QSopt Reference Manual

Version 1.0 – October 27, 2003

Contents

Preface

The *QSopt* library provides a set of functions for creating, manipulating, and solving linear-programming problems. The library is written in the C programming language, and it is distributed as an archive file `qsopt.a` and a header file `qsopt.h`. This reference manual contains descriptions of the functions available in the *QSopt* library.

Chapter 1

Introduction

Linear programming is a widely used tool for computing optimal solutions to problems involving the allocation of scare resources. A classic reference is *Linear Programming*, V. Chvátal (W. H. Freeman and Company, New York, 1983), and a recently updated reference is *Linear Programming: Foundations and Extensions*, R. J. Vanderbei (Kluwer Academic Publishers, Boston, 2001).

Linear Programming Problems

A linear-programming (LP) problem is one of maximizing or minimizing a linear function subject to linear equality and linear inequality constraints. A general form of an LP problem is

$$\begin{gathered}
\text{Maximize } \; c_1x_1 + c_2x_2 + \cdots + c_nx_n \\
\text{Subject to} \\
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \le b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \le b_2 \\
\vdots \\
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \le b_m \\
l_1 \le x_1 \le u_1 \\
l_2 \le x_2 \le u_2 \\
\vdots \\
l_n \le x_n \le u_n
\end{gathered}$$

where $x_1, x_2, \ldots, x_n$ are variables and the remaining elements are input data. Each of the input data can be any (rational) number; the $l_1, l_2, \ldots, l_n$ can also be assigned the value of $-$infinity ($-\infty$) and the $u_1, u_2, \ldots, u_n$ can also be assigned the value of $+$infinity ($+\infty$). The "Maximize" can alternatively be "Minimize", and each "$\le$" relation can alternatively be an "$=$" or a "$\ge$" relation.

There is common terminology associated with the parts of an LP problem. The linear function $c_1x_1 + c_2x_2 + \cdots + c_nx_n$ that we seek to maximize (or minimize) is called the *objective function*, and $c_1, c_2, \ldots, c_n$ are called the objective coefficients. The inequalities $a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \le b_i$ (for $i = 1, \ldots, m$) are referred to as the *constraints*, and the values $b_1, b_2, \ldots, b_m$ are called the *right-hand-side* values. The values $l_1, l_2, \ldots, l_n$ are called *lower bounds*, and the values $u_1, u_2, \ldots, u_n$ are called *upper bounds*. Note that by setting $l_i = -\infty$ and $u_i = +\infty$, we allow the variable x_i to have no explicit bounds; such a variable is called *free*.

It is often convenient to express an LP problem in matrix notation. To do this, we define a matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

and vectors

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad c = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}, \quad l = \begin{pmatrix} l_1 \\ l_2 \\ \vdots \\ l_n \end{pmatrix}, \quad u = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}.$$

The general LP problem can then be formulated as

$$\begin{gathered} \text{Maximize } \; c^T x \\ \text{Subject to} \\ Ax \leq b \\ l \leq x \leq u \end{gathered}$$

where c^T is the transpose of the c-vector. The matrix A is called the *constraint matrix*. We will sometimes refer to the ith constraint as the ith "row" and refer to the jth variable as the jth "column".

As an example of an LP problem, consider the following formulation with 3 variables ($n = 3$) and 2 constraints ($m = 2$).

$$\begin{gathered} \text{Maximize } \; 3.0x_1 + 2.0x_2 + 4.0x_3 \\ \text{Subject to} \\ 3.1x_1 + 2.3x_2 + 1.4x_3 \leq 12.2 \\ 5.0x_1 + 1.1x_2 = 10.0 \\ 0.0 \leq x_1 \leq +\infty \\ -\infty \leq x_2 \leq +\infty \\ 0.0 \leq x_3 \leq 10.0 \end{gathered}$$

In describing such an LP problem, it is standard practice to assume that each variable x_i has lower bound $l_i = 0$ and upper bound $u_i = +\infty$, if the bounds are not explicitly given in the model. So this example could be expressed as

$$\begin{gathered} \text{Maximize } \; 3.0x_1 + 2.0x_2 + 4.0x_3 \\ \text{Subject to} \\ 3.1x_1 + 2.3x_2 + 1.4x_3 \leq 12.2 \\ 5.0x_1 + 1.1x_2 = 10.0 \\ x_2 \text{ free} \\ x_3 \leq 10.0 \end{gathered}$$

where (as we mentioned above) "free" is the shorthand way of expressing the fact that x_2 has no explicit bounds.

The QSopt Library

The *QSopt* library provides a set of functions for creating, manipulating, and solving LP problems. The library is written in the (ANSI) C programming language, and it is distributed as an archive file `qsopt.a` and a header file `qsopt.h`. Thread-safe versions of the library are available for use in re-entrant programming applications.

This reference manual contains descriptions of the functions available in the *QSopt* library. To make the best use of the library, the user should be familiar with the C programming language; excellent references for C are *The C Programming Language, Second Edition*, B. W. Kernighan and D. M. Ritchie (Prentice Hall, Englewood Cliffs, 1988) and *A C Reference Manual*, S. P. Harbison, III and G. L. Steele, Jr. (Prentice Hall, Upper Saddle River, 2002).

Chapter 2

Creating an LP Problem

Nearly all *QSopt* functions work with a single problem data structure that stores all available information about the problem as well as the associated choice of solution algorithm. A handle to the data structure is provided to the user by the `QSprob` data type.

The first step to working with an LP problem is to obtain an initialized `QSprob` from the *QSopt* library. There are three available functions that return initialized `QSprob` objects, allowing the user to start with an empty problem, a problem described in a file (in MPS or LP format), or a problem described by a set of user data. These four functions are described below, together with a function that allows the user to free the memory associated with a `QSprob` when work on a given problem is completed.

		Page
QScreate_prob	initialize a new data structure with an empty problem	4
QSread_prob	read a problem from a named file	5
QSload_prob	build a problem from user data	6
QSfree_prob	free the data structure for a problem	8

QScreate_prob

Initialize a new problem data structure for an empty problem.

Synopsis

```
QSprob QScreate_prob (const char *name, int objsense)
```

Arguments

`name` – a string specifying the name of the problem (can be `NULL`)

`objsense` – specifies if the problem is a minimization problem or a maximization problem; to specify a minimization problem set `objsense` to `QS_MIN` (that is, 1) and to specify a maximization problem set `objsense` to `QS_MAX` (that is, -1).

Returns

An initialized `QSprob` object, providing a handle to an empty problem. If an error occurred, the `QSprob` object is set to `NULL`.

Description

The `QSprob` provided by this function is the starting point for building an LP problem from within an application. The `QSprob` is a handle to an empty problem that can be filled-in using the problem modification routines described in Chapter 5. To specify a minimization problem set `objsense` to `QS_MIN` (that is, 1), and to specify a maximization problem set `objsense` to `QS_MAX` (that is, -1); if `objsense` is given some other value, the objective function defaults to minimization.

A standard method is to follow the `QScreate_prob()` call with calls to `QSnew_row()` to initialize the constraints, then calls to `QSadd_col()` to create the variables and to build the coefficients in the constraint matrix. Alternatively, one can first use `QSnew_col()` to initialize the variables, then use `QSadd_row()` to create the constraints and the constraint matrix. (Or mix the column and row operations, if that is more natural in the given application.)

Example

```
QSprob p;

p = QScreate_prob ("binky", QS_MIN);
if (p == (QSprob) NULL) {
    fprintf (stderr, "Unable to initialize LP data structure\n");
} else {
    /* Build the LP with calls to QSnew_row () and QSadd_col () */
}
```

QSread_prob

Read a problem from a named file.

Synopsis

```
QSprob QSread_prob (const char *filename, const char *filetype)
```

Arguments

`filename` – a string specifying the name of the file containing the problem
`filetype` – use `"LP"` to read a file in LP format or `"MPS"` to read a file in MPS format

Returns

An initialized `QSprob` object, providing a handle to the problem described in the specified file. If an error occurred, the `QSprob` object is set to `NULL`.

Description

This function can read LP problems in either MPS format or in LP format. To specify MPS format, set `filetype` to `"MPS"` (the letters can be any mix of uppercase and lowercase), and to specify LP format, set `filetype` to `"LP"`. (If some other string is specified, an error code will be returned.)

MPS format is a standard file format for recording LP problems. The format was created at IBM in the 1960s, and the original standard included fixed-column information (a common feature in the days of punched cards). An introduction to the format can be found at

`plato.la.asu.edu/pub/mps_format.txt`.

The *QSopt* MPS reader follows a number of other modern LP solvers in removing the fixed-column aspects of the format. A complete description of the format used in *QSopt* is given in Chapter 10.

LP format is a well-established format for recording algebraic representations of LP problems, in a natural, row by row, manner. The *QSopt* LP format is compatible with the standard usage of the LP file format incorporated in other LP solvers. A complete description of the format used in *QSopt* is given in Chapter 10.

Example

```
QSprob p;

/* Read a problem in MPS format given in the file binky.mps */

p = QSread_prob ("binky.mps", "MPS");
if (p == (QSprob) NULL) {
    fprintf (stderr, "Unable to read and load the LP\n");
} else {
    /* We can solve and/or modify the problem using the handle p */
}
```

QSload_prob

Build a problem from user data.

Synopsis

```
QSprob QSload_prob (const char *probname, int ncols, int nrows,
    int *cmatcnt, int *cmatbeg, int *cmatind, double *cmatval,
    int objsense, double *obj, double *rhs, char *sense, double *lower,
    double *upper, const char **colnames, const char **rownames)
```

Arguments

`probname` – a string containing the name of the problem (can be `NULL`).

`ncols` – specifies the number of columns.

`nrows` – specifies the number of rows.

`cmatcnt` – an array of length `ncols`; the jth entry specifies the number of non-zeros in the jth column.

`cmatbeg` – an array of length `ncols`; the jth entry specifies the location of the start of the entries for the jth column in the arrays `cmatind` and `cmatval`.

`cmatind` – an array that contains the row indices of the non-zero entries in the constraint matrix. The indices of the jth column must be stored consecutively starting at entry number `cmatbeg[j]` (there are `cmatcnt[j]` entries for the jth column).

