LOQO User's Manual – Version 2.27

Robert J. Vanderbei

Statistics and Operations Research
Technical Report No. SOR-96-07

February 28, 1997

Princeton University
School of Engineering and Applied Science
Department of Civil Engineering and Operations Research
Princeton, New Jersey 08544

LOQO USER'S MANUAL – VERSION 2.27

ROBERT J. VANDERBEI

ABSTRACT. LOQO is a system for solving convex optimization problems. It is based on an infeasible primal-dual interior-point method. For linear programs, it reads industry standard MPS formated input files. For convex quadratic problems, we have extended the definition of the MPS format to allow one to specify quadratic terms in the objective function. We call this extended format the QPS format.
In addition to an executable program, LOQO also comes with a subroutine library that can be used to solve general convex optimization problems. These problems are solved by forming a quadratic approximation to the problem at each iteration of the infeasible interior-point method.
This manual describes
(1) how to install LOQO on your hardware,
(2) how to formulate and solve linear programs in MPS format,
(3) how to formulate and solve quadratic programs in QPS format,
(4) how to use AMPL together with LOQO to solve linear or quadratic programs, and
(5) how to use the subroutine library to formulate and solve convex optimization problems.

1. INSTALLATION

The normal mechanism for distribution is by downloading from the following website:

`http://www.princeton.edu/~rvdb/loqoexecs.html`

At that website is a list of the supported hardware platforms. To download, simply click on the platform that is appropriate. For sake of discussion, suppose that the appropriate platform is **`SGI (IRIX 6.2) Executable`**. Then, the downloaded file will be called **`sgi_IRIX6.2.tar.gz`**. While not required, it is a good idea to put this file in an empty directory called **`loqo`**. This file is a compressed collection of files bundled together with the **`tar`** command. To expand out the original files, execute the following commands:

```
gunzip sgi_IRIX6.2.tar
tar xvf sgi_IRIX6.2.tar
```

The tar command will extract several files from **`sgi_IRIX6.2.tar`**. Here is a list of some of the files you will find.

`loqo`	An executable code, which solves linear programming problems that are presented in industry-standard MPS form (see **`afiro.mps`** for an example) and quadratic programming problems in an extension of MPS form (see **`afiro.qps`** for an example).
`loqo.c`	A file containing the main program for **`loqo`**. It is included as an example on how to use the LOQO function library.
`loqo2.c`	A modification of **`loqo.c`** illustrating how to modify the variable initialization routine and/or the stopping rule for application specific programming.

Research supported by AFOSR through grant AFOSR-91-0359 and by NSF through grant CCR-9403789.

loqo.h A header file containing the function prototypes for each function in the LOQO function library. This file must be **#include**'d in any program file in which calls to the LOQO function library are made (**loqo.c** is an example of this).

libloqo.a An archive file containing the LOQO function library.

Some of these files are regular text files while others are platform dependent binary files. The platform dependent binaries are in a subdirectory. You should link (or copy or move) them up to the current directory:

```
ln -s sgi_IRIX6.2/install .
ln -s sgi_IRIX6.2/libloqo.a .
ln -s sgi_IRIX6.2/loqo .
```

Now, to check that you have all the files, type

```
ls -l
```

If they all seem to be there, you may want to remove **sgi_IRIX6.2.tar** by typing

```
rm sgi_IRIX6.2.tar
```

since it is a fairly large file that is now redundant. The **ls** command will also show you the read/write/execute permissions on these files. Check to make sure that you have read permission on all of these files and execute permission on **loqo**, and **install**.

Before you can use LOQO you need to run **install**. To do this, simply type

```
./install
```

If you are logged in as 'root' when you execute this command, **install** will set things so that anyone with a login on your machine will be able to use the system – otherwise, only you yourself will have access to it.

You may want to have your system administrator move some of the files as follows:

```
mv loqo /usr/local/bin
mv loqo.h /usr/local/include
mv libloqo.a /usr/local/lib
```

2. Solving Linear and Quadratic Programs in MPS Format

Solving linear programs that are already encoded in MPS format is easy. For example, to solve the linear program stored in **myfirstlp.mps**, you simply type

```
loqo myfirstlp.mps
```

Loqo will display on your screen an iteration log giving information regarding the solution process. When it is done it will put the optimal solution (primal, dual and reduced costs) into a file called **myfirstlp.out** (which is derived from the **myfirstlp** on the **NAME** line of **myfirstlp.mps**). The solution file can then be perused using any file editor (such as **vi** or **emacs**).

Standard UNIX features can be used with **loqo**. For example, if you want to save the iteration log into a file called say **myfirstlp.log**, simply type

```
loqo myfirstlp.mps > myfirstlp.log
```

If you want to time the entire solution process, use

```
time loqo myfirstlp.mps
```

2.1. MPS File Format. Input files follow the standard MPS format (for a detailed description, see [2]) for linear programs and are an extension of this format in the case of quadratic programs. The easiest way to describe the format is to look at an example. Consider the following quadratic program:

$$\text{minimize } 3x_1 - 2x_2 + x_3 - 4x_4 + \frac{1}{2}(x_3 - 2x_4)^2$$

$$\begin{array}{rrrrcr} x_1 & +x_2 & -4x_3 & +2x_4 & \geq & 4 \\ -3x_1 & +x_2 & -2x_3 & & \leq & 6 \\ & +x_2 & & -x_4 & = & -1 \\ x_1 & +x_2 & -x_3 & & = & 0 \end{array}$$

$$x_1 \text{ free}, \quad -100 \leq x_2 \leq 100, \quad x_3, x_4 \geq 0.$$

The input file for this quadratic program looks like this:

```
          1         2         3         4         5         6
12345678901234567890123456789012345678901234567890123456789012345678890
NAME      myfirstlp
ROWS
 G  r1
 L  r2
 E  r3
 E  r4
 N  obj
COLUMNS
    x1        r1                1.     r2               -3.
    x1        r4                1.     obj               3.
    x2        r1                1.     r2                1.
    x2        r3                1.     r4                1.
    x2        obj              -2.
    x3        r1               -4.     r2               -2.
    x3        r4               -1.     obj               1.
    x4        r1                2.     r3               -1.
    x4        obj              -4.
RHS
    rhs       r1                4.     r2                6.
    rhs       r3               -1.
BOUNDS
 FR           x1
 LO           x2             -100.
 UP           x2              100.
QUADS
    x3        x3                1.
    x3        x4               -2.
    x4        x4                4.
ENDATA
```

Upper case labels must be upper case and represent MPS format keywords. Lower case labels could have been upper or lower case. They represent information particular to this example. Column alignment is important and so a column counter has been shown across the top (tabs are not allowed).

The ROWS section assigns a name to each row and indicates whether it is a greater than row (G), a less than row (L), an equality row (E), or a nonconstrained row (N). Nonconstrained rows refer to the linear part of the objective function.

The COLUMNS section contains column and row label pairs for each nonzero in the constraint matrix together with the coefficient of the corresponding nonzero element. Note that either one or two nonzeros can be specified on each line of the file. There is no requirement about whether one or two values are specified on a given line although the trend is to specify just one nonzero per line (this uses slightly more disk space, but disk storage space is cheap and the one-per-line format is easier to read). All the nonzeros for a given column must appear together, but the row labels within that column can appear in any order.

The RHS section is where the values of nonzero right-hand side values are given. The label "rhs" is optional.

By default all variables are assumed to be nonnegative. If some variables have other bounds, then a BOUNDS section must be included. The label FR indicates that a variable is free. The labels LO and UP indicate lower and upper bounds for the specified variable.

If the problem has quadratic terms in the objective, their coefficients can be specified by including a QUADS section. The format of the QUADS section is the same as the COLUMNS section except that the labels are column-column pairs instead of column-row pairs. Note that only diagonal and below diagonal elements are specified. The above diagonal elements are filled in automatically.

