.5` format where possible. The special values 0, 1 and -1 are printed as `.`, `1.0` and `-1.0`. Bounds outside the range $(-10^{20}, 10^{20})$ appear as the word `None`.

The `file` option operates independently. If $f > 0$, the final solution is output to the Solution file, with numerical values in `1p,e16.5` format.

`Summary file`	f	Default = 6 (typically)
`Summary level`	l	Default = 0
`Summary frequency`	k	Default = 100

The Summary file provides a brief form of the iteration log and the exit condition. It also includes error messages. In an interactive environment, the output normally appears at the screen and allows a run to be monitored.

The default Summary file is defined in your system-specific documentation. If f is specified, it must be a valid Fortran unit number in the range $0 \le f \le 99$. It should be different from the Print file. `Summary file 0` suppresses summary output.

For problems with linear constraints, `Summary level 0` produces brief output. `Summary level 1` gives a few additional messages. `Summary frequency` k produces one line of the iteration log

every k minor iterations. `Summary frequency 1` causes every minor iteration to be logged. `Summary frequency 0` is shorthand for $k = 99999$.

For problems with nonlinear constraints, `Summary level 0` produces one line of output per major iteration. This provides a short summary of the progress of the optimization. The `Summary frequency` is ignored. If `Summary level` > 0, certain quantities are printed at the start of each major iteration, and minor iterations are logged according to the `Summary frequency`.

`Timing level` l Default = 2

$l = 0$ suppresses timing.

$l = 1$ times input, solve and output.

$l = 2$ times input, solve, output, `funcon` and `funobj`.

The values $l = -1$ and -2 are the same as 1 and 2, except the times are not printed at the end. If you are calling subroutine `minoss`, you may print the times in your own format by accessing the following common block:

```
parameter        ( ntime = 5 )
common    /m1tim / tlast(ntime), tsum(ntime), numt(ntime), ltime
```

where

`numt(`k`)` is the number of times clock k has been turned on.
`tlast(`k`)` is the time at which clock k was last turned on.
`tsum(`k`)` is the total time elapsed while clock k was on.
`ltime` is the `Timing level` l.

For $k = 1$ to 5, clock k times input, solve, output, `funcon` and `funobj` respectively. See subroutines `m1time` and `m1timp` for further details. For `Timing level 2`, MINOS and `minoss` both call `m1time` at the end of a run. This prints the "total time" statistics using a loop of the form

```
do k = 1, ntime
   call m1timp( k, 'Time', tsum(k) )
end do
```

Chapter 4

September 23, 2003

MPS Files

One way to input much of the data for a problem is via a file in the classical "MPS format" designed for mathematical programming systems of the 1960s and 70s. Stand-alone MINOS always uses MPS files. Alternatively, a driver program may call subroutine `mirmps` to read an MPS file and then pass the resulting data to subroutine `minoss`. The program could also call subroutine `miwmps` to output an MPS file describing the current problem.

4.1 MPS FILE HEADERS

For linear programs, an MPS file provides *all* of the problem data, including names for the variables and constraints. For nonlinear problems, an MPS file provides all data for linear constraints, as well as the sparsity pattern of the Jacobian (the gradients of nonlinear constraints).

In contrast to the free format allowed in the SPECS file, a very *fixed* format must be used for the MPS file. (Each item of data must appear in specific columns.) Various *headers* divide the MPS file into several sections as follows:

```
NAME
ROWS
  ..
COLUMNS
  ..
RHS
  ..
RANGES
  ..
BOUNDS
  ..
ENDATA
```

Each header must begin in column 1. The RANGES and BOUNDS sections are optional. The lines indicated by "`..`" all have the following format:

Columns	2–3	5–12	15–22	25–36	40–47	50–61
Contents	*Key*	*Name0*	*Name1*	*Value1*	*Name2*	*Value2*

Comment lines contain an asterisk (`*`) in column 1 followed by any characters.

MPS files may be created by hand, by your own special-purpose program, or by matrix generators such as GAMMA, MAGEN and OMNI. Other modeling languages and optimization systems such as GAMS and CPLEX can also output a model in MPS format.

Beware that variations are inevitable in almost any "standard" format. MINOS needs some minor extensions to allow for nonlinear problems.[1] Restrictions and extensions are listed in Section 4.9.

[1] The format used for problems in CUTE format [?] is a far greater generalization of MPS format for a large class of nonlinear problems.

4.2 THE NAME HEADER

```
NAME          MODEL001
```

This line contains the word `NAME` in columns 1–4 and a name for the problem in columns 15–22. (The name may be from 1 to 8 characters of any kind, or it may be blank.) The name is used to label the solution output, and it appears on the first line of each Basis file.

The NAME line is normally the first line in the MPS file, but it may be preceded or followed by comment lines.

4.3 THE ROWS SECTION

```
ROWS
 E  Fun01
 G  Fun02
 L  Capital1
 N  Cost
```

General constraints are commonly referred to as *rows*. The ROWS section contains one line for each constraint (i.e., for each row). *Key* defines what type the constraint is, and *Name0* gives the constraint an 8-character name. The various row-types are as follows:

Key	*Row-type*
`E`	$=$
`G`	$\geq$
`L`	$\leq$
`N`	Objective
`N`	Free

The 1-character *Key* may be in column 2 or column 3.

Row-types `E`, `G` and `L` are easily understood in terms of a linear function $a^T x$ and a right-hand side β. They are used to specify constraints of the form

$$a^T x = \beta, \qquad a^T x \geq \beta \qquad \text{and} \qquad a^T x \leq \beta$$

respectively. Nonzero elements of the row-vector a are entered in the COLUMNS section, and if β is nonzero it is entered in the RHS section.

Row-type `N` stands for "Not binding", also known as "Free". It is used to define the *objective row*, and also to prevent a constraint from actually being a constraint. (Note that $-\infty \leq a^T x \leq +\infty$ is not really a constraint at all. Type `N` rows are implemented by giving them infinite bounds of this kind.)

The *objective row* is a free row that specifies the vectors c and d in the objective function $F(x) + c^T x + d^T y$. It is taken to be the *first free row*, unless some other free row is specified by the `Objective` keyword in the Specs file.

The ROWS section need not contain any free rows if $c = d = 0$. If there are some nonlinear objective variables, the objective function will then be $F(x)$ as defined by subroutine `FUNOBJ`. Otherwise, *no* objective function exists and MINOS will terminate at the first point that satisfies the constraints.

If the ROWS section does contain free rows but none of them is intended to be an objective row, then some dummy name such as `OBJECTIVE = NONE` should be specified in the Specs file to prevent the first free row from being selected. (If the objective function is $F(x)$ with no linear terms, `Objective = funobj` would be a mnemonic reminder.)

Row-names for Nonlinear Constraints

The names of nonlinear constraints must be listed *first* in the ROWS section, and their order must be consistent with the computation of the array `f(*)` in subroutine `funcon`.

In particular, the objective row (if any) must appear after the list of nonlinear row names. For simplicity we suggest that potential objective rows be placed last:

```
ROWS
 G   Fun01          nonlinear constraints first
 G   Fun02
 .
 E   Lin01          now linear constraints
 E   Lin02
 .
 N   Cost01         objective rows last
 N   Cost02
```

4.4 THE COLUMNS SECTION

```
1   5.....12  15....22  25........36  40....47  50........61

COLUMNS
    x01       Fun06          1.0      Row09         -3.0
    x01       Row08          2.5      Row12          1.123456
    x01       Row03      -11.111111
    x02       Fun02          1.0
    x02       Cost01         5.0
```

For each variable x_j, the COLUMNS section defines a name for x_j and lists the nonzero entries a_{ij} in the corresponding column of the constraint matrix. The nonzeros for the first column must be grouped together before those for the second column, and so on. If a column has several nonzeros, it does not matter what order they appear in (as long as they all appear before the next column).

In general, *Key* is blank (except for comments), *Name0* is the column name, and *Name1*, *Value1* give a row name and value for some coefficient in that column. If there is another row name and value for the same column, they may appear as *Name2*, *Value2* on the same line, or they may be on the next line.

If either *Name1* or *Name2* is blank, the corresponding value is ignored.

Values are input using Fortran format `bn, e12.0`. This allows values to be entered in several forms; for example, `1.2345678`, `1.2345678e+0`, `123.45678e-2` and `12345678e-07` all represent the same number. It is usually best to include an explicit decimal point or an `e` (or `d`). Note that the `bn` format treats blanks (spaces) as null (not 0), so that entries such as `1e-2` do not have to be right-justified in their field.

In the above example, variable `x01` has 5 nonzero coefficients in the constraints named `Fun06`, `Row09`, `Row08`, `Row12` and `Row03`. The row names and values may be in an arbitrary order, but they must all appear before the entries for column `x02`.

There is no need to specify columns for the slack variables; they are incorporated implicitly.

Nonlinear Variables

Nonlinear variables must appear *first* in the COLUMNS section, ordered in a manner that is consistent with the array `x(*)` in the user subroutines `funobj`and/or `funcon`. In the example

$$\text{minimize} \qquad (x + y + z)^2 + 3z + 5w \tag{1}$$

$$\begin{aligned} \text{subject to} \quad & x^2 + y^2 + z = 2 && (2)\\ & x^4 + y^4 + w = 4 && (3)\\ & 2x + 4y \ge 0 && (4)\\ & z \ge 0, \quad w \ge 0 && (5) \end{aligned}$$

we have three nonlinear objective variables (x, y, z), two nonlinear Jacobian variables (x, y), one linear variable w, two nonlinear constraints, one linear constraint, and some simple bounds. The nonlinear constraints and variables should always be ordered in a similar way, at the *top left-hand corner* of the constraint matrix. The latter is therefore of the form

$$A = \begin{pmatrix} J_k & A_1 \\ A_2 & A_3 \end{pmatrix}$$

where J_k is the Jacobian matrix. The variables associated with J_k and A_2 must appear *first* in the COLUMNS section, and their order must be consistent with the array `x(*)` in subroutine `funcon`. Similarly, entries belonging to J_k must appear in an order that is consistent with the array `g(*)` in subroutine `funcon`.

For convenience, let the first n_1 columns of A be

$$\begin{pmatrix} J_k \\ A_2 \end{pmatrix} = \begin{pmatrix} j_1 & j_2 & \dots & j_{n_1} \\ a_1 & a_2 & \dots & a_{n_1} \end{pmatrix},$$

where j_1 is the first column of J_k and a_1 is the first column of A_2. The coefficients of j_1 and a_1 must appear before the coefficients of j_2 and a_2 (and so on for all columns). Usually, those belonging to j_1 will appear before any in a_1, but this is not essential. (If certain linear constraints are made nonlinear at a later date, this means that entries in the COLUMNS section will not have to be reordered. However, the corresponding row names will need be moved towards the top of the ROWS section.)

If `Jacobian = Dense`, the elements of J_k need not be specified in the MPS file. If `Jacobian = Sparse`, *all* nonzero elements of J_k must be specified. Any variable coefficients should be given a dummy value, such as zero. These dummy entries identify the location of the elements; their actual values are computed later by subroutine `funcon` or by finite differences.

If all constraint gradients are known (`Derivative level = 2` or `3`), any Jacobian elements that are constant may be given their correct values in the COLUMNS section, and then they need not be reset by subroutine `funcon`. This includes values that are identically zero—such elements do not have to be specified anywhere (in the MPS file or in `funcon`). In other words, Jacobian elements are assumed to be zero unless otherwise specified.

Note that `x(*)` need not have the same dimension in subroutines `funobj` and `funcon` (i.e., the parameter `n` may differ), in the event that different numbers are specified by the `Nonlinear objective` and `Nonlinear Jacobian` keywords. However the shorter set of nonlinear variables must occur at the beginning of the longer set, and the ordering of variables in the COLUMNS section must match both sets.

A nonlinear objective function will often involve variables that occur only linearly in the constraints. In such cases we recommend that the objective variables be placed *after* the Jacobian variables in the COLUMNS section, since the Jacobian will then be as small as possible. (See the variable z in the example above.)

4.5 THE RHS SECTION

```
1   5.....12  15....22  25........36  40....47  50........61
```

```
RHS
    RHS01     Fun01          1.0       Row09           -3.0
    RHS01     Row08          2.5       Row12         1.123456
    RHS01     Row03       -11.111111
    RHS02     Fun02          1.0
    RHS02     Fun04          5.0
```

This section specifies the elements of b_1 and b_2 in (2)–(3). Together these vectors comprise what is called the right-hand side. Only the nonzero coefficients need to be specified. They may appear in any order. The format is exactly the same as in the COLUMNS section, with *Name0* giving a name to the right-hand side.

If $b_1 = 0$ and $b_2 = 0$, the `RHS` header line must appear as usual, but no rhs coefficients need follow.

The RHS section may contain more than one right-hand side. The *first* one will be used unless some other name is specified in the SPECS file.

4.6 THE RANGES SECTION (OPTIONAL)

```
1   5.....12  15....22  25........36    40....47  50........61

ROWS
 E  Fun01
 E  Fun02
 G  Capital1
 L  Capital2
 .
COLUMNS
 .
RHS
    RHS01     Fun01          4.0       Fun02            4.0
 .
RANGES
    RANGE01   Fun01          1.0       Fun02           -1.0
    RANGE01   Capital1       1.0       Capital2         1.0
```

Ranges are used for constraints of the form

$$l \le a^T x \le u,$$

where both l and u are finite. The range of the constraint is $r = u - l$. Either l or u is specified in the RHS section (as b say), and r is defined in the RANGES section. The resulting l and u depend on the row-type of the constraint and the sign of r as follows:

Row-type	*Sign of r*	*Lower limit, l*	*Upper limit, u*
E	$+$	b	$b + \lvert r\rvert$
E	$-$	$b - \lvert r\rvert$	b
G	$+$ or $-$	b	$b + \lvert r\rvert$
L	$+$ or $-$	$b - \lvert r\rvert$	b

The format is exactly the same as in the COLUMNS section, with *Name0* giving a name to the

range set. The constraints listed above will have the following limits:

$$\begin{aligned} 4.0 \le \texttt{Fun01} &\le 5.0, \\ 3.0 \le \texttt{Fun02} &\le 4.0, \\ 4.0 \le \texttt{Capital1} &\le 5.0, \\ 3.0 \le \texttt{Capital2} &\le 4.0. \end{aligned}$$

The RANGES section may contain more than one set of ranges. The *first* set will be used unless some other name is specified in the SPECS file.

4.7 THE BOUNDS SECTION (OPTIONAL)

```
1   5.....12   15....22   25........36

BOUNDS
 UP BOUND01    x01             4.0
 UP BOUND01    x02             4.0
 .
 LO BOUND01    x04            -1.0
 UP BOUND01    x04             4.0
 .
 FR BOUND01    x06
 UP BOUND01    x06             4.0
 .
```

The default bounds on all variables x_j (excluding slacks) are $0 \le x_j \le \infty$. If necessary, the default values 0 and ∞ can be changed in the SPECS file to $l \le x_j \le u$ by the `Lower` and `Upper` keywords respectively.

If uniform bounds of this kind are not suitable, any number of alternative values may be specified in the BOUNDS section. As usual, several sets of bounds may be given, and the first set will be used unless some other name is specified in the SPECS file.

In this section, *Key* gives the type of bound required, *Name0* is the name of the bound set, and *Name1* and *Value1* are the column name and bound value. (*Name2* and *Value2* are ignored.)

Let l and u be the default bounds just mentioned, and let x and b be the column and value specified. The various bound-types allowed are as follows:

Key	*Bound-type*	*Resultingbounds*			
`LO`	Lower bound	b	$\le x \le$	u	
`UP`	Upper bound	l	$\le x \le$	b	
`FX`	Fixed variable	b	$\le x \le$	b	(i.e., $x = b$)
`FR`	Free variable	$-\infty$	$\le x \le$	$+\infty$	
`MI`	Minus infinity	$-\infty$	$\le x \le$	u	
`PL`	Plus infinity	l	$\le x \le$	$+\infty$	

The effect of the examples above is to give the following bounds:

$$\begin{aligned} l &\le \texttt{x01} \le 4.0 \\ l &\le \texttt{x02} \le 4.0 \\ -1.0 &\le \texttt{x04} \le 4.0 \\ -\infty &\le \texttt{x06} \le 4.0 \end{aligned}$$

Note that types `FR`, `MI`, or `PL` should always be used to specify "infinite" bounds; they imply values of $\pm 10^{20}$, which are treated specially.

Nonlinear Problems

Bounds are often needed to avoid singularities in the nonlinear functions. For example, if the functions involve $\log x_j$, a bound of the form $x_j \geq 10^{-4}$ or $x_j \geq 10^{-5}$ is generally necessary to avoid evaluating the log at zero or negative values of x_j.

The bounds must take into account the `Feasibility tolerance` t, whose default value is 10^{-6}. Subroutines `funobj` and `funcon` are not called until the linear constraints and bounds are satisfied to within the specified tolerance. Thus, it would not be safe to specify the bound $x_j \geq 10^{-7}$ unless the feasibility tolerance were reduced to $t = 10^{-8}$ (say).

The `INITIAL` *Bounds Set*

In general, variables will initially have the value zero, if zero lies between the associated upper and lower bounds. Otherwise, the initial value will be the bound closest to zero.

The name `INITIAL` is reserved for a special bounds set that may be used to assign other initial values. If an `INITIAL` bounds set is used, it must appear after any normal bound sets. A warning is given if it is the first set encountered after the `BOUNDS` line.

The `INITIAL` bounds set also influences the CRASH procedure for constructing an initial basis (unless a basis file is provided). Broadly speaking, CRASH favors certain variables, ignores certain others, and treats the remainder as neutral. The following example illustrates the various cases:

```
FR INITIAL   x1            1.0
FX INITIAL   x2            2.0
LO INITIAL   x3
UP INITIAL   x4
MI INITIAL   x5            5.0
PL INITIAL   x6            6.0
```

1. `x1` will then be favored by CRASH for inclusion in the initial basis. Free rows (type `N`) and free columns (type `FR` in a normal bounds set) will also be favored. The initial value of `x1` will be 1.0.

2. If possible, `x2` will initially be superbasic at the value 2.0. If the number of `FX INITIAL`s has already reached the `Superbasics limit`, `x2` will initially be nonbasic at the same value 2.0.

3. `x3` and `x4` will initially be nonbasic at their respective lower and upper bounds (or at value zero if those bounds are infinite).

4. `x5` and `x6` will initially be nonbasic at the specified values 5.0 and 6.0.

The last five bound types (`FX`, `LO`, `UP`, `MI`, `PL`) prevent the associated variables from being included in the initial basis.

`FR INITIAL` or `FX INITIAL` should be used if good values are known for variables that are likely to lie between their bounds in an optimal solution. Type `FR` is preferred if many such values are to be specified; however, the values may change when the basic variables are reset to satisfy $Ax + Is = 0$. Type `FX` guarantees the specified starting value, but should not be used excessively if the optimal solution is likely to be close to a vertex.

`LO INITIAL` or `UP INITIAL` should be used for variables that are likely to be on their lower or upper bound at a solution.

`MI INITIAL` and `PL INITIAL` are included for completeness.

As with normal bound sets, variables may be listed in any order. (For each entry a linear search is made through the column names, starting at the name on the previous entry. Thus, for large problems it helps to follow the order of the variables in the COLUMNS section, at least to some extent.)

The `INITIAL` bounds set is ignored if a basis file is supplied.

4.8 COMMENTS

Any line in the MPS file may contain an asterisk "*" in column 1 and arbitrary data in columns 2–72. Such lines are treated as comments. They appear in the printed listing but are otherwise ignored.

4.9 RESTRICTIONS AND EXTENSIONS IN MPS FORMAT

1. Fixed format is used.
2. Names cannot be longer than 8 characters.
3. Blanks are significant in the 8-character name fields. We recommend that all names be left-justified with no imbedded blanks. In particular, names referred to in the SPECS file *must* be left-justified in the MPS file. For example, `Objective = Cost02` specifies an 8-character name whose last two characters are blank.
4. Scale factors cannot be entered in the ROWS section.
5. It does not matter if there is no row of type `N`.
6. There must be at least one row in the ROWS section, even for problems with no general constraints. (It may have row-type `N`.)
7. Nonlinear constraints must appear before linear constraints in the ROWS section.
8. Markers of the form

   ```
   'MARKER'  'INTORG'
   'MARKER'  'INTEND'
   ```

   may appear in the COLUMNS section, but they will not cause intervening variables to take integer values.
9. Subroutine `mirmps` returns an integer array of indicators `hint(*)` to record the intention of the above markers.
10. Numerical values may be entered in Fortran's `e` or `f` format. Spaces within the 12-character fields are treated as if they were absent.
11. Nonlinear variables must appear before linear variables in the COLUMNS section.
12. If RANGES and BOUNDS sections are both present, the RANGES section must appear first.
13. In the BOUNDS section, if an `UP` entry specifies a zero upper bound, the corresponding lower bound is *not* affected. (Beware—in some MP systems, the lower bound is converted to $-\infty$.)
14. The ranges name `LAGRANGE` has a special meaning.
15. The bounds name `INITIAL` has a special meaning.

Chapter 5

September 23, 2003

Basis Files

Basis files are used to record full or partial information about the variables and the constraints. They may be saved at the end of a run in order to restart the run if necessary, or to provide a good starting point for some closely related problem.

Three formats are available for saving basis information. They are invoked by options of the form

```
New Basis file      10
Backup    file      11
Punch     file      20
Dump      file      30
```

New Basis and Backup files have the same format. They are saved every k-th iteration, in that order, where k is the `Save frequency`. The file numbers may be anything convenient in the range 1 to 99, or 0 for files that are not wanted (the default value).

New Basis, Punch and Dump files are saved at the end of a run, in that order. They may be re-loaded at the start of a later run via options of the form

```
Old basis file      10
Insert    file      20
Load      file      30
```

Only one such file will actually be loaded. If more than one positive file number is specified, the order of precedence is as shown. If no basis files are specified, one of the `Crash options` takes effect.