`cmatval` – an array that contains the values of the non-zero entries in the constraint matrix. The non-zero values of the jth column must be stored consecutively starting at entry number `cmatbeg[j]` (there are `cmatcnt[j]` entries for the jth column).

`objsense` – specifies if the problem is a minimization problem or a maximization problem; to specify a minimization problem set `objsense` to `QS_MIN` (that is, 1) and to specify a maximization problem set `objsense` to `QS_MAX` (that is, -1).

`obj` – an array of length `ncols`; the jth entry is the coefficient of the jth variable in the objective function.

`rhs` – an array of length `nrows`; the ith entry is the right-hand-side value of the ith constraint.

`sense` – an array of length `nrows`; the ith entry specifies the sense of the ith constraint; to specify an equation use `'E'`, to specify a $\leq$ constraint use `'L'`, and to specify a $\geq$ constraint use `'G'`.

`lower` – an array of length `ncols`; the jth entry is the lower bound for the jth variable (use `-QS_MAXDOUBLE` if the jth variable has no lower bound).

`upper` – an array of length `ncols`; the jth entry is the upper bound for the jth variable (use `QS_MAXDOUBLE` if the jth variable has no upper bound).

`colnames` – an array of length `ncols`; the jth entry is a string that specifies the name of the jth variable (the `colnames` field can be `NULL`).

`rownames` – an array of length `nrows`; the ith entry is a string that specifies the name of the ith constraint (the `rownames` field can be `NULL`).

Returns

An initialized `QSprob` object, providing a handle to the problem described by the specified data. If an error occurred, the `QSprob` object is set to `NULL`.

Description

The arguments to `QSload_prob()` provide a method to completely describe an LP problem; the cost of this completeness is the frightening appearance of the argument list itself. Most applications can be written more cleanly using `QScreate_prob()` to initialize an empty `QSprob`, and then filling-in the LP data piece by piece with the problem modification routines described in Chapter 5. In some cases, however, it is convenient to load everything in a single stroke. For example, if an application stores all data for an LP in a problem-specific format, then it is natural to use `QSload_prob()` to load all of the information into the *QSopt* solver.

Note that the specification of both `cmatcnt` and `cmatbeg` is an overkill, since if the data arrays are packed densely, then the `cmatbeg` values can be determined by the `cmatcnt` values. The point in using both `cmatcnt` and `cmatbeg` is to allow applications the freedom of providing gaps in the data arrays. (This may be helpful, for example, when a application builds its LP data before exact counts of the number of non-zeros in each column are available.)

When specifying the constraint coefficients, objective coefficients, right-and-aside values, and upper and lower bounds, it is important not to give any value larger in magnitude than `QS_MAXDOUBLE` (1e30). This restriction is due to the internal structures used in the *QSopt* solvers.

Example

```
/*                                                      */
/*  Using QSload_prob to load the following LP problem  */
/*       Maximize  3.0x + 2.0y + 4.0z                   */
/*       Subject to                                     */
/*                 3.1x + 2.3y + 1.4z <= 12.2           */
/*                 5.0x + 1.1y         = 10.0           */
/*                 x >= 2.0                             */
/*                 y free                               */
/*                 1.0 <= z <= 10.0                     */
/*                                                      */

int rval;
QSprob p;
int cmatcnt[3]    = { 2, 2, 1 };
int cmatbeg[3]    = { 0, 2, 4 };
int cmatind[5]    = { 0, 1, 0, 1, 0 };
double cmatval[5] = { 3.1, 5.0, 2.3, 1.1, 1.4 };
double obj[3]     = { 3.0, 2.0, 4.0 };
double rhs[2]     = { 12.2, 10.0 };
char sense[2]     = { 'L', 'E' };
double lower[3]   = { 2.0, -QS_MAXDOUBLE, 1.0 };
double upper[3]   = { QS_MAXDOUBLE, QS_MAXDOUBLE, 10.0 };
const char *colnames[3] = { "x", "y", "z" };
```

```
p = QSload_prob ("small", 3, 2, cmatcnt, cmatbeg, cmatind, cmatval,
                  QS_MAX, obj, rhs, sense, lower, upper, colnames,
                  (char **) NULL);

if (p == (QSprob) NULL) {
    fprintf (stderr, "Unable to load the LP problem\n");
} else {
    /* As a check, write the LP to a file */
    rval = QSwrite_prob (p, "small.lp", "LP");
    if (rval) {
        fprintf (stderr, "Could not write LP, error code %d\n", rval);
    }
}
```

QSfree_prob

Free the data structure for a problem.

Synopsis

```
void QSfree_prob (QSprob p)
```

Arguments

`p` – a handle to a problem.

Returns

No return value.

Description

When work is complete on a given problem `p`, `QSfree_prob(p)` should be called to free the memory associated with the `QSprob` data structure. This operation is important in applications where LP problems are repeatedly created and solved, permitting the application to reuse the available memory.

Note that after a call to `QSfree_prob(p)`, the specified `QSprob p` will no longer contain any information related to the original LP problem.

Example

```
/* p is an initialized QSprob */

QSfree_prob (p);
```

Chapter 3

Optimizing an LP Problem

The *QSopt* library provides access to two procedures for solving linear programming problems (and the LP relaxations associated with mixed-integer programming problems). The solution procedures are the primal and dual simplex algorithms. For each of these procedures, the user can specify a number of parameters to determine the particular variant of the algorithm that is applied (for example, the dual steepest-edge simplex algorithm). The parameters for the algorithms are specified by using the `QSset_param()` function.

		Page
QSopt_dual	solve the LP problem with the dual simplex algorithm	9
QSopt_primal	solve the LP problem with the primal simplex algorithm	11

QSopt_dual

Solve the LP problem with the dual simplex algorithm.

Synopsis

```
int QSopt_dual (QSprob p, int *status)
```

Arguments

`p` – a handle to an initialized problem.
`status` – returns a code to indicate the status of the final solution.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

The dual simplex algorithm is the method of choice for many forms of LP problems when solving the problem from scratch. The algorithm is also an effective solution procedure for re-optimizing problems after the addition of constraints (adding constraints keeps the current dual solution feasible).

After a call to `QSopt_dual()`, the `status` variable should be checked to determine the status of the solution. In the most common cases, either an optimal solution is found (indicated by a value of `QS_LP_OPTIMAL`), or the problem is shown to be in infeasible (`QS_LP_INFEASIBLE`), or the problem is shown to be unbounded (`QS_LP_UNBOUNDED`). A full list of the possible values for `status` is given below.

Value	Constant	Meaning
1	QS_LP_OPTIMAL	Optimal solution found
2	QS_LP_INFEASIBLE	LP problem has no feasible solution
3	QS_LP_UNBOUNDED	LP problem objective is unbounded
4	QS_LP_ITER_LIMIT	Iteration limit reached
5	QS_LP_TIME_LIMIT	Time limit reached
6	QS_LP_UNSOLVED	Could not solve the LP problem

By default `QSopt_dual()` will solve the specified LP problem using the dual steepest-edge pricing rule. This rule is effective for many difficult LP problems, but for problems that are relatively easy to solve (even though they may be very large in size) a less time-consuming pricing rule may be preferable. To select an alternative pricing rule, the `QSset_param()` function can be used with the `whichparam` argument set to the value given below.

Constant	Parameter
QS_PARAM_DUAL_PRICING	Dual pricing

The *QSopt* library has available three pricing rules for the dual simplex algorithm. The pricing rule can be specified in `QSset_param()` by setting the `newvalue` argument to one of the values given below.

Constant	Rule
QS_PRICE_DDANTZIG	Dual Dantzig
QS_PRICE_DSTEEP	Dual steepest-edge
QS_PRICE_DMULTPARTIAL	Dual multiple-partial

Several parameters associated with the optimization algorithm (as well as with `QSopt_primal()`) can also be set with `QSset_param()`. These parameters are given in the following table.

Parameter	Possible Values	Default
QS_PARAM_SIMPLEX_DISPLAY	0 or 1	0
QS_PARAM_SIMPLEX_SCALING	0 or 1	1
QS_PARAM_SIMPLEX_MAX_ITERATIONS	Any positive integer	300000

In a call to `QSset_param()`, the parameter is specified in the `whichparam` argument and the value is specified in the `newvalue` argument. The `QS_PARAM_SIMPLEX_DISPLAY` parameter controls whether or not detailed information on the progress of the optimization algorithm is displayed (0 is no, and 1 is yes) and the `QS_PARAM_SIMPLEX_SCALING` parameter controls whether or not the problem data is scaled at the start of the optimization algorithm. The maximum number of pivots allowed in the algorithm can be set with the `QS_PARAM_SIMPLEX_MAX_ITERATIONS` parameter.

Additional parameters can be set with `QSset_param_double()`, as indicated below.

Parameter	Possible Values	Default
QS_PARAM_SIMPLEX_MAX_TIME	Any positive number	10000.0

The `QS_PARAM_SIMPLEX_MAX_TIME` parameter sets the maximum allowed time (in seconds) for the optimization algorithm.

Example

```
int status, rval;

/* p is a QSprob, a handle to an existing LP problem */

rval = QSopt_dual (p, &status);
if (rval) {
    fprintf (stderr, "QSopt_dual failed with return code %d\n", rval);
} else {
    switch (status) {
    case QS_LP_OPTIMAL:
        printf ("Found optimal solution to LP\n");
        break;
    case QS_LP_INFEASIBLE:
        printf ("No feasible solution exists for the LP\n");
        break;
    case QS_LP_UNBOUNDED:
        printf ("The LP objective is unbounded\n");
        break;
    default:
        printf ("LP could not be solved, status = %d\n", status);
        break;
    }
}
```

QSopt_primal

Solve the LP problem with the primal simplex algorithm.

Synopsis

```
int QSopt_primal (QSprob p, int *status)
```

Arguments

`p` – a handle to an initialized problem.
`status` – returns a code to indicate the status of the final solution.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

In general, there is not a clear-cut winner between the primal and dual simplex algorithms. If performance is important in an application, it may pay to make tests of both methods to determine which function to use in the given setting.

After a call to `QSopt_primal()`, the `status` variable should be checked to determine the status of the solution. The possible values of `status` are given above in the description of `QSopt_dual()`.

The `QSset_param()` function can be used to select the pricing rule that is used in `QSopt_primal()`. The appropriate value for the `whichparam` argument of `QSset_param()` is given below.