2.2. Spec Files. LOQO has only a small number of user adjustable parameters. The parameters have default values which are usually appropriate, but other values can be specified by including in the MPS file appropriate keywords and, if required, corresponding values. These keywords (and values) must appear one per line and all must appear before the **NAME** line in the MPS file. If preferred, these keywords (and values) can be put into a separate file which is then included on the **loqo** command line. For example, suppose the file containing the keywords and values is called **spec**. Then to solve **myfirstlp.mps** using this specfile, you'd type:

```
loqo spec myfirstlp.mps
```

A list of the parameter keywords can be found in Appendix A.

2.3. Termination Conditions. Once **loqo** starts iterating toward an optimal solution, there are a number of ways that the iterations can terminate. Here is a list of the termination conditions that can appear at the end of the iteration log and how they should be interpreted:

OPTIMAL SOLUTION FOUND Indicates that an optimal solution to the optimization problem was found. The default criteria for optimality are that the primal and dual agree to 8 significant figures and that the primal and dual are feasible to the **1.0e-6** relative error level.

SUBOPTIMAL SOLUTION FOUND If at some iteration, the primal and the dual problems are feasible and at the next iteration the degree of infeasibility (in either the primal or the dual) increases significantly, then **loqo** will decide that numerical instabilities are beginning to play heavily and will back up to the previous solution and terminate with this message. The amount of increase in the infeasibility required to trigger this response is tied to the value of **INFTOL2**. Hence, if you want to force **loqo** to go further, simply set this parameter to a value larger than the default.

ITERATION LIMIT **Loqo** will only attempt 200 iterations. Experience has shown that if an optimal solution has not been found within this number of iterations, more iterations will not help. Typically, **loqo** solves problems in somewhere between 10 and 60 iterations.

PRIMAL INFEASIBLE If at some iteration, the primal is infeasible, the dual is feasible and at the next iteration the degree of infeasibility of the primal increases significantly, then **loqo** will conclude that the problem is primal infeasible. If you are certain that this is not the case, you can force **loqo** to go further by rerunning with **INFTOL2** set to a larger value than the default.

DUAL INFEASIBLE If at some iteration, the primal is feasible, the dual is infeasible and at the next iteration the degree of infeasibility of the dual increases significantly, then **loqo** will conclude that the problem is dual infeasible. If you are certain that this is not the case, you can force **loqo** to go further by rerunning with **INFTOL2** set to a larger value than the default.

PRIMAL and/or DUAL INFEASIBLE If at some iteration, the primal and the dual are infeasible and at the next iteration the degree of infeasibility in either the primal or the dual increases significantly, then **loqo** will conclude that the problem is either primal or dual infeasible. If you are certain that this is not the case, you can force **loqo** to go further by rerunning with **INFTOL2** set to a larger value than the default.

PRIMAL INFEASIBLE (INCONSISTENT EQUATIONS) This type of infeasibility is only detected at the first iteration. If **loqo** terminates here and you are sure that it should go on, set the parameter **EPSSOL** to a larger value than its default.

3. CALLING LOQO FROM WITHIN AMPL

AMPL is a language for expressing mathematical optimization problems (see [1]). For those users having AMPL installed on their system, it is easy to use LOQO to solve linear or quadratic programs that are formulated using AMPL. We shall illustrate how this is done with an example.

3.1. The Markowitz Model. Markowitz received the 1990 Nobel Prize in Economics for his portfolio optimization model in which the tradeoff between risk and reward is explicitly treated. We shall briefly describe this model in its simplest form. Given a collection of potential investments (indexed, say, from 1 to n), let R_j denote the return in the next time period on investment j, $j = 1, \ldots, n$. In general, R_j is a random variable, although some investments may be essentially deterministic.

A *portfolio* is determined by specifying what fraction of one's assets to put into each investment. That is, a portfolio is a collection of nonnegative numbers x_j, $j = 1, \ldots, n$ that sum to one. The return (on each dollar) one would obtain using a given portfolio is given by

$$R = \sum_j x_j R_j .$$

The *reward* associated with such a portfolio is defined as the expected return:

$$\mathbf{E}R = \sum_j x_j \mathbf{E}R_j .$$

Similarly, the *risk* is defined as the variance of the return:

$$\begin{aligned} \mathbf{Var}(R) &= \mathbf{E}(R - \mathbf{E}R)^2 \\ &= \mathbf{E}(\sum_j x_j (R_j - \mathbf{E}R_j))^2 \\ &= \mathbf{E}(\sum_j x_j \tilde{R}_j)^2, \end{aligned}$$

where $\tilde{R}_j = R_j - \mathbf{E}R_j$. One would like to maximize the reward while minimizing the risk. In the Markowitz model, one forms a linear combination of the mean and the variance (parametrized here by μ) and minimizes that:

$$\begin{array}{ll} \text{maximize} & \sum_j x_j \mathbf{E}R_j - \mu \mathbf{E}(\sum_j x_j \tilde{R}_j)^2 \\ \text{subject to} & \sum_j x_j = 1 \\ & x_j \geq 0 \, j = 1, 2, \ldots, n. \end{array}$$

Here, μ is a positive parameter that represents the importance of risk relative to reward. That is, high values of μ will tend to minimize risk at the expense of reward whereas low values put more weight on reward.

Of course, the distribution of the R_j's is not known theoretically but is arrived at empirically by looking at historical data. Hence, if $R_j(t)$ denotes the return on investment j at time t (in the past) and these values are known for all j and for $t = 1, 2, \ldots, T$, then expectations can be replaced by sample means as follows:

$$\mathbf{E}R_j = \frac{1}{T} \sum_{t=1}^{T} R_j(t).$$

The full model, expressed in AMPL and solved using LOQO, looks like this:

```
param n integer > 0 default 500; # number of investment opportunities
param T integer > 0 default 20;  # number of historical samples

param mu default 1.0;

param R {1..T,1..n} := Uniform01(); # return for each asset at each time
                                    # (in lieu of actual data,
                                    # we use a random number generator).

param mean {j in 1..n}              # mean return for each asset
        := ( sum{i in 1..T} R[i,j] ) / T;

param Rtilde {i in 1..T,j in 1..n}  # returns adjusted for their means
        := R[i,j] - mean[j];

var x{1..n} >= 0;

minimize linear_combination:
      mu *                                                  # weight
      sum{i in 1..T} (sum{j in 1..n} Rtilde[i,j]*x[j])^2  # variance
      -
      sum{j in 1..n} mean[j]*x[j]                           # mean
      ;

subject to total_mass:
    sum{j in 1..n} x[j] = 1;

option solver loqo;

solve;

printf: "Optimal Portfolio: \n";
```

```
printf {j in 1..n: x[j]>0.001}: "    %3d %10.7f \n", j, x[j];

printf: "Mean = %10.7f, Variance = %10.5f \n",
        sum{j in 1..n} mean[j]*x[j],
        sum{i in 1..T} (sum{j in 1..n} Rtilde[i,j]*x[j])^2;
```

If we suppose that this model is stored in a file called **markowitz.mod**, then the model can be solved by typing:

```
ampl markowitz.mod
```

The output that is produced looks like this:

```
LOQO: optimal solution (18 iterations)
primal objective -0.6442429645
  dual objective -0.6442429804
Optimal Portfolio:
     55  0.0947938
    110  0.0065181
    117  0.0798031
    133  0.0939989
    139  0.0019010
    149  0.0659400
    151  0.1004992
    204  0.0010392
    222  0.0655397
    240  0.0659591
    302  0.0065312
    311  0.0075340
    392  0.1939483
    414  0.0533822
    423  0.0087265
    428  0.0212861
    444  0.0551152
    465  0.0128579
    496  0.0385583
    497  0.0260679
Mean =  0.6489705, Variance =    0.00473
```

The adjustable parameters described in Appendix A can be set within AMPL. For example, to request a more elaborate iteration log, one would include in **markowitz.mod** the following line somewhere before the solver is called

```
option loqo_options "verbose=2";
```

Several options can be adjusted in one line. For example,

```
option loqo_options "verbose=2 itnlim=5 primal";
```

sets the verbosity level to 2, the maximum number of iterations to 5 and requests the primal-favored ordering (see Appendix A for a description of this and other parameters).