Figures 5.1–5.3 illustrate the data formats used for basis files. 80-character fixed-length records are suitable in all cases. (36-character records would be adequate for Punch and Dump files.) The files shown correspond to the optimal solution for the Economic Growth model described in Section 8.4. Selected column numbers are included to define significant data fields. The model has 10 nonlinear constraints, 10 linear constraints, and 30 variables.

5.1 NEW AND OLD BASIS FILES

These files contain the most compact representation of the state of each variable and constraint. They are intended for restarting the solution of a problem at a point that was reached by an earlier run on the *same problem* or a related problem with the *same dimensions*. (Perhaps the `Iterations limit` was previously too small, or some bounds have been altered.)

As illustrated in Figure 5.1, the following information is recorded in New Basis file.

1. The problem name, the iteration number when the file was created, the status of the solution (`OPTIMAL SOLN`, `INFEASIBLE`, `UNBOUNDED`, `EXCESS ITNS`, `ERROR CONDN`, or `PROCEEDING`), the number of infeasibilities, and the current objective value (or the sum of infeasibilities).

2. The `OBJECTIVE`, `RHS`, `RANGES` and `BOUNDS` names, `M` $= m$, the number of rows in the constraint matrix, `N` $= n$, the number of columns in the constraint matrix, and `SB` $= nS$, the number of superbasic variables.

```
.........1.........2.........3.........4.........5.........6.........7.........8

MANNE10   ITN      17    OPTIMAL SOLN  NINF      0       OBJ   2.670098627239E+00
OBJ=FUNOBJ   RHS=         RNG=RANGE1   BND=BOUND1   M=      20 N=      30 SB=    7
03333333330323333333323322222111111111110000000000
       1    3.05000000000000E+00  0
       2    3.12665036215043E+00  3
       3    3.21443007884476E+00  3
       4    3.30400426867706E+00  3
       5    3.39521996012654E+00  3
       6    3.48787828288392E+00  3
       7    3.58172349003054E+00  3
       8    3.67642941254574E+00  3
       9    3.77158333557364E+00  3
      10    3.86666666666667E+00  3
      11    9.50000000000000E-01  0
      12    9.68418313307234E-01  3
      13    9.97801045227854E-01  2
      14    1.02820030935737E+00  3
      15    1.05967005676750E+00  3
      16    1.09227178532333E+00  3
      17    1.12607611310996E+00  3
      18    1.16116410684441E+00  3
      19    1.19762896604086E+00  3
      20    1.21394308356457E+00  3
      21    7.66503621504308E-02  3
      22    8.77797166943282E-02  2
      23    8.95741898323014E-02  3
      24    9.12156914494767E-02  3
      25    9.26583227573872E-02  2
      26    9.38452071466108E-02  2
      27    9.47059225152014E-02  2
      28    9.51539230279001E-02  2
      29    9.50833310930291E-02  2
       0
```

Figure 5.1: Format of NEW and OLD BASIS files

3. A set of $(n + m - 1)/80 + 1$ lines indicating the state of the n variables x and the m slack variables, in that order. One character `hs`(j) is recorded for each $j = 1{:}n + m$ as follows, written with `format(80i1)`.

`hs`(j)	*State of the j-th variable*
0	Nonbasic at lower bound
1	Nonbasic at upper bound
2	Superbasic
3	Basic

   If variable j is *fixed* (lower bound = upper bound), then `hs`(j) may be 0 or 1. The same is true if variable j is *free* (infinite bounds) and still nonbasic, although free variables will almost always be basic.

4. A set of lines of the form

$$j \qquad x_j \qquad hs_j$$

written with `format(i8, 1p, e24.14, i3)` and terminated by an entry with $j = 0$, where j denotes the j-th variable and x_j is a real value. The j-th variable is either the j-th column or the $(j - n)$-th slack, if $j > n$. This list is normally empty for linear problems (unless there happen to be some superbasic variables). For nonlinear problems, all basic, superbasic and nonliner variables are listed in their natural order. The value hs_j matches the corresponding digit in the `hs(`j`)` lines, as a helpful reminder. (It is not used if the data is reloaded as an OLD BASIS file.)

Loading a New Basis file

A file that has been saved as an OLD BASIS file may be input at the beginning of a later run as a NEW BASIS file.

1. The first line is input and printed but otherwise not used.
2. The values labeled `M` and `N` on the second line must agree with m and n, the dimensions of the current problem. The value labeled `SB` is input and printed but is not used.
3. The next set of lines must contain $n + m$ entries `hs(`j`)` = `0`, `1`, `2` or `3`, including exactly m values `3`, denoting the basic variables.
4. Normally the list of j and x_j values will include an entry for every variable whose state is `hs(`j`)` = 2 (the superbasic variables) in the preceding lines. The hs_j value is not input or used.
5. Further (j, x_j, hs_j) entries may be included in any order. Again the hs_j value is not used.

5.2 PUNCH AND INSERT FILES

These files provide compatibility with commercial mathematical programming systems. The PUNCH file from a previous run may be used as an INSERT file for a later run on the same problem. It may also be possible to modify the INSERT file and/or problem and still obtain a useful advanced basis.

The MPS format has been slightly generalized to allow the saving and reloading of nonbasic solutions. It is illustrated in Figure 5.2. Apart from the first and last line, each entry has the following form:

Columns	2–3	5–12	15–22	25–36
Contents	*Key*	*Name1*	*Name2*	*Value*

The various keys are best defined in terms of the action they cause on input. It is assumed that the basis is initially set to be the full set of slack variables, and that column variables are initially at their smallest bound in absolute magnitude.

Key	*Action to be taken during* INSERT
`XL`	Make variable *Name1* basic and slack *Name2* nonbasic at its lower bound.
`XU`	Make variable *Name1* basic and slack *Name2* nonbasic at its upper bound.
`LL`	Make variable *Name1* nonbasic at its lower bound.
`UL`	Make variable *Name1* nonbasic at its upper bound.
`SB`	Make variable *Name1* superbasic at the specified *Value*.

Note that *Name1* may be a column name or a row name, but (on `XL` and `XU` lines) *Name2* must be a row name. In all cases, row names indicate the associated slack variable, and if *Name1* is a nonlinear variable, its *Value* is recorded for possible use in defining the initial Jacobian matrix.

The key `SB` is an addition to the standard MPS format to allow for nonbasic solutions.

```
.........1.........2.........3.........4

NAME          MANNE10   PUNCH/INSERT
 LL KAP001               3.05000E+00
 XU KAP002    MON001     3.12665E+00
 XU KAP003    MON002     3.21443E+00
 XU KAP004    MON003     3.30400E+00
 XU KAP005    MON004     3.39522E+00
 XU KAP006    MON005     3.48788E+00
 XU KAP007    MON006     3.58172E+00
 XU KAP008    MON007     3.67643E+00
 XU KAP009    MON008     3.77158E+00
 XU KAP010    MON009     3.86667E+00
 LL CON001               9.50000E-01
 XU CON002    MON010     9.68418E-01
 SB CON003               9.97801E-01
 XL CON004    CAP002     1.02820E+00
 XL CON005    CAP003     1.05967E+00
 XL CON006    CAP004     1.09227E+00
 XL CON007    CAP005     1.12608E+00
 XL CON008    CAP006     1.16116E+00
 XL CON009    CAP007     1.19763E+00
 XL CON010    CAP008     1.21394E+00
 XL INV001    CAP009     7.66504E-02
 SB INV002               8.77797E-02
 XL INV003    CAP010     8.95742E-02
 XL INV004    TERMINV    9.12157E-02
 SB INV005               9.26583E-02
 SB INV006               9.38452E-02
 SB INV007               9.47059E-02
 SB INV008               9.51539E-02
 SB INV009               9.50833E-02
 UL INV010               1.16000E-01
ENDATA
```

Figure 5.2: Format of PUNCH and INSERT files

```
.........1.........2.........3.........4

NAME          MANNE10      DUMP/LOAD
 LL KAP001               3.05000E+00
 BS KAP002               3.12665E+00
 BS KAP003               3.21443E+00
 BS KAP004               3.30400E+00
 BS KAP005               3.39522E+00
 BS KAP006               3.48788E+00
 BS KAP007               3.58172E+00
 BS KAP008               3.67643E+00
 BS KAP009               3.77158E+00
 BS KAP010               3.86667E+00
 LL CON001               9.50000E-01
 BS CON002               9.68418E-01
 SB CON003               9.97801E-01
 BS CON004               1.02820E+00
 BS CON005               1.05967E+00
 BS CON006               1.09227E+00
 BS CON007               1.12608E+00
 BS CON008               1.16116E+00
 BS CON009               1.19763E+00
 BS CON010               1.21394E+00
 BS INV001               7.66504E-02
 SB INV002               8.77797E-02
 BS INV003               8.95742E-02
 BS INV004               9.12157E-02
 SB INV005               9.26583E-02
 SB INV006               9.38452E-02
 SB INV007               9.47059E-02
 SB INV008               9.51539E-02
 SB INV009               9.50833E-02
 UL INV010               1.16000E-01
 UL MON001               0.00000E+00
 UL MON002               0.00000E+00
 UL MON003               0.00000E+00
 UL MON004               0.00000E+00
 UL MON005               0.00000E+00
 UL MON006               0.00000E+00
 UL MON007               0.00000E+00
 UL MON008               0.00000E+00
 UL MON009               0.00000E+00
 UL MON010               0.00000E+00
 LL CAP002               0.00000E+00
 LL CAP003               0.00000E+00
 LL CAP004               0.00000E+00
 LL CAP005               0.00000E+00
 LL CAP006               0.00000E+00
 LL CAP007               0.00000E+00
 LL CAP008               0.00000E+00
 LL CAP009               0.00000E+00
 LL CAP010               0.00000E+00
 LL TERMINV              0.00000E+00
ENDATA
```

Figure 5.3: Format of DUMP and LOAD files

Notes on PUNCH Data

1. Variables are output in natural order. For example, on the first `XL` or `XU` line, *Name1* will be the first basic column and *Name2* will be the first row whose slack is not basic. (The slack could be nonbasic or superbasic.)

2. `LL` lines are *not* output for nonbasic variables if the corresponding lower bound value is zero.

3. Superbasic slacks are output last.

4. PUNCH and INSERT files deal with the status and values of *slack variables*. This is in contrast to the printed solution and the SOLUTION file, which deal with *rows*.

Notes on INSERT Data

1. Before an INSERT file is read, column variables are made nonbasic at their smallest bound in absolute magnitude, and the slack variables are made basic.

2. Preferably an INSERT file should be an unmodified PUNCH file from an earlier run on the same problem. If some rows have been added to the problem, the INSERT file need not be altered. (The slacks for the new rows will be in the basis.)

3. Entries will be ignored if *Name1* is already basic or superbasic. `XL` and `XU` lines will be ignored if *Name2* is not basic.

4. `SB` lines may be added before the `ENDATA` line, to specify additional superbasic columns or slacks.

5. An `SB` line will not alter the status of *Name1* if the `Superbasics limit` has been reached. However, the associated *Value* will be retained if *Name1* is a Jacobian variable.

5.3 DUMP AND LOAD FILES

These files are similar to PUNCH and INSERT files, but they record solution information in a manner that is more direct and more easily modified. In particular, no distinction is made between columns and slacks. Apart from the first and last line, each entry has the form

Columns	2–3	5–12	25–36
Contents	*Key*	*Name*	*Value*

as illustrated in Figure 5.3. The keys `LL`, `UL`, `BS` and `SB` mean Lower Limit, Upper Limit, Basic and Superbasic respectively.

Notes on DUMP Data

1. A line is output for every variable, columns followed by slacks.

2. Nonbasic free variables will be output with either `LL` or `UL` keys and with *Value* zero.

Notes on LOAD Data

1. Before a LOAD file is read, all columns and slacks are made nonbasic at their smallest bound in absolute magnitude. The basis is initially empty.

2. Each `LL`, `UL` or `BS` line causes *Name* to adopt the specified status. The associated *Value* will be retained if *Name* is a Jacobian variable.

3. An `SB` line causes *Name* to become superbasic at the specified *Value*.

4. An entry will be ignored if *Name* is already basic or superbasic. (Thus, only the first `BS` or `SB` line takes effect for any given *Name*.)

5. An `SB` line will not alter the status of *Name* if the `Superbasics limit` has been reached, but the associated *Value* will be retained if *Name* is a Jacobian variable.

6. (*Partial basis*) If fewer than m variables are specified to be basic, a tentative basis list will be constructed by adding the requisite number of slacks, starting from the first row and taking those that were not previously specified to be basic or superbasic. (If the resulting basis proves to be singular, the basis factorization routine will replace a number of basic variables by other slacks.) The starting point obtained in this way will not necessarily be "good".

7. (*Too many basics*) If m variables have already been specified as basic, any further `BS` keys will be treated as though they were `SB`. This feature may be useful for combining solutions to smaller problems.

5.4 RESTARTING MODIFIED PROBLEMS

Any of the above three starting methods (Old Basis, Insert and Load files) may be preferable to the cold start (Crash) options. The best choice depends on the extent to which a problem has been modified, and whether it is more convenient to specify variables by number or by name. The following notes offer some rules of thumb.

Protection

In general there is no danger of specifying infinite values. For example, if a variable is specified to be nonbasic at an upper bound that happens to be $+\infty$, it will be made nonbasic at its lower bound. Conversely if its lower bound is $-\infty$. If the variable is *free* (both bounds infinite), it will be made nonbasic at value zero. No warning message is issued.

Default Status

If the status of a variable is not explicitly given, it will initially be nonbasic at the bound that is smallest in absolute magnitude. Ties are broken in favor of lower bounds, and free variables will again take the value zero.

Restarting with Different Bounds

Suppose that a problem is to be restarted after the bounds on some variable `x` have been altered. Any of the basis files may be used, but the starting point obtained depends on the status of `x` at the time the basis is saved.

If `x` is basic or superbasic, the starting point will be the same as before (unless some nonbasic variables also have their bounds altered). If `x` is basic, its initial value may lie outside the new bounds.

If `x` was previously *fixed*, it is likely to be nonbasic at its *lower* bound. Increasing its upper bound will not affect the solution.

In contrast, if `x` was nonbasic at its *upper* bound and that bound is *reduced*, the starting values for an arbitrary number of basic variables could be changed (since they will be recomputed from the nonbasic and superbasic variables). This may not be of great consequence, but sometimes it may be worthwhile to retain the old solution precisely. To do this, one must make `x` superbasic at the original bound value.

For example, if `x` was nonbasic at an upper bound of 5.0 (which has now been changed), one should insert a line of the form

```
j                    5.0
```

near the end of an OLD BASIS file, or the line

```
SB x                    5.0
```

near the end of an INSERT or LOAD file. Note that the SPECS file must specify a `Superbasics limit` at least as large as the number of variables involved, even for purely linear problems.

Sequences of Problems

Whenever practical, a series of related problems should be ordered so that the *most tightly constrained* cases are solved first. Their solutions will often provide feasible starting points for subsequent relaxed problems, as long the above precautions are taken.

Altering Bounds with the `CYCLE` Option

Sequences of problems will sometimes be defined in conjunction with the `Cycle` facilities. Various alterations can be made to each problem from within your own subroutine `matmod`. In particular, it is straightforward to alter the bounds on any of the columns or slacks.

Chapter 6

September 23, 2003

Output

Subroutine `mistart` specifies unit numbers for the PRINT and SUMMARY files described in this section. The files can be redirected (or suppressed) via the `Print file` and `Summary file` options.

6.1 THE PRINT FILE

The following information is output to the PRINT file during the solution process. The longest line of output is 124?? characters.

- A listing of the SPECS file, if any.
- The selected options.
- An estimate of the storage needed and the amount available.
- Some statistics about the problem data.
- The storage available for the LU factors of the basis matrix.
- A log from the scaling procedure, if `Scale option` > 0.
- Notes about the initial basis obtained from CRASH or a BASIS file.
- The major iteration log.
- The minor iteration log.
- Basis factorization statistics.
- The EXIT condition and some statistics about the solution obtained.
- The printed solution, if requested.

The last five items are described in the following sections.

6.1.1 The major iteration log

Problems with nonlinear constraints require several *major iterations* to reach a solution, each involving the solution of an *LC subproblem* (a linearly constrained subproblem that generates search directions for x and λ). If `Print level` $= 0$, one line of information is output to the PRINT file each major iteration. Problem `t4manne` gives the log shown in Figure 6.1.

```
Major minor   total ninf step     objective     Feasible Optimal  nsb   ncon     LU penalty BSwap
    1     1T      1    0 0.0E+00  0.00000000E+00 0.0E+00 1.2E+01    8      4     31 1.0E-01     0
    2    13      14    0 1.0E+00  2.67011596E+00 4.4E-06 2.8E-03    7     23     56 1.0E-01     8
Completion Full    now requested
    3     3      17    0 1.0E+00  2.67009870E+00 3.1E-08 1.4E-06    7     29     41 1.0E-01     0
    4     0      17    0 1.0E+00  2.67009863E+00 5.6E-17 1.4E-06    7     30     41 1.0E-02     0
```

Figure 6.1: The Major Iteration log

Label	*Description*
`Major`	The current major iteration number.
`minor`	is the number of iterations required by both the feasibility and optimality phases of the QP subproblem. Generally, `Mnr` will be 1 in the later iterations, since theoretical analysis predicts that the correct active set will be identified near the solution (see §**??**).
`total`	The total number of minor iterations.
`ninf`	The number of infeasibilities in the LC subproblem. Normally 0, because the bounds on the linearized constraints are relaxed in several stages until the constraints are "feasible".
`step`	The step length α taken along the current search direction p. The variables x have just been changed to $x + \alpha p$. On reasonably well-behaved problems, `step` $= 1.0$ as the solution is approached, meaning the new estimate of (x, λ) is the solution of the LC subproblem.
`objective`	The value of true objective function.
`Feasible`	The value of `rowerr`, the maximum component of the scaled nonlinear constraint residual (**??**). The solution is regarded as acceptably feasible if `Feasbl` is less than the `Row tolerance`.
`Optimal`	The value of `maxgap`, the maximum complementarity gap (**??**). It is an estimate of the degree of nonoptimality of the reduced costs. Both `Feasible` and `Optimal` are small in the neighborhood of a solution.
`nsb`	The current number of superbasic variables.
`ncon`	The number of times subroutine `funcon` has been called to evaluate the nonlinear constraint functions. The Jacobian has been evaluated or approximated essentially the same number of times. (Function evaluations needed to estimate the Jacobian by finite differences are not included.)
`LU`	The number of nonzeros in the sparse LU factors of the basis matrix on completion of the LC subproblem. (The factors are computed at the start of each major iteration, and updated during minor iterations whenever a basis change occurs.) As the solution is approached and the minor iterations decrease towards zero, `LU` reflects the number of nonzeros in the LU factors at the start of the LC subproblem.
`penalty`	The penalty parameter ρ_k used in the modified augmented Lagrangian that defines the objective function for the LC subproblem.
`BSwap`	The number of columns of the basis matrix B that were swapped with columns of S to improve the condition of B. The swaps are determined by an LU factorization of the rectangular matrix $B_S = (\,B \;\; S\,)^T$ with stability being favored more than sparsity.

```
   Itn ph pp     rg    +sbs  -sbs   -bs  step    pivot   ninf  sinf,objective     L     U ncp  nobj  ncon  nsb Hmod cond(H) conv
     1  1  1 -1.0E+00     2     2     1 3.0E+01  1.0E+00    1  1.35000000E+02     0    19   0
     2  1  1 -1.0E+00    27    27   102 7.0E+01  1.0E+00    1  1.05000000E+02     0    19   0
     3  1  1 -1.0E+00     3     3    27 3.0E+01 -1.0E+00    1  3.50000000E+01     1    19   0
     4  1  1 -1.0E+00    28    28    26 4.9E-11  1.0E+00    1  5.00000000E+00     1    20   0
     5  1  1 -1.0E+00    47    47     2 4.9E-11  1.0E+00    1  5.00000000E+00     1    20   0
     6  1  1  1.0E+00    27    27   101 5.0E+00 -1.0E+00    1  5.00000000E+00     2    20   0

Itn      6 -- feasible solution.  Objective =  -1.818044887E+02

     7  3  1 -1.7E+01    87     0     0 1.0E+00  0.0E+00    0 -2.77020571E+02     4    21   0     6     0    1  1 0 1.0E+00 FFTT
     8  3  1 -1.7E+01    72     0     0 1.9E-01  0.0E+00    0 -3.05336895E+02     4    21   0     8     0    2  1 0 5.5E+00 FFTT
     9  3  1 -2.3E+01    41     0     0 1.0E+00  0.0E+00    0 -4.43743832E+02     4    21   0     9     0    3  1 0 6.5E+00 FFFF
    10  4  1  6.6E-01     0     0     0 6.0E+00  0.0E+00    0 -5.64075338E+02     4    21   0    11     0    3  1 0 3.5E+00 FFTT
...

   Itn ph pp     rg    +sbs  -sbs   -bs  step    pivot   ninf  sinf,objective     L     U ncp  nobj  ncon  nsb Hmod cond(H) conv
   161  4  1  8.8E-03     0    73    71 4.2E+00  1.0E+00    0 -1.73532497E+03     4    20   0   340     0   17  1 1 9.6E+00 TTTF
   162  3  1 -3.5E-02     6     0     0 1.5E+00  0.0E+00    0 -1.73533264E+03     4    20   0   342     0   18  1 0 1.3E+02 TTFF
   163  4  1  2.9E-02     0     0     0 4.5E+00  0.0E+00    0 -1.73533617E+03     4    20   0   344     0   18  1 0 2.0E+01 TTFF
   164  4  1  2.1E-02     0     0     0 2.3E+01  0.0E+00    0 -1.73538331E+03     4    20   0   347     0   18  1 0 9.8E+00 TTFF
   165  4  1  3.0E-02     0     0     0 5.0E+00  0.0E+00    0 -1.73552261E+03     4    20   0   349     0   18  1 0 2.1E+01 TTFF
   166  4  1  1.2E-02     0     0     0 1.0E+00  0.0E+00    0 -1.73556089E+03     4    20   0   350     0   18  1 0 2.2E+01 TTTF
tolrg  reduced to  1.162E-03      lvltol = 1
   167  4  1  2.3E-03     0     0     0 1.0E+00  0.0E+00    0 -1.73556922E+03     4    20   0   351     0   18  1 0 2.2E+01 TTFF
   168  4  1  1.2E-03     0     0     0 7.9E-01  0.0E+00    0 -1.73556953E+03     4    20   0   353     0   18  1 0 2.1E+01 TTFF
   169  4  1  1.0E-04     0     0     0 1.0E+00  0.0E+00    0 -1.73556958E+03     4    20   0   354     0   18  1 0 2.0E+01 TTTT
tolrg  reduced to  1.013E-05      lvltol = 1
   170  4  1  2.9E-05     0     0     0 1.1E+00  0.0E+00    0 -1.73556958E+03     4    20   0   356     0   18  1 0 1.7E+01 TTFF
   171  4  1  1.0E-05     0     0     0 1.0E+00  0.0E+00    0 -1.73556958E+03     4    20   0   357     0   18  1 0 1.7E+01 TTFF
   172  4  1  1.5E-06     0     0     0 1.2E+00  0.0E+00    0 -1.73556958E+03     4    20   0   359     0   18  1 0 1.7E+01 TTTF
tolrg  reduced to  1.000E-06      lvltol = 2
   173  4  1  2.4E-07     0     0     0 1.0E+00  0.0E+00    0 -1.73556958E+03     4    20   0   360     0   18  1 0 1.7E+01 TTTF

Biggest dj =  3.583E-03 (variable     25)   norm rg =  2.402E-07   norm pi =  1.000E+00
```

Figure 6.2: The Minor Iteration log

6.1.2 The minor iteration log

If `Print level` ≥ 1, one line of information is output to the PRINT file every kth minor iteration, where k is the specified `Print frequency` (default $k =$ `100`). A heading is printed periodically. Problem `t5weapon` gives the log shown in Figure 6.2.