Constant	Parameter
`QS_PARAM_PRIMAL_PRICING`	Primal pricing

The *QSopt* library has available four pricing rules for the primal simplex algorithm. The pricing rule can be specified in `QSset_param()` by setting the `newvalue` argument to one of the values given below.

Constant	Rule
`QS_PRICE_PDANTZIG`	Primal Dantzig
`QS_PRICE_PDEVEX`	Primal Devex
`QS_PRICE_PSTEEP`	Primal steepest-edge
`QS_PRICE_PMULTPARTIAL`	Primal multiple-partial

Additional parameters associated with the optimization algorithm can be set using `QSset_param()` and `QSset_param_double()`; see the documentation for `QSopt_dual()` given above.

Example

```
int status, rval;

/* p is a QSprob, a handle to an existing LP problem */

rval = QSopt_primal (p, &status);
if (rval)
    fprintf (stderr, "QSopt_primal failed with return code %d\n", rval);
} else {
    switch (status) {
    case QS_LP_OPTIMAL:
        printf ("Found optimal solution to LP\n");
        break;
    case QS_LP_INFEASIBLE:
        printf ("No feasible solution exists for the LP\n");
        break;
    case QS_LP_UNBOUNDED:
        printf ("The LP objective is unbounded\n");
        break;
    default:
        printf ("LP could not be solved, status = %d\n", status);
        break;
    }
}
```

Chapter 4

Accessing an LP Solution

After a problem has been solved by `QSopt_dual()` or `QSopt_primal()`, information about the components of the solution can be obtained through a variety of *QSopt* library functions. The optimal basis can be obtained via the functions described in Chapter 7 *Working with an LP Basis*, and in this chapter we describe functions to retrieve the objective function value, the solution vector, the reduced costs, the values of the dual variables, and the values of the slack variables associated with the constraints.

Except for the objective function value, all information is returned in arrays. The arrays need to be allocated by the calling routine. For the solution vector and for the values of the reduced costs, the arrays must be length at least `ncols` (the number of columns [variables] in the problem). For the values of the dual variables and for the values of the slack variables, the arrays must be of length at least `nrows` (the number of rows [constraints] in the problem). The appropriate values of `ncols` and `nrows` can be obtained by calls to the functions `QSget_colcount()` and `QSget_rowcount()`, respectively.

QSget_status

Obtain the solution status.

Synopsis

```
int QSget_status (QSprob p, int *status)
```

Arguments

p – a handle to an initialized problem.
status – returns the solution status of the LP problem.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

If the problem p has been modified since the last time QSopt_dual() or QSopt_primal() has been called to solve p, then status is set to QS_LP_MODIFIED (this has numerical value 100). Otherwise, status is set to the value that was associated with the last call to QSopt_dual() or QSopt_primal(), as specified in the table given on page 10.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int rval, status;

rval = QSget_satus (p, &status);
if (rval) {
    fprintf (stderr, "Unable to obtain status, error code %d\n", rval);
} else {
    switch (status) {
    case QS_LP_OPTIMAL:
        printf ("An optimal solution is available\n");
        break;
    case QS_LP_INFEASIBLE:
        printf ("The LP has no feasible solution\n");
        break;
    case QS_LP_UNBOUNDED:
        printf ("The LP has unbounded objective value\n");
        break;
    case QS_ITER_LIMIT:
        printf ("The optimization algorithm reached its iteration limit\n");
        break;
    case QS_TIME_LIMIT:
        printf ("The optimization algorithm reached its time limit\n");
        break;
    case QS_LP_UNSOLVED:
        printf ("The optimization algorithm could not solve the LP\n");
        break;
    case QS_LP_MODIFIED:
        printf ("The LP was modified since last optimization call\n");
        break;
    default:
        printf ("Unknown solution status: %d\n", status);
        break;
    }

}
```

QSget_solution

Copy various solution data into arrays.

Synopsis

```
int QSget_solution (QSprob p, double *value, double *x, double *pi,
    double *slack, double *rc)
```

Arguments

`p` – a handle to an initialized problem.

`value` – returns the value of the objective function (this field can be `NULL`).

`x` – returns the solution vector as an array (this field can be `NULL`, if it is not `NULL` then it should point to an array of length at least `ncols`, the number of columns in the problem).

`pi` – returns the values of the dual variables as an array (this field can be `NULL`, if it is not `NULL` then it should point to an array of length at least `nrows`, the number of rows in the problem).

`slack` – returns the values of the slack variables associated with the constraints (this field can be `NULL`, if it is not `NULL` then it should point to an array of length at least `nrows`).

`rc` – returns the reduced costs of the variables (this field can be `NULL`, if it is not `NULL` then it should point to an array of length at least `ncols`).

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred. Note that `QSget_solution()` returns an error code if it is called with a specified problem that has not been solved with one of the optimization functions (`QSopt_dual()` or `QSopt_primal()`) since the last time the problem was loaded or modified.

Description

This function allows the user to obtain different components of the solution in one call (if some component is not needed, the field can be given as `NULL`). To obtain individual components, it is easier to use the specialized functions described later in this chapter.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */
/* Assume that p was solved with QSopt_dual() or QSopt_primal, and */
/* that the status code indicates that an optimal solution was     */
/* obtained.  We will grab and print all solution components.      */

int rval, ncols, nrows, i, j;
double value;
double *x, *pi, *slack, *rc;
char **colnames, **rownames;

ncols = QSget_colcount (p);
nrows = QSget_rowcount (p);

/* Get the variable names and the constraint names, for printing */

colnames = (char **) malloc (ncols * sizeof (char *));
```

```
rownames = (char **) malloc (nrows * sizeof (char *));
rval = QSget_colnames (p, colnames);
if (rval) {
    fprintf (stderr, "Could not get column names, error code %d\n", rval);
    /* free the memory for colnames and rownames                 */
    /* return an error code                                      */
}
rval = QSget_rownames (p, rownames);
if (rval) {
    fprintf (stderr, "Could not get row names, error code %d\n", rval);
    /* free the memory for colnames and rownames                 */
    /* return an error code                                      */
}

/* Allocate memory for the solution components.                 */

x  = (double *) malloc (ncols * sizeof (double));
rc = (double *) malloc (ncols * sizeof (double));
pi = (double *) malloc (nrows * sizeof (double));
slack = (double *) malloc (nrows * sizeof (double));

rval = QSget_solution (p, &value, x, pi, slack, rc);
if (rval) {
    fprintf (stderr, "Could not get solution, error code %d\n", rval);
    /* free the memory for x, rc, pi, slack, colnames, rownames  */
    /* return an error code                                      */
}

printf ("Objective Value = %.6f\n", value);
for (j = 0; j < ncols; j++) {
    printf ("%s = %.6f, reduced cost = %.6f\n", colnames[j], x[j], rc[j]);
}
for (i = 0; i < nrows; i++) {
    printf ("dual %s = %.6f, slack = %.6f\n", rownames[i], pi[i], slack[i]);
}

/* Free the memory, including the individual names of cols and rows */

free (x);  free (rc);  free (pi);  free (slack);
for (j = 0; j < ncols; j++) {
    QSfree (colnames[j]);   /* Use QSfree for memory from QSopt functs  */
}
free (colnames);
for (i = 0; i < nrows; i++) {
    QSfree (rownames[i]);   /* Use QSfree for memory from QSopt functs  */
}
free (rownames);
```

QSget_objval

Get the current objective function value.

Synopsis

```
int QSget_objval (QSprob p, double *value)
```

Arguments

p – a handle to an initialized problem.
value – returns the value of the objective function.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred. Note that `QSget_objval()` returns an error code if it is called with a specified problem that has not been solved with one of the optimization functions (`QSopt_dual()` or `QSopt_primal()`) since the last time the problem was loaded or modified.

Description

After a successful call to one of the optimization routines (`QSopt_dual()` or `QSopt_primal()`), this function can be used to obtain the value of the objective function.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int rval;
double value;

rval = QSget_objval (p, &value);
if (rval) {
    fprintf (stderr, "Could not get objective value, error code %d\n", rval);
} else {
    printf ("Objective value = %.6f\n", value);
}
```

QSget_x_array

Copy the solution vector into an array.

Synopsis

```
int QSget_x_array (QSprob p, double *x)
```

Arguments

p – a handle to an initialized problem.
x – returns the solution vector as an array; this field should point to an array of length at least ncols, the number of columns in the problem.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred. Note that `QSget_x_array()` returns an error code if it is called with a specified problem that has not been solved with one of the optimization functions (`QSopt_dual()` or `QSopt_primal()`) since the last time the problem was loaded or modified.

Description

This function returns the solution to the LP problem as a dense vector, that is, the values of all variables (both zero and non-zero) are returned in the array `x`.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int rval, ncols;
double *x;

ncols = QSget_colcount (p);
x = (double *) malloc (ncols * sizeof (double));

rval = QSget_x_array (p, x);
if (rval) {
    fprintf (stderr, "Could not get x-vector, error code %d\n", rval);
} else {
    /* To print solution with names, see QSget_solution() example. */
}

free (x);
```

QSget_pi_array

Copy the values of the dual variables into an array.

Synopsis

```
int QSget_pi_array (QSprob p, double *pi)
```

Arguments

`p` – a handle to an initialized problem.

`pi` – returns the values of the dual variables as an array; this should point to an array of length at least `nrows`, the number of rows in the problem.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred. Note that `QSget_pi_array()` returns an error code if it is called with a specified problem that has not been solved with one of the optimization functions (`QSopt_dual()` or `QSopt_primal()`) since the last time the problem was loaded or modified.