4. Solving Convex Programming Problems

One of the most common (and natural) approaches to convex programming is a technique called *sequential quadratic programming*. In this method, a quadratic approximation to the convex programming problem is formed and solved to optimality. Then, a new quadratic approximation is formed around the previously obtained QP solution. This new QP is solved to optimality and the process is continued until the optimality conditions for the convex program are met.

In LOQO, a similar but better approach is taken. Instead of solving each quadratic program to optimality, we reform the quadratic approximation at every iteration of the interior-point method. This allows us to solve the convex programming problem in approximately the same amount of time as it takes to solve just one quadratic program. The QP solver in LOQO has hooks that allow a user to modify his or her problem at the beginning of each iteration of the interior-point method. These hooks can be used to update the quadratic approximation to a convex programming problem. Furthermore, the LOQO subroutine library contains functions that hide the details from the user thereby making it relatively easy to formulate and solve convex programming problems.

In this section, we shall describe these functions and how to use them. We start with an example.

4.1. A Facility Location Model. Consider the problem of deciding on the location at which to build a warehouse given that one wishes to minimize the sum of the distances to a given set of customer locations. Let a_j, $j = 1, 2, \ldots, n$, denote the customer locations (as points in $\mathbb{R}^2$) and let x denote the as yet undetermined location of the warehouse (also a point in $\mathbb{R}^2$). The objective is to minimize the sum of the distances from the warehouse to each customer

$$\sum_{j=1}^{n} \|x - a_j\|.$$

Here, the norm $\|\cdot\|$ denotes the Euclidean distance. We should note that had we wished to minimize the sum of the *squared* distances, then the solution would be very simple. Indeed, x would be the centroid of the customer locations

$$x = \frac{\sum_{j=1}^{n} a_j}{n}.$$

However, transportation costs are better modeled as proportional to distance, not squared distance, and so the centroid is not the best location for the warehouse.

As a specific example, suppose that there are 3 customers and their locations are $(0, -1)$, $(0, 1)$, and $(10, 0)$. We can use LOQO to solve this convex optimization problem. The idea is to put the linear parts of the problem into an MPS file and then to specify the nonlinearities separately. The easiest way to make an MPS file is to use a modeling language, such as AMPL. Here is the AMPL formulation of the facility location problem (which is stored in **`fac_loc.mod`**):

```
param n := 3;   # number of facility
param d := 2;   # dimension of geographical space

set CUSTS := 1..n;      # set of facilities

param a{CUSTS, 1..2};   # geographical coordinates of each facility

var x{1..2};

minimize tot_dist:
    sum {j in CUSTS} sqrt( sum {i in 1..2} (x[i] - a[j,i])^2 );
```

```
data;

param a: 1   2   :=
      1  0  -1
      2  0   1
      3 10   0 ;

option presolve 0;
option auxfiles rc;
write mfac_loc;
```

Note that the AMPL model does not invoke any solver (such as LOQO). Instead, it simply writes out some files. Setting the **presolve** option to zero directs AMPL not to do any problem simplification. We want it just exactly the way we've formulated it. The last line in the AMPL file instructs AMPL to write an MPS file called **fac_loc.mps**. Here is the file it produces:

```
NAME          fac_loc
ROWS
 N  R0001
COLUMNS
    C0001     R0001     0
    C0002     R0001     0
RHS
BOUNDS
 FR BOUND     C0001
 FR BOUND     C0002
ENDATA
```

It simply says that the problem has two variables, called C0001 and C0002 and that both of these variables are free variables. It also sets up that linear part of the objective function, which is called R0001. But it puts all zeros in for the data.

The nonlinearities must be specified separately. We do this by preparing a short C program (which is stored in **fac_loc.c**):

```
#include <math.h>
#include "loqo.h"

#define n 3  /* number of customers */
#define d 2  /* dimension of Euclidean space */

void eucl (  double *z, double *param,
             double *pval, double *grad, double **hessian );

main( int argc, char *argv[])
{
        LOQO   *lp;
        int    i,j;

        char *collabs[d]          =   { "C0001", "C0002" };
```

```
        double obj_param[n][d]    = { {  0.0, -1.0 },
                                      {  0.0,  1.0 },
                                      { 10.0,  0.0 } } ;

        lp = openlp();

        argc--; argv++;
        readlp(argc,argv,lp);

        for (j=0; j<n; j++) {
            nlobjterm(eucl, 2, collabs, 2, obj_param[j]);
        }

        solvelp( lp );

        writesol(lp, "fac_loc.out");
    }
```

In the function **`main()`**, the first order of business is to call **`openlp()`** which sets up a data structure (called **`LOQO`**) to hold an optimization problem and returns a pointer to this data structure (called **`lp`**).

Next, **`readlp()`** is called. It is passed a count **`argc`** of how many command-line arguments there were when main was called together with an array of strings **`argv`** containing each of these command-line arguments. It also gets a pointer to the **`LOQO`** data structure that was created with the call to **`openlp()`**. **`Readlp()`** expects each of the command-line arguments to be the names of files. It attempts to read these files one at a time in the order that they appear. These files, taken in their totality, should constitute a standard MPS file description of the linear part of the problem. (It is a feature of LOQO that it allows one to break an MPS file into several pieces and then to list them separately on the command line with the understanding the **`readlp()`** will read each fragment in order.) The linear information that it reads in these files is stored in the **`LOQO`** data structure to which **`lp`** points.

Next we must define the nonlinearities. There are n nonlinear terms in the objective function. For each nonlinear term of the form

$$f(x_{j_1}, x_{j_2}, \ldots, x_{j_k}; p_1, \ldots, p_m),$$

we must make a call to the function **`nlobjterm()`**. This function has five arguments:

(1) A pointer to the function that computes the values of f, its gradient, and its hessian.
(2) An integer indicating the number of variables, k, appearing in the function f.
(3) An array of strings, each string being a column label from the MPS file associated with the relevant variable.
(4) An integer indicating the number of auxillary parameters, m, on which the function depends.
(5) An array of double precision numbers containing the parameters $p_1, p_2, \ldots, p_m$.

In the facility location problem, the function is the Euclidean distance from a given point. The parameters are the coordinates of the given point.

After finishing the loop that sets up the n terms in the nonlinear objective function, we are ready to solve the convex optimization problem. We do this by calling **`solvelp()`** passing it the **`LOQO`** pointer **`lp`**. As its name indicates, the function **`solvelp()`** solves the optimization problem. The optimal values of the primal and the dual variables are stored in the **`LOQO`** data structure to which **`lp`** points.

The function **`writesol()`** creates a file containing the primal and the dual solution values.

For each nonlinear function that appears in the model formulation, one must prepare a C function that evaluates this function together with its gradient and hessian. In the facility location model, there is only one nonlinear function: the

Euclidean distance function

$$f(z) = \|z - a\| = \sqrt{(z_1 - a_1)^2 + (z_2 - a_2)^2}.$$

Hence, we must define one corresponding C function. We've called it **`eucl`**. The C function must have five arguments:

(1) An array containing the arguments to the mathematical function.
(2) An array containing the parameters that define the mathematical function.
(3) A pointer to a double precision variable which, on return, will contain the value of the nonlinear function.
(4) An array which will, on return, contain the gradient of the nonlinear function.
(5) A two-dimensional array which will contain the Hessian of the nonlinear function.

The code fragment shown above contains a prototype for **`eucl`** and passes a pointer to this function to **`nlobjterm`** but does not define this function. Here is its definition (also found in **`fac_loc.c`**):

```
void eucl ( double *z, double *param,
            double *pval, double *grad, double **hessian )
{
        double z0, z1, val, val3;

        z0 = z[0] - param[0];
        z1 = z[1] - param[1];

        val = sqrt(z0*z0 + z1*z1);
        val3 = val*val*val;

        *pval = val;                                    /* f   */

        grad[0] = z0/val;                               /* f'  */
        grad[1] = z1/val;                               /* f'  */

        hessian[0][0] = z1*z1/val3;                     /* f'' */
        hessian[0][1] = hessian[1][0] = -z0*z1/val3;    /* f'' */
        hessian[1][1] = z0*z0/val3;                     /* f'' */
}
```

The two code fragments can be found together in a file called **`fac_loc.c`** and the MPS file is stored in a file called **`fac_loc.mps`**. These two files contain all the information necessary to describe a particular instance of the facility location model.