Label *Description*

`Itn` The current minor iteration number.

`ph` The current phase of the solution procedure:

1. Phase 1 simplex method, trying to satisfy the linear constraints. The current solution is an infeasible vertex.
2. Phase 2 simplex method, solving a linear program.
3. Reduced-gradient method. A nonbasic variable has just become superbasic.
4. Reduced-gradient method, optimizing the current set of superbasic variables.

`pp` The Partial Price indicator. The variable selected by the last PRICE operation came from the `pp`th partition of A and I. `pp` is set to zero when the basis is refactored.

A PRICE operation is defined to be the process by which a nonbasic variable is selected to become a new superbasic. The selected variable is denoted by `jq`. Variable `jq` often becomes basic immediately. Otherwise it remains superbasic, unless it reaches its opposite bound and becomes nonbasic again. If `Partial price` is in effect, variable `jq` is selected from $A_{\tt pp}$ or $I_{\tt pp}$, the `pp`th segments of the constraint matrix $(\,A \;\; I\,)$.

`rg` In Phase 1, 2 or 3, this is d_j, the reduced cost (reduced gradient) of the variable `jq` selected by PRICE at the start of the present iteration. Algebraically, $d_j = g_j - \pi^T a_j$ for $j =$ `jq`, where g_j is the gradient of the current objective function, π is the vector of dual variables for the problem (or LC subproblem), and a_j is the jth column of the current $(A \quad I)$.

In Phase 4, `rg` is the largest reduced gradient among the superbasic variables.

`+sbs` The variable `jq` selected by PRICE to be added to the superbasic set.

`-sbs` The variable chosen to leave the set of superbasics. It has become basic if the entry under `-bs` is nonzero; otherwise it has become nonbasic.

`-bs` The variable removed from the basis (if any) to become nonbasic.

`step` The step length α taken along the current search direction p. The variables x have just been changed to $x + \alpha p$.

`pivot` If column a_q replaces the rth column of the basis B, `pivot` is the rth element of a vector y satisfying $By = a_q$. Wherever possible, `step` is chosen to avoid extremely small values of `pivot` (because they cause the basis to be nearly singular). In rare cases, it may be necessary to increase the `Pivot tolerance` to exclude very small elements of y from consideration during the computation of `step`.

`ninf` The number of infeasibilities *before* the present iteration. This number decreases monotonically.

`sinf,objective` If `ninf` > 0, this is `sinf`, the sum of infeasibilities before the present iteration. It usually decreases at each nonzero `step`, but if `ninf` decreases by 2 or more, `sinf` may occasionally increase.

Otherwise it is the value of the current objective function *after* the present iteration. For linear programs, it means the true linear objective function. For problems with linear constraints, it means the sum of the linear objective and the value returned by subroutine `funobj`. For problems with nonlinear constraints, it is the quantity just described if `Lagrangian = No`; otherwise it is the value of the augmented Lagrangian for the current major iterations (which tends to the true objective as convergence is approached).

`L` The number of nonzeros representing the basis factor L. Immediately after a basis factorization $B = LU$, this is `lenL`, the number of subdiagonal elements in the columns of a lower triangular matrix. Further nonzeros are added to `L` when various columns of B are later replaced. (Thus, `L` increases monotonically.)

`U` The number of nonzeros in the basis factor U. Immediately after a basis factorization, this is `lenU`, the number of diagonal and superdiagonal elements in the rows of an upper-triangular matrix. As columns of B are replaced, the matrix U is maintained explicitly (in sparse form). The value of `U` may fluctuate up or down; in general it will tend to increase.

`ncp` The number of compressions required to recover storage in the data structure for U. This includes the number of compressions needed during the previous basis factorization. Normally `ncp` should increase very slowly. If not, the amount of workspace available to MINOS should be increased by a significant amount. As a suggestion, the work array `z(*)` should be extended by $2($`L`$+$`U`$)$ elements.

The following items are printed if the problem is nonlinear or if the superbasic set is non-empty (i.e., if the current solution is not a vertex).

Label	*Description*
`nobj`	The number of times subroutine `funobj` has been called.
`ncon`	The number of times subroutine `funcon` has been called.
`nsb`	The current number of superbasic variables.

`Hmod` An indication of the type of modifications made to the triangular matrix R that is used to approximate the reduced Hessian matrix. Two integers i_1 and i_2 are shown. They will remain zero for linear problems. If $i_1 = 1$, a BFGS quasi-Newton update has been made to R, to account for a move within the current subspace. (This will not occur if the solution is infeasible.) If $i_2 = 1$, R has been modified to account for a change in basis. This will sometimes occur even if the solution is infeasible (if a feasible point was obtained at some earlier stage).

Both updates are implemented by triangularizing the matrix $R+vw^T$ for some vectors v and w. If an update fails for numerical reasons, i_1 or i_2 will be set to 2, and the resulting R will be nearly singular. (However, this is highly unlikely.)

`cond(H)` An estimate of the condition number of the reduced Hessian. It is the square of the ratio of the largest and smallest diagonals of the upper triangular matrix R—a lower bound on the condition number of the matrix $R^T R$ that approximates the reduced Hessian. `cond(H)` gives a rough indication of whether or not the optimization procedure is having difficulty. The reduced-gradient algorithm will make slow progress if `cond(H)` becomes as large as 10^8, and will probably fail to find a better solution if `cond(H)` reaches 10^{12} or more.

To guard against high values of `cond(H)`, attention should be given to the scaling of the variables and the constraints. In some cases it may be necessary to add upper or lower bounds to certain variables to keep them a reasonable distance from singularities in the nonlinear functions or their derivatives.

`conv` A set of four logical variables C_1, C_2, C_3, C_4 that are used to determine when to discontinue optimization in the current subspace (Phase 4) and consider releasing a nonbasic variable from its bound (the PRICE operation of Phase 3). Let `rg` be the norm of the reduced gradient, as described above. The meaning of the variables C_j is as follows:

C_1 is `true` if the change in x was sufficiently small;
C_2 is `true` if the change in the objective was sufficiently small;
C_3 is `true` if `rg` is smaller than some loose tolerance `TOLRG`;
C_4 is `true` if `rg` is smaller than some tighter tolerance.

The test used is of the form

if (C_1 *and* C_2 *and* C_3) *or* C_4 *then go to Phase 3.*

At present, `tolrg` $= t|$`dj`$|$, where t is the `Subspace tolerance` (nominally 0.5) and `dj` is the reduced-gradient norm at the most recent Phase 3 iteration. The "tighter tolerance" is the maximum of 0.1 `tolrg` and $10^{-7}\|\pi\|$. Only the tolerance t can be altered at run-time.

6.1.3 Crash statistics

The following items are output to the PRINT file when `start` = `'Cold'` and no basis file is loaded. They refer to the number of columns that the CRASH procedure selects during several passes through A while searching for a triangular basis matrix.

Label	*Description*
`Slacks`	is the number of slacks selected initially.
`Free cols`	is the number of free columns in the basis, including those whose bounds are rather far apart.
`Preferred`	is the number of "preferred" columns in the basis (i.e., `hs`$(j) = 3$ for some $j \leq n$). It will be a subset of the columns for which `hs`$(j) = 3$ was specified.
`Unit`	is the number of unit columns in the basis.
`Double`	is the number of columns in the basis containing 2 nonzeros.
`Triangle`	is the number of triangular columns in the basis with 3 or more nonzeros.
`Pad`	is the number of slacks used to pad the basis (to make it a nonsingular triangle).

6.1.4 Basis factorization statistics

If `Print level` ≥ 1, the following items are output to the PRINT file whenever the basis B or the rectangular matrix $B_S = (\, B \;\; S\,)^T$ is factorized. Note that B_S may be factorized at the start of just some of the major iterations. It is immediately followed by a factorization of B itself.

Gaussian elimination is used to compute a sparse LU factorization of B or B_S, where PLP^T and PUQ are lower and upper triangular matrices for some permutation matrices P and Q. Stability is ensured as described under `LU factor tolerance` in §**??**.

Label	*Description*
`Factorize`	The number of factorizations since the start of the run.
`Demand`	A code giving the reason for the present factorization.
`Itn`	The current iteration number.
`Nonlin`	The number of nonlinear variables in the current basis B.
`Linear`	The number of linear variables in B.
`Slacks`	The number of slack variables in B.
`m`	The number of rows in the matrix being factorized (B or B_S).
`n`	The number of columns in the matrix being factorized. Preceded by "=" if the matrix is B; by "¿" if it is B_S.
`Elems`	The number of nonzero elements in B or B_S.
`Amax`	The largest nonzero in B or B_S.
`Density`	The density of the matrix (percentage of nonzeros).

`Merit` The average Markowitz merit count for the elements chosen to be the diagonals of PUQ. Each merit count is defined to be $(c-1)(r-1)$ where c and r are the number of nonzeros in the column and row containing the element at the time it is selected to be the next diagonal. `Merit` is the average of `m` such quantities. It gives an indication of how much work was required to preserve sparsity during the factorization.

`lenL` The number of nonzeros in the factor L.

`L+U` The number of nonzeros in both L and U.

`Cmprssns` The number of times the data structure holding the partially factored matrix needed to be compressed to recover unused storage. Ideally this number should be zero. If it is more than 3 or 4, the amount of workspace available to MINOS should be increased for efficiency.

`Incres` The percentage increase in the number of nonzeros in L and U relative to the number of nonzeros in B or B_S.

`Utri` The size of the "backward triangle" in B or B_S. These top rows of U come directly from the matrix.

`lenU` The number of nonzeros in the factor U.

`Ltol` The maximum allowed size of nonzeros in L. Usually equal to the `LU factor tolerance`.

`Umax` The maximum nonzero in U.

`Ugrwth` The ratio Umax / Amax.

`Ltri` The size of the "forward triangle" in B or B_S. These initial columns of L come directly from the matrix.

`dense1` is the number of columns remaining when the density of the basis matrix being factorized reached 0.3.

`Lmax` The maximum nonzero in L (no larger than `Ltol`).

`Akmax` The maximum nonzero arising during the factorizaiton. (Printed only if Theshold Complete Pivoting is in effect.)

`Agrwth` The ratio Akmax / Amax. (Printed only if Theshold Complete Pivoting is in effect.)

`bump` The number of columns of B or B_S excluding `Utri` and `Ltri`.

`dense2` The number of columns remaining when the density of the basis matrix being factorized reached 0.6.

`DUmax` The largest diagonal of U (really PUQ).

`DUmin` The smallest diagonal of U.

`condU` The ratio `DUmax/DUmin`. As long as `Ltol` is not large (say 10.0 or less), `condU` is an estimate of the condition number of B. If this number is extremely large, the basis is nearly singular and some numerical difficulties might occur. (However, an effort is made to avoid near-singularity by using slacks to replace columns of B that would have made `Umin` extremely small. Messages are issued to this effect, and the modified basis is refactored.)

6.1.5 EXIT conditions

When the solution procedure terminates, an `EXIT --` message is printed to summarize the final result. Here we describe each message and suggest possible courses of action.

The number associated with each EXIT is the output value of the integer variable `inform`.

> The following messages arise when the SPECS file is found to contain no further problems.

```
-2  EXIT -- input error.   MINOS encountered end-of-file or an
    endrun card before finding a SPECS file on unit  nn
```

The SPECS file may not be properly assigned. Its unit number `nn` is defined at compile time in subroutine `snInit`, and normally it is the system card input stream.

Otherwise, the SPECS file may be empty, or cards containing the keywords `Skip` or `Endrun` may imply that all problems should be ignored (see §**??**).

```
-1  ENDRUN
```

This message is printed at the end of a run if MINOS terminates of its own accord. Otherwise, the operating system will have intervened for one of many possible reasons (excess time, missing file, arithmetic error in user routines, etc.).

> The following messages arise when a solution exists (though it may not be optimal). A BASIS file may be saved, and the solution will be output to the PRINT or SOLUTION files if requested.

```
 0  EXIT -- optimal solution found
```

This is the message we all hope to see! It is certainly preferable to every other message, and we naturally want to believe what it says, because this is surely one situation where *the computer knows best*. There may be cause for celebration if the objective function has reached an astonishing new high (or low).

In all cases, a distinct level of caution is in order, even if it can wait until next morning. For example, if the objective value *is* much better than expected, we may have obtained an optimal solution to the wrong problem! Almost any item of data could have that effect if it has the wrong value. Verifying that the problem has been defined correctly is one of the more difficult tasks for a model builder. It is good practice in the function subroutines to print any data that is input during the first entry.

If nonlinearities exist, one must always ask the question: could there be more than one local optimum? When the constraints are linear and the objective is known to be convex (e.g., a sum of squares) then all will be well if we are *minimizing* the objective: a local minimum is a global minimum in the sense that no other point has a lower function value. (However, many points could have the *same* objective value, particularly if the objective is largely linear.) Conversely, if we are *maximizing* a convex function, a local maximum cannot be expected to be global, unless there are sufficient constraints to confine the feasible region.

Similar statements could be made about nonlinear constraints defining convex or concave regions. However, the functions of a problem are more likely to be neither convex nor concave. Always specify a good starting point if possible, especially for nonlinear variables, and include reasonable upper and lower bounds on the variables to confine the solution to the specific region of interest. We expect modelers to *know something about their problem*, and to make use of that knowledge as well as they can.

One other caution about "`Optimal solution`"s. Some of the variables or slacks may lie outside their bounds more than desired, especially if scaling was requested. `Max Primal infeas` refers to the largest bound infeasibility and which variable (or slack) is involved. If it is too large, consider

restarting with a smaller `Feasibility tolerance` (say 10 times smaller) and perhaps `Scale option 0`.

Similarly, `Max Dual infeas` indicates which variable is most likely to be at a non-optimal value. Broadly speaking, if

$$\texttt{Max Dual infeas/Norm of pi} = 10^{-d},$$

then the objective function would probably change in the dth significant digit if optimization could be continued. If d seems too large, consider restarting with smaller `Optimality tolerance`s.

Finally, `Nonlinear constraint violn` shows the maximum infeasibility for nonlinear rows. If it seems too large, consider restarting with a smaller `Row tolerance`.

```
 1  EXIT -- the problem is infeasible
```

When the constraints are linear, this message can probably be trusted. Feasibility is measured with respect to the upper and lower bounds on the variables and slacks. Among all the points satisfying the general constraints $Ax + s = 0$, there is apparently no point that satisfies the bounds on x and s. Violations as small as the `Feasibility tolerance` are ignored, but at least one component of x or s violates a bound by more than the tolerance.

When nonlinear constraints are present, infeasibility is *much* harder to recognize correctly. Even if a feasible solution exists, the current linearization of the constraints may not contain a feasible point. In an attempt to deal with this situation, when solving each linearly constrained (LC) subproblem, MINOS is prepared to relax the bounds on the slacks associated with nonlinear rows.

If an LC subproblem proves to be infeasible or unbounded (or if the Lagrange multiplier estimates for the nonlinear constraints become large), MINOS enters so-called "nonlinear elastic" mode. The subproblem includes the original QP objective and the sum of the infeasibilities—suitably weighted using the `Elastic weight` parameter. In elastic mode, some of the bounds on the nonlinear rows "elastic"—i.e., they are allowed to violate their specified bounds. Variables subject to elastic bounds are known as *elastic variables*. An elastic variable is free to violate one or both of its original upper or lower bounds. If the original problem has a feasible solution and the elastic weight is sufficiently large, a feasible point eventually will be obtained for the perturbed constraints, and optimization can continue on the subproblem. If the nonlinear problem has no feasible solution, MINOS will tend to determine a "good" infeasible point if the elastic weight is sufficiently large. (If the elastic weight were infinite, MINOS would locally minimize the nonlinear constraint violations subject to the linear constraints and bounds.)

Unfortunately, even though MINOS locally minimizes the nonlinear constraint violations, there may still exist other regions in which the nonlinear constraints are satisfied. Wherever possible, nonlinear constraints should be defined in such a way that feasible points are known to exist when the constraints are linearized.

```
 2  EXIT -- the problem is unbounded  (or badly scaled)
    EXIT -- violation limit exceeded -- the problem may be unbounded
```

For linear problems, unboundedness is detected by the simplex method when a nonbasic variable can apparently be increased or decreased by an arbitrary amount without causing a basic variable to violate a bound. A message prior to the EXIT message will give the index of the nonbasic variable. Consider adding an upper or lower bound to the variable. Also, examine the constraints that have nonzeros in the associated column, to see if they have been formulated as intended.

Very rarely, the scaling of the problem could be so poor that numerical error will give an erroneous indication of unboundedness. Consider using the `Scale` option.

For nonlinear problems, MINOS monitors both the size of the current objective function and the size of the change in the variables at each step. If either of these is very large (as judged by the `Unbounded` parameters—see §**??**), the problem is terminated and declared UNBOUNDED. To avoid large function values, it may be necessary to impose bounds on some of the variables in order to keep them away from singularities in the nonlinear functions.

The second message indicates an abnormal termination while enforcing the limit on the constraint violations. This exit implies that the objective is not bounded below in the feasible region defined by expanding the bounds by the value of the `Violation limit`.

```
3  EXIT -- major iteration limit exceeded
   EXIT -- minor iteration limit exceeded
   EXIT -- too many iterations
```

Either the `Iterations limit` or the `Major iterations limit` was exceeded before the required solution could be found. Check the iteration log to be sure that progress was being made. If so, restart the run using a basis file that was saved (or should have been saved!) at the end of the run.

```
4  EXIT -- requested accuracy could not be achieved
```

A feasible solution has been found, but the requested accuracy in the dual infeasibilities could not be achieved. An abnormal termination has occurred, but MINOS is within 10^{-2} of satisfying the `Major optimality tolerance`. Check that the `Major optimality tolerance` is not too small.

```
5  EXIT -- the superbasics limit is too small:  nnn
```

The problem appears to be more nonlinear than anticipated. The current set of basic and superbasic variables have been optimized as much as possible and a PRICE operation is necessary to continue, but there are already `nnn` superbasics (and no room for any more).

In general, raise the `Superbasics limit` s by a reasonable amount, bearing in mind the storage needed for the reduced Hessian (about $\frac{1}{2}s^2$ double words).

```
6  EXIT -- constraint and objective values could not be calculated
```

This exit occurs if a value `mode` ≤ -1 is set during some call to `funobj` or `funcon`. MINOS assumes that you want the problem to be abandoned forthwith.

In some environments, this exit means that your subroutines were not successfully linked to MINOS. If the default versions of `funobj` and `funcon` are ever called, they issue a warning message and then set `mode` to terminate the run.

```
7  EXIT -- subroutine funobj seems to be giving incorrect gradients
```

A check has been made on some individual elements of the objective gradient array at the first point that satisfies the linear constraints. At least one component `gObj(`j`)` is being set to a value that disagrees markedly with a forward-difference estimate of $\partial f/\partial x_j$. (The relative difference between the computed and estimated values is 1.0 or more.) This exit is a safeguard, since MINOS will usually fail to make progress when the computed gradients are seriously inaccurate. In the process it may expend considerable effort before terminating with EXIT 9 below.

Check the function and gradient computation *very carefully* in `funobj`. A simple omission (such as forgetting to divide `fObj` by 2) could explain everything. If `fObj` or `gObj(`j`)` is very large, then give serious thought to scaling the function or the nonlinear variables.

If you feel *certain* that the computed `gObj(`j`)` is correct (and that the forward-difference estimate is therefore wrong), you can specify `Verify level 0` to prevent individual elements from being checked. However, the optimization procedure may have difficulty.

```
8  EXIT -- subroutine funcon seems to be giving incorrect gradients
```

This is analogous to the preceding exit. At least one of the computed Jacobian elements is significantly different from an estimate obtained by forward-differencing the constraint vector $F(x)$. Follow the advice given above, trying to ensure that the arrays `fCon` and `gCon` are being set correctly in `funcon`.

```
9  EXIT -- the current point cannot be improved upon
```

Several circumstances could lead to this exit.

1. Subroutines `funobj` or `funcon` could be returning accurate function values but inaccurate gradients (or vice versa). This is the most likely cause. Study the comments given for EXIT 7 and 8, and do your best to ensure that the coding is correct.