Description

This function returns the dual solution to the LP problem as a dense vector, that is, the values of all dual variables (both zero and non-zero) are returned in the array `pi`. Note that the values of the dual variables associated with lower and upper bounds are given implicitly by the array `pi`.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */
```

```
int rval, nrows;
double *pi;

nrows = QSget_rowcount (p);
pi = (double *) malloc (nrows * sizeof (double));

rval = QSget_pi_array (p, pi);
if (rval) {
    fprintf (stderr, "Could not get dual values, error code %d\n", rval);
} else {
    /* To print dual values with names, see QSget_solution() example. */
}

free (pi);
```

QSget_slack_array

Copy the constraint slack values into an array.

Synopsis

```
int QSget_slack_array (QSprob p, double *slack)
```

Arguments

`p` – a handle to an initialized problem.

`slack` – returns the values of the slack variables associated with the constraints; this field should point to an array of length at least `nrows`, the number of rows in the problem.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred. Note that `QSget_slack_array()` returns an error code if it is called with a specified problem that has not been solved with one of the optimization functions (`QSopt_dual()` or `QSopt_primal()`) since the last time the problem was loaded or modified.

Description

This function returns the slack values associated with the constraints of the LP problem. The values are given as a dense vector, that is, the values of all slacks (both zero and non-zero) are returned in the array `slack`.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int rval, nrows;
double *slack;

nrows = QSget_rowcount (p);
slack = (double *) malloc (nrows * sizeof (double));

rval = QSget_slack_array (p, slack);
if (rval) {
    fprintf (stderr, "Could not get slacks, error code %d\n", rval);
} else {
    /* To print slack values with names, see QSget_solution() example. */
}

free (slack);
```

QSget_rc_array

Copy the reduced costs into an array.

Synopsis

```
int QSget_rc_array (QSprob p, double *rc)
```

Arguments

`p` – a handle to an initialized problem.
`rc` – returns the reduced costs of the variables; this field should point to an array of length at least `ncols`, the number of columns in the problem.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred. Note that `QSget_rc_array()` returns an error code if it is called with a specified problem that has not been solved with one of the optimization functions (`QSopt_dual()` or `QSopt_primal()`) since the last time the problem was loaded or modified.

Description

This function returns the reduced costs of the variables in the LP problem as a dense vector, that is, values for all variables (both zero and non-zero) are returned in the array `rc`.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int rval, ncols;
double *rc;

ncols = QSget_colcount (p);
rc = (double *) malloc (ncols * sizeof (double));

rval = QSget_rc_array (p, rc);
if (rval) {
    fprintf (stderr, "Could not get reduced costs, error code %d\n", rval);
} else {
    /* To print reduced costs with names, see QSget_solution() example. */
}

free (rc);
```

QSget_named_x

Obtain the solution value of a named variable.

Synopsis

```
int QSget_named_x (QSprob p, const char *colname, double *val)
```

Arguments

`p` – a handle to an initialized problem.
`colname` – a string specifying the name of a variable.
`val` – returns the solution value of the named variable.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred. Note that `QSget_named_x()` returns an error code if it is called with a specified problem that has not been solved with one of the optimization functions (`QSopt_dual()` or `QSopt_primal()`) since the last time the problem was loaded or modified. An error will also occur if the named variable is not in the LP problem.

Description

After a successful call to one of the optimization routines (`QSopt_dual()` or `QSopt_primal()`), this function can be used to obtain the value of a variable specified by its name.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */
/* Get the solution value of the variable z.                       */

int rval;
double val;

rval = QSget_named_x (p, "z", &val);
if (rval) {
    fprintf (stderr, "Could not get the x-value, error code %d\n", rval);
} else {
    printf ("Solution value = %.6f\n", val);
}
```

QSget_named_rc

Obtain the reduced cost of a named variable.

Synopsis

```
int QSget_named_rc (QSprob p, const char *colname, double *val)
```

Arguments

`p` – a handle to an initialized problem.
`colname` – a string specifying the name of a variable.
`val` – returns the reduced cost of the named variable.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Note that `QSget_named_rc()` returns an error code if it is called with a specified problem that has not been solved with one of the optimization functions (`QSopt_dual()` or `QSopt_primal()`) since the last time the problem was loaded or modified. An error will also occur if the named variable is not in the LP problem.

Description

After a successful call to one of the optimization routines (`QSopt_dual()` or `QSopt_primal()`), this function can be used to obtain the reduced cost of a variable specified by its name.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */
/* Get the reduced cost of the variable z.                         */

int rval;
double val;

rval = QSget_named_rc (p, "z", &val);
if (rval) {
    fprintf (stderr, "Could not get the rc-value, error code %d\n", rval);
} else {
    printf ("Reduced cost = %.6f\n", val);
}
```

QSget_named_pi

Obtain the dual value associated with a named row.

Synopsis

```
int QSget_named_pi (QSprob p, const char *rowname, double *val)
```

Arguments

`p` – a handle to an initialized problem.
`rowname` – a string specifying the name of a constraint.
`val` – returns the dual value associated with the named constraint.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred. Note that `QSget_named_pi()` returns an error code if it is called with a specified problem that has not been solved with one of the optimization functions (`QSopt_dual()` or `QSopt_primal()`) since the last time the problem was loaded or modified. An error will also occur if the named constraint is not in the LP problem.

Description

After a successful call to one of the optimization routines (`QSopt_dual()` or `QSopt_primal()`), this function can be used to obtain the value of the dual variable associated with a constraint specified by name.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */
/* Get the value of the dual variable for constraint r2.           */
```

```
int rval;
double val;

rval = QSget_named_pi (p, "r2", &val);
if (rval) {
    fprintf (stderr, "Could not get the dual value, error code %d\n", rval);
} else {
    printf ("Dual value = %.6f\n", val);
}
```

QSget_named_slack

Obtain the slack value associated with a named row.

Synopsis

```
int QSget_named_slack (QSprob p, const char *rowname, double *val)
```

Arguments

`p` – a handle to an initialized problem.
`rowname` – a string specifying the name of a constraint.
`val` – returns the slack value associated with the named constraint.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred. Note that `QSget_named_slack()` returns an error code if it is called with a specified problem that has not been solved with one of the optimization functions (`QSopt_dual()` or `QSopt_primal()`) since the last time the problem was loaded or modified. An error will also occur if the named constraint is not in the LP problem.

Description

After a successful call to one of the optimization routines (`QSopt_dual()` or `QSopt_primal()`), this function can be used to obtain the slack value (corresponding to the LP solution) for the named constraint.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */
/* Get the slack value for the constraint r2.                      */

int rval;
double val;

rval = QSget_named_slack (p, "r2", &val);
if (rval) {
    fprintf (stderr, "Could not get the slack value, error code %d\n", rval);
} else {
    printf ("Slack value = %.6f\n", val);
}
```

Chapter 5

Modifying an LP Problem

In many applications, a linear-programming problem needs to be modified during the course of the solution procedure, for example when new constraints are discovered or new variables need to be added. The *QSopt* library provides a variety of functions to address this need. Note that this collection of functions can also be used to build a problem from scratch, after a call to `QScreate_prob()` to initialize an empty problem.

QSnew_col

Create a new empty column (variable) in the problem.

Synopsis

```
int  QSnew_col (QSprob p, double obj, double lower, double upper,
     const char *name)
```

Arguments

`p` – a handle to an initialized problem.
`obj` – the objective function coefficient for the variable.
`lower` – the lower bound for the variable (use `-QS_MAXDOUBLE` if no lower bound).
`upper` – the upper bound for the variable (use `QS_MAXDOUBLE` if no upper bound).
`name` – the name of the variable (can be `NULL`).

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

The `QSnew_col()` function can be used to add a new variable to an existing LP problem. This function should be used in cases where the new variable does not appear in the existing rows of the LP problem, and in cases where the variable's coefficients in the existing rows are not yet known in the application. (In other cases, `QSadd_col()` should be used to add the variable.)

As an example of a possible use of `QSnew_col()`, the function can be called to initialize the variables of an LP after a call to `QScreate_prob()`; the constraint matrix is then created with calls to `QSadd_row()`. A call to `QSnew_col()` can also be used before adding a constraint that introduces a new variable to an existing problem, permitting the coefficients of the new variable to be specified in the arguments of `QSadd_row()`.

When specifying `obj`, `lower`, and `upper`, it is important not to give any value larger in magnitude than `QS_MAXDOUBLE` (1e30). This restriction is due to the internal structures used in the *QSopt* solvers.

Example

```
/* Assume p is initialized to the problem in the QSload_prob example. */
/* Add a variable w, with an objective coefficient of 1.1, and with   */
/* 1.0 <= w <= 2.0.                                                    */

int rval;
const char *name = "w";

rval = QSnew_col (p, 1.1, 1.0, 2.0, name);
if (rval) {
    fprintf (stderr, "QSnew_col failed with return code %d\n", rval);
}
```

QSadd_cols

Add a set of columns (variables) to the problem.

Synopsis

```
int QSadd_cols (QSprob p, int num, int *cmatcnt, int *cmatbeg,
    int *cmatind, double *cmatval, double *obj, double *lower,
    double *upper, const char **names)
```

Arguments

`p` – a handle to an initialized problem.
`num` – the number of columns to be added.
`cmatcnt` – an array of length `num` (see `QSload_prob()`).
`cmatbeg` – an array of length `num` (see `QSload_prob()`).
`cmatind` – an array of row indices (see `QSload_prob()`).
`cmatval` – an array of matrix coefficients (see `QSload_prob()`).
`obj` – an array of length `num` specifying objective function coefficients for the variables.
`lower` – an array of length `num` specifying lower bounds for the variables.
`upper` – an array of length `num` specifying upper bounds for the variables.
`names` – an array of length `num` specifying names of variables (`names` can be `NULL`).

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

Use `QSadd_cols()` to add a collection of variables to an LP problem, including the non-zero coefficients of the variables in the existing constraints of the problem. (If the new variables do not appear in the existing rows, then it is simpler to use `QSnew_col()` to add the variables. Also, to add only a single variable, it is simpler to use `QSadd_col()`.) The arguments to specify the constraint coefficients follow the same pattern as in `QSload_prob()`.

When specifying `cmatval`, `obj`, `lower`, and `upper`, it is important not to give any value larger in magnitude than `QS_MAXDOUBLE` (1e30). This restriction is due to the internal structures used in the *QSopt* solvers.

Example

```
/* Assume p is initialized to the problem in the QSload_prob example. */
/* Add variables v and w to obtain the following LP problem.         */
/*      Maximize  3.0x + 2.0y + 4.0z + 5.1v + 1.1w                    */
/*      Subject to                                                    */
/*                3.1x + 2.3y + 1.4z + 3.5v + 2.1w <= 12.2            */
/*                5.0x + 1.1y              + 3.0w  = 10.0             */
/*                x >= 2.0                                            */
/*                y free                                              */
/*                1.0 <= z <= 10.0                                    */
/*                0.0 <= v <= 2.0                                     */
/*                1.0 <= w <= 2.0                                     */

int rval;
int cmatcnt[2] = { 1, 2 };
int cmatbeg[2] = { 0, 1 };
int cmatind[3] = { 0, 0, 1 };
double cmatval[3] = { 3.5, 2.1, 3.0 };
double obj[2] = { 5.1, 1.1 };
double lower[2] = { 0.0, 1.0 };
double upper[2] = { 2.0, 2.0 };
const char *names[2] = { "v", "w" };

rval = QSadd_cols (p, 2, cmatcnt, cmatbeg, cmatind, cmatval, obj,
                  lower, upper, names);
```