We are now ready to solve this instance of the model. To this end, let us assume that the files **`loqo.h`** and **`libloqo.a`** sit in the same directory as the two model files. Then we can solve the problem by typing

```
cc -O fac_loc.c libloqo.a -lm -o fac_loc
fac_loc fac_loc.mps
```

The iteration log produced by this run should look something like this:

```
variables: non-neg      0,  free        2,  bdd          0,  total        2
constraints: eq         0,  ineq        0,  ranged       0,  total        0
nonzeros:    A          0,  Q           4
nonzeros:    L          3,  arith_ops               6
```

```
------------------------------------------------------------------------
      |         Primal            |           Dual            | Sig
Iter  |  Obj Value      Infeas    |   Obj Value      Infeas   | Fig  Status
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   1   1.2000000e+01   2.83e+02     1.2000000e+01   1.58e+02    30
   2   1.1742154e+01   1.42e+01     1.1819860e+01   8.60e+00     2
   3   1.1732180e+01   7.13e-01     1.1742578e+01   4.55e-01     3
   4   1.1732051e+01   3.56e-02     1.1732669e+01   2.29e-02     4
   5   1.1732051e+01   1.78e-03     1.1732082e+01   1.15e-03     6
   6   1.1732051e+01   8.91e-05     1.1732052e+01   5.74e-05     7
   7   1.1732051e+01   4.45e-06     1.1732051e+01   2.87e-06     8    PF DF
----------------------
OPTIMAL SOLUTION FOUND
```

The file, **fac_loc.out** looks like this:

```
COLUMNS SECTION
   index       label  primal_val reduced_cst    lower_bd    upper_bd
       0    X1        5.7735e-01  0.0000e+00   -Infinity    Infinity
       1    X2       -0.0000e+00  0.0000e+00   -Infinity    Infinity
ROWS SECTION
   index       label    dual_val  row_actvty rght_hnd_sd       range
ENDOUT
```

Before leaving this example, we should note that an improved version of **fac_loc.c** would allow the number of customers n to be a variable and would read from a separate data file the n location vectors.

4.2. A Constrained Facility Location Model. Now, suppose that in addition to the desire to minimize the sum of the distances to the customers, we also have constraints that the warehouse can be no further than a fixed distance from each of several (say m) factories that supply goods to the warehouse. Let b_i denote the position in $\mathbb{R}^2$ of the i-th factory and let d_i denote the maximum distance that the i-th factory can be from the warehouse. Then the i-th distance constraint can be written as

$$\|x - b_i\| \le d_i$$

or as

$$\|x - b_i\|^2 \le d_i^2.$$

It turns out that the second representation is better and so we add constraints of this type to the model developed before. The updated AMPL model, stored in **fac_loc2.mod** looks like this:

```
param m := 2;
param n := 3;
param d := 2;

set CUSTS := 1..n;
set FACTS := 1..m;

param a{CUSTS, 1..2};
param b{FACTS, 1..2};
```

```
param dist{FACTS};

var x{1..2};

minimize tot_dist:
    sum {j in CUSTS} sqrt( sum {i in 1..2} (x[i] - a[j,i])^2 );

subject to not_far{k in FACTS}:
    sum {i in 1..2} (x[i] - b[k,i])^2 <= dist[k]^2;

data;

param a: 1   2   :=
      1  0  -1
      2  0   1
      3 10   0 ;

param b: 1   2   :=
      1  5  -2
      2  6   1 ;

param dist :=
      1  3
      2  2 ;


option presolve 0;
option auxfiles rc;
write mfac_loc2;
```

and the MPS file it produces looks like this:

```
NAME          fac_loc2
ROWS
 L  R0001
 L  R0002
 N  R0003
COLUMNS
    C0001     R0001     0
    C0001     R0002     0
    C0001     R0003     0
    C0002     R0001     0
    C0002     R0002     0
    C0002     R0003     0
RHS
    B         R0001     9
    B         R0002     4
```

```
BOUNDS
 FR BOUND     C0001
 FR BOUND     C0002
ENDATA
```

Nonlinear terms appearing in constraints are specified in much the same manner as for nonlinear objective terms. The function **main()** in **fac_loc2.c** is almost the same as **main()** in **fac_loc.c** except that the following lines must be inserted after the call to **readlp()** and before the called to **solvelp()**:

```
for (i=0; i<m; i++) {
    sprintf(row, "R000%1d", i+1);
    nlconstr(eucl2, row, 2, collabs, 2, const_param[i]);
}
```

The function **nlconstr()** is similar to **nlobjterm()**. It has six arguments, five of which are the same as before. The new argument is inserted between the first two old arguments. It is a string indicating the row label (from the MPS file) in which the nonlinear objective term appears. Here is a list of the six parameters and their meanings:

(1) A pointer to the function that computes the values of $f()$, its gradient, and its hessian.
(2) A string indicating the row label from the MPS file associated with this nonlinear constraint term.
(3) An integer indicating the number of variables (k) appearing in the function $f()$ has.
(4) An array of strings, each string being a column label from the MPS file associated with the relevant variable.
(5) An integer indicating the number of auxillary parameters (m) on which the function depends.
(6) An array of double precision numbers containing the parameters $p_1, p_2, \ldots, p_m$.

Of course, the number of factories **m** is 2 and their positions are at at $(5, -2)$ and $(6, 1)$. As before, the positions are given in the parameters that are passed to the nonlinear function. In this case these parameters are stored in an array called **const_param**. Hence, the following lines must appear near the top of **main()**:

```
double const_param[m][2] = { {  5.0, -2.0 },
                             {  6.0,  1.0 } };
```

and the number of factories **m** could be **#define**'ed:

```
#define m 2  /* number of factories */
```

The function **eucl2()** which computes the value, gradient, and hessian of the square of the Euclidean distance from a specified point must be prototyped at the top of the file and must be defined somewhere, say at the end of the file. Its definition is as follows:

```
void eucl2( double *z, double *param,
            double *pval, double *grad, double **hessian )
{
        double z0, z1, val;

        z0 = z[0] - param[0];
        z1 = z[1] - param[1];

        val = z0*z0 + z1*z1;

        *pval = val;                                    /* f   */
```

```
        grad[0] = 2*z0;                                 /* f'  */
        grad[1] = 2*z1;                                 /* f'  */

        hessian[0][0] = 2;                              /* f'' */
        hessian[0][1] = hessian[1][0] = 0;              /* f'' */
        hessian[1][1] = 2;                              /* f'' */
}
```

The entire model can be found in the files **fac_loc2.mps** and **fac_loc2.c** distributed with the LOQO software. The model is compiled and run as before. This time it takes 12 iterations to converge to a solution having 9 figures of agreement between the primal and dual objective functions. Looking at the solution file we see that the optimal location for the warehouse is at (4.0795, 0.44187).