2. The function and gradient values could be consistent, but their precision could be too low. For example, accidental use of a `real` data type when `double precision` was intended would lead to a relative function precision of about 10^{-6} instead of something like 10^{-15}. The default `Optimality tolerance` of 10^{-6} would need to be raised to about 10^{-3} for optimality to be declared (at a rather suboptimal point). Of course, it is better to revise the function coding to obtain as much precision as economically possible.

3. If function values are obtained from an expensive iterative process, they may be accurate to rather few significant figures, and gradients will probably not be available. One should specify

`Function precision`	t
`Major optimality tolerance`	$\sqrt{t}$

   but even then, if t is as large as 10^{-5} or 10^{-6} (only 5 or 6 significant figures), the same exit condition may occur. At present the only remedy is to increase the accuracy of the function calculation.

`10 EXIT -- cannot satisfy the general constraints`
An LU factorization of the basis has just been obtained and used to recompute the basic variables x_B, given the present values of the superbasic and nonbasic variables. A step of "iterative refinement" has also been applied to increase the accuracy of x_B. However, a row check has revealed that the resulting solution does not satisfy the current constraints $Ax - s = 0$ sufficiently well.

This probably means that the current basis is very ill-conditioned. Try `Scale option 1` if scaling has not yet been used and there are some linear constraints and variables.

For certain highly structured basis matrices (notably those with band structure), a systematic growth may occur in the factor U. Consult the description of `Umax`, `Umin` and `Growth` in §6.1.4, and set the `LU factor tolerance` to 2.0 (or possibly even smaller, but not less than 1.0).

`12 EXIT -- terminated from subroutine s1User`
The user has set the value `iAbort` = 1 in subroutine `s1User`. MINOS assumes that you want the problem to be abandoned forthwith.

> If the following exits occur during the *first* basis factorization, the primal and dual variables `x` and `pi` will have their original input values. BASIS files will be saved if requested, but certain values in the printed solution will not be meaningful.

`20 EXIT -- not enough integer/real storage for the basis factors`
The main integer or real storage array `iw(*)` or `rw(*)` is apparently not large enough for this problem. The routine declaring `iw` and `rw` should be recompiled with a larger dimensions for those arrays. The new values should also be assigned to `leniw` and `lenrw`.

An estimate of the additional storage required is given in messages preceding the exit.

`21 EXIT -- error in basis package`
A preceding message will describe the error in more detail. One such message says that the current basis has more than one element in row i and column j. This could be caused by a corresponding error in the input parameters `a(*)`, `ha(*)`, and `ka(*)`.

`22 EXIT -- singular basis after nnn factorization attempts`
This exit is highly unlikely to occur. The first factorization attempt will have found the basis to be structurally or numerically singular. (Some diagonals of the triangular matrix U were respectively zero or smaller than a certain tolerance.) The associated variables are replaced by slacks and the

modified basis is refactorized, but singularity persists. This must mean that the problem is badly scaled, or the `LU factor tolerance` is too much larger than 1.0.

> If the following messages arise, either an Old Basis file could not be loaded properly, or some fatal system error has occurred. New Basis files cannot be saved, and there is no solution to print. The problem is abandoned.

`30  EXIT -- the basis file dimensions do not match this problem`
On the first line of the Old Basis file, the dimensions labeled `m` and `n` are different from those associated with the problem that has just been defined. You have probably loaded a file that belongs to another problem.

Remember, if you have added rows or columns to `a(*)`, `ha(*)` and `ka(*)`, you will have to alter `m` and `n` *and* the map beginning on the third line (a hazardous operation). It may be easier to restart with a Punch or Dump file from an earlier version of the problem.

`31  EXIT -- the basis file state vector does not match this problem`
For some reason, the Old Basis file is incompatible with the present problem, or is not consistent within itself. The number of basic entries in the state vector (i.e., the number of `3`'s in the map) is not the same as `m` on the first line, or some of the `2`'s in the map did not have a corresponding "$j \quad x_j$" entry following the map.

`32  EXIT -- system error.  Wrong no. of basic variables:  nnn`
This exit should never happen. It may indicate that the wrong MINOS source files have been compiled, or incorrect parameters have been used in the call to subroutine `minoss`.

Check that all integer variables and arrays are declared `integer` in your calling program (including those beginning with `h`!), and that all "real" variables and arrays are declared consistently. (They should be `double precision` on most machines.)

> The following messages arise if additional storage is needed to allow optimization to begin. The problem is abandoned.

`42  EXIT -- not enough 8-character storage to start solving the problem`
The main character storage array `cw(*)` is not large enough.

`43  EXIT -- not enough integer storage to start solving the problem`
The main integer storage array `iw(*)` is not large enough to provide workspace for the optimization procedure. See the advice given for Exit 20.

`44  EXIT -- not enough real storage to start solving the problem`
The main storage array `rw(*)` is not large enough to provide workspace for the optimization procedure. Be sure that the `Superbasics limit` is not unreasonably large. Otherwise, see the advice for EXIT 20.

6.1.6 Solution output

At the end of a run, the final solution is output to the Print file in accordance with the `Solution` keyword. Some header information appears first to identify the problem and the final state of the optimization procedure. A ROWS section and a COLUMNS section then follow, giving one line of information for each row and column. The format used is similar to certain commercial systems, though there is no industry standard.

An example of the printed solution is given in §6. In general, numerical values are output with format `f16.5`. The maximum record length is 111 characters, including the first (carriage-control) character.

To reduce clutter, a dot "`.`" is printed for any numerical value that is exactly zero. The values ± 1 are also printed specially as `1.0` and `-1.0`. Infinite bounds ($\pm 10^{20}$ or larger) are printed as `None`.

Note: If two problems are the same except that one minimizes an objective $f(x)$ and the other maximizes $-f(x)$, their solutions will be the same but the signs of the dual variables π_i and the reduced gradients d_j will be reversed.

The ROWS section

General linear constraints take the form $l \le Ax \le u$. The ith constraint is therefore of the form

$$\alpha \le a^T x \le \beta,$$

and the value of $a^T x$ is called the *row activity*. Internally, the linear constraints take the form $Ax - s = 0$, where the slack variables s should satisfy the bounds $l \le s \le u$. For the ith "row", it is the slack variable s_i that is directly available, and it is sometimes convenient to refer to its state. Slacks may be basic or nonbasic (but not superbasic).

Nonlinear constraints $\alpha \le F_i(x) + a^T x \le \beta$ are treated similarly, except that the row activity and degree of infeasibility are computed directly from $F_i(x) + a^T x$ rather than from s_i.

Label	*Description*
`Number`	The value $n + i$. This is the internal number used to refer to the ith slack in the iteration log.
`Row`	The name of the ith row.
`State`	The state of the ith row relative to the bounds α and β. The various states possible are as follows.

`LL`	The row is at its lower limit, α.
`UL`	The row is at its upper limit, β.
`EQ`	The limits are the same ($\alpha = \beta$).
`BS`	The constraint is not binding. s_i is basic.

A key is sometimes printed before the `State` to give some additional information about the state of the slack variable.

`A`	*Alternative optimum possible*. The slack is nonbasic, but its reduced gradient is essentially zero. This means that if the slack were allowed to start moving from its current value, there would be no change in the objective function. The values of the basic and superbasic variables *might* change, giving a genuine alternative solution. The values of the dual variables *might* also change.
`D`	*Degenerate*. The slack is basic, but it is equal to (or very close to) one of its bounds.
`I`	*Infeasible*. The slack is basic and is currently violating one of its bounds by more than the `Feasibility tolerance`.
`N`	*Not precisely optimal*. The slack is nonbasic. Its reduced gradient is larger than the `Optimality tolerance` .

Note: If `Scale option > 0`, the tests for assigning `A`, `D`, `I`, `N` are made on the scaled problem, since the keys are then more likely to be meaningful.

`Activity` The row value $a^T x$ (or $F_i(x) + a^T x$ for nonlinear rows).

`Slack activity` The amount by which the row differs from its nearest bound. (For free rows, it is taken to be minus the `Activity`.)

`Lower limit` α, the lower bound on the row.

`Upper limit` β, the upper bound on the row.

`Dual activity` The value of the dual variable π_i, often called the shadow price (or simplex multiplier) for the ith constraint. The full vector π always satisfies $B^T\pi = g_B$, where B is the current basis matrix and g_B contains the associated gradients for the current objective function.

`I` The constraint number, i.

The COLUMNS section

Here we talk about the "column variables" x_j, $j = 1 : n$. We assume that a typical variable has bounds $\alpha \le x_j \le \beta$.

Label	*Description*
`Number`	The column number, j. This is the internal number used to refer to x_j in the iteration log.
`Column`	The name of x_j.
`State`	The state of x_j relative to the bounds α and β. The various states possible are as follows.

`LL`	x_j is nonbasic at its lower limit, α.
`UL`	x_j is nonbasic at its upper limit, β.
`EQ`	x_j is nonbasic and fixed at the value $\alpha = \beta$.
`FR`	x_j is nonbasic at some value strictly between its bounds: $\alpha < x_j < \beta$.
`BS`	x_j is basic. Usually $\alpha < x_j < \beta$.
`SBS`	x_j is superbasic. Usually $\alpha < x_j < \beta$.

A key is sometimes printed before the `State` to give some additional information about the state of x_j.

`A`	*Alternative optimum possible.* The variable is nonbasic, but its reduced gradient is essentially zero. This means that if x_j were allowed to start moving from its current value, there would be no change in the objective function. The values of the basic and superbasic variables *might* change, giving a genuine alternative solution. The values of the dual variables *might* also change.
`D`	*Degenerate.* x_j is basic, but it is equal to (or very close to) one of its bounds.
`I`	*Infeasible.* x_j is basic and is currently violating one of its bounds by more than the `Feasibility tolerance`.

`N` *Not precisely optimal.* x_j is nonbasic. Its reduced gradient is larger than the `Optimality tolerance` .

Note: If `Scale option > 0`, the tests for assigning `A`, `D`, `I`, `N` are made on the scaled problem, since the keys are then more likely to be meaningful.

`Activity` The value of the variable x_j.

`Obj Gradient` g_j, the jth component of the gradient of the (linear or nonlinear) objective function. (If any x_j is infeasible, g_j is the gradient of the sum of infeasibilities.)

`Lower limit` α, the lower bound on x_j.

`Upper limit` β, the upper bound on x_j.

`Reduced gradnt` The reduced gradient $d_j = g_j - \pi^T a_j$, where a_j is the jth column of the constraint matrix (or the jth column of the Jacobian at the start of the final major iteration).

`M+J` The value $m + j$.

6.2 THE SOLUTION FILE

The information in a printed solution (§6.1.6) may be output as a Solution file, according to the `Solution file` option (which may refer to the Print file if so desired). Infinite bounds appear as $\pm 10^{20}$ rather than `None`. Other numerical values are output with format `1p, e16.6`.

A Solution file is intended to be read from disk by a self-contained program that extracts and saves certain values as required for possible further computation. Typically the first 14 records would be ignored. Each subsequent record may be read using

```
format(i8, 2x, 2a4, 1x, a1, 1x, a3, 5e16.6, i7)
```

adapted to suit the occasion. The end of the ROWS section is marked by a record that starts with a `1` and is otherwise blank. If this and the next 4 records are skipped, the COLUMNS section can then be read under the same format. (There should be no need for `backspace` statements.)

6.3 THE SUMMARY FILE

If `Summary file` > 0, the following information is output to the Summary file. (It is a brief form of the Print file.) All output lines are less than 72 characters.

- The `Begin` line from the Specs file, if any.
- The basis file loaded, if any.
- A brief Major iteration log.
- A brief Minor iteration log.
- The EXIT condition and a summary of the final solution.

The following Summary file is from example problem `t6wood` using `Print level 0` and `Major damping parameter 0.5`.

```
     ==============================
     M I N O S  5.51     (Nov 2002)
     ==============================

Begin t6wood   (WOPLANT test problem; optimal obj = -15.55716)

Name   WOPLANT
===>  Note:  row  OBJ       selected as linear part of objective.
Rows          9
Columns      12
Elements     73

Scale option  2,    Partial price   1

Itn      0 -- linear constraints satisfied.

This is problem t6wood.  Derivative level =  3

funcon  sets      36   out of      50   constraint gradients.

Major minor  step     objective  Feasible Optimal  nsb   ncon penalty BSwap
    1     0T 0.0E+00  0.00000E+00 5.9E-01 1.1E+01    0      4 1.0E+00     0
    2    22  5.0E-01 -1.56839E+01 2.7E-01 1.6E+01    3     47 1.0E+00     0
    3    10  6.0E-01 -1.51527E+01 1.5E-01 9.9E+00    2     68 1.0E+00     2
    4    21  5.7E-01 -1.53638E+01 6.4E-02 3.6E+00    3    113 1.0E+00     1
    5    15  1.0E+00 -1.55604E+01 2.7E-02 1.4E-01    3    144 1.0E+00     0
    6     5  1.0E+00 -1.55531E+01 6.4E-03 2.2E-01    3    154 1.0E+00     0
    7     4  1.0E+00 -1.55569E+01 3.1E-04 7.0E-04    3    160 1.0E-01     0
    8     2  1.0E+00 -1.55572E+01 1.6E-08 1.1E-04    3    163 1.0E-02     0
    9     1  1.0E+00 -1.55572E+01 5.1E-14 2.2E-06    3    165 1.0E-03     0

EXIT -- optimal solution found

Problem name                 WOPLANT
No. of iterations                  80   Objective value     -1.5557160112E+01
No. of major iterations             9   Linear objective    -1.5557160112E+01
Penalty parameter            0.000100   Nonlinear objective  0.0000000000E+00
No. of calls to funobj              0   No. of calls to funcon            165
No. of superbasics                  3   No. of basic nonlinears             6
No. of degenerate steps             0   Percentage                       0.00
Norm of x   (scaled)          9.8E-01   Norm of pi  (scaled)          1.8E+02
Norm of x                     3.2E+01   Norm of pi                    1.6E+01
Max Prim inf(scaled)        0 0.0E+00   Max Dual inf(scaled)        1 2.2E-06
Max Primal infeas           0 0.0E+00   Max Dual infeas             1 5.8E-08
Nonlinear constraint violn    5.1E-14



Solution printed on file   9

funcon called with nstate =   2

Time for MPS input                            0.00 seconds
Time for solving problem                      0.04 seconds
Time for solution output                      0.00 seconds
Time for constraint functions                 0.00 seconds
Time for objective function                   0.00 seconds
Endrun
```

Chapter 7

September 23, 2003

Subroutine minoss

This chapter describes minoss, the subroutine version of MINOS. Later sections describe an auxiliary routine (mispec) for reading a SPECS file, and some additional routines for specifying individual lines of such a file as part of the calling program.

Note that subroutine mispec must be called before the first call to minoss, even if a SPECS file is not being read.

In the subroutine specifications, "double precision" entities are appropriate for most machines, but in same cases (e.g. on Cray and Convex systems) they should be changed to their "single precision" equivalents. In some installations, integer*4 may have been changed to integer*2 throughout the MINOS source code, to conserve storage. Otherwise, both integer*4 and plain integer are intended to mean 4-byte words.

7.1 SUBROUTINE MINOSS

Problem data is passed to minoss as parameters, rather than from an MPS file. This is generally more efficient and convenient for applications that would normally use a "matrix generator".

Specification

```
      subroutine minoss( start, m, n, nb, ne, nname,
     $                   nncon, nnobj, nnjac,
     $                   iobj, objadd, names,
     $                   a, ha, ka, bl, bu, name1, name2,
     $                   hs, xn, pi, rc,
     $                   inform, mincor, ns, ninf, sinf, obj,
     $                   z, nwcore )

      implicit           double precision (a-h,o-z)
      character*(*)      start
      integer            m, n, nb, ne, nname,
     $                   nncon, nnobj, nnjac, iobj,
     $                   inform, mincor, ns, ninf, nwcore
      double precision   objadd, sinf, obj
      character*8        names(5)
      integer*4          ha(ne), hs(nb)
      integer            ka(n+1), name1(nname), name2(nname)
      double precision   a(ne), bl(nb), bu(nb)
      double precision   xn(nb), pi(m), rc(nb), z(nwcore)
```

On entry:

start specifies how a starting basis (and certain other items) are to be obtained.

start = 'Cold' means that Crash should be used to choose an initial basis (unless a basis file is provided).

`start` = `'Warm'` means that a basis is already defined in `hs` (probably from an earlier call).

`start` = `'Hot'` or `'Hot FHS'` implies a Hot start. `hs` defines a basis and an earlier call has defined certain other things that should also be kept. The problem dimensions and the array `z(*)` must not have changed.

`F` refers to the *LU* factors of the basis.
`H` refers to the approximate reduced Hessian *R*.
`S` refers to column and row scales.

`start` = `'Hot H'` (for example) means that only the Hessian is defined.

`start` = `'Basis file'` is the same as `start` = `'Cold'` (but is more meaningful if an Old Basis, Insert or Load file is provided).

`m` is m, the number of general constraints. For LP problems this means the number of rows in the constraint matrix A. If `integer*4` has been replaced by `integer*2` throughout the Fortran source code, `m` should not exceed 16383. Otherwise there is essentially no upper limit.

In principle, `m` > 0, though sometimes `m` $= 0$ may be acceptable. (Strictly speaking, Fortran declarations of the form `double precision pi(m)` require `m` to be positive. In debug mode, compilers will probably enforce this, but optimized code may sometimes run successfully with `m` $= 0$.)

`n` is n, the number of variables (excluding slacks). For LP problems, this is the number of columns in A (> 0).

`nb` is $nb = n + m$ (the number of bounds in `bl` or `bu`).

`ne` is ne, the number of nonzero entries in A (including the Jacobian for any nonlinear constraints). In principle, `ne` > 0, though again `m` $= 0$, `ne` $= 0$ may work with some compilers.

`nname` is the number of column and row names provided in the arrays `name1` and `name2`. If `nname` $= 1$, there are *no* names. Generic names will be used in the printed solution. Otherwise, `nname` $= nb$ and all names must be provided.

`nncon` is m_1, the number of nonlinear constraints (≥ 0).

`nnobj` is n_1', the number of nonlinear objective variables (≥ 0).

`nnjac` is n_1'', the number of nonlinear Jacobian variables (≥ 0). If `nncon` $= 0$, `nnjac` $= 0$. If `nncon` > 0, `nnjac` > 0.

`iobj` says which row of A is a free row containing a linear objective vector c. If there is no such vector, `iobj` $= 0$. Otherwise, this row must come after any nonlinear rows, so that `nncon` $<$ `iobj` $\leq m$.

`objadd` is a constant that will be added to the objective. Typically `objadd` = `0.0d+0`.

`names(5)` is a set of 8-character names for the problem, the linear objective, the rhs, the ranges and bounds. (This is a hangover from MPS files. The names are used in the printed solution and in some of the basis files.)

`a(ne)` is the constraint matrix (Jacobian), stored column-wise.

`ha(ne)` is a list of row indices for each nonzero in `a(*)`.

`ka(n+1)` is a set of pointers to the beginning of each column of the constraint matrix within `a(*)` and `ha(*)`. It is essential that $\texttt{ka}(1) = 1$ and $\texttt{ka}(n+1) = ne + 1$.

1. If the problem has a nonlinear objective, the first `nnobj` columns of `a` and `ha` belong to the nonlinear objective variables. Subroutine `funobj` deals with these variables.
2. If the problem has nonlinear constraints, the first `nnjac` columns of `a` and `ha` belong to the nonlinear Jacobian variables, and the first `nncon` rows of `a` and `ha` belong to the nonlinear constraints. Subroutine `funcon` deals with these variables and constraints.
3. If $\texttt{nnobj} > 0$ and $\texttt{nnjac} > 0$, the two sets of nonlinear variables overlap. The total number of nonlinear variables is $\texttt{nn} = \max(\texttt{nnobj}, \texttt{nnjac})$.
4. The Jacobian forms the top left corner of `a` and `ha`. If a Jacobian column j ($1 \le j \le \texttt{nnjac}$) contains any entries $\texttt{a}(k)$, $\texttt{ha}(k)$ associated with nonlinear constraints ($1 \le \texttt{ha}(k) \le \texttt{nncon}$), those entries must come before any other (linear) entries.
5. The row indices $\texttt{ha}(k)$ for a column may be in any order (subject to Jacobian entries appearing first). Subroutine `funcon` must define Jacobian entries in the same order.
6. Columns of A should contain at least one entry, so that $\texttt{ka}(j) < \texttt{ka}(j+1)$ for every j. If a column has no meaningful entry, include a dummy entry $\texttt{a}(k) =$ `0.0d+0`, $\texttt{ha}(k) = 1$.

`bl(nb)` is the lower bounds on the variables and slacks (x, s).

The first n entries of `bl`, `bu`, `hs` and `xn` refer to the variables x. The last m entries refer to the slacks s.

`bu(nb)` is the upper bounds on (x, s).

Beware: MINOS represents general constraints as $Ax + s = 0$. Constraints of the form $l \le Ax \le u$ therefore mean $l \le -s \le u$, so that $-u \le s \le -l$. The last m components of `bl` and `bu` are $-u$ and $-l$.

`name1(nname), name2(nname)` are integer arrays.

If $\texttt{nname} = 1$, `name1` and `name2` are not used. The printed solution will use generic names for the columns and rows. If $\texttt{nname} = nb$, $\texttt{name1}(j)$ and $\texttt{name2}(j)$ should contain the name of the j-th variable in `2a4` format ($j = 1$ to nb). If $j = n + i$, the j-th variable is the i-th row.

`hs(nb)` sometimes contains a set of initial states for each variable x, or for each variable and slack (x, s). See next lines.

`xn(nb)` sometimes contains a set of initial values for each variable x, or for each variable and slack (x, s).

1. For cold starts, you must define $\texttt{hs}(j)$ and $\texttt{xn}(j)$, $j = 1$ to n. (The values for $j = n+1$ to nb need not be set.) If nothing special is known about the problem, or if there is no wish to provide special information, you may set $\texttt{hs}(j) = 0$, $\texttt{xn}(j) = 0.0$ for all $j = 1$ to n. All variables will be eligible for the initial basis.