```
if (rval) {
    fprintf (stderr, "Add columns failed, error code %d\n", rval);
} else {
    rval = QSwrite_prob (p, "newsmall.lp", "LP");
    if (rval) {
        fprintf (stderr, "Could not write LP, error code %d\n", rval);
    }
}
```

QSadd_col

Add a single column (variable) to the problem.

Synopsis

```
int QSadd_col (QSprob p, int cnt, int *cmatind, double *cmatval,
    double obj, double lower, double upper, const char *name)
```

Arguments

`p` – a handle to an initialized problem.
`cnt` – the number of non-zero entries in the column.
`cmatind` – an array of length `cnt` specifying the row indices of the non-zero entries.
`cmatval` – an array of length `cnt` specifying the values of the matrix coefficients for the non-zero entries.
`obj` – the objective function coefficient for the variable.
`lower` – the lower bound for the variable (use `-QS_MAXDOUBLE` if no lower bound).
`upper` – the upper bound for the variable (use `QS_MAXDOUBLE` if no upper bound).
`name` – the name of the variable (can be `NULL`).

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

The `QSadd_col()` function provides a simpler interface than `QSadd_cols()` for adding a single variable to an existing LP, for example, the argument `obj` is a `double` rather than an array of `doubles` (of length 1 when adding only a single variable).

When specifying `cmatval`, `obj`, `lower`, and `upper`, it is important not to give any value larger in magnitude than `QS_MAXDOUBLE` (1e30). This restriction is due to the internal structures used in the *QSopt* solvers.

Note that when adding many variables, it is more efficient to use `QSadd_cols()` rather than repeated calls to `QSadd_col()`, due to the internal data management in the *QSopt* library.

Example

```
/* Assume p is initialized to the problem in the QSload_prob example. */
/* Add a variable w, with an objective coefficient of 1.1, with a 2.1 */
/* in the first constraint, with a 3.0 in the second constraint, and  */
/* 1.0 <= w <= 2.0.                                                   */

int rval;
const char *name = "w";
int cmatind[2] = { 0, 1 };
int cmatval[2] = { 2.1, 3.0 };

rval = QSadd_col (p, 2, cmatind, cmatval, 1.1, 1.0, 2.0, name);
if (rval) {
    fprintf (stderr, "QSadd_col failed with return code %d\n", rval);
}
```

QSdelete_cols

Delete a set of columns from the problem.

Synopsis

`int QSdelete_cols (QSprob p, int num, int *dellist)`

Arguments

`p` – a handle to an initialized problem.
`num` – the number of columns to be deleted.
`dellist` – an array of length `num` specifying the indices of the columns to be deleted.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

Use this function to delete a set of variables from an existing LP problem, specifying the variables as a list of indices. (Note that the variables are numbered from 0 up to `ncols-1`, where `ncols` is the number of columns in the problem.)

Example

```
/* Assume p is an initialized LP problem with at least 5 columns.  */
/* Use QSdelete_cols() to remove columns 0, 2, and 4.              */

int rval;
int dellist[3] = { 0, 2, 4 };

rval = QSdelete_cols (p, 3, dellist);
if (rval) {
    fprintf (stderr, "QSdelete_cols failed with return code %d\n", rval);
}
```

QSdelete_col

Delete a single column from the problem.

Synopsis

`int QSdelete_col (QSprob p, int colindex)`

Arguments

`p` – a handle to an initialized problem.

`colindex` – the index of the column to be deleted.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

Use this function to delete a single variable from an existing LP problem; the interface is slightly easier than that of `QSdelete_cols()`. (Note that the variables are numbered from 0 up to `ncols-1`, where `ncols` is the number of columns in the problem.)

Example

```
/* Assume p is an initialized LP problem with at least 5 columns.   */
/* Use QSdelete_col() to remove column 4.                           */

int rval;

rval = QSdelete_col (p, 4);
if (rval) {
    fprintf (stderr, "QSdelete_col failed with return code %d\n", rval);
}
```

QSdelete_setcols

Delete a set of columns specified by flags.

Synopsis

```
int QSdelete_setcols (QSprob p, int *flags)
```

Arguments

`p` – a handle to an initialized problem.

`flags` – an array of length `ncols` (the number of columns in the problem) specifying a 0 or 1 for each column; the columns corresponding to the 1's will be deleted.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

Use this function to delete a set of variables (columns) from an existing LP problem, specifying the variables as an array of flags, with a 0 entry indicating that a variable should not be deleted and a 1 entry indicated that a variable should be deleted.

Example

```
/* Assume p is an initialized LP problem with 5 columns.  */
/* Use QSdelete_setcols() to remove columns 0, 2, and 4.  */

int rval;
int flags[5] = { 1, 0, 1, 0, 1 };

rval = QSdelete_setcols (p, flags);
if (rval) {
    fprintf (stderr, "QSdelete_setcols failed with return code %d\n", rval);
}
```

QSdelete_named_column

Delete a column specified by name.

Synopsis

```
int QSdelete_named_column (QSprob p, const char *colname)
```

Arguments

`p` – a handle to an initialized problem.
`colname` – a string specifying the name of a column (variable) to be deleted.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred. If the column is not in the LP problem, an error will be returned.

Description

Use this function to delete a single variable from an existing LP problem by specifying the name of the variable; the interface is slightly easier than that of `QSdelete_named_columns_list()`.

Example

```
  /* Assume p is an initialized LP problem. */
  /* Delete variable z.                     */

  int rval;

  rval = QSdelete_named_column (p, "z");
  if (rval) {
      fprintf (stderr, "could not delete variable, error code %d\n", rval);
  }
```

QSdelete_named_columns_list

Delete a list of columns, specified by names.

Synopsis

```
int QSdelete_named_columns_list (QSprob p, int num, const char **colnames)
```

Arguments

`p` – a handle to an initialized problem.
`num` – the number of columns to be deleted.
`colnames` – an array of length `num`; the entries in the array are strings specifying the names of the columns (variables) to be deleted.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred. If one or more of the columns is not in the LP problem, an error will be returned.

Description

This function can be used to delete a set of variables from an existing LP problem by specifying the names of the variables in a list. It is more efficient to use this function then to use repeated calls to `QSdelete_named_column()`.

Example

```
/* Assume p is an initialized LP problem.  */
/* Delete variables v and w.               */

int rval;
const char *dlist[2] = { "v", "w" };

rval = QSdelete_named_columns_list (p, 2, dlist);
if (rval) {
    fprintf (stderr, "could not delete variables, error code %d\n", rval);
}
```

QSnew_row

Create a new empty row (constraint) in the problem

Synopsis

```
int QSnew_row (QSprob p, double rhs, char sense, const char *name)
```

Arguments

`p` – a handle to an initialized problem.
`rhs` – the right-hand-side value of the constraint.
`sense` – the sense of the constraint (to specify an equation use `'E'`, to specify a $\leq$ constraint use `'L'`, and to specify a $\geq$ constraint use `'G'`).
`name` – the name of the constraint (can be `NULL`).

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

The `QSnew_row()` function can be used to add a new constraint to an existing LP problem. This function should be used in cases where the new constraint has no non-zero coefficients in the existing columns of the LP problem, and in cases where the constraint's coefficients in the existing columns are not yet known in the application. (In other cases, `QSadd_row()` should be used to add the constraint.)

As an example of a possible use of `QSnew_row()`, the function can be called to initialize the constraints of an LP after a call to `QScreate_prob()`; the constraint matrix is then created with calls to `QSadd_col()`.

When specifying `rhs`, it is important not to give any value larger in magnitude than `QS_MAXDOUBLE` (1e30). This restriction is due to the internal structures used in the *QSopt* solvers.

Example

```
/* Assume p is initialized to the problem in the QSload_prob example. */
/* Add a >= constraint, with right-hand-side value 6.0 (and no name   */
/* specified for the row).                                             */

int rval;

rval = QSnew_row (p, 6.0, 'G', (const char *) NULL);
if (rval) {
    fprintf (stderr, "QSnew_row failed with return code %d\n", rval);
}
```

QSadd_rows

Add a set of rows (constraints) to the problem.

Synopsis

```
int QSadd_rows (QSprob p, int num, int *rmatcnt, int *rmatbeg,
    int *rmatind, double *rmatval, double *rhs, char *sense,
    char **names)
```

Arguments

`p` – a handle to an initialized problem.

`num` – the number of rows to be added.

`rmatcnt` – an array of length `num`; the ith entry specifies the number of non-zero coefficients in the ith row to be added to the LP problem.

`rmatbeg` – an array of length `num`; the ith entry specifies the location of the start of the ith row in the `rmatind` and `rmatval` arrays.

`rmatind` – an array that contains the column indices of the non-zero coefficients in the rows to be added. The indices for the ith row must be stored consecutively starting at entry number `rmatbeg[i]` (there are `rmatcnt[i]` indices for the ith row).

`rmatval` – an array that contains the values of the non-zero coefficients in the rows to be added. The coefficients for the ith row must be stored consecutively starting at entry number `rmatbeg[i]` (there are `rmatcnt[i]` non-zero coefficients for the ith row).

`rhs` – an array of length `num`; the ith entry is the right-hand-side value of the ith constraint.

`sense` – an array of length `num`; the ith entry specifies the sense of the ith constraint; to specify an equation use `'E'`, to specify a $\leq$ constraint use `'L'`, and to specify a $\geq$ constraint use `'G'`.

`names` – an array of length `num`; the ith entry is a string that specifies the name of the ith constraint to be added (the `names` field can be `NULL`).

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

Use `QSadd_rows()` to add a collection of constraints to an LP problem, including the non-zero coefficients of the constraints in the existing columns of the problem. The arguments to specify the non-zero coefficients follow the same pattern as in `QSload_prob()`.

When specifying `rmatval` and `rhs`, it is important not to give any value larger in magnitude than `QS_MAXDOUBLE` (1e30). This restriction is due to the internal structures used in the *QSopt* solvers.

Note that to add single constraint, it is simpler to use `QSadd_row()`.)

Example