4.3. A Utility Approach to Portfolio Optimization. Consider the Markowitz model for portfolio optimization discussed in Section 3.1. The VonNeumann/Morgenstern theory of utility suggests that one should optimize the expected utility of the return instead of the linear combination of risk and reward described earlier. Using the utility approach with a log utility function, the portfolio optimization model can be formulated as follows:

```
param n integer > 0 default 500; # number of investment opportunities
param T integer > 0 default 20;  # number of historical samples

param R {1..T,1..n} := Uniform01(); # return for each asset at each time

var x{1..n} >= 0;
var y{1..T};

minimize negative_expected_utility:
             -sum{i in 1..T} log(y[i]);

subject to total_mass:
    sum{j in 1..n} x[j] = 1;

subject to definitional_constraints {i in 1..T}:
    y[i] = sum{j in 1..n} R[i,j]*x[j];

option auxfiles "c";               # to get row and column files
write mutility;                       # output into MPS format file
```

Here, we have used AMPL to define the convex optimization problem, but as before we have not indicated that we want AMPL to call any solver. Instead, we are simply asking AMPL to write an MPS file containing the linear part of the model (the **write mutility** statement tells AMPL to create an MPS file called **utility.mps**). We've also requested that AMPL output an auxiliary file called **utility.col** containing a list of the variable names in the order in which they appear in the MPS file. The reason is that currently the AMPL/solver interface routines provided with the AMPL software do not produce functions that give values in the Hessian matrix (except if the problem is a QP). But, as we've seen, we need this information. Therefore, we must still produce it by hand. This is not too hard. A quick perusal of the **utility.col** shows that the first 20 variables correspond to **y[1]** to **y[T]**. Then, looking in **utility.mps** we see that the MPS file refers to these variables as **C0001** to **C0020**. Therefore, the C program file, called say **utility.c** should look like this:

```
#include <math.h>
#include "loqo.h"

#define n 20

void logutil (  double *z, double *pval, double *grad, double **hessian );

main( int argc, char *argv[])
{
    LOQO   *lp;
    int    j;

    char *collabs[n][1]       = { { "C0001" },
                                  { "C0002" },
                                  { "C0003" },
                                  { "C0004" },
                                  { "C0005" },
                                  { "C0006" },
                                  { "C0007" },
                                  { "C0008" },
                                  { "C0009" },
                                  { "C0010" },
                                  { "C0011" },
                                  { "C0012" },
                                  { "C0013" },
                                  { "C0014" },
                                  { "C0015" },
                                  { "C0016" },
                                  { "C0017" },
                                  { "C0018" },
                                  { "C0019" },
                                  { "C0020" } } ;
    double obj_shifts[1]      =   {    0.0  } ;

    lp = openlp();

    argc--; argv++;
    readlp(argc,argv,lp);

    for (j=0; j<n; j++) {
        nlobjterm(logutil, 1, collabs[j], obj_shifts);
    }

    solvelp( lp );

    writesol(lp, "utility.out");
}

void logutil ( double *z, double *pval, double *grad, double **hessian )
```

```
{
    double z0, val;

    z0 = z[0];

    val = -log(z0);

    *pval = val;                                    /* f   */

    grad[0] = -1/z0;                                /* f'  */

    hessian[0][0] = 1/(z0*z0);                      /* f'' */
}
```

The following commands compile and run the model;

```
cc -O utility.c libloqo.a -lm -o utility
utility utility.mps
```

This model converges in just 16 iterations (taking less than one second on a typical 10 MFLOP workstation).

5. Modeling Hints

Every attempt has been made to make LOQO as robust as possible on a wide spectrum of problem instances. However, there are certain suggestions that the modeler should take heed of to obtain maximum performance.

5.1. Artificial Variables. *Splitting free variables.* Some existing codes for solving linear programs are unable to handle free variables. As a consequence, many problems have been formulated with free variables split into the difference between two nonnegative variables. This trick does not present any difficulties for algorithms based on the simplex method, but it does tend to cause problems for interior-point methods and, in particular, for LOQO. Since LOQO is designed to be able to handle problems with free variables, we suggest that they be left as free variables and indicated as such in the input file.

Artificial Big-M Variables. Some problems have artificial variables added to guarantee feasibility using the traditional Big-M method. Putting huge values anywhere in a problem invites numerical problems. LOQO has its own feasibility phase and so we suggest that any Big-M type artificial variables be left out.

5.2. Separable Equivalents. The algorithm implemented in LOQO only works on convex quadratic programs. This means that Q must be positive semi-definite if the problem is a minimization (and negative semi-definite if the problem is a maximization). LOQO checks this condition and prints a warning whenever it discovers a Q that violates the semidefiniteness condition.

This raises an interesting question. How do you know that a matrix Q is positive semi-definite? Generally the best way to prove this is to exhibit another matrix F for which

$$Q = F^T F.$$

Here, F does not need to be a square matrix. In fact, it is quite likely that you, the modeller, will know of an F and this F may have many fewer rows than columns. It will also most likely be sparser than Q. In this case, it is much better to replace the nonseparable quadratic term

$$\frac{1}{2} x^T Q x$$

in the objective function with an equivalent separable term

$$\frac{1}{2}y^T y$$

and simply add the following constraints to the problem:

$$Fx - y = 0.$$

Let us illustrate this concept with the Markowitz model presented in Section 3.1. By setting the verbosity level to 2, one discovers the following statistics associated with **markowitz.mod**:

```
variables: non-neg    500,  free        0,  bdd          0,  total      500
constraints: eq         1,  ineq        0,  ranged       0,  total        1
nonzeros:    A        500,  Q      250000
nonzeros:    L     125751,  arith_ops           42168001
```

The second entry on the third line gives the number of nonzeros in the matrix Q defining the quadratic terms. Here it is 250000 which is exactly 500 squared. This indicates that Q is a dense 500×500 matrix.

Now, let us consider a slight modification to the model, which we have stored in a new file called **markowitz2.mod**:

```
param n integer > 0 default 500; # number of investment opportunities
param T integer > 0 default 20;  # number of historical samples

param mu default 1.0;

param R {1..T,1..n} := Uniform01(); # return for each asset at each time
                                    # (in lieu of actual data,
                                    # we use a random number generator).

param mean {j in 1..n}              # mean return for each asset
        := ( sum{i in 1..T} R[i,j] ) / T;

param Rtilde {i in 1..T,j in 1..n}  # returns adjusted for their means
        := R[i,j] - mean[j];

var x{1..n} >= 0;
var y{1..T};

minimize linear_combination:
      mu *                                          # weight
      sum{i in 1..T} y[i]^2                         # variance
      -
      sum{j in 1..n} mean[j]*x[j]                   # mean
      ;

subject to total_mass:
    sum{j in 1..n} x[j] = 1;

subject to definitional_constraints {i in 1..T}:
    y[i] = sum{j in 1..n} Rtilde[i,j]*x[j];
```

```
option solver "/usr/people/rvdb/ampl/loqo";
option loqo_options "verbose=2";

solve;

printf: "Optimal Portfolio: \n";
printf {j in 1..n: x[j]>0.001}: "    %3d %10.7f \n", j, x[j];

printf: "Mean = %10.7f, Variance = %10.5f \n",
        sum{j in 1..n} mean[j]*x[j],
        sum{i in 1..T} (sum{j in 1..n} Rtilde[i,j]*x[j])^2;
```

If we make a timed run of this model (by typing **`time ampl markowitz2.mod`**), the first few lines of output look like this:

```
LOQO: Verbosity level (VERBOSE) = 2
variables: non-neg    500,  free       20,  bdd          0,  total      520
constraints: eq        21,  ineq        0,  ranged       0,  total       21
nonzeros:    A      10520,  Q          20
nonzeros:    L      11271,  arith_ops            256121
```

Note that there are now 20 more constraints but at the same time the number of nonzeros in Q is only 20. Furthermore, the number of arithmetic operations (which correlates closely with true run-times – at least for large problems) is only 256121 as compared with 42168001 in **`markowitz.mod`**. This suggest that the second formulation should run perhaps a hundred times faster than the first. Indeed, running both models on the same hardware platform one finds that **`markowitz2.mod`** solves in 5.95 seconds whereas **`markowitz.mod`** takes 188.75 seconds, which translates to a speedup by more than a factor of 60.

5.3. Dense Columns. Some problems are naturally formulated with the constraint matrix having a small number of columns that are significantly denser than the other columns. From an efficiency point of view, dense columns have been a red herring for interior-point methods. However, LOQO incorporates certain specific techniques to avoid the inefficiencies often encountered on models with dense columns.