   Less trivially, to say that variable j will probably be equal to one of its bounds, set $\texttt{hs}(j) = 4$ and $\texttt{xn}(j) = \texttt{bl}(j)$ or $\texttt{hs}(j) = 5$ and $\texttt{xn}(j) = \texttt{bu}(j)$ as appropriate.

2. For Cold starts with no basis file, a Crash procedure is used to select an initial basis. The initial basis matrix will be triangular (ignoring certain small entries in each column). The values `hs`$(j) = 0, 1, 2, 3, 4, 5$ have the following meaning:

   If `hs`$(j) = 0$, 1 or 3, Crash considers that column j is eligible for the basis, with preference given to 3.

   If `hs`$(j) = 2$, 4 or 5, Crash ignores column j.

   After Crash, columns for which `hs`$(j) = 2$ are made superbasic. Other columns not selected for the basis are made nonbasic at the value `xn`(j) if `bl`$(j) \leq$ `xn`$(j) \leq$ `bu`(j), or at the value `bl`(j) or `bu`(j) closest to `xn`(j).

3. For Warm or Hot starts, all of `hs(1:nb)` is assumed to be set to the values 0, 1, 2 or 3 (probably from some previous call) and all of `xn(1:nb)` must have values.

If `start` = `'Cold'` or `'Basis file'` and an Old Basis, Insert or Load file is provided, `hs` and `xn` need not be set at all.

`pi(m)` contains an estimate of the vector of Lagrange multipliers (shadow prices) for the nonlinear constraints. The first `nncon` components must be defined. They will be used as λ_k in the subproblem objective function for the first major iteration. If nothing is known about λ_k, set `pi`(i) = `0.0d+0`, $i = 1$ to `nncon`.

`ns` need not be specified for Cold starts, but should retain its value from a previous call when a Warm or Hot start is used.

`z(nwcore)` is a (large) array that provides all workspace. Problems involving m general constraints typically need `nwcore` at least $100m$. See the output parameter `mincor` below.

On exit:

`hs(nb)` is the final state vector. If the solution is optimal or feasible, the entries of `hs` usually have the following meaning:

`hs`(j)	*State of variable j*	*Usual value of* `xn`(j)
0	nonbasic	`bl`(j)
1	nonbasic	`bu`(j)
2	superbasic	Between `bl`(j) and `bu`(j)
3	basic	Between `bl`(j) and `bu`(j)

Basic and superbasic variables may be outside their bounds by as much as the `Feasibility tolerance`. Note that if scaling is specified, the `Feasibility tolerance` applies to the variables of the *scaled* problem. In this case, the variables of the original problem may be as much as 0.1 outside their bounds, but this is unlikely unless the problem is very badly scaled. Check the "Primal infeasibility" printed after the `EXIT` message.

Very occasionally some nonbasic variables may be outside their bounds by as much as the `Feasibility tolerance`, and there may be some nonbasics for which `xn`(j) lies strictly between its bounds.

If `ninf` > 0, some basic and superbasic variables may be outside their bounds by an arbitrary amount (bounded by `sinf` if scaling was not used).

`xn(nb)` is the final variables and slacks (x, s).

`pi(m)` is the vector of dual variables π (a set of Lagrange multipliers for the general constraints).

`rc(nb)` is a vector of reduced costs, $g - (\,A \quad I\,)^T\pi$, where g is the gradient of the objective function if `xn` is feasible, or the gradient of the Phase-1 objective otherwise. If `ninf` $= 0$, the last m entries are $-\pi$.

`inform` says what happened, as described more fully in Chapter 6.3.

`inform`	*Meaning*
0	Optimal solution found.
1	The problem is infeasible.
2	The problem is unbounded (or badly scaled).
3	Too many iterations.
4	Apparent stall. The solution has not changed for a large number of iterations (e.g. 1000).
5	The `Superbasics limit` is too small.
6	Subroutine `funobj` or `funcon` requested termination by returning `mode` < 0.
7	`funobj` seems to be giving incorrect gradients.
8	`funcon` seems to be giving incorrect gradients.
9	The current point cannot be improved.
10	Numerical error in trying to satisfy the linear constraints (or the linearized nonlinear constraints). The basis is very ill-conditioned.
11	Cannot find a superbasic to replace a basic variable.
12	Basis factorization requested twice in a row. Should probably be treated as `inform` $= 9$.
13	Near-optimal solution found. Should probably be treated as `inform` $= 9$.

`inform`	*Meaning*
20	Not enough storage for the basis factorization.
21	Error in basis package.
22	The basis is singular after several attempts to factorize it (and add slacks where necessary).
30	An OLD BASIS file had dimensions that did not match the current problem.
32	System error. Wrong number of basic variables.
40	Fatal errors in the MPS file.
41	Not enough storage to read the MPS file.
42	Not enough storage to solve the problem.

`mincor` says how much storage is needed to solve the problem. If `inform` $= 42$, the work array `z(nwcore)` was too small. `minoss` may be called again with `nwcore` suitably larger than `mincor`. (The bigger the better, since it is not certain how much storage the basis factors need.)

`ns` is the final number of superbasics.

`ninf` is the number of infeasibilities.

`sinf` is the sum of infeasibilities.

`obj` is the value of the objective function. If `ninf` $= 0$, `obj` includes the nonlinear objective if any. If `ninf` > 0, `obj` is just the linear objective if any.

7.2 SUBROUTINE MISPEC

This subroutine must be called before the first call to `minoss`. It opens the SPECS, PRINT and SUMMARY files (if they exist), sets all options to default values, and reads the SPECS file if any. File numbers must be in the range 1 to 99, or 0 if the associated file does not exist.

Specification

```
subroutine mispec( ispecs, iprint, isumm, nwcore, inform )

integer            ispecs, iprint, isumm, nwcore, inform
```

On entry:

`ispecs` says whether or not a SPECS file exists. If `ispecs` > 0, a file is read from the specified Fortran file number. Typically `ispecs` = 4.

`iprint` says if a PRINT file is to be created. Typically `iprint` = 9.

`isumm` says if a SUMMARY file is to be created. Typically `isumm` = 6. In an interactive environment, this value usually denotes the screen.

`nwcore` is the length of the workspace array `z(*)` that is later passed to `minoss`.

On exit:

`inform` is 0 if there was no SPECS file, or if the SPECS file was successfully read. Otherwise, it returns the number of errors encountered.

7.3 SUBROUTINES MIOPT, MIOPTI, MIOPTR

These subroutines may be called from the program that calls `minoss`. They specify a single option that might otherwise be defined in one line of a SPECS file.

Specification

```
subroutine miopt ( buffer,         iprint, isumm, inform )
subroutine miopti( buffer, ivalue, iprint, isumm, inform )
subroutine mioptr( buffer, rvalue, iprint, isumm, inform )

character*(*)      buffer
integer            ivalue
double precision   rvalue
integer            iprint, isumm, inform
```

On entry:

`buffer` is a string to be decoded as if it were a line of a SPECS file. For `miopt`, the maximum length of `buffer` is 72 characters. Use `miopt` if the string contains all of the data associated with a particular keyword. For example,

```
call miopt ( 'Iterations 1000',    iprint, isumm, inform )
```

is suitable if the value 1000 is known at compile time.

For `miopti` and `mioptr` the maximum length of `buffer` is 55 characters.

`ivalue` is an integer value associated with the keyword in `buffer`. Use `miopti` if it is convenient to define the value at run time. For example,

```
itnlim = 1000
if (m .gt. 500) itnlim = 8000
call miopti( 'Iterations', itnlim, iprint, isumm, inform )
```

allows the iteration limit to be computed.

`rvalue` is a floating-point value associated with the keyword in `buffer`. Use `mioptr` if it is convenient to define the value at run time. For example,

```
factol = 100.0d+0
if ( illcon ) factol = 5.0d+0
call mioptr( 'LU factor tol', factol, iprint, isumm, inform )
```

allows the *LU* stability tolerance to be computed.

`iprint` is a file number for printing each line of data, along with any error messages. `iprint` = 0 suppresses this output.

`isumm` is a file number for printing any error messages. `isumm` = 0 suppresses this output.

`inform` should be 0.

On exit:

`inform` is the number of errors encountered so far.

7.4 EXAMPLE USE OF MINOSS

File `minost.for` contains a Fortran test program to illustrate the use of subroutines `mispec`, `minoss`, `miopt`, `miopti` and `mioptr`. The test program reads a SPECS file, generates test problem MANNE (see Pages 98–108 of the User's Guide), sets some options not specified in the SPECS file, then calls `minoss` to solve the problem.

The SPECS file is in `minost.spc`. The required function subroutines `funobj` and `funcon` are part of the MINOS source file `mi05funs.for`.

To use the test program, compile and link `minost.for` and all of the MINOS source files, excluding the stand-alone MINOS main program (`mi00main.for`). See file `unix.mak` or `minost.mak`.

To run the resulting binary file, see file `unix.run` or `vminost.com`.

Good luck with your own use of `minoss`!

File minost.for

```
*     ------------------------------------------------------------------
*     File minost.for
*     This is a main program to test subroutine minoss, which is
*     part of MINOS 5.5.  It generates the problem called MANNE on
*     Pages 98-108 of the MINOS 5.1 User's Guide, then asks minoss
*     to solve it.
*
*
*     11 Nov 1991: First version.
*     27 Nov 1991: miopt, miopti, mioptr used to alter some options
*                  for a second call to minoss.
*     10 Apr 1992: objadd added as input parameter to minoss.
*     26 Jun 1992: integer*2 changed to integer*4.
*     15 Oct 1993: t4data now outputs pi.
*     24 Jan 1995: MINOS inadvertently scales all of xn before solving,
*                  so t4data sets dummy values for the slacks after all.
*     05 Feb 1998: No longer have to set Jacobian = dense or sparse
*                  when MINOS is called as a subroutine.
*     ------------------------------------------------------------------

      program            minost

      implicit           double precision (a-h,o-z)

      parameter        ( maxm   = 100,
     $                   maxn   = 150,
     $                   maxnb  = maxm + maxn,
     $                   maxne  = 500,
     $                   nname  = 1 )

      character*8        names(5)
      integer*4          ha(maxne) , hs(maxnb)
      integer            ka(maxn+1), name1(nname), name2(nname)
      double precision   a(maxne)  , bl(maxnb)   , bu(maxnb),
     $                   xn(maxnb) , pi(maxm)    , rc(maxnb)

      parameter          ( nwcore = 50000 )
      double precision   z(nwcore)
*     ------------------------------------------------------------------
```

```
*     Give names to the Problem, Objective, Rhs, Ranges and Bounds.

      names(1) = 'manne10 '
      names(2) = 'funobj  '
      names(3) = 'zero    '
      names(4) = 'range1  '
      names(5) = 'bound1  '

*     Specify some of the MINOS files.
*     ispecs  is the Specs file   (0 if none).
*     iprint  is the Print file   (0 if none).
*     isumm   is the Summary file (0 if none).
*     (mispec opens these files via mifile and m1open.)
*     nout    is an output file used here by mitest.

      ispecs = 4
      iprint = 9
      isumm  = 6
      nout   = 6

*     ------------------------------------------------------------------
*     Set options to default values.
*     Read a Specs file (if ispecs > 0).
*     ------------------------------------------------------------------
      call mispec( ispecs, iprint, isumm, nwcore, inform )

      if (inform .ge. 2) then
         write(nout, *) 'ispecs > 0 but no Specs file found'
         stop
      end if

*     ------------------------------------------------------------------
*     Generate a 10-period problem (nt = 10).
*     Instead of hardwiring nt here, we could do the following:
*     1. Say    Nonlinear constraints  10      in the Specs file.
*     2. At the top of this program include the following common block:
*               common    /m8len / njac  ,nncon ,nncon0,nnjac
*     3. Say    nt = nncon    in the line below.
*     ------------------------------------------------------------------
      nt     = 10
      call t4data( nt, maxm, maxn, maxnb, maxne, inform,
     $             m, n, nb, ne, nncon, nnobj, nnjac,
     $             a, ha, ka, bl, bu, hs, xn, pi )

      if (inform .ge. 1) then
         write(nout, *) 'Not enough storage to generate a problem ',
     $                  'with  nt =', nt
         stop
      end if

*     ------------------------------------------------------------------
*     Specify options that were not set in the Specs file.
*     i1 and i2 may refer to the Print and Summary file respectively.
*     Setting them to 0 suppresses printing.
*     ------------------------------------------------------------------
```

```
      i1     = 0
      i2     = 0
      ltime  = 2
      call miopti( 'Timing level      ', ltime, i1, i2, inform )

*     ------------------------------------------------------------------
*     Go for it, using a Cold start.
*     iobj   = 0 means there is no linear objective row in a(*).
*     objadd = 0.0 means there is no constant to be added to the
*            objective.
*     hs     need not be set if a basis file is to be input.
*            Otherwise, each hs(1:n) should be 0, 1, 2, 3, 4, or 5.
*            The values are used by the Crash procedure m2crsh
*            to choose an initial basis B.
*            If hs(j) = 0 or 1, column j is eligible for B.
*            If hs(j) = 2, column j is initially superbasic (not in B).
*            If hs(j) = 3, column j is eligible for B and is given
*                          preference over columns with hs(j) = 0 or 1.
*            If hs(j) = 4 or 5, column j is initially nonbasic.
*     ------------------------------------------------------------------
      iobj   = 0
      objadd = 0.0

*     For straightforward applications we would call minoss just once,
*     giving it all of z(*) for workspace.
*     Here we call it twice to illustrate situations where z(*) can be
*     expanded to suit the problem size.
*
*     For the first call, set lenz foolishly small and let minoss
*     tell us (via mincor) how big it would like z(*) to be.

      lenz   = 2
      call minoss( 'Cold', m, n, nb, ne, nname,
     $             nncon, nnobj, nnjac,
     $             iobj, objadd, names,
     $             a, ha, ka, bl, bu, name1, name2,
     $             hs, xn, pi, rc,
     $             inform, mincor, ns, ninf, sinf, obj,
     $             z, lenz )

      write(nout, *) ' '
      write(nout, *) 'Estimate of required workspace: mincor =', mincor

*     Since nwcor2 was not big enough, we will now have inform = 42.
*     Make z(*) longer and try again.  mincor SHOULD be enough.
*     (In general we should allow more to give the LU factors
*     as much room as possible).  For example,
*     mincor = mincor + 5*m + 1000   might be enough.)
*
*     Note that we can't say z(*) is longer than nwcore here.
*     minoss will return inform = 42 again if mincor > nwcore.

      lenz   = min( mincor, nwcore )

      call minoss( 'Cold', m, n, nb, ne, nname,
```

```
     $             nncon, nnobj, nnjac,
     $             iobj, objadd, names,
     $             a, ha, ka, bl, bu, name1, name2,
     $             hs, xn, pi, rc,
     $             inform, mincor, ns, ninf, sinf, obj,
     $             z, lenz )

      write(nout, *) ' '
      write(nout, *) 'minoss finished.'
      write(nout, *) 'inform =', inform
      write(nout, *) 'ninf   =', ninf
      write(nout, *) 'sinf   =', sinf
      write(nout, *) 'obj    =', obj
      if (inform .ge. 20) go to 900

*     ------------------------------------------------------------------
*     Alter some options and test the Warm start.
*     ------------------------------------------------------------------

*     The following illustrates the use of miopt, miopti and mioptr
*     to set specific options.  If necessary, we could ensure that
*     all unspecified options take default values
*     by first calling miopt ( 'Defaults', ... ).
*     Beware that certain parameters would then need to be redefined.
      write(nout, *) ' '
      write(nout, *) 'Alter options and test Warm start:'

      inform = 0
      itnlim = 20
      penpar = 0.01
      call miopt ( '                  ',         iprint, isumm, inform )
*---  call miopt ( 'Defaults          ',         iprint, isumm, inform )
*---  call miopti( 'Problem number    ',   1114, iprint, isumm, inform )
*---  call miopt ( 'Maximize          ',         iprint, isumm, inform )
      call miopt ( 'Derivative level 3',         iprint, isumm, inform )
*---  call miopt ( 'Print  level     0',         iprint, isumm, inform )
      call miopt ( 'Verify level     0',         iprint, isumm, inform )
      call miopt ( 'Scale option     0',         iprint, isumm, inform )
      call miopti( 'Iterations        ', itnlim, iprint, isumm, inform )
      call mioptr( 'Penalty parameter ', penpar, iprint, isumm, inform )

      if (inform .gt. 0) then
         write(nout, *) 'NOTE: Some of the options were not recognized'
      end if

*     Test the Warm start.
*     hs(*) specifies a complete basis from the previous call.
*     A Warm start uses hs(*) directly, without calling Crash.
*
*     Warm and Hot starts are normally used after minoss has solved a
*     problem with the SAME DIMENSIONS but perhaps altered data.
*     Here we have not altered the data, so very few iterations
*     should be required.

      call minoss( 'Warm', m, n, nb, ne, nname,
```

```
     $             nncon, nnobj, nnjac,
     $             iobj, objadd, names,
     $             a, ha, ka, bl, bu, name1, name2,
     $             hs, xn, pi, rc,
     $             inform, mincor, ns, ninf, sinf, obj,
     $             z, nwcore )

      write(nout, *) ' '
      write(nout, *) 'minoss finished again.'
      write(nout, *) 'inform =', inform
      write(nout, *) 'obj    =', obj
      if (inform .ge. 20) go to 900

*     ------------------------------------------------------------------
*     Alter more options (perhaps) and test the Hot start.
*     As with a Warm start, hs(*) specifies a basis from the
*     previous call.  In addition, up to three items from the previous
*     call can be reused.  They are denoted by F, H and S as follows:
*     'Hot F'    means use the existing basis FACTORS (B = LU).
*     'Hot H'    means use the existing reduced HESSIAN approximation.
*     'Hot S'    means use the existing column and row SCALES.
*     'Hot FS'   means use the Factors and Scales but not the Hessian.
*     'Hot FHS'  means use all three items.
*     'Hot'      is equivalent to 'Hot FHS'.
*     The letters F,H,S may be in any order.
*     Note that 'Hot' keeps existing scales.  Must say
*     'Hot H' or 'Hot ...' or something longer than 4 characters
*     if new scales are wanted.
*     ------------------------------------------------------------------
      write(nout, *) ' '
      write(nout, *) 'Test Hot start:'
      call miopt ( '                  ',         iprint, isumm, inform )
      call miopt ( 'Scale option     2',         iprint, isumm, inform )

      call minoss( 'Hot H', m, n, nb, ne, nname,
     $             nncon, nnobj, nnjac,
     $             iobj, objadd, names,
     $             a, ha, ka, bl, bu, name1, name2,
     $             hs, xn, pi, rc,
     $             inform, mincor, ns, ninf, sinf, obj,
     $             z, nwcore )

      write(nout, *) ' '
      write(nout, *) 'minoss finished again.'
      write(nout, *) 'inform =', inform
      write(nout, *) 'obj    =', obj
      if (inform .ge. 20) go to 900
      stop

*     ------------------------------------------------------------------
*     Error exit.
*     ------------------------------------------------------------------
  900 write(nout, *) ' '
      write(nout, *) 'STOPPING because of error condition'
      stop
```

```
*     end of main program to test subroutine minoss
      end

************************************************************************

      subroutine t4data( nt, maxm, maxn, maxnb, maxne, inform,
     $                   m, n, nb, ne, nncon, nnobj, nnjac,
     $                   a, ha, ka, bl, bu, hs, xn, pi )

      implicit           double precision (a-h,o-z)
      integer*4          ha(maxne), hs(maxnb)
      integer            ka(maxn+1)
      double precision   a(maxne) , bl(maxnb), bu(maxnb),
     $                   xn(maxnb), pi(maxm)

*     ------------------------------------------------------------------
*     t4data  generates data for the test problem t4manne
*     (called problem MANNE in the MINOS 5.1 User's Guide).
*     The constraints take the form
*              f(x) + A*x + s = 0,
*     where the Jacobian for f(x) + Ax is stored in a(*), and any
*     terms coming from f(x) are in the TOP LEFT-HAND CORNER of a(*),
*     with dimensions  nncon x nnjac.
*     Note that the right-hand side is zero.
*     s is a set of slack variables whose bounds contain any constants
*     that might have formed a right-hand side.
*
*     The objective function is
*             F(x) + c'x
*     where c would be row iobj of A (but there is no such row in
*     this example).  F(x) involves only the FIRST nnobj variables.
*
*     On entry,
*     nt      is T, the number of time periods.
*     maxm, maxn, maxnb, maxne are upper limits on m, n, nb, ne.
*
*     On exit,
*     inform  is 0 if there is enough storage, 1 otherwise.
*     m       is the number of nonlinear and linear constraints.
*     n       is the number of variables.
*     nb      is n + m.
*     ne      is the number of nonzeros in a(*).
*     nncon   is the number of nonlinear constraints (they come first).
*     nnobj   is the number of nonlinear objective variables.
*     nnjac   is the number of nonlinear Jacobian variables.
*     a       is the constraint matrix (Jacobian), stored column-wise.
*     ha      is the list of row indices for each nonzero in a(*).
*     ka      is a set of pointers to the beginning of each column of a.
*     bl      is the lower bounds on x and s.
*     bu      is the upper bounds on x and s.
*     hs(1:n) is a set of initial states for each x  (0,1,2,3,4,5).
*     xn(1:n) is a set of initial values for x.
*     pi(1:m) is a set of initial values for the dual variables pi.
*
```

```
*     09 Jul 1992: No need to initialize xn and hs for the slacks.
*     15 Oct 1993: pi is now an output parameter.  (Should have been
*                  all along.)
*     24 Jan 1995: MINOS inadvertently scales all of xn before solving,
*                  so we set dummy values for the slacks after all.
*     ------------------------------------------------------------------

      parameter      ( zero   = 0.0d+0,   one    = 1.0d+0,
     $                 dummy  = 0.1d+0,   growth = .03d+0,
     $                 bplus  = 1.0d+20,  bminus = - bplus )

*     nt defines the dimension of the problem.

      m      = nt*2
      n      = nt*3
      nb     = n + m
      nncon  = nt
      nnobj  = nt*2
      nnjac  = nt
      ne     = nt*6 - 1

*     Check if there is enough storage.

      inform = 0
      if (m      .gt. maxm ) inform = 1
      if (n      .gt. maxn ) inform = 1
      if (nb     .gt. maxnb) inform = 1
      if (ne     .gt. maxne) inform = 1
      if (inform .gt.   0  ) return

*     Generate columns for Capital (Kt, t = 1 to nt).
*     The first nt rows are nonlinear, and the next nt are linear.
*     The Jacobian is an nt x nt diagonal.
*     We generate the sparsity pattern here.
*     We put in dummy numerical values of 0.1 for the gradients.
*     Real values for the gradients are computed by t4con.

      ne     = 0
      do 100  k = 1, nt

*        There is one Jacobian nonzero per column.

         ne     = ne + 1
         ka(k)  = ne
         ha(ne) = k
         a(ne)  = dummy

*        The linear constraints form an upper bidiagonal pattern.

         if (k .gt. 1) then
            ne     = ne + 1
            ha(ne) = nt + k - 1
            a(ne)  = one
         end if
```

```
         ne     = ne + 1
         ha(ne) = nt + k
         a(ne)  = - one
  100 continue

*     The last nonzero is special.

      a(ne)  = growth

*     Generate columns for Consumption (Ct for t = 1 to nt).
*     They form -I in the first nt rows.
*     jC and jI are base indices for the Ct and It variables.

      jC    = nt
      jI    = nt*2

      do 200 k = 1, nt
         ne       = ne + 1
         ka(jC+k) = ne
         ha(ne)   = k
         a(ne)    = - one
  200 continue

*     Generate columns for Investment (It for t = 1 to nt).
*     They form -I in the first nt rows and -I in the last nt rows.

      do 300 k = 1, nt
         ne       = ne + 1
         ka(jI+k) = ne
         ha(ne)   = k
         a(ne)    = - one
         ne       = ne + 1
         a(ne)    = - one
         ha(ne)   = nt + k
  300 continue

*     ka(*) has one extra element.

      ka(n+1) = ne + 1

*     Set lower and upper bounds for Kt, Ct, It.
*     Also initial values and initial states for all variables.
*     The Jacobian variables are the most important.
*     Set hs(k) = 2 to make them initially superbasic.
*     The others might as well be on their smallest bounds (hs(j) = 0).

      do 400  k = 1, nt
         bl(   k) = 3.05d+0
         bu(   k) = bplus
         bl(jC+k) = 0.95d+0
         bu(jC+k) = bplus
         bl(jI+k) = 0.05d+0
         bu(jI+k) = bplus

         xn(   k) = 3.0d+0 + (k - 1)/10.0d+0
```

```
         xn(jC+k) = bl(jC+k)
         xn(jI+k) = bl(jI+k)

         hs(   k) = 2
         hs(jC+k) = 0
         hs(jI+k) = 0
  400 continue

*     The first Capital is fixed.
*     The last three Investments are bounded.

      bu(1)       = bl(1)
      xn(1)       = bl(1)
      hs(1)       = 0
      bu(jI+nt-2) = 0.112d+0
      bu(jI+nt-1) = 0.114d+0
      bu(jI+nt  ) = 0.116d+0

*     Set bounds on the slacks.
*     The nt nonlinear (Money)    rows are >=.
*     The nt    linear (CapacitY) rows are <=.
*     We no longer need to set initial values and states for slacks.
*     24 Jan 1995: MINOS inadvertently scales all of xn before solving,
*                  so we set dummy values for the slacks after all.

      jM     = n
      jY     = n + nt

      do 500    k = 1, nt
         bl(jM+k) = bminus
         bu(jM+k) = zero
         bl(jY+k) = zero
         bu(jY+k) = bplus

         xn(jM+k) = zero
         xn(jY+k) = zero
*-       hs(jM+k) = 0
*-       hs(jY+k) = 0
  500 continue

*     The last Money and Capacity rows have a Range.

      bl(jM+nt) = - 10.0d+0
      bu(jY+nt) =   20.0d+0

*     Initialize pi.
*     5.4 requires only pi(1:nncon) to be initialized.
*     5.5 may want all of pi to be initialized (not yet sure).

      do 600 i = 1, nt
         pi(i)    = - one
         pi(nt+i) = + one
  600 continue

*     end of t4data
```

```
      end
```