```
/* Assume p is initialized to the problem in the QSload_prob example. */
/* Add two new rows to obtain the following LP.                       */
/*      Maximize  3.0x + 2.0y + 4.0z                                  */
/*      Subject to                                                    */
/*                3.1x + 2.3y + 1.4z <= 12.2                          */
/*                5.0x + 1.1y        = 10.0                           */
/*                0.5x + 2.0y + 4.0z >=  6.0    New                   */
/*                       3.0y + 2.0z <=  8.0    New                   */
/*                x >= 2.0                                            */
/*                y free                                              */
/*                1.0 <= z <= 10.0                                    */
/*                                                                    */

int rval;
int rmatcnt[2] = { 3, 2 };
int rmatbeg[2] = { 0, 3 };
int rmatind[5] = { 0, 1, 2, 1, 2 };
double rmatval[5] = { 0.5, 2.0, 4.0, 3.0, 2.0 };
double rhs[2] = { 6.0, 8.0 };
char sense[2] = { 'G', 'L' };

rval = QSadd_rows (p, 2, rmatcnt, rmatbeg, rmatind, rmatval, rhs,
                   sense, (char **) NULL);
if (rval) {
    fprintf (stderr, "Add rows failed, error code %d\n", rval);
} else {
    rval = QSwrite_prob (p, "new2small.lp", "LP");
    if (rval) {
        fprintf (stderr, "Could not write LP, error code %d\n", rval);
    }
}
```

QSadd_row

Add a single row (constraint) to the problem.

Synopsis

```
int QSadd_row (QSprob p, int cnt, int *rmatind, double *rmatval,
    double rhs, char sense, const char *name)
```

Arguments

`p` – a handle to an initialized problem.

`cnt` – the number of non-zero entries in the row.

`rmatind` – an array of length `cnt` specifying the column indices of the non-zero entries.

`rmatval` – an array of length `cnt` specifying the values of the matrix coefficients for the non-zero entries.

`rhs` – the right-hand-side value of the constraint.

`sense` – the sense of the constraint; to specify an equation use `'E'`, to specify a $\leq$ constraint use `'L'`, and to specify a $\geq$ constraint use `'G'`.

`name` – the name of the constraint (can be `NULL`).

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

The `QSadd_row()` function provides a simpler interface than `QSadd_rows()` for adding a single constraint to an existing LP, for example, the argument `rhs` is a `double` rather than an array of `doubles` (of length 1 when adding only a single constraint).

When specifying `rmatval` and `rhs`, it is important not to give any value larger in magnitude than `QS_MAXDOUBLE` (1e30). This restriction is due to the internal structures used in the *QSopt* solvers.

Note that when adding many constraints, it is more efficient to use `QSadd_rows()` rather than repeated calls to `QSadd_row()`, due to the internal data management in the *QSopt* library.

Example

```
/* Assume p is initialized to the problem in the QSload_prob example. */
/* Add the constraint 0.5x + 2.0y + 4.0z >= 6.0 to the LP.           */

int rval;
int rmatind[3] = { 0, 1, 2 };
int rmatval[3] = { 0.5, 2.0, 4.0 };

rval = QSadd_row (p, 2, rmatind, rmatval, 6.0, 'G', (const char *) NULL);
if (rval) {
    fprintf (stderr, "QSadd_row failed with return code %d\n", rval);
}
```

QSdelete_rows

Delete a set of rows from the problem.

Synopsis

```
int QSdelete_rows (QSprob p, int num, int *dellist)
```

Arguments

`p` – a handle to an initialized problem.
`num` – the number of rows to be deleted.
`dellist` – an array of length `num` specifying the indices of the rows to be deleted.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

Use this function to delete a set of constraints from an existing LP problem, specifying the constraints as a list of indices. (Note that the constraints are numbered from 0 up to `nrows-1`, where `nrows` is the number of rows in the problem.)

Example

```
/* Assume p is an initialized LP problem with at least 4 rows.  */
/* Use QSdelete_rows() to remove rows 1 and 3.                  */

int rval;
int dellist[2] = { 1, 3 };

rval = QSdelete_rows (p, 2, dellist);
if (rval) {
    fprintf (stderr, "QSdelete_rows failed with return code %d\n", rval);
}
```

QSdelete_row

Delete a single row from the problem.

Synopsis

`int QSdelete_row (QSprob p, int rowindex)`

Arguments

`p` – a handle to an initialized problem.
`rowindex` – the index of the row to be deleted.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

Use this function to delete a single constraint from an existing LP problem; the interface is slightly easier than that of `QSdelete_rows()`. (Note that the constraints are numbered from 0 up to `nrows-1`, where `nrows` is the number of rows in the problem.)

Example

```
/* Assume p is an initialized LP problem with at least 4 rows.  */
/* Use QSdelete_row() to remove row 3.                          */

int rval;

rval = QSdelete_row (p, 3);
if (rval) {
    fprintf (stderr, "QSdelete_row failed with return code %d\n", rval);
}
```

QSdelete_setrows

Delete a set of rows specified by flags.

Synopsis

`int QSdelete_setrows (QSprob p, int *flags)`

Arguments

`p` – a handle to an initialized problem.

`flags` – an array of length `nrows` (the number of rows in the problem) specifying a 0 or 1 for each row; the rows corresponding to the 1's will be deleted.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

Use this function to delete a set of constraints from an existing LP problem, specifying the constraints as an array of flags, with a 0 entry indicating that a constraint should not be deleted and a 1 entry indicated that a constraint should be deleted.

Example

```
/* Assume p is an initialized LP problem with 4 rows.  */
/* Use QSdelete_setrows() to remove rows 1 and 3.      */

int rval;
int flags[4] = { 0, 1, 0, 1 };

rval = QSdelete_setrows (p, flags);
if (rval) {
    fprintf (stderr, "QSdelete_setrows failed with return code %d\n", rval);
}
```

QSdelete_named_row

Delete a row specified by name.

Synopsis

```
int QSdelete_named_row (QSprob p, const char *rowname)
```

Arguments

`p` – a handle to an initialized problem.
`rowname` – a string specifying the name of a row (constraint) to be deleted.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred. If the row is not in the LP problem, an error will be returned.

Description

Use this function to delete a single constraint from an existing LP problem by specifying the name of the constraint; the interface is slightly easier than that of `QSdelete_named_rows_list()`.

Example

```
/* Assume p is an initialized LP problem.  */
/* Delete constraint r2.                   */

int rval;

rval = QSdelete_named_row (p, "r2");
if (rval) {
    fprintf (stderr, "could not delete row, error code %d\n", rval);
}
```

QSdelete_named_rows_list

Delete a list of rows, specified by names.

Synopsis

```
int QSdelete_named_rows_list (QSprob p, int num, const char **rownames)
```

Arguments

`p` – a handle to an initialized problem.
`num` – the number of rows to be deleted.
`rownames` – an array of length `num`; the entries in the array are strings specifying the names of the rows (constraints) to be deleted.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred. If one or more of the rows is not in the LP problem, an error will be returned.

Description

This function can be used to delete a set constraints from an existing LP problem by specifying the names of the constraints in a list. It is more efficient to use this function then to use repeated calls to `QSdelete_named_row()`.

Example

```
  /* Assume p is an initialized LP problem.  */
  /* Delete constraints r0 and r2.           */

  int rval;
  const char *dlist[2] = { "r0", "r2" };

  rval = QSdelete_named_rows_list (p, 2, dlist);
  if (rval) {
      fprintf (stderr, "could not delete rows, error code %d\n", rval);
  }
```

QSchange_coef

Change a coefficient in the constraint matrix.

Synopsis

```
int QSchange_coef (QSprob p, int rowindex, int colindex, double coef)
```

Arguments

`p` – a handle to an initialized problem.
`rowindex` – the index of the row containing the coefficient.

`colindex` – the index of the column containing the coefficient.
`coef` – the new value of the coefficient.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

Use this function to change a single coefficient in the constraint matrix; it can be used to modify an existing non-zero coefficient or to add a new non-zero coefficient.

Note that the function assumes that both the row and column exist; use `QSadd_row` or `QSadd_col` if the coefficient introduces a new row or column.

Example

```
/* Change the matrix coefficient in row 1, col 2, to 1.5   */

int rval = 0;

rval = QSchange_coef (p, 1, 2, 1.5);
if (rval) {
    fprintf (stderr, "could not change coef, error code %d\n", rval);
}
```

QSchange_objcoef

Change a coefficient in the objective function.

Synopsis

```
int QSchange_objcoef (QSprob p, int indx, double coef)
```

Arguments

`p` – a handle to an initialized problem.
`indx` – the index of the variable whose objective coefficient will be changed.
`coef` – the new value of the coefficient.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

Use this function to change a single coefficient in the objective function; it can be used to modify an existing non-zero coefficient or to add a new non-zero coefficient to the objective.

Note that the function assumes that the column (variable) exists; use `QSnew_col` if the coefficient introduces a new column.

Example

```
/* Change the objective coefficient of variable y to 5.0   */

int rval = 0;
int yindex;

/* Find the index of of y */
```

```
rval = QSget_column_index (p, "y", &yindex);
if (rval) {
    fprintf (stderr, "could not get index, error code %d\n", rval);
} else {
    if (yindex == -1) {
        printf ("y is not a variable in the problem\n");
    } else {
        rval = QSchange_objcoef (p, yindex, 5.0);
        if (rval) {
            fprintf (stderr, "could not change coef, error %d\n", rval);
        }
    }
}
```

QSchange_objsense

Change the sense of the objective function.

Synopsis

```
int QSchange_objsense (QSprob p, int newsense)
```

Arguments

`p` – a handle to an initialized problem.

`newsense` – the new sense of the objective; to specify a minimization problem set `newsense` to `QS_MIN` (that is, 1) and to specify a maximization problem set `newsense` to `QS_MAX` (that is, -1).

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

Use this function to switch from minimizing the objective function to maximizing the objective function, and vice versa.

Example