Recently discovered "tricks" (which are incorporated into LOQO) have largely overcome the problems associated with dense columns, however, the user should be aware that the presense of dense columns could be the source of numerical difficulties. Often it is easy to reformulate a problem having dense columns in such a way that the new formulation avoids dense columns. For example, if variable x appears in a large number of constraints, we would suggest introducing several different variables, $x_1, \ldots, x_k$, all representing the same original variable x and using x_1 in some of the constraints, x_2 in some others, etc. Of course, $k-1$ new constraints must be added to equate each of these new variables to each other. Hence, the new problem will have $k-1$ more variables and $k-1$ more constraints, but it will have a constraint matrix that doesn't have dense columns. Often it is better to solve a slightly larger problem if the larger constraint matrix has an improved sparsity structure.

6. The Function Library

The easiest way to explain how to use the function library is to look at an example. Here is a commented listing of an abbreviated version of **`loqo.c`**:

```
#include <loqo.h>         /* defines LOQO structure and prototypes
                             functions */
#include <string.h>

main( int argc, char *argv[] )
{
    char    fname[80];    /* solution file name */
    LOQO    *lp;          /* a pointer to a LOQO structure */

    lp = openlp();        /* up to 20 problems can be open at once */

    argc--; argv++;       /* remove 'loqo' from command line args */
    readlp(argc,argv,lp); /* pass command line args to readlp */

    solve_lp( lp );       /* solve the optimization problem */

    inv_clo();            /* free up space reserved for solving systems
                             of equations */

    writesol(lp,"solution");   /* write solution into a file */

    closelp(lp);          /* close this problem */

    return(0);
}
```

The user interface to the LOQO function library is patterned after the customary file manipulation functions defined in **`stdio.h`**. That is, there is an open statement that simply returns a pointer to a structure containing all the relevant information for the other library functions and there is a close function that releases this pointer for future use. In this case, the structure containing all the relevant information is called **`LOQO`**. Here is the definition of this structure:

```
typedef struct loqo {
    int m;          /* number of rows */
    int n;          /* number of columns */
    int nz;         /* number of nonzeros */
    double *A;      /* pointer to array of nonzero values in A */
    int *iA;        /* pointer to array of corresponding row indices */
    int *kA;        /* pointer to array of indices into A (and iA)
                       indicating where each new column of A begins */
    double *b;      /* pointer to array containing right-hand side */
    double *c;      /* pointer to array containing objective function */
    double f;       /* fixed adjustment to objective function */
    double *r;      /* pointer to array containing range vector */
    double *l;      /* pointer to array containing lower bounds */
    double *u;      /* pointer to array containing upper bounds */
    int *varsgn;    /* array indicating which variables were declared to
                            be non-positive */
    char **rowlab;  /* array of strings containing row labels */
    char **collab;  /* array of strings containing column labels */
```

```
    int qnz;        /* number of nonzeros in lower triangle of Q */
    double *Q;      /* pointer to array of nonzero values of Q */
    int *iQ;        /* pointer to array of corresponding row indices */
    int *kQ;        /* pointer to array of indices into Q (and iQ)
                       indicating where each new column of Q begins */

    double *At;     /* pointer to array of nonzero values in At */
    int *iAt;       /* pointer to array of corresponding row indices */
    int *kAt;       /* pointer to array of indices into At (and iAt) */

    int *bndmark;   /* pointer to array of bound marks */
    int *rngmark;   /* pointer to array of range marks */

    double *w;      /* pointer to array containing primal surpluses */
    double *x;      /* pointer to array containing primal solution */
    double *y;      /* pointer to array containing dual solution */
    double *z;      /* pointer to array containing dual slacks */
    double *p;      /* pointer to array containing range slacks */
    double *q;      /* pointer to array containing dual range slacks */
    double *s;      /* pointer to array containing dual for ub slacks */
    double *t;      /* pointer to array containing upper bound slacks */
    double *v;      /* pointer to array containing dual for range (w) */
    double *ub;     /* pointer to array containing shifted up bounds */

    int max;        /* max = -1, min = 1 */
    double inftol;  /* infeasibility tolerance */
    int sf_req;     /* significant figures requested */
    int verbose;    /* level of verbosity */
    char name[15];  /* string containing problem name */
    char obj[11];   /* string containing objective function name */
    char rhs[11];   /* string containing right-hand side name */
    char ranges[11];/* string containing range set name */
    char bounds[11];/* string containing bound set name */

    int  (*stopping_rule)(); /* pointer to stopping rule fcn */
    void (*init_vars)();     /* pointer to initialization fcn */

    int    iter;        /* current iteration number */
    double pres;        /* primal residual (i.e. infeasibility) */
    double dres;        /* dual   residual (i.e. infeasibility) */
    int    sigfig;      /* significant figures */
    double primal_obj;  /* primal objective value */
    double dual_obj;    /* dual   objective value */

    int flag;        /* 0=unopened, 1 = opened */
} LOQO;
```

If you wish to generate your own customized reports, simply replace the call to **writesol** with a call to your own output routine. Similarly, if you wish to write your own problem generator and do not want to put its output into an MPS file but rather go straight into LOQO, then you simply need to replace the call to **readlp** by a call to one of your own functions which generates the optimization problem and stores it in **lp**.

For those unfamiliar with C structures, the various components are accessed by typing **lp->** followed by the member name. For example, the number of constraints is **lp->m** and the j-th element of the objective vector is **lp->c[j]**. Since it is inconvenient to have to include the **lp->** prefix all the time, a common trick is to copy the needed parts of the data structure into local variables of the same name. For example, here is a simplified version of the function **writesol**:

```
void  writesol( LOQO *lp, char fname[] )
{
    int     m, n, *varsgn;
    double  *x, *y, *z;
    char    **rowlab, **collab;

    int     i,j;
    FILE    *fp;

    m       = lp->m;
    n       = lp->n;
    varsgn  = lp->varsgn;
    rowlab  = lp->rowlab;
    collab  = lp->collab;
    x       = lp->x;
    y       = lp->y;
    z       = lp->z;

    for (j=0; j<n; j++) {
            x[j]  = varsgn[j]*x[j];
            z[j]  = varsgn[j]*z[j];
    }
    if ( ( fp = fopen(fname, "w") ) == NULL ) error(2,fname);
    fprintf(fp,"COLUMNS SECTION\n");
    fprintf(fp,"   index       label primal_value reduced cost\n");
    for (j=0; j<n; j++) fprintf(fp,"%8d  %10s  %10.3e  %10.3e \n",
                                   j,collab[j],x[j],z[j]);
    fprintf(fp,"ROWS SECTION\n");
    fprintf(fp,"   index       label  dual_value  \n");
    for (i=0; i<m; i++) fprintf(fp,"%8d  %10s  %10.3e \n",
                                   i,rowlab[i],y[i]);
    fclose(fp);
}
```

You should bear in mind that if you change any of the local variables that correspond to members of the **LOQO** structure and you want these changes propagated back to the calling routine, you must copy the new version back into the **LOQO** structure. For example, if **m** were to change, you'd have to put

```
lp->m = m;
```

somewhere after you changed `m`.

REFERENCES

[1] R. Fourer, B. Kernighan, and D.M. Gay. *AMPL*. Scientific Press, 1993.
[2] J.L. Nazareth. *Computer Solutions of Linear Programs*. Oxford University Press, 1987.

APPENDIX A. ADJUSTABLE PARAMETERS

Here is a list of the parameter keywords with a description of each keyword's meaning and how to use it:

BOUNDS *str* Specifies the name of the bounds set. *Str* must be a string that matches one of the bounds-set labels in the bounds section of the MPS file. The default is to use the first encountered bounds set.

DENSE *n* The ordering heuritics mentioned above are actually implemented as modifications of the usual heuristics into two-tier versions of the basic heuristic. This is necessary since the reduced KKT system is not positive semi-definite. For each column of the constraint matrix, there is an associated column in the reduced KKT system. Generally, speaking these are the tier-one columns. These tier-one columns are intended to be eliminated before the tier-two columns. However, it is sometimes possible to see tremendous improvements in solution time if a small number of these columns are assigned to tier-two. The columns whose reassignment could make the biggest impact are those columns which have the most nonzeros (i.e. dense columns). LOQO has a built in heuristic that tries to determine a reasonable threshold above which a column will be declared dense and put into tier-two. However, the heuristic can be overridden by setting **DENSE** to any value you want.