File minost.spc

```
Begin manne10  (10-period economic growth model)

   Problem number          1114
   Maximize
   Major iterations           8
   Minor iterations          20
   Penalty parameter        0.1

   Hessian dimension         10
   Derivative level           3
*   Verify gradients
   Verify level               0

   Scale option               2
   Scale option               1
   Iterations                50
   Print level (jflxb)    00000
   Print frequency            1
   Summary level              0
   Summary frequency          1
End Manne10
```

7.5 MINOS(IIS): DEBUGGING INFEASIBLE MODELS

If the linear constraints in a model cannot be satisfied, MINOS will exit with the message "The problem is infeasible". This usually implies some formulation error in the model. The printed solution shows which variables or slacks lie outside their bounds, and by how much. However, the exact cause of infeasibility may be difficult to detect.

In such cases, further analysis is provided by MINOS(IIS), a modified version of MINOS available from John Chinneck at Carleton University:

> J. W. Chinneck (1993). MINOS(IIS) 4.2 User's Manual, Report SCE-93-17, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada K1S 5B6.
>
> Phone: (613)788-5733, Fax: (613)788-5727, Email: chinneck@sce.carleton.ca.

Chapter 8

September 23, 2003

Library Subroutines

This chapter describes additional library subroutines that were not discussed in the previous chapter on `minoss`. These are as follows:

```
mititle  Obtain MINOS version/date string if you want it.
mistart  Initialize MINOS.  CALL THIS BEFORE OTHER ROUTINES.
micore   Estimate length of work array z(*).
minos    Alternative solve routine to minoss allowing constraint bounds
         the same as SNOPT.  (But it calls \minoss.)

micmps   Count MPS file rows, columns, elements.
micjac   Count Jacobian elements.
mirmps   Read  an MPS file.
miwmps   Write an MPS file.
```

8.1 SUBROUTINE MITITLE

This subroutine simply defines the label `title` as a 30-character string detailing the current version of MINOS.

Specification

```
      subroutine mititle( title )
  character*30        title

  title  = 'M I N O S  5.51     (Nov 2002)'
```

8.2 SUBROUTINE MISTART

This *must* be the first subroutine called from your code, i.e. must be called before any other MINOS library subroutines (including those described in the previous chapter). It opens the default files Print Summary and also Specs if it is required. It also initialises the title and sets the Specs options to their default values.

Specification

```
subroutine mistart( iprint, isumm, ispecs )


 call mifile( 1 )      ! Open the PRINT, SUMMARY and SPECS files.
 call m1init( )        ! Set a few constants.
```

```
call mititle( title ) ! Get title.

call m1page( 1 )      ! Indicate new page, then print title.
call m3dflt( 1 )      ! Set the options to default values.
```

On entry:

ispecs says whether or not a SPECS file exists. If ispecs > 0, a file is read from the specified Fortran file number. The default value is ispecs $= 4$.

iprint says if a PRINT file is to be created. The default value is iprint $= 9$.

isumm says if a SUMMARY file is to be created. The default value is isumm $= 6$. In an interactive environment, this value usually denotes the screen.

8.3 SUBROUTINE MICORE

This subroutine estimates the amount of core required to solve a problem when the size is specified in terms of the number of rows, columns and elements, and the size of the nonlinear portion of the problem (and number of Jacobian elements if it is sparse).

Specification

```
      subroutine micore( m, n, ne, nscl, maxr, maxs,
     $                   nnobj, nncon, nnjac, nejac,
     $                   mincor )


      integer            m, n, ne, nscl, maxr, maxs,
     $                   nnobj, nncon, nnjac, nejac,
     $                   mincor
```

On entry:

m is m, the number of general constraints.

n is n, the number of variables (excluding slacks).

ne is ne, the number of nonzero entries in A (including the Jacobian for any nonlinear constraints).

nscl is the size of the scaling array ascale.

maxr is the maximum Hessian size (Hessian dimension in the SPECS).

maxs is the maximum Superbasics limit (Superbasics limit in the SPECS).

nnobj is n_1', the number of nonlinear objective variables (≥ 0).

nncon is m_1, the number of nonlinear constraints (≥ 0).

nnjac is n_1'', the number of nonlinear Jacobian variables (≥ 0). If nncon $= 0$, nnjac $= 0$. If nncon > 0, nnjac > 0.

nejac is the number of Jacobian elements (allowing for possible sparsity).

On exit:

`mincor` is the minimum size of the array `real*8` z(mincor).

8.4 SUBROUTINE MINOS

This subroutine has the same input and output parameters as `minoss`. However `minos`allows the constraint bounds bl and bu to be input in the form $\texttt{bl} \leq Ax \leq \texttt{bu}$ and $bl \leq x \leq \texttt{bu}$.

Hence, the constraints are $Ax - s = 0, \texttt{bl} \leq (x, s) \leq \texttt{bu}$, rather than $Ax + s = 0, \texttt{bl} \leq (x, s) \leq \texttt{bu}$ as in `minoss`. `pi` and the last m components of `bl`, `bu`, `hs`, `xn`, `rc` are "back to front", but the other parameters are the same as in `minoss`.

Specification

```
      subroutine minos ( start, m, n, nb, ne, nname,
     $                   nncon, nnobj, nnjac,
     $                   iobj, objadd, names,
     $                   a, ha, ka, bl, bu, name1, name2,
     $                   hs, xn, pi, rc,
     $                   inform, mincor, ns, ninf, sinf, obj,
     $                   z, nwcore )

      implicit           double precision (a-h,o-z)
      character*(*)      start
      integer            m, n, nb, ne, nname,
     $                   nncon, nnobj, nnjac, iobj,
     $                   inform, mincor, ns, ninf, nwcore
      double precision   objadd, sinf, obj
      character*8        names(5)
      integer*4          ha(ne), hs(nb)
      integer            ka(n+1), name1(nname), name2(nname)
      double precision   a(ne), bl(nb), bu(nb)
      double precision   xn(nb), pi(m), rc(nb), z(nwcore)
```

On entry:

(See the parameter list for `minoss`).

On exit:

(See the parameter list for `minoss`).

8.5 SUBROUTINE MICMPS

This subroutine is used to determine the size of the probem as measured by the MPS file.

Specification

```
      subroutine micmps( imps, m, n, ne, nint, inform )

      implicit           none
```

```
      integer            imps, m, n, ne, nint, inform
```

On entry:

`imps` is the unit number for the MPS file.

On exit:

`m` is m, the number of general constraints.

`n` is n, the number of variables (excluding slacks).

`ne` is ne, the number of nonzero entries in A (including the Jacobian for any nonlinear constraints).

`nint` is the number of integer bound types in the FIRST BOUNDS section.

`inform` = 0 if the MPS file was processed correctly = 1 if a NAME record wasn't found = 2 if a ROWS record wasn't found = 3 if a COLUMNS record wasn't found = 4 if an ENDATA record wasn't found = 5 if an unexpected END OF FILE occurred.

8.6 SUBROUTINE MICJAC

This subroutine counts how many elements are in the $m_1 x n_1''$top left-hand corner of the mxn sparse matrix defined by ha and ka.

Specification

```
      subroutine micjac( m, n, ne, nncon, nnjac, nejac,
      $                  ha, ka )


      integer            m, n, ne, nncon, nnjac, nejac
      integer            ha(ne), ka(n+1)
```

On entry:

`m` is m, the number of general constraints.

`n` is n, the number of variables (excluding slacks).

`ne` is ne, the number of nonzero entries in A (including the Jacobian for any nonlinear constraints).

`nncon` is m_1, the number of nonlinear constraints (≥ 0).

`nnjac` is n_1'', the number of nonlinear Jacobian variables (≥ 0).

`ha(ne)` is a list of row indices for each nonzero in `a(*)`.

`ka(n+1)` is a set of pointers to the beginning of each column of the constraint matrix within `a(*)` and `ha(*)`.

On exit:

`nejac` is the number of non-zero Jacobian elements.

8.7 SUBROUTINE MIRMPS

Subroutine `mirmps`inputs constraint data for a linear or nonlinear program in MPS format, consisting of NAME, ROWS, COLUMNS, RHS, RANGES and BOUNDS sections in that order. The RANGES and BOUNDS sections are optional. In the LP case, MPS format defines a set of constraints of the form $l \leq x \leq u, \texttt{b1} \leq Ax \leq \texttt{b2}$, where l and u are specified by the BOUNDS section, and $b1$ and $b2$ are defined somewhat indirectly by the ROWS, RHS and RANGES sections. `mirmps`converts these constraints into the equivalent form $Ax + s = 0, \texttt{bl} \leq (x,\, s) \leq \texttt{bu}$,

where s is a set of slack variables. This is the way MINOS deals with the data. The first n components of bl and bu are the same as l and u. The last m components are $-b2$ and $-b1$.

MPS format gives 8-character names to the rows and columns of A. One of the rows of A may be regarded as a linear objective row. This will be row $iobj$, where $iobj = 0$ means there is no such row.

The data defines a linear program if $nncon = nnjac = nnobj = 0$. The nonlinear case is the same except for a few details.

1. If $nncon = nnjac = 0$ but $nnobj > 0$, the first $nnobj$ columns are associated with a nonlinear objective function.
2. If $nncon > 0$, then $nnjac > 0$ and $nnobj$ may be zero or positive. The first $nncon$ rows and the first $nnjac$ columns are associated with a set of nonlinear constraints.
3. Let $nn = max(nnjac, nnobj)$. The first nn columns correspond to ”nonlinear variables”.
4. If an objective row is specified ($iobj > 0$), then it must be such that $iobj > nncon$.
5. ”Small” elements (below the A_ij tolerance) are ignored only if they lie outside the $nncon$ by $nnjac$ Jacobian, i.e. outside the top-left corner of A.
6. No warning is given if some of the first nn columns are empty.

Specification

```
      subroutine mirmps( imps, maxm, maxn, maxnb, maxne,
     $                   nncon, nnjac, nnobj,
     $                   m, n, nb, ne, nint,
     $                   iobj, objadd, names,
     $                   a, ha, ka, bl, bu, name1, name2,
     $                   hint, hs, xn, pi,
     $                   inform, ns, z, nwcore )

      implicit           double precision (a-h,o-z)
      character*8        names(5)
      integer            ha(maxne) , hint(maxn)  , hs(maxnb)
      integer            ka(maxn+1), name1(maxnb), name2(maxnb)
      double precision   a(maxne)  , bl(maxnb)   , bu(maxnb)
      double precision   xn(maxnb) , pi(maxm)    , z(nwcore)
```

On entry:

`imps` is the unit containing the MPS file. On some systems, it may be necessary to open file *imps* before calling `mirmps`.

`maxm` is an overestimate of the number of rows in the ROWS section of the MPS file.

`maxn` is an overestimate of the number of columns in the COLUMNS section of the MPS file.

`maxnb` is $maxm + maxn$.

`maxne` is an overestimate of the number of elements (matrix coefficients) in the COLUMNS section.

`nncon` is the number of nonlinear constraints in the problem. These must be the *FIRST* rows in the ROWS section.

`nnjac` is the number of nonlinear Jacobian variables in the problem. These must be the *FIRST* columns in the COLUMNS section.

`nnobj` is the number of nonlinear objective variables in the problem. These must be the *FIRST* columns in the COLUMNS section, overlapping where necessary with the Jacobian variables.

`names` is an array of five 8-character names. names(1) need not be specified... it will be changed to the name on the NAME record of the MPS file. names(2) is the name of the objective row to be selected from the ROWS section, or blank if `mirmps`should select the first type N row encountered. Similarly, names(3), names(4) and names(5) are the names of the RHS, RANGES and BOUNDS to be selected from the RHS, RANGES and BOUNDS sections respectively, or blank if `mirmps`should select the first ones encountered.

`z` is a workspace array of length *nwcore*. It is needed to hold the row-name hash table and a few other things.

`nwcore` is the length of $z(*)$. It should be at least $4 * maxm$.

On exit:

`m` is m, the number of general constraints.

`n` is n, the number of variables (excluding slacks).

`nb` is $nb = n + m$ (the number of bounds in `bl` or `bu`).

`ne` is ne, the number of nonzero entries in A (including the Jacobian for any nonlinear constraints).

`nint` is the number of integer variables detected.

`iobj` is the row number of the specified objective row, or zero if no such row was found.

`objadd` is a real constant extracted from row iobj of the RHS. It is zero if the RHS contained no objective entry. MINOS adds objadd to the objective function.

`names` $names(1)$–$names(5)$ contain the names of the Problem, Objective row, RHS, RANGES and BOUNDS respectively.

`a` $a(*)$ contains the ne entries for each column of the matrix specified in the COLUMNS section.

`ha` $ha(*)$ contains the corresponding row indices.

`ka` $ka(j)(j = 1ton)$ points to the beginning of column j in the parallel arrays $a(*), ha(*)$. $ka(n+1) = ne + 1$.

`bl` $bl(*)$ contains nb lower bounds for the columns and slacks. If there is no lower bound on $x(j)$, then $bl(j) = -1.0d + 20$.

`bu` $bu(*)$ contains nb lower bounds for the columns and slacks. If there is no upper bound on $x(j)$, then $bu(j) = +1.0d + 20$.

`name1,name2` $name1(*), name2(*)$ contain nb column and row names in 2a4 format. The $j - th$ column name is stored in $name1(j)$ and $name2(j)$. The $i - th$ row name is stored in $name1(k)$ and $name2(k)$, where $k = n + i$.

`hint` $hint(j) = 0$ if $x(j)$ is continuous, $= 1$ if $x(j)$ is integer.

`hs` $hs(*)$ contains an initial state for each column and slack.

`xn` $xn(*)$ contains an initial value for each column and slack.

If there is no INITIAL bounds set, $xn(j) = 0$ if that value lies between $bl(j)$ and $bu(j)$, $=$ the bound closest to zero otherwise, $hs(j) = 0$ if $xn(j) < bu(j)$, $= 1$ if $xn(j) = bu(j)$.

If there is an INITIAL bounds set, $xn(j)$ and $hs(j)$ are set as follows. Suppose the j-th variable has the name Xj, and suppose any numerical value specified happens to be 3.

		$xn(j)$	$hs(j)$
FR INITIAL Xj	3.0	3.0	-1
FX INITIAL Xj	3.0	3.0	2
LO INITIAL Xj		$bl(j)$	4
UP INITIAL Xj		$bu(j)$	5
MI INITIAL Xj	3.0	3.0	4
PL INITIAL Xj	3.0	3.0	5

`pi` $pi(*)$ contains a vector defined by a special RHS called LAGRANGE. If the MPS file contains no such RHS, $pi(i) = 0.0, i = 1 : m$.

`inform` $inform = 0$ if no fatal errors were encountered, $= 40$ if the ROWS or COLUMNS sections were empty or $iobj > 0$ but $iobj \leq nncon$, $= 41$ if `maxm`, `maxn` or `maxne` were too small.

`ns` is the number of `FX INITIAL` entries in the INITIAL bounds set.

8.8 SUBROUTINE MIWMPS

The subroutine `miwmps` writes an MPS file to file number `mps`. All parameters except `mps` are the same as for `minoss` . They are all input parameters.

Specification

```
       subroutine miwmps( mps, m, n, nb, ne, nname, names,
     $                    a, ha, ka, bl, bu, name1, name2 )

      implicit           double precision (a-h,o-z)
      character*8        names(5)
      integer            ha(ne)
      integer            ka(n+1), name1(nname), name2(nname)
      double precision   a(ne), bl(nb), bu(nb)
```

On entry:

(See the parameter list for `minoss`.)

`mps` is the unit number for the MPS file.

`m` is m, the number of general constraints.

`n` is n, the number of variables (excluding slacks).

`nb` is $nb = n + m$ (the number of bounds in `bl` or `bu`).

`ne` is ne, the number of nonzero entries in A (including the Jacobian for any nonlinear constraints).

`nname` is the number of column and row names provided in the arrays `name1` and `name2`.

`names(5)` is a set of 8-character names for the problem, the linear objective, the rhs, the ranges and bounds. (This is a hangover from MPS files. The names are used in the printed solution and in some of the basis files.)

`a(ne)` is the constraint matrix (Jacobian), stored column-wise.

`ha(ne)` is a list of row indices for each nonzero in `a(*)`.

`ka(n+1)` is a set of pointers to the beginning of each column of the constraint matrix within `a(*)` and `ha(*)`. It is essential that `ka(1)` $= 1$ and `ka`$(n + 1) = ne + 1$. (See also the discussion in Section 7.1)

`bl(nb)` is the lower bounds on the variables and slacks (x, s).

The first n entries of `bl`, `bu`, `hs` and `xn` refer to the variables x. The last m entries refer to the slacks s.

`bu(nb)` is the upper bounds on (x, s).

`name1(nname), name2(nname)` are integer arrays.

(See also the discussion in Section 7.1)

Chapter 9

September 23, 2003

Examples

The following sections define some example problems and show the input required to solve them using MINOS. The last example in section 8.4 is test problem MANNE as supplied on the distribution tape. For this example we also give the output produced by MINOS.

As the examples show, certain Fortran routines may be required to run a particular problem, depending on the problem and on the Fortran installation:

- A main program to allocate workspace
- Subroutine `FUNOBJ` to define a nonlinear objective function (if any)
- Subroutine `FUNCON` to define nonlinear constraint functions (if any)
- Subroutine `MATMOD` for special applications

The following input items are *always* required:

- A SPECS file
- An MPS file

Additional input may include a BASIS file and data read by the Fortran routines.

Load modules and file specifications are inevitably machine-dependent. A resident expert will be needed to install MINOS on your particular machine and to recommend job control or operating system commands. On some machines it will be possible to run *linear programs* through MINOS without compiling any routines or linking them to the MINOS code file. For nonlinear problems, some compilation and linking is unavoidable.