```
/* p is an initialized QSprob, change it to maximization */

int rval;

rval = QSchange_sense (p, QS_MAX);
if (rval) {
    fprintf (stderr, "QSchange_objsense failed with return code %d\n", rval);
}
```

QSchange_rhscoef

Change a coefficient in the right-hand-side vector.

Synopsis

```
int QSchange_rhscoef (QSprob p, int indx, double coef)
```

Arguments

`p` – a handle to an initialized problem.
`indx` – the index of the row whose right-hand-side coefficient will be changed.
`coef` – the new value of the coefficient.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

Use this function to change a single coefficient in the right-hand-side vector; it can be used to modify an existing non-zero coefficient or to add a new non-zero coefficient to the right-hand-side vector.

Note that the function assumes that the row (constraint) exists; use `QSnew_row` if the coefficient introduces a new row.

Example

```
/* Change the right-hand-side coefficient in row 1 to 7.1  */

int rval = 0;

rval = QSchange_rhscoef (p, 1, 7.1);
if (rval) {
    fprintf (stderr, "could not change rhs value, error code %d\n", rval);
}
```

QSchange_senses

Change the sense of a set of constraints.

Synopsis

```
int QSchange_senses (QSprob p, int num, int *rowlist, char *sense)
```

Arguments

`p` – a handle to an initialized problem.
`num` – the number of constraints to be changed.
`rowlist` – an array of length `num` specifying the indices of the constraints to be changed.
`sense` – an array of length `num` specifying the new sense of each of the constraints; entries should be `'E'`, `'L'`, or `'G'`.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

The sense of each constraint in an LP problem is either $=$, $\leq$, or $\geq$. This function can be used to change the sense of each constraint in a given list, specifying new values using the encoding `'E'` for $=$, `'L'` for $\leq$, and `'G'` for $\geq$. To change the sense of a single constraint it is easier to use the function `QSchange_sense()`.

Example

```
/* Assume p is initialized to the problem in the QSload_prob example. */
/* Change the first constraint to an "=" and change the second       */
/* constraint to a "<=".                                              */

int rval;
int rowlist[2] = { 0, 1 };
char sense[2] = { 'E', 'L' };

rval = QSchange_senses (p, 2, rowlist, sense);
if (rval) {
    fprintf (stderr, "Could not change the senses, error code %d\n",rval);
}
```

QSchange_sense

Change the sense of a single constraint.

Synopsis

`int QSchange_sense (QSprob p, int rowindex, char sense)`

Arguments

`p` – a handle to an initialized problem.
`rowindex` – the index of the constraint to be changed.
`sense` – the new sense of the constraint; the value should be `'E'`, `'L'`, or `'G'`.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

The sense of each constraint in an LP problem is either $=$, $\leq$, or $\geq$. This function can be used to change the sense of a single constraint by specifying a new value using the encoding `'E'` for $=$, `'L'` for $\leq$, and `'G'` for $\geq$.

Example

```
/* Assume p is initialized to the problem in the QSload_prob example. */
/* Change the second constraint to "<=".                              */

int rval;

rval = QSchange_sense (p, 1, 'L');
if (rval) {
    fprintf (stderr, "Could not change the sense, error code %d\n",rval);
}
```

QSchange_bounds

Change the lower or upper bounds for a set of variables.

Synopsis

```
int QSchange_bounds (QSprob p, int num, int *collist, char *lu,
    double *bounds)
```

Arguments

`p` – a handle to an initialized problem.
`num` – the number of bounds to be changed.
`collist` – an array of length `num` specifying the indices of the variables whose bounds will be changed.
`lu` – an array of length `num` specifying the type of each bound to be changed (`'L'` for a lower bound, `'U'` for an upper bound).
`bounds` – an array of length `num` specifying the values of the new bounds.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

This function can be used to change the lower and upper bounds for a list of variables. Bounds on variables can be removed by specifying `-QS_MAXDOUBLE` as a lower bound, or `QS_MAXDOUBLE` as an upper bound. (To change a bound on a single variable it is easier to use the function `QSchange_bound()`.)

When specifying the upper and lower bounds, it is important not to give any value larger in magnitude than `QS_MAXDOUBLE` (1e30). This restriction is due to the internal structures used in the *QSopt* solvers.

Example

```
/* Assume p is initialized to the problem in the QSload_prob example. */
/* Change the bounds on variable y to 0 <= y <= 1, and change the     */
/* upper bound on x to 3, giving x the bounds 2 <= x <= 3.            */

int rval;
int collist[3] = { 1, 1, 0};
char lu[3] = { 'L', 'U', 'U'};
double bounds[3] = { 0.0, 1.0, 3.0 };

rval = QSchange_bounds (p, 3, collist, lu, bounds);
if (rval) {
    fprintf (stderr, "Could not change the bounds, error code %d\n",rval);
}
```

QSchange_bound

Change the lower or upper bound for a single variable.

Synopsis

```
int QSchange_bound (QSprob p, int indx, char lu, double bound)
```

Arguments

`p` – a handle to an initialized problem.
`indx` – the index of the variable whose bound will be changed.
`lu` – an `'L'` to change the lower bound or a `'U'` to change an upper bound.

`bound` – the value of the new bound.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

This function can be used to change the lower or upper bound of a single variable. A bound on a variable can be removed by specifying `-QS_MAXDOUBLE` as a lower bound, or `QS_MAXDOUBLE` as an upper bound. A common use of this function is to fix a variable at its lower or upper bound, or to change a bound in a step of a branch-and-bound algorithm.

When specifying the lower or upper bound, it is important not to give any value larger in magnitude than `QS_MAXDOUBLE` (1e30). This restriction is due to the internal structures used in the *QSopt* solvers.

Example

```
/* Assume p is initialized to the problem in the QSload_prob example. */
/* Change the lower bound on variable y to 0.                          */

int rval;
rval = QSchange_bound (p, 1, 'L', 0.0);
if (rval) {
    fprintf (stderr, "Could not change the bound, error code %d\n",rval);
}
```

Chapter 6

Accessing LP Problem Data

The *QSopt* library provides a number of functions for obtaining information about the content of the problem associated with a specified `QSprob` object.

		Page
QSget_probname	copy the problem name	45
QSget_objname	copy the objective name	46
QSget_colcount	return the number of columns (variables) in the problem	47
QSget_rowcount	return the number of rows (constraints) in the problem	47
QSget_nzcount	return the number of non-zeros in the constraint matrix	48
QSget_obj	copy the objective function coefficients into an array	48
QSget_rhs	copy the right-hand-side values into an array	49
QSget_bound	obtain a lower or upper bound for a variable	50
QSget_bounds	copy the lower and upper bounds into arrays	51
QSget_columns	copy all columns	52
QSget_columns_list	copy a list of columns	54
QSget_rows	copy all rows	55
QSget_rows_list	copy a list of rows	57
QSget_column_index	obtain the index of a named column	58
QSget_row_index	obtain the index of a named row	59
QSget_colnames	copy the names of the columns in the problem	60
QSget_rownames	copy the names of the rows in the problem	61
QSget_intcount	obtain the number of integer variables	62
QSget_intflags	get flags indicating the integer variables	63

QSget_probname

Copy the problem name.

Synopsis

```
char* QSget_probname (QSprob p)
```

Arguments

`p` – a handle to an initialized problem.

Returns

A pointer to a string specifying the name of the problem. The function returns `NULL` if an error occurred.

Description

The pointer returned by `QSget_probname()` is a handle to a string that is allocated in the function. When the string is no longer needed, the memory should be freed by the user; since the memory for the string is allocated by the *QSopt* library, the memory must be freed with a call to `QSfree()` and not by the system `free()` function—see the documentation for `QSfree()` on page 73. (Note that without the call to `QSfree ()`, the *QSopt* library will not free the memory for the string, even after a call to `QSfree_prob(p)`.

Example

```
char *name;

/* p is an initialized QSprob */

name = QSget_probname (p);
if (name == (char *) NULL) {
    fprintf (stderr, "Could not obtain the name of the LP\n");
} else {
    printf ("Problem Name: %s\n", name);
    QSfree (name);  /* Use QSfree for mem allocated by QSopt */
}
```

QSget_objname

Copy the objective name.

Synopsis

```
char* QSget_objname (QSprob p)
```

Arguments

`p` – a handle to an initialized problem.

Returns

A pointer to a string specifying the name of the objective for the problem. The function returns `NULL` if an error occurred.

Description

The pointer returned by `QSget_objname()` is a handle to a string that is allocated in the function. When the string is no longer needed, the memory should be freed by the user; since the memory for the string is allocated by the *QSopt* library, the memory must be freed with a call to `QSfree()` and not by the system `free()` function—see the documentation for `QSfree()` on page 73. (Note that without the call to `QSfree ()`, the *QSopt* library will not free the memory for the string, even after a call to `QSfree_prob(p)`.

Example

```
char *objname;

/* p is an initialized QSprob */

objname = QSget_objname (p);
if (objname == (char *) NULL) {
    fprintf (stderr, "Could not obtain the objective name for the LP\n");
} else {
    printf ("Objective Name: %s\n", objname);
    QSfree (objname);  /* Use QSfree for mem allocated by QSopt */
}
```

QSget_colcount

Return the number of columns in the problem.

Synopsis

```
int QSget_colcount (QSprob p)
```

Arguments

p – a handle to an initialized problem.

Returns

The number of columns (variables) in the problem. If p has not been initialized, QSget_colcount() will return 0.

Description

The value returned by QSget_colcount() is the number of variables in the problem, it does not include any slack variables or artificial variables that may have been added during the solution procedure. A common use of this function is to obtain the value of ncols in order to use the functions that access the components of the LP solution.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int ncols;

ncols = QSget_colcount (p);
printf ("Number of variables: %d\n", ncols);
```

QSget_rowcount

Return the number of rows (constraints) in the problem.

Synopsis

```
int QSget_rowcount (QSprob p)
```

Arguments

p – a handle to an initialized problem.

Returns

The number of rows (constraints) in the problem. If p has not been initialized, `QSget_rowcount()` will return 0.