DUAL Requests that the ordering heuristic be set to favor the dual problem. This is typically prefered if the number of constraints far exceeds the number of variables or if the problem has a large number of dense columns. More generally, it is prefered if the matrix AA^T has more nonzeros than the matrix $A^T A$. By default LOQO uses a heuristic to decide if it is better to use the primal-favored or the dual-favored ordering.

EPSNUM *eps* At the heart of LOQO is a factorization routine that factors the so-called reduced KKT system into the product of a lower triangular matrix L times a diagonal matrix D times the transpose of L. If the reduced KKT system is not of full rank, then a zero will appear on each diagonal element of D for which the corresponding equation can be written as a linear combination of preceding equations. **EPSNUM** is a tolerance — if $D_{jj} \leq$ **EPSNUM**, then the j^{th} row of the reduced KKT system is declared a dependent row. The default value for **EPSNUM** is 0.0.

EPSSOL *eps* Having dependent rows in the reduced KKT system is not by itself an indication of trouble. All that is required is that when solving the system using the forward and backward substitution procedures, it is required that when encountering a row that has been declared dependent, the right-hand side element must also be zero. If it is not, then the system of equations is inconsistent and a message to this effect is printed. **EPSSOL** is a zero tolerance for deciding how small this right-hand side element must be to be considered equivalent to a zero. The default is **1.0e-6**.

INFTOL *eps* Specifies the infeasibility tolerance for the primal and for the dual problems. The default is **1.0e-5**.

INFTOL2 *eps* Specifies the infeasibility tolerance used by the stopping rule to decide if matters are deteriorating. That is, if the new infeasibility is greater than the old infeasibility by more than INFTOL2 then stop and declare the problem infeasible. The default is **1.0**.

ITERLIM Specifies a maximum number of iterations to perform. Generally speaking the an optimal solution hasn't been found after about 50 or 60 iterations, it is quite likely that something is wrong with the model (or with LOQO itself) and it is best to quit. The default is **200**.

MAX Requests that the problem be a maximization instead of a minimization.

MIN Requests that the problem be a minimization (this is the default).

MINDEG This keyword requests the minimum degree heuristic (this is the default).

MINLOCFIL This keyword requests the minimum-local-fill heuristic. This heuristic is slower than the minimum degree heuristic, but sometimes it generates significantly better orderings yielding an overall win.

NOREORD The rows and columns of the reduced KKT system are symmetrically permuted using a heuristic that aims to minimize the amount of fill-in in L. Two heuristics are available: *minimum degree* and *minimum-local-fill* (which is also called minimum-deficiency). If you wish to use neither of these heuristics and simply solve the system in the original order, include the **NOREORD** keyword.

OBJ *str* Specifies the name of the objective function. *Str* must be a string that matches one of the **N** rows in the rows section of the MPS file. The default is to use the first encountered **N** row.

PRIMAL Requests that the ordering heuristic be set to favor the primal problem. This is typically prefered if the number of variables far exceeds the number of constraints or if the problem has a large number of dense rows. More generally, it is prefered if the matrix AA^T has fewer nonzeros than the matrix $A^T A$. By default LOQO uses a heuristic to decide if it is better to use the primal-favored or the dual-favored ordering.

RANGES *str* Specifies the name of the range set. *Str* must be a string that matches one of the range-set labels in the ranges section of the MPS file. The default is to use the first encountered range set.

RHS *str* Specifies the name of the right-hand side. *Str* must be a string that matches one of the right-hand side labels in the right-hand side section of the MPS file. The default is to use the first encountered right-hand side.

SIGFIG *n* Specifies the number of significant figures to which the primal and dual objective function values must agree for a solution to be declared optimal. The default is 8.

TIMLIM *tmax* Sets a maximum time in seconds to let the system run. The default is forever.

VERBOSE *n* Larger values of *n* result in more statistical information printed on standard output. Zero indicates no printing to standard output. The default value is 1.

APPENDIX B. EXAMPLE.

B.1. An MPS file. Here is a partial listing of the MPS file BOEING2. The entire file contains 970 lines.

```
NAME          BOEING2
ROWS
 G  REVENUES
 G  ACOCOSTS
 N  OBJECTIV
 L  FUELAVAL
 G  SYSTDEPT
 G  ACMILES
 G  ASMILES
      .
      .
      .
 L  DCLGAORD
 L  DCLGACLE
 L  DCCLELGA
 G  MCORDBOS
 G  MCLGAORD
COLUMNS
    PBOSORD0  REVENUES          .075   OBJECTIV        -.075
    PBOSORD0  PASSNGRS             1.  RPMILES        .86441
    PBOSORD0  LFRPMASM       -.86441   DMBOSORD           1.
    PBOSORD0  LF1003S1            -1.
    PBOSORD1  REVENUES          .075   OBJECTIV        -.075
    PBOSORD1  PASSNGRS             1.  RPMILES        .87605
       .         .              .       .              .
       .         .              .       .              .
       .         .              .       .              .
    N1201AC4  FUELAVAL        .70359   SYSTDEPT           1.
    N1201AC4  ACMILES         .18557   FLAV*4           .8063
    N1201AC4  ATONMILE        2.7836   LFTNMILE        1.3918
    N1201AC4  LF1201C1         11.25   CONTLGA4           1.
    N1201AC4  CONTBOS4            -1.
RHS
    RHS1      FUELAVAL       100000.   PASSNGRS         9431.
    RHS1      SYSTDEPT           50.   FLAV*1             30.
    RHS1      FLAV*2             45.   DMBOSORD          302.
    RHS1      DMBOSLGA         2352.   DMBOSCLE          142.
    RHS1      DMORDBOS          302.   DMORDLGA          515.
    RHS1      DMORDCLE          619.   DMLGABOS         2743.
     .          .              .       .              .
     .          .              .       .              .
     .          .              .       .              .
    RHS1      MSCLEBOS             1.  MSCLEORD           6.
    RHS1      MSCLELGA             3.  MCORDBOS           1.
    RHS1      MCLGAORD             2.  DCBOSORD          12.
```

```
    RHS1      DCBOSCLE          16.   DCORDBOS          24.
    RHS1      DCORDLGA          13.   DCLGAORD          45.
    RHS1      DCLGACLE          16.   DCCLELGA           5.
    RHS1      NOPTCLE0          24.
RANGES
    RANGE1    DMBOSORD          61.   DMBOSLGA         471.
    RANGE1    DMBOSCLE          29.   DMORDBOS          61.
    RANGE1    DMORDLGA         103.   DMORDCLE         124.
    RANGE1    DMLGABOS         549.   DMLGAORD         143.
    RANGE1    DMLGACLE         104.   DMCLEBOS          27.
    RANGE1    DMCLEORD         143.   DMCLELGA          82.
    RANGE1    DCBOSORD          12.   DCBOSCLE          3.2
    RANGE1    DCORDBOS          4.8   DCORDLGA          2.6
    RANGE1    DCLGAORD           9.   DCLGACLE          3.2
    RANGE1    DCCLELGA           5.
BOUNDS
 LO INTBOU    GRDTIMN1        -100.
 UP INTBOU    GRDTIMN1           0.
 LO INTBOU    GRDTIMN2         -90.
 UP INTBOU    GRDTIMN2           0.
 LO INTBOU    GRDTIMN3         -45.
 UP INTBOU    GRDTIMN3           0.
 LO INTBOU    GRDTIMN4         -45.
 UP INTBOU    GRDTIMN4           0.
     .           .              .
     .           .              .
     .           .              .
 UP INTBOU    N1100AC4           7.
 UP INTBOU    N1102AC2           7.
 UP INTBOU    N1102AC4           7.
 UP INTBOU    N1200AC2          14.
 UP INTBOU    N1200AC4           7.
 UP INTBOU    N1201AC2          14.
 UP INTBOU    N1201AC4           7.
ENDATA
```