For some installations it may also be convenient to have your own copy of subroutine `MIFILE`, to define certain file attributes in (non-standard) Fortran, rather than via operating system commands. The resident expert will know best.

Good luck! We hope the examples that follow are general enough to set you on the right track.

9.1 LINEAR PROGRAMMING

One of the classical applications of the simplex method was to the so-called *diet problem*. Given the nutritional content of a selection of foods, the cost of each food, and the consumer's minimum daily requirements, the problem is to find the combination that is least expensive. The following example is taken from Chvátal (1983).

$$\text{minimize} \quad c^T x \qquad \text{subject to} \qquad Ax \ge b, \qquad 0 \le x \le u,$$

where

$$A = \begin{pmatrix} 110 & 205 & 160 & 160 & 420 & 260 \\ 4 & 32 & 13 & 8 & 4 & 14 \\ 2 & 12 & 54 & 285 & 22 & 80 \end{pmatrix}, \qquad b = \begin{pmatrix} 2000 \\ 55 \\ 800 \end{pmatrix},$$

and

$$c = \begin{pmatrix} 3 & 24 & 13 & 9 & 20 & 19 \end{pmatrix}^T, \qquad u = \begin{pmatrix} 4 & 3 & 2 & 8 & 2 & 2 \end{pmatrix}^T.$$

Main program (not needed for some installations)

```
      DOUBLE PRECISION    Z(10000)
      DATA           NWCORE/10000/
C
      CALL MINOS1( Z,NWCORE )
      STOP
      END
```

Dummy user routines (not needed for some installations)

```
      SUBROUTINE FUNOBJ
      ENTRY FUNCON
      ENTRY MATMOD
      RETURN
      END
```

SPECS File

```
   BEGIN DIET PROBLEM
      MINIMIZE
      ROWS                     20
      COLUMNS                  30
      ELEMENTS                 50

      SUMMARY FILE              9
      SUMMARY FREQUENCY         1  * (for small problems only)}
      NEW BASIS FILE           11
   END DIET PROBLEM
```

MPS File

```
NAME          DIET
ROWS
 G   ENERGY
 G   PROTEIN
 G   CALCIUM
 N   COST
COLUMNS
     OATMEAL   ENERGY     110.0     PROTEIN      4.0
     OATMEAL   CALCIUM      2.0     COST         3.0
     CHICKEN   ENERGY     205.0     PROTEIN     32.0
     CHICKEN   CALCIUM     12.0     COST        24.0
     EGGS      ENERGY     160.0     PROTEIN     13.0
     EGGS      CALCIUM     54.0     COST        13.0
     MILK      ENERGY     160.0     PROTEIN      8.0
     MILK      CALCIUM    285.0     COST         9.0
     PIE       ENERGY     420.0     PROTEIN      4.0
     PIE       CALCIUM     22.0     COST        20.0
     PORKBEAN  ENERGY     260.0     PROTEIN     14.0
     PORKBEAN  CALCIUM     80.0     COST        19.0
RHS
     DEMANDS   ENERGY    2000.0     PROTEIN     55.0
     DEMANDS   CALCIUM    800.0
BOUNDS
 UP  SERVINGS  OATMEAL      4.0
 UP  SERVINGS  CHICKEN      3.0
 UP  SERVINGS  EGGS         2.0
 UP  SERVINGS  MILK         8.0
 UP  SERVINGS  PIE          2.0
 UP  SERVINGS  PORKBEAN     2.0
ENDATA
```

Notes on the Diet Problem

1. For small problems such as this, the SPECS file does not really need to specify certain parameters, because the default values are large enough. However, we include them as a reminder for more substantial models.

2. In the MPS file we put the objective row last. Looking ahead, this is one way of ensuring that it does not get mixed up with nonlinear constraints, whose names must appear *first* in the ROWS section.

3. The constraint matrix is unusual in being 100% *dense*. Most models have at least a few zeros in each column and in b. They would not need to appear in the COLUMNS and RHS sections.

4. MINOS takes three iterations to solve the problem. The optimal objective is $c^Tx = 92.5$. The optimal solution is $x = (\,4,\ 0,\ 0,\ 4.5,\ 2,\ 0\,)^T$ and $s = (\,0,\ -5,\ -534.5\,)^T$. The optimal dual variables are $\pi = (\,0.05625,\ 0,\ 0\,)^T$.

9.2 UNCONSTRAINED OPTIMIZATION

The following is a classical unconstrained problem, due to Rosenbrock (1960):

$$\text{minimize} \quad F(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2.$$

We use it to illustrate the data required to minimize a function with no general constraints. Bounds on the variables are easily included; we specify $-10 \le x_1 \le 5$ and $-10 \le x_2 \le 10$.

Calculation of $F(x)$ and its gradients

```
      SUBROUTINE FUNOBJ( MODE,N,X,F,G,NSTATE,NPROB,Z,NWCORE )
      IMPLICIT\ \ \ \ \ \ \ \ \ \ REAL*8(A-H,O-Z)
      DOUBLE PRECISION\ \ \ X(N), G(N), Z(NWCORE)
C
C     ROSENBROCK'S BANANA FUNCTION.
C
      X1     =     X(1)
      X2     =     X(2)
      T1     =     X2 - X1**2
      T2     =     1.0 - X1
      F      =   100.0 * T1**2   +    T2**2
      G(1)   = - 400.0 * T1 * X1    -   2.0 * T2
      G(2)   =   200.0 * T1
      RETURN
C
C     END OF FUNOBJ FOR ROSENBROCK
      END
```

SPECS File

```
BEGIN ROSENBROCK
   OBJECTIVE    =    FUNOBJ
   NONLINEAR VARIABLES        2
   SUPERBASICS LIMIT          3

   LOWER BOUND              -10.0
   UPPER BOUND               10.0

   SUMMARY FILE               9
   SUMMARY FREQUENCY          1
   ITERATIONS LIMIT          50
END ROSENBROCK
```

MPS File

```
NAME              ROSENBROCK
ROWS
 N   DUMMYROW
COLUMNS
     X1
     X2
RHS
```

```
BOUNDS
 UP  BOUND1      X1              5.0
 FX  INITIAL     X1             -1.2
 FX  INITIAL     X2              1.0
ENDATA
```

Notes on Rosenbrock's function

1. There is nothing special about subroutine `FUNOBJ`. It returns the function value $F(x)$ and two gradient values $g_j = \partial F/\partial x_j$ on every entry. If `G(1)` or `G(2)` were not assigned values, MINOS would "notice" and proceed to estimate either or both by finite differences.

2. The SPECS file apparently does not need to estimate the dimensions of the constraint matrix A, which is supposed to be void anyway. But in fact, MINOS will represent A as a $1 \times n_1$ matrix containing n_1 elements that are all zero. For very large unconstrained problems, the `COLUMNS` and `ELEMENTS` keywords must be specified accordingly.

3. The SPECS file must specify the exact number of nonlinear variables, n_1. To allow a little elbow room, the `SUPERBASICS LIMIT` must be set to $n_1 + 1$, unless it is known that some of the bounds will be active at the solution.

4. The MPS file must specify at least one row. Here it is a dummy free row (type `N` = non-binding constraint). The basis matrix will remain $B = 1$ throughout, corresponding to the slack variable on the free row.

5. The COLUMNS section contains just a list of the variable names. The RHS header card must appear, but a free row has no right-hand-side entry.

6. Uniform bounds $-10 \leq x_j \leq 10$ are specified in the SPECS file as a matter of good practice. Their presence does not imply additional work. If the `LOWER` and `UPPER BOUND` keywords did *not* appear, the variables would implicitly have the bounds $0 \leq x_j \leq \infty$, which will not always be appropriate.

7. With the uniform bounds specified, only one additional card is needed in the BOUNDS section to impose the restriction $x_1 \leq 5$.

8. The `INITIAL` bound set illustrates how the starting point $(x_1, x_2) = (-1.2, 1.0)$ is specified. These cards must appear at the end of the BOUNDS section. Since the `SUPERBASICS LIMIT` is sufficiently high, both variables will initially be superbasic at the indicated values.

9. If the `INITIAL` bound set were absent (and if no BASIS file were loaded), x_1 and x_2 would initially be nonbasic at the bound that is smaller in absolute value (with ties broken in favor of lower bounds); in this case, $x_1 = u_1 = 5$ and $x_2 = l_2 = -10$.

10. From the standard starting point shown, a quasi-Newton method with a moderately accurate linesearch takes about 20 iterations and 60 function and gradient evaluations to reach the unique solution $x_1 = x_2 = 1.0$.

9.3 LINEARLY CONSTRAINED OPTIMIZATION

Quadratic programming (QP) is a particular case of linearly constrained optimization, in which the objective function $F(x)$ includes linear and quadratic terms. There is no special way of informing MINOS that $F(x)$ is quadratic, but the algorithms in MINOS will tend to perform more efficiently

on quadratics than on other nonlinear functions. The following items are required to solve the quadratic program

$$\text{minimize} \quad F(x) = \frac{1}{2}x^TQx + c^Tx \qquad \text{subject to} \qquad Ax \le b, \quad x \ge 0$$

for the particular data

$$Q = \begin{pmatrix} 4 & 2 & 2 \\ 2 & 4 & 0 \\ 2 & 0 & 2 \end{pmatrix}, \qquad c = \begin{pmatrix} -8 \\ -6 \\ -4 \end{pmatrix}, \qquad A = \begin{pmatrix} 1 & 1 & 2 \end{pmatrix}, \qquad b = 3.$$

Calculation of quadratic term and its gradients

```
      SUBROUTINE FUNOBJ( MODE,N,X,F,G,NSTATE,NPROB,Z,NWCORE )
      IMPLICIT           REAL*8(A-H,O-Z)
      DOUBLE PRECISION   X(N), G(N), Z(NWCORE)
      COMMON    /QPCOMM/ Q(50,50)
C
C     Computation of   F = 1/2 x'Qx,  g = Qx.
C     The COMMON statement and subroutine SETQ are problem
      dependent.
C
C
      IF (NSTATE .EQ. 1) CALL SETQ( Q, 50, N )
      F      = 0.0
C
      DO 200 I = 1, N
         GRAD  = 0.0
         DO 100 J = 1, N
            GRAD  = GRAD + Q(I,J)*X(J)
  100    CONTINUE
         F     = F + X(I)*GRAD
         G(I)  = GRAD
  200    CONTINUE
C
         F     = 0.5*F
         RETURN
C
C        END OF FUNOBJ FOR QP
         END
```

SPECS File

```
BEGIN QP
   NONLINEAR VARIABLES        3
   SUPERBASICS LIMIT          3

   SUMMARY FILE               9
   SUMMARY FREQUENCY          1
   ITERATIONS LIMIT          50
END QP
```

MPS File

```
NAME          QP
ROWS
 L   A
 N   C
COLUMNS
     X1        A          1.0        C        -8.0
     X2        A          1.0        C        -6.0
     X3        A          2.0        C        -4.0
RHS
     B         A          3.0
ENDATA
```

Notes on the QP example

1. In subroutine `FUNOBJ` we assume that the array `Q(*,*)` is initialized during the first entry by another subroutine `SETQ`, which is problem-dependent. The `COMMON` statement is also problem-dependent and is included to ensure that `Q` will retain its values for later entries. (In some Fortran implementations, local variables are not retained between entries.)

2. The quadratic form will often involve only some of the variables. In such cases the variables should be ordered so that the nonzero rows and columns of Q come first, thus:

$$Q = \begin{pmatrix} \bar{Q} & \\ & 0 \end{pmatrix}.$$

3. The parameter `N` and the number of `NONLINEAR VARIABLES` would then be the dimension of $\bar{Q}$.

4. `FUNOBJ` could have computed the linear term $c^T x$ (and its gradient c). However we have included c as an objective row in the MPS file, in the same manner as for linear programs. This is more general, because c could contain entries for all variables, not just those associated with $\bar{Q}$.

5. Beware—if $c \neq 0$, the factor $\frac{1}{2}$ makes a vital difference to the function being minimized.

6. The optimal solution to the QP problem as stated is

$$x = (1.3333,\ 0.77777,\ 0.44444), \qquad \tfrac{1}{2}x^T Q x = 8.2222,$$
$$c^T x = -17.111, \qquad F(x) = -8.8888.$$

Test Problems WEAPON **and** ETAMACRO

The MINOS distribution tape contains data for these two linearly constrained problems. The SPECS file for ETAMACRO is as follows. It is set up to solve a linear form of the problem first, and then use the optimal basis as a starting point for the nonlinear form.

Linear Least Squares

Data-fitting can give rise to a *constrained linear least-squares problem* of the form

$$\text{minimize} \quad \|Xx - y\|_2 \qquad \text{subject to} \qquad Ax \ge b, \quad l \le x \le u.$$

This problem may be solved with MINOS as it stands, by coding subroutine `FUNOBJ` to compute the objective function $F(x) = \frac{1}{2}\|Xx - y\|_2^2$ and its gradient $g(x) = X^T(Xx - y)$. If X is a sparse matrix, it may be more convenient to express the problem in the form

$$\text{minimize} \quad F(r) = \frac{1}{2}r^Tr \text{ subject to } \begin{pmatrix} I & X \\ & A \end{pmatrix} \begin{pmatrix} r \\ x \end{pmatrix} \begin{pmatrix} = \\ \ge \end{pmatrix} \begin{pmatrix} y \\ b \end{pmatrix}, \; r \text{ free}, \; l \le x \le u.$$

Notes on the least-squares problem

1. As usual, the constraints in $Ax \ge b$ may include all types of inequality.

2. $r = y - Xx$ is the *residual vector* and r^Tr is the *sum of squares*.

3. The objective function is easily programmed as $F(r) = \frac{1}{2}r^Tr$ and $g(r) = r$.

4. More stable methods are known for the least-squares problem. If there are no constraints at all, several codes are available for minimizing $\|Xx - y\|_2$ when X is either dense or sparse. When there are equality constraints only ($Ax = b$), we know of one specialized method that can treat X and A as sparse matrices (Björck and Duff, 1980). For the more general case with inequalities and bounds, MINOS is one of very few systems that could attempt to solve problems in which X and A are sparse. *However*, if n (the dimension of x) is very large, MINOS will not be efficient unless almost n constraints and bounds are active at the solution.

5. If it is expected that most of the elements of x will be away from their bounds, it will be worthwhile to specify `MULTIPLE PRICE 10` (say). This will allow up to 10 variables at a time to be added to the set currently being optimized, instead of the usual 1.

The Discrete ℓ_1 Problem

An apparently similar data-fitting problem is

$$\text{minimize} \quad \|Xx - y\|_1 \qquad \text{subject to} \qquad Ax \geq b$$

where $\|r\|_1 \equiv \sum |r_i|$. However, this problem is best solved by means of the following purely *linear* program:

$$\begin{array}{lcl} & \text{maximize}_{\lambda,\ \mu} & y^T\lambda + b^T\mu \\ \text{subject to} & X^T\lambda + A^T\mu = 0, & -1 \leq \lambda_i \leq 1, \qquad \mu \geq 0. \end{array}$$

Notes on the ℓ_1 problem

1. The solution x is recovered as the dual variables, i.e., the Lagrange multipliers associated with the general constraints.

2. The optimal value of $\|Xx - y\|_1$ is the sum of the absolute values of the reduced costs associated with λ. (It is also the maximal value of $y^T\lambda + b^T\mu$.)

3. If a particular row in $Ax \geq b$ is required to be an equality constraint, the corresponding component of μ should be a free variable.

4. It does not appear simple to include the bounds $l \leq x \leq u$ except as part of $Ax \geq b$. If there are many finite bounds, it may be best to solve the original problem directly as a linear program, thus:

$$\begin{array}{ll} & \text{minimize}_{r,\ s} \quad e^Tr + e^Ts \\ \text{subject to} & \begin{pmatrix} & & A \\ I & -I & X \end{pmatrix} \begin{pmatrix} r \\ s \\ x \end{pmatrix} \begin{pmatrix} \geq \\ = \end{pmatrix} \begin{pmatrix} b \\ y \end{pmatrix}, \quad r,\ s \geq 0, \qquad l \leq x \leq u, \end{array}$$

where $e^T = (\,1 \;\; 1 \;\; \ldots 1\,)$.

9.4 NONLINEARLY CONSTRAINED OPTIMIZATION

Two example problems are described here to illustrate the subroutines and data required to specify a problem with nonlinear constraints. The first example is small, dense and highly nonlinear; it shows how the Jacobian matrix may be handled most simply (as a dense matrix) when there are very few nonlinear constraints or variables. The second example has both linear and nonlinear constraints, and illustrates most of the features that will be present in large-scale applications where it is essential to treat the Jacobian as a sparse matrix.

Problem MHW4D (Wright (1976), example 4, starting point D)

$$\text{minimize} \quad (x_1 - 1)^2 + (x_1 - x_2)^2 + (x_2 - x_3)^3 + (x_3 - x_4)^4 + (x_4 - x_5)^4$$

$$\begin{aligned} \text{subject to} \quad x_1 + x_2^2 + x_3^3 &= 3\sqrt{2} + 2, \\ x_2 - x_3^2 + x_4 &= 2\sqrt{2} - 2, \\ x_1 x_5 &= 2. \end{aligned}$$

Starting point: $x_0 = (-1, 2, 1, -2, -2)$

Notes for problem MHW4D

1. The function subroutines include code for a second problem (Wright, 1976, example 9). The parameter `NPROB` is used to branch to the appropriate calculation.
2. In subroutine `FUNOBJ`, `F` is the value of the objective function $F(x)$ and `G` contains the corresponding 5 partial derivatives.
3. In subroutine `FUNCON`, `F` is an array containing the vector of constraint functions $f(x)$, and `G` holds the Jacobian matrix; thus, the i-th row of `G` contains the partial derivatives for the i-th constraint. In this example the Jacobian is best treated as a dense matrix, so `G` is a two-dimensional array. Note that several elements of `G` are zero; they do not need to be explicitly set.
4. Subroutine `FUNCON` will be called before subroutine `FUNOBJ`. The parameter `NSTATE` is used to print a message on the very first entry to `FUNCON`. This is just a matter of good practice, since it is often convenient to compile MINOS and the function routines into an executable code file, and one can easily forget which particular function routines were used.
5. The SPECS file shown contains keywords that should in general be specified for small, dense problems (i.e., ones whose default values would not be ideal).
6. The COLUMNS section of the MPS file contains only the names of the variables, since they are all "nonlinear", and because there are no linear constraints.
7. The BOUNDS section specifies only the initial point. Uniform bounds on the variables are given in the SPECS file.
8. Since `FX` indicators are used for the `INITIAL` bounds, the `SUPERBASICS LIMIT` needs to be at least 5 in this case, plus 1 for elbow room during the optimization.
9. This example has several local minima, and the performance of MINOS is very dependent on the initial point x_0. See Wright (1976) or Murtagh and Saunders (1982) for computational details.

Problem MHW4D**; computation of the objective function**

Problem MHW4D**; computation of the constraint functions**

Problem MHW4D**; the SPECS file**

Problem MHW4D**; the MPS file**

Problem MANNE (Manne, 1979)

$$\text{maximize} \quad \sum_{t=1}^{T} \beta_t \log C_t$$

$$\begin{array}{llcll} \text{subject to} & \alpha_t K_t^b & \geq & C_t + I_t, & 1 \leq t \leq T, & \text{(nonlinear constraints)} \\ & K_{t+1} & \leq & K_t + I_t, & 1 \leq < T, & \text{(linear constraints)} \\ & gK_T & \leq & I_T, & & \end{array}$$

with various ranges and bounds.

The variables here are K_t, C_t and I_t, representing capital, consumption and investment during T time periods. The first T constraints are nonlinear because the variables K_t are raised to the power $b = 0.25$. The problem is described more fully in Murtagh and Saunders (1982), where results are given for the case $T = 100$.

The main program and subroutines shown on the following pages are part of the file `HEAD1` on the MINOS distribution tape (see sections 7.1 and 7.4). The SPECS data and MPS data are contained in the file `MANNE DATA`; they apply to the case $T = 10$.

Notes for problem MANNE

1. For efficiency, the Jacobian variables K_t are made the first T components of x, followed by the objective variables C_t. Since the objective does not involve K_t, subroutine `FUNOBJ` must set the first T components of the objective gradient to zero. The parameter `N` will have the value $2T$. Verification of the objective gradients may as well start at variable $T + 1$.
2. For subroutine `FUNCON`, `N` will be T. The Jacobian matrix is particularly simple in this example; in fact $J(x)$ has only one nonzero element per column (i.e., it is diagonal). The parameter `NJAC` will therefore be T also. It is used only to dimension the array `G`.
3. `NSTATE` enables `B`, `AT` and `BT` to be initialized on the first entry to `FUNCON`, for subsequent use in both of the function subroutines. (Remember that the first call to `FUNCON` occurs before the first call to `FUNOBJ`.) The name chosen for the labeled `COMMON` block holding these quantities must be different from the other `COMMON` names used by MINOS, as listed in section 7.3.
4. `NSTATE` is also used to produce some output on the final call to `FUNCON`.
5. The `COMMON` block `M1FILE` is one of those used by MINOS; see section 1.6. For test purposes we also use `COMMON` block `M8DIFF` to access the variable `LDERIV`.
6. The SPECS file uses keywords that you should become familiar with before running large problems. Other values will be appropriate for other applications.
7. The MPS file specifies a sparse $T \times T$ Jacobian in the top left corner of the constraint matrix. An arbitrary value of `0.1` has been used for the nonzero variable coefficients. A zero or blank numeric field would be equally good.