B.2. A log file. Here is a partial listing of the corresponding log file when LOQO is used to solve the problem in Section B.1:

```
        +-----------------------------------------------+
        |                                               |
        |            LOQO: Version 2.11                 |
        |       (C) Princeton University, 1992-1995     |
        |                                               |
        +-----------------------------------------------+
variables: non-neg     89,  free        0,  bdd         54,  total      143
constraints: eq         4,  ineq      143,  ranged      19,  total      167
nonzeros:    A       1339,  Q           0
nonzeros:    L       4363,  arith_ops              89262
-------------------------------------------------------------------------
      |         Primal            |           Dual            | Sig
Iter  |  Obj Value      Infeas    |   Obj Value      Infeas   | Fig  Status
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   1   4.7237992e+03   5.75e-01     7.2271163e+09   8.32e+03
   2   4.7735777e+03   5.74e-01     7.2010923e+09   8.29e+03
   3   5.5976588e+03   5.51e-01     7.0116304e+09   8.08e+03
   4   6.4798130e+03   5.30e-01     6.7046013e+09   7.76e+03
   5   8.5615694e+03   4.48e-01     5.3385958e+09   6.24e+03
   6   6.0632486e+03   3.11e-01     3.8072059e+09   4.45e+03
   7   5.2019929e+03   2.32e-01     2.1263747e+09   2.56e+03
   8   4.6772801e+03   1.68e-01     1.3486633e+09   1.70e+03
   9   3.8920450e+03   9.93e-02     8.3550490e+08   1.07e+03
  10   3.2774096e+03   8.96e-02     6.9535403e+08   9.11e+02
  11   1.3682424e+03   4.08e-02     5.3629338e+08   7.21e+02
  12   1.2199505e+03   3.31e-02    -4.9872847e+07   4.91e+01
  13   3.3982074e+02   9.83e-03    -2.8819898e+07   5.85e+00
  14   8.6940743e+01   2.69e-03    -1.5355669e+07   1.96e+00
  15   3.0164084e+01   1.03e-03    -4.6581330e+06   3.01e-01
  16  -1.4731024e+01   5.61e-04    -2.4296725e+06   1.17e-01
  17  -5.2302263e+01   3.32e-04    -1.6526514e+06   7.12e-02
  18  -1.2075975e+02   1.36e-04    -7.9671784e+05   2.19e-02
  19  -1.5180495e+02   4.23e-05    -3.2575147e+05   5.57e-03
  20  -1.7334366e+02   2.61e-06    -7.2457483e+04   7.56e-04          PF
  21  -1.7628397e+02   1.33e-07    -6.3383861e+03   6.24e-05          PF
  22  -2.0157336e+02   7.25e-09    -2.1985311e+03   2.01e-05          PF
  23  -2.4811212e+02   2.32e-09    -8.5903751e+02   5.76e-06          PF DF
  24  -2.6365903e+02   1.41e-09    -4.6257064e+02   1.58e-06          PF DF
  25  -2.9579539e+02   4.56e-10    -3.7439386e+02   5.81e-07     1    PF DF
  26  -3.0246863e+02   2.63e-10    -3.4693510e+02   2.78e-07     1    PF DF
  27  -3.0931667e+02   1.03e-10    -3.2266089e+02   5.60e-08     1    PF DF
  28  -3.1352719e+02   1.96e-11    -3.1897090e+02   2.92e-08     2    PF DF
  29  -3.1468541e+02   1.03e-11    -3.1576272e+02   5.13e-09     2    PF DF
  30  -3.1490420e+02   2.32e-11    -3.1508096e+02   3.97e-10     3    PF DF
     dependent  rows:    1
```

```
  31  -3.1501125e+02   1.51e-12    -3.1502190e+02   3.42e-11     4    PF DF
  32  -3.1501835e+02   7.58e-14    -3.1501889e+02   6.82e-11     6    PF DF
  33  -3.1501871e+02   3.88e-15    -3.1501874e+02   2.92e-11     7    PF DF
     dependent  rows:    1
  34  -3.1501873e+02   5.43e-16    -3.1501873e+02   6.46e-11     8    PF DF
----------------------
OPTIMAL SOLUTION FOUND
```

B.3. A solution file. And finally, here is part of the solution file:

```
COLUMNS SECTION
   index       label  primal_val reduced_cst    lower_bd    upper_bd   OB_flag
       0    PBOSORD0  3.0200e+02  4.6896e-12  0.0000e+00    Infinity
       1    PBOSORD1  2.7237e-07  6.9453e-03  0.0000e+00    Infinity
       2    PBOSORD2  2.0514e-07  2.4120e-02  0.0000e+00    Infinity
       3    PBOSORD3  8.7781e-08  4.5699e-02  0.0000e+00    Infinity
       4    PBOSORD4  2.6571e-07  1.2973e-02  0.0000e+00    Infinity
       5    PBOSLGA0  7.1200e+02  1.7384e-12  0.0000e+00    Infinity
       6    PBOSLGA1  2.6800e+02  4.1145e-12  0.0000e+00    Infinity
       7    PBOSLGA2  7.3947e-07  6.9637e-03  0.0000e+00    Infinity
       .       .           .           .           .           .
       .       .           .           .           .           .
       .       .           .           .           .           .
     137    N1102AC2  3.1029e-08  1.2656e-01  0.0000e+00  7.0000e+00
     138    N1102AC4  1.5372e-08  2.4910e-01  0.0000e+00  7.0000e+00
     139    N1200AC2  1.4000e+01  9.2694e-11  0.0000e+00  1.4000e+01
     140    N1200AC4  1.9873e-08  2.8403e-01  0.0000e+00  7.0000e+00
     141    N1201AC2  1.3321e+01  1.0305e-10  0.0000e+00  1.4000e+01
     142    N1201AC4  1.3578e-08  3.0478e-01  0.0000e+00  7.0000e+00
ROWS SECTION
   index       label    dual_val  row_actvty rght_hnd_sd       range   OB_flag
       0    REVENUES  2.9171e-12  4.4431e+02  0.0000e+00    Infinity
       1    ACOCOSTS  1.0767e-11  1.2929e+02  0.0000e+00    Infinity
       2    OBJECTIV -4.7238e-97 -3.1502e+02   -Infinity    Infinity
       3    FUELAVAL -1.4524e-14 -1.8197e+02 -1.0000e+05    Infinity
       4    SYSTDEPT  6.9938e-11  1.2307e+02  5.0000e+01    Infinity
       5    ACMILES   7.5904e-11  4.9225e+01  0.0000e+00    Infinity
       6    ASMILES   2.2615e-13  5.5609e+03  0.0000e+00    Infinity
       7    PASSNGRS  1.4260e-10  9.4560e+03  9.4310e+03    Infinity
       .       .           .           .           .           .
       .       .           .           .           .           .
       .       .           .           .           .           .
     159    DCBOSCLE  3.4040e-01 -1.6000e+01 -1.6000e+01  3.2000e+00
     160    DCORDBOS  7.5000e-01 -2.4000e+01 -2.4000e+01  4.8000e+00
     161    DCORDLGA  6.8000e-01 -1.3000e+01 -1.3000e+01  2.6000e+00
     162    DCLGAORD  3.1818e-01 -4.5000e+01 -4.5000e+01  9.0000e+00
     163    DCLGACLE  1.6366e-01 -1.6000e+01 -1.6000e+01  3.2000e+00
     164    DCCLELGA  3.7000e-01 -5.0000e+00 -5.0000e+00  5.0000e+00
     165    MCORDBOS  1.3976e-09  2.6937e+00  1.0000e+00    Infinity
     166    MCLGAORD  6.5306e-10  4.0000e+00  2.0000e+00    Infinity
ENDOUT
```

ROBERT J. VANDERBEI, PROGRAM IN STATISTICS AND OPERATIONS RESEARCH, PRINCETON UNIVERSITY, PRINCETON, NJ 08544
E-mail address: rvdb@princeton.edu