Problem MANNE**; main program and calculation of the objective function**

Problem MANNE**; calculation of the constraint functions**

Problem MANNE; the SPECS file

Problem MANNE; the MPS file

Problem MANNE**; the MPS file**, continued

Problem MANNE; output from MINOS

Problem MANNE**; output from MINOS**, continued

Problem MANNE**; output from MINOS**, continued

Problem MANNE**; output from MINOS**, continued

Problem MANNE**; output from MINOS**, continued

9.5 USE OF SUBROUTINE MATMOD

The following example illustrates the construction of a sequence of problems, based on the Diet problem of section 8.1. It assumes that the following cards have been added to the SPECS file:

```
CYCLE LIMIT         3
CYCLE PRINT         3
CYCLE TOLERANCE     2.0
PHANTOM COLUMNS     1    (or more)
PHANTOM ELEMENTS    3    (or more)
```

1. Solution of the original problem constitutes cycle 1.

2. After cycle 1, MATMOD will be called twice with NCYCLE = 2 and 3 respectively, denoting the beginning of cycles 2 and 3. The value of N will include the normal columns and the phantom columns; in this case, N = 6 + 1 = 7. Likewise, NE includes normal and phantom elements; in this case, NE = 24 + 3 = 27.

3. For cycle 2, we alter the cost coefficient on the variable called CHICKEN. This happens to be the second variable, but for illustrative purposes we use the MINOS subroutine M3NAME to search the list of column names to find the appropriate index. In this case, M3NAME will return the value JCHICK = 2.

4. Similarly, we use M3NAME to search the list of row names to find the index for the objective row, whose name is known to be COST. In this case, M3NAME will return the value JCOST = 11. Since rows are stored after the N columns, this means that the objective is row number JCOST−N = 4. (As it happens, this value is already available in the COMMON variable IOBJ.)

5. This example assumes that CHICKEN already had a nonzero cost coefficient, since it is not possible to increase the number of entries in existing columns. If the cost coefficient was previously zero, it would have to be entered as such in the MPS file, and the SPECS file would have to set AIJ TOLERANCE = 0.0 to prevent zero coefficients from being rejected.

6. For cycle 3, we generate one new column by calling upon the MINOS subroutine MATCOL. The PHANTOM COLUMNS and PHANTOM ELEMENTS keywords must define sufficient storage for this new column. (The estimates defined by the normal COLUMNS and ELEMENTS keywords must also allow for the phantom columns and elements.)

7. For illustrative purposes, we make use of the specified CYCLE TOLERANCE and the value of X(1) in the current solution, to decide whether to proceed with cycle 3.

8. After the call to MATCOL, the COMMON variable JNEW points to the new column. It allows us to set a finite upper bound on the associated variable. If there had been insufficient storage, or if COL(*) contained no significant elements, MATERR would have been increased from 0 to 1. Usually, this means that the sequence of cycles should be terminated (by setting FINISH = .TRUE.).

9.6 THINGS TO REMEMBER

Use the following space to record the fruits of your experience. They may be useful reminders the next time you come to run MINOS. (We suggest you use a pencil.)

Bibliography

[Bar71] R. H. Bartels (1971). A stabilization of the simplex method, *Numerische Mathematik* 16, 414–434.

[BG69] R. H. Bartels and G. H. Golub (1969). The simplex method of linear programming using the *LU* decomposition, *Communications of the ACM* 12, 266–268.

[BD80] Å. Björck and I. S. Duff (1980). A direct method for the solution of sparse linear least squares problems, *Linear Algebra and its Applications* 34, 43–67.

[BM68] J. Bracken and G. P. McCormick (1968). *Selected Applications of Nonlinear Programming*, John Wiley and Sons, New York and Toronto.

[BKM88] A. Brooke, D. Kendrick and A. Meeraus (1988). *GAMS: A User's Guide*, The Scientific Press, Redwood City, California.

[Chv83] V. Chvátal (1983). *Linear Programming*, W. H. Freeman and Company, New York and San Francisco.

[Dan51] G. B. Dantzig (1951). Maximization of a linear function of variables subject to linear inequalities, in T. C. Koopmans (ed.), *Activity Analysis of Production and Allocation*, Proceedings of Linear Programming Conference, June 20–24, 1949, John Wiley and Sons, New York, 359–373.

[Dan63] G. B. Dantzig (1963). *Linear Programming and Extensions*, Princeton University Press, Princeton, New Jersey.

[Dav59] W. C. Davidon (1959). Variable metric methods for minimization, A.E.C. Research and Development Report ANL-5990, Argonne National Laboratory, Argonne, Illinois.

[DS83] J. E. Dennis, Jr. and R. B. Schnabel (1983). *Numerical Methods for Unconstrained Optimization*, Prentice-Hall, Englewood Cliffs, New Jersey.

[Fou82] R. Fourer (1982). Solving staircase linear programs by the simplex method, *Mathematical Programming* 23, 274–313.

[FGK92] R. Fourer, D. Gay and B. Kernighan (1992). *AMPL: A Modeling Language for Mathematical Programming*, The Scientific Press, South San Francisco, California.

[Fri02] M. P. Friedlander. *A Globally Convergent Linearly Constrained Lagrangian Method for Nonlinear Optimization.* PhD thesis, Dept of Management Science and Engineering, Stanford University, 2002. http://www.stanford.edu/group/SOL/dissertations.html.

[GMW81] P. E. Gill, W. Murray and M. H. Wright (1981). *Practical Optimization*, Academic Press, London.

[GMSW79] P. E. Gill, W. Murray, M. A. Saunders and M. H. Wright (1979). Two step-length algorithms for numerical optimization, Report SOL 79-25, Department of Operations Research, Stanford University, California.

[GMSW87] P. E. Gill, W. Murray, M. A. Saunders and M. H. Wright (1987). Maintaining *LU* factors of a general sparse matrix, *Linear Algebra and its Applications* 88/89, 239–270.

[GMSW89] P. E. Gill, W. Murray, M. A. Saunders and M. H. Wright (1989). A practical anti-cycling procedure for linearly constrained optimization, *Mathematical Programming* 45, 437–474.

[Him72] D. M. Himmelblau (1972). *Applied Nonlinear Programming*, McGraw-Hill.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid and F. T. Krogh (1979). Basic Linear Algebra Subprograms for Fortran usage, *ACM Transactions on Mathematical Software* 5, 308–323 and (Algorithm) 324–325.

[Man77] A. S. Manne (1977). ETA-MACRO: A Model of Energy-Economy Interactions, in C. J. Hitch (ed.), *Modeling Energy-Economy Interactions*, Resources for the Future, Washington, DC. Also in R. Pindyck (ed.), *Advances in the Economics of Energy and Resources*, Vol. 2: *The Production and Pricing of Energy Resources*, JAI Press, Inc., Greenwich, Connecticut, 1979, 205–233.

[Man79] A. S. Manne (1979). Private communication.

[MS78] B. A. Murtagh and M. A. Saunders (1978). Large-scale linearly constrained optimization, *Mathematical Programming* 14, 41–72.

[MS82] B. A. Murtagh and M. A. Saunders (1982). A projected Lagrangian algorithm and its implementation for sparse nonlinear constraints, *Mathematical Programming Study* 16, *Algorithms for Constrained Minimization of Smooth Nonlinear Functions*, 84–117.

[Pre80] P. V. Preckel (1980). Modules for use with MINOS/AUGMENTED in solving sequences of mathematical programs, Report SOL 80-15, Department of Operations Research, Stanford University, California.

[Reid76] J. K. Reid (1976). Fortran subroutines for handling sparse linear programming bases, Report R8269, Atomic Energy Research Establishment, Harwell, England.

[Reid82] J. K. Reid (1982). A sparsity-exploiting variant of the Bartels-Golub decomposition for linear programming bases, *Mathematical Programming* 24, 55–69.

[Rob72] S. M. Robinson (1972). A quadratically convergent algorithm for general nonlinear programming problems, *Mathematical Programming* 3, 145–156.

[RK72] J. B. Rosen and J. Kreuser (1972). A gradient projection algorithm for nonlinear constraints, in F. A. Lootsma (ed.), *Numerical Methods for Nonlinear Optimization*, Academic Press, London and New York, 297–300.

[Ros60] H. H. Rosenbrock (1960). An automatic method for finding the greatest or least value of a function, *Computer Journal* 3, 175–184.

[Sau76] M. A. Saunders (1976). A fast, stable implementation of the simplex method using Bartels-Golub updating, in J. R. Bunch and D. J. Rose (eds.), *Sparse Matrix Computations*, Academic Press, New York, 213–226.

[Wolf62] P. Wolfe (1962). The reduced-gradient method, unpublished manuscript, RAND Corporation, Santa Monica, California.

[Wri76] M. H. Wright (1976). Numerical methods for nonlinearly constrained optimization, PhD thesis, Computer Science Department, Stanford University, California.

Appendix A

September 23, 2003

System Information

A.1 DISTRIBUTION FILES

The MINOS source code and test problems are distributed as a set of Fortran and data files.

- For installation instructions, please see file `m1minos.doc`.
- Certain other `*.doc` files give information for specific machines.
- File `readme` lists changes not documented elsewhere.

Troubleshooting

If you encounter difficulty with compiling or linking, please check the following items. The Fortran files are referred to here by names of the form `*.for`. (On Unix systems, they are renamed `*.f`.)

1. Most current machines require double-precision arithmetic. Check that the Fortran files use appropriate declarations. For example, file `mi00main.for` should contain the line
   ```
         implicit           double precision (a-h,o-z)
   ```
   Single precision is correct on a few machines (notably Cray and Convex). These use
   ```
         implicit           real (a-h,o-z)
   ```
   throughout.

2. File `mi00main.for` declares an array `z(nwcore)` for MINOS to use as workspace. Make `nwcore` as large as possible, bearing in mind the maximum problem size that is likely to be encountered. Very roughly, linear programs with m rows may require `nwcore` $\geq 100m$.

3. File `mi05funs.for` contains nonlinear function routines for the supplied test problems. Use this file initially to run the test cases, but replace it later with your own functions.

4. On most machines, use file `mi10unix.for`. Check a few machine-dependencies in the following subroutines. The requirements are described in the source code.
   `m1open` opens files.
   `m1init` sets the machine precision, `eps`. Typically $2^{-52} =$ `2.22d-16` in IEEE arithmetic.
   `m1cpu` calls the system timer. On some Unix systems, the timer is `etime`. If the name is unknown, set `time = -1.0` as shown in the source code.

5. For DEC OpenVMS systems, use file `mi10vms.for`. All machine-dependent subroutines are ready to go. In addition, `minos2` uses dynamic memory allocation.

6. In file `mi35inpt.for`, subroutine `m3hash` is suitable for most machines. In rare cases it may need to be altered if MPS data files are not input correctly. Again, the requirements are described in the source code.

A.2 SOURCE FILES

The Fortran source code is divided into several files, each containing several subroutines or functions. The naming convention used should minimize the risk of a clash with user-written routines.

`mi00main.for` Main program for Stand-alone MINOS.

```
Program MINOS
```

`mi05funs.for` Function routines for test problems.

```
funobj   funcon   matmod
t2obj    t3obj    t4obj    t4con    t5obj    t6con    t7obj
```

`mi10unix.for` Machine-dependent routines. (Use `mi10vms.for` for OpenVMS.)

```
minoss   minos1   minos2   minos3
mifile   mispec   misolv
m1clos   m1envt   m1init
m1open   m1page   m1time   m1timp   m1cpu
```

`mi15blas.for` Basic Linear Algebra Subprograms (a subset).

```
dasum    daxpy    dcopy    ddot     dnrm2    dscal    idamax
```

These routines are members of the Level 1 BLAS (Lawson, *et al.*, 1979). It may be possible to replace them by versions that have been tuned to your particular machine.

Single-precision versions of MINOS use `sasum`, `saxpy`, etc.

```
dddiv    ddscl    dload    dnorm1
hcopy    hload    icopy    iload    iload1
```

These are additional utility routines that could be tuned to your machine. `dload` is used the most, to set a vector to zero.

`mi20amat.for` Core allocation and constraint matrix routines.

```
m2core   m2amat   m2aprd   m2apr1   m2apr5
m2crsh   m2scal   m2scla   m2unpk   matcol
```

`mi25bfac.for` Basis factorization routines.

```
m2bfac   m2bmap   m2belm   m2bsol   m2sing
lu1fac   lu1fad   lu1gau   lu1mar   lu1pen
lu1max   lu1or1   lu1or2   lu1or3   lu1or4
lu1pq1   lu1pq2   lu1pq3   lu1rec
lu6chk   lu6sol   lu7add   lu7elm   lu7for   lu7zap   lu8rpc
```

`mi30spec.for` SPECS file input.

```
miopt    miopti   mioptr   m3char   m3dflt   m3key
m3file   oplook   opnumb   opscan   optokn   opuppr
```

`mi35inpt.for` MPS file input.

```
m3getp   m3hash   m3imov
m3inpt   m3mpsa   m3mpsb   m3mpsc   m3read
```

`mi40bfil.for` BASIS file input/output and SOLUTION printing.

```
m4getb   m4chek   m4id     m4name   m4inst   m4load   m4oldb
m4savb   m4dump   m4newb   m4pnch   m4rc     m4infs
m4rept   m4soln   m4solp   m4stat
```

`mi50lp.for` Primal simplex method.

```
m5bsx    m5chzr   m5dgen   m5frmc   m5hs     m5log    m5lpit
m5pric   m5rc     m5setp   m5setx   m5solv
```

`mi60srch.for` Linesearch and function evaluation.

```
m6dmmy   m6fcon   m6fobj   m6fun    m6fun1   m6grd    m6grd1
m6dobj   m6dcon   m6srch   srchc    srchq
```

`mi65rmod.for` Maintaining the quasi-Newton factor R

```
m6bfgs   m6bswp   m6radd   m6rcnd   m6rdel
m6rmod   m6rset   m6rsol   m6swap
```

`mi70nobj.for` Nonlinear objective; reduced-gradient algorithm.

```
     m7bsg    m7chkd   m7chkg   m7chzq   m7fixb
     m7rg     m7rgit   m7sdir   m7sscv
```

`mi80ncon.for` Nonlinear constraints; projected Lagrangian algorithm.

```
     m8ajac   m8augl   m8aug1   m8chkj   m8prtj   m8sclj
     m8setj   m8viol
```

`minosl.for` For installations solving linear programs only.

```
     Program MINOSL
     funobj   funcon   etc. (dummy entries)
```

The last file `minosl.for` is included as a substitute for files `mi00main.for`, `mi60srch.for`, `mi65rmod.for`, `mi70nobj.for`, `mi80ncon.for`, if MINOS is to be used to solve linear programs only. It reduces the compiled code size by about 100K bytes. It is recommended for use on microcomputers and machines that do not have virtual memory.

A.3 COMMON BLOCKS

Certain Fortran `COMMON` blocks are used in the MINOS source code to communicate between subroutines. Their names are listed below.

```
m1env    m1eps    m1file   m1savz   m1tim    m1word
m2file   m2len    m2lu1    m2lu2    m2lu3    m2lu4    m2mapa   m2mapz
m2parm
m3len    m3loc    m3mps1   m3mps2   m3mps3   m3mps4   m3mps5   m3scal
m5len    m5loc    m5freq   m5inf    m5lobj   m5log1   m5log2   m5log3
m5log4   m5lp1    m5lp2    m5prc    m5step   m5tols
m7len    m7loc    m7cg1    m7cg2    m7conv   m7phes   m7tols
m8len    m8loc    m8al1    m8al2    m8diff   m8func   m8save   m8veri
cycle1   cycle2   cyclcm
```

A complete listing of the `COMMON` blocks and their contents appears in subroutine `minos3`. (Also see Section 2.6.) It may be convenient to make use of these occasionally; for example,

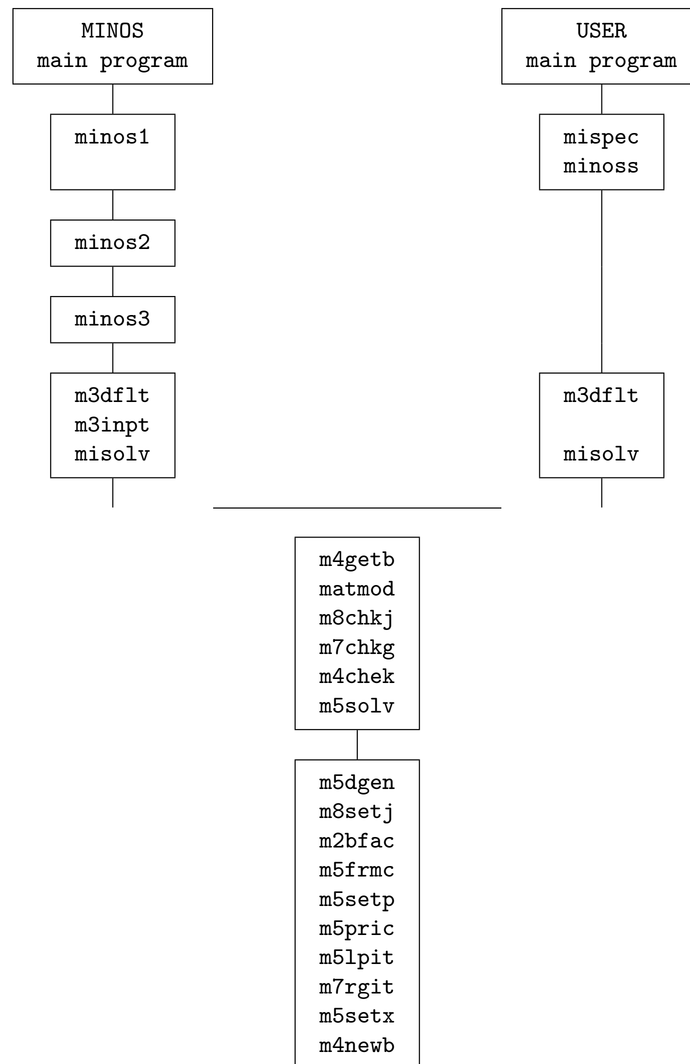
```
common    /m1file/ iread,iprint,isumm
```

gives the unit numbers for the PRINT file and the SUMMARY file.

As supplied, MINOS does not use blank `COMMON`. However, in some installations it may be desirable to store the workspace array `Z` there.

A.4 SUBROUTINE STRUCTURE

The following picture illustrates the top levels of the subroutine hierarchy for Stand-alone MINOS and for user programs that call subroutine `minoss`.



1. For Stand-alone MINOS, `minos1` reads the SPECS file. For each `begin`–`end` sequence found, it allocates storage and calls `minos2`.

2. In some implementations (e.g. file `mi10vms.for`), `minos2` expands the work array `z(*)` if necessary. It then calls `minos3` to finish processing the current problem.

3. `minos3` reads an MPS file, loads a basis file (if any), and checks gradients. According to the `Cycle limit`, it then solves one or more related problems.

4. For User programs, `mispec` reads a SPECS file (if any). It must be called before `minoss`, even if no SPECS file is provided.

A.5 MATRIX DATA STRUCTURE

In the MINOS source code, the constraint matrix A is stored column-wise in sparse format in the arrays `a`, `ha`, `ka`, as defined in the specifications of subroutine `minoss` (Section **??**). The matrix I associated with the slack variables is represented implicitly. If the objective function contains linear terms $c^T x + d^T y$, then $(\, c^T \;\; d^T \,)$ is included as the `iobj`-th row of A. (See the `common` block `m5lobj` below.)

If there are nonlinear constraints, the top left-hand corner of A is loaded with the current Jacobian matrix at the start of each major iteration.

The following `common` blocks contain dimensions and other items relating to the storage of A.

```
common    /m3len / m     ,n     ,nb    ,nscl
```

`m` — m, the number of rows in A, including the linear objective row (if any).

`n` — n, the number of columns in A, possibly including c "phantom columns".

`nb` — $n + m =$ `n+m`, the total number of variables in the problem, including the slacks.

`nscl` — Either `nb` or `1`, depending on whether `Scale` has been specified or not.

```
common    /m2mapa/ ne    ,nka   ,la    ,lha   ,lka
```

`ne` — The number of nonzero elements in A, possibly including e "phantom elements".

`nka` — $n + 1 =$ `n+1`, the number of pointers in the array `ka`.

`la` — The address of `a(*)` in the work array `z(*)`.

`lha` — The address of `ha(*)` in the work array `z(*)`.

`lka` — The address of `ka(*)` in the work array `z(*)`.

Beware that for `minoss`, the arrays `a`, `ha`, `ka`, `bl`, `bu`, `name1`, `name2`, `hs`, `xn`, `pi`, `rc`, are *not* stored in `z` in this way, since they are parameters to `minoss`.

```
common    /m5len / maxr  ,maxs  ,mbs   ,nn    ,nn0   ,nr    ,nx
```

`maxr` — The `Hessian dimension`.

`maxs` — The `Superbasics limit`.

`mbs` — `m+maxs`, the maximum number of basic and superbasic variables.

`nn` — $n_1 = \max\{\texttt{nnobj}, \texttt{nnjac}\}$, the number of `Nonlinear variables`.

`nn0` — $\max\{\texttt{1}, \texttt{nn}\}$.

`nr` — The dimension of the array `r` that is used to approximate the reduced Hessian, R.

`nx` — $\max\{\texttt{mbs}, \texttt{nn}\}$.

```
common    /m5lobj/ sinf  ,wtobj ,minimz,ninf  ,iobj
```

`sinf` — The current sum of infeasibilities.

`wtobj` — The scalar w used in the composite objective technique.

`minimz` — $+1$ if the objective is to be minimized; -1 if it is to be maximized.

`ninf` — The current number of infeasibilities.

`iobj` — The row number for the linear objective. (If `iobj` is zero, there is no such row.)

```
common    /m7len / fobj  ,fobj2 ,nnobj ,nnobj0
```

`fobj` The current value of the function value `f` returned by `funobj`.

`fobj2` A temporary value of `fobj`.

`nnobj` n'_1, the number of `Nonlinear objective variables`.

`nnobj0` $\max\{1, \texttt{nnobj}\}$.

```
common    /m8len / njac  ,nncon ,nncon0,nnjac
```

`njac` The number of elements in the Jacobian.

`nncon` m_1, the number of `Nonlinear constraints`.

`nncon0` $\max\{1, \texttt{nncon}\}$.

`nnjac` n''_1, the number of `Nonlinear Jacobian variables`.