Description

The value returned by `QSget_rowcount()` is the number of constraints in the problem; it does not include the objective row. A common use of this function is to obtain the value of `nrows` in order to use the functions that access the components of the LP solution.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int nrows;

nrows = QSget_rowcount (p);
printf ("Number of constraints: %d\n", nrows);
```

QSget_nzcount

Return the number of non-zeros in the constraint matrix.

Synopsis

```
int QSget_nzcount (QSprob p)
```

Arguments

p – a handle to an initialized problem.

Returns

The number of non-zero entries in the constraint matrix. If p has not been initialized, `QSget_nzcount()` will return 0.

Description

The value returned by `QSget_nzcount()` is the number of non-zero entries in the constraint matrix, it does not include the non-zero coefficients in the objective function.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int nzcount;

nzcount = QSget_nzcount (p);
printf ("Number of non-zero constraint coefficients: %d\n", nzcount);
```

QSget_obj

Copy the objective function coefficients into an array.

Synopsis

```
int QSget_obj (QSprob p, double *obj)
```

Arguments

`p` – a handle to an initialized problem.
`obj` – returns the objective function coefficients as an array; this field should point to an array of length at least `ncols`, the number of columns in the problem.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

This function returns the coefficients of the objective function as a dense vector, that is, all coefficients (both zero and non-zero) are returned in the array `obj`. The calling function must allocate the memory for the `obj` array; it should be of length at least `ncols`.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int rval, ncols;
double *obj;

ncols = QSget_colcount (p);
obj = (double *) malloc (ncols * sizeof (double));

rval = QSget_obj (p, obj);
if (rval) {
    fprintf (stderr, "Could not get objective, error code %d\n", rval);
} else {
    printf ("Objective Function\n");
    for (i = 0; i < ncols; i++) {
        printf ("%f\n", obj[i]);
    }
}

free (obj);
```

QSget_rhs

Copy the right-hand-side values into an array.

Synopsis

```
int QSget_rhs (QSprob p, double *rhs)
```

Arguments

`p` – a handle to an initialized problem.

`rhs` – returns the right-hand-side values as an array; this field should point to an array of length at least `nrows`, the number of rows in the problem.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

This function returns the right-hand-side values of the constraints (the b values in the linear system $Ax \leq b$) as a dense vector, that is, all coefficients (both zero and non-zero) are returned in the array `rhs`. The calling function must allocate the memory for the `rhs` array; it should be of length at least `nrows`, the number of rows in the problem.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int rval, nrows;
double *rhs;

nrows = QSget_rowcount (p);
rhs = (double *) malloc (nrows * sizeof (double));

rval = QSget_rhs (p, rhs);
if (rval) {
    fprintf (stderr, "Could not get right-hand-side, error code %d\n", rval);
} else {
    printf ("Right-Hand-Side\n");
    for (i = 0; i < nrows; i++) {
        printf ("%f\n", rhs[i]);
    }
}

free (rhs);
```

QSget_bound

Obtain a lower or upper bound for a variable.

Synopsis

`int QSget_bound (QSprob p, int colindex, char lu, double *bound)`

Arguments

`p` – a handle to an initialized problem.
`colindex` – the index of the variable.
`lu` – an `'L'` to obtain a lower bound or a `'U'` to obtain an upper bound.
`bound` – returns the value of the specified lower or upper bound.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

This function can be used to obtain the lower or upper bound on a single variable. If the bound on the specified variable does not exist, `bound` will be set to `-QS_MAXDOUBLE` in the case of lower bounds and set to `QS_MAXDOUBLE` in the case of upper bounds.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int rval;
double bound;

rval = QSget_bound (p, 2, 'U', &bound);
if (rval) {
    fprintf (stderr, "could not obtain bound, error code %d\n", rval);
} else {
    printf ("Upper bound of variable 2 is %f\n", bound);
}
```

QSget_bounds

Copy the lower and upper bounds into arrays.

Synopsis

```
int QSget_bounds (QSprob p, double *lower, double *upper)
```

Arguments

`p` – a handle to an initialized problem.

`lower` – returns the variable lower bounds as an array; this field can be NULL, if it is not NULL then it should point to an array of length at least `ncols`, the number of columns in the problem.

`upper` – returns the variable upper bounds as an array; this field can be NULL, if it is not NULL then it should point to an array of length at least `ncols`.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

This function returns the lower and upper bounds on all variables (if one of the two arrays is not needed, its argument can be set to NULL). The calling function must allocate the memory for the two arrays (unless they are set to NULL); the arrays should be of length at least `ncols`, the number of columns in the problem. If some variable does not have a lower bound, the corresponding value of `lower` will be set to `-QS_MAXDOUBLE`. Similarly, if some variable does not have an upper bound, the corresponding value of `upper` will be set to `QS_MAXDOUBLE`.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int j, rval, ncols;
double *lower, *upper;

ncols = QSget_colcount (p);
lower = (double *) malloc (ncols * sizeof (double));
upper = (double *) malloc (ncols * sizeof (double));
```

```
rval = QSget_bounds (p, lower, upper);
if (rval) {
    fprintf (stderr, "could not obtain bounds, error code %d\n", rval);
} else {
    for (j = 0; j < ncols; j++) {
        if (lower[j] == -QS_MAXDOUBLE) {
            printf ("-infinity");
        } else {
            printf ("%f", lower[j]);
        }
          printf (" <= Variable %d <= ", j);
        if (upper[j] == QS_MAXDOUBLE) {
            printf ("+infinity\n");
        } else {
            printf ("%f\n", upper[j]);
        }
    }
}

free (lower);
free (upper);
```

QSget_columns

Copy all columns.

Synopsis

```
int QSget_columns (QSprob p, int **colcnt, int **colbeg, int **colind,
    double **colval, double **obj, double **lower, double **upper,
    char ***names)
```

Arguments

`p` – a handle to an initialized problem.

`colcnt` – returns an array of length `ncols` (the number of columns in the problem), with the jth entry specifying the number of non-zero coefficients in the jth column of the constraint matrix; this field can be `NULL`.

`colbeg` – returns an array of length `ncols`, with the jth entry specifying the location of the start of the jth column in the `colind` and `colval` arrays; this field can be `NULL`.

`colind` – returns an array that contains the row indices of the columns; the indices for jth column are stored consecutively starting at entry number `colbeg[j]` (there are `colcnt[j]` indices for the jth column); this field can be `NULL`.

`colval` – returns an array that contains the non-zero coefficients in the columns; the coefficients for the jth column are stored consecutively starting at entry number `colbeg[j]` (there are `colcnt[j]` coefficients for jth column); this field can be `NULL`.

`obj` – returns an array of length `ncols`, giving the objective function coefficients; this field can be `NULL`.

`lower` – returns an array of length `ncols`, giving the lower bounds for the variables (if a variable has no lower bound, the value `-QS_MAXDOUBLE` is given); this field can be `NULL`.

`upper` – returns an array of length `ncols`, giving the upper bounds for the variables (if a variable has no upper bound, the value `QS_MAXDOUBLE` is given); this field can be `NULL`.

`names` – returns an array of length `ncols`, where the jth entry is a string that specifies the name of jth variable; this field can be `NULL`.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

A complete description of the variables for the problem can be obtained using this function. Each of the arguments returns an array that is allocated by the function `QSget_columns()`; if some piece of information is not needed, the corresponding argument for the function should be set to `NULL`. The calling function should free the memory for each of the arrays (and for the strings in the array `names`) when the information is no longer needed; since the memory for the arrays and strings is allocated by the *QSopt* library, the memory must be freed with calls to `QSfree()` and not by the system `free()` function—see the documentation for `QSfree()` on page 73. The form of the column information is the same as that used in the `QSload_prob()` function.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

double *colval = NULL, *obj = NULL, *lower = NULL, *upper = NULL;
int *colcnt = NULL, *colbeg = NULL, *colind = NULL;
char **names = NULL;
int ncols, j, k, rval;

ncols = QSget_colcount (p);
rval = QSget_columns (p, &colcnt, &colbeg, &colind, &colval, &obj, &lower,
                      &upper, &names);
if (rval) {
    fprintf (stderr, "could not obtain cols, error code %d\n", rval);
} else {
    for (j = 0; j < ncols; j++) {
        printf ("%s OBJ = %f, LOWER = %f, UPPER = %f\n",
                names[j], obj[j], lower[j], upper[j]);
        printf ("   Coefficients: ");
        for (k = colbeg[j]; k < colbeg[j] + colcnt[j]; k++) {
            printf ("(%d, %f) ", colind[k], colval[k]);
        }
        printf ("\n");
    }
}

QSfree (colcnt); QSfree (colbeg); QSfree (colind); QSfree (colval);
QSfree (obj);
QSfree (lower); QSfree (upper);
if (names) {
    for (j = 0; j < ncols; j++) {
        QSfree (names[j]);
    }
    QSfree (names);
}
```

QSget_columns_list

Copy a list of columns.

Synopsis

```
int QSget_columns_list (QSprob p, int num, int *collist,
    int **colcnt, int **colbeg, int **colind, double **colval,
    double **obj, double **lower, double **upper, char ***names)
```

Arguments

`p` – a handle to an initialized problem.

`num` – the length of the array `collist`.

`collist` – an array of length `num`; each entry specifies the index of a column (so a value between 0 and `ncols - 1`, where `ncols` is the number of columns in the problem).

`colcnt` – returns an array of length `num`, with the jth entry specifying the number of non-zero coefficients in column `collist[j]` of the constraint matrix; this field can be `NULL`.

`colbeg` – returns an array of length `num`, with the jth entry specifying the location of the start of column `collist[j]` in the `colind` and `colval` arrays; this field can be `NULL`.

`colind` – returns an array that contains the row indices of the non-zero coefficients in the columns specified in the `collist` array; the indices for column `collist[j]` are stored consecutively starting at entry number `colbeg[j]` (there are `colcnt[j]` indices for `collist[j]`); this field can be `NULL`.

`colval` – returns an array that contains the non-zero coefficients in the columns specified by the `collist` array; the coefficients for column `collist[j]` are stored consecutively starting at entry number `colbeg[j]` (there are `colcnt[j]` coefficients for `collist[j]`); this field can be `NULL`.

`obj` – returns an array of length `num`, giving the objective function coefficients for the columns specified in the `collist` array; this field can be `NULL`.

`lower` – returns an array of length `num`, giving the lower bounds for the columns specified in the `collist` array (if a variable has no lower bound, the value `-QS_MAXDOUBLE` is given); this field can be `NULL`.

`upper` – returns an array of length `num`, giving the upper bounds for the columns specified in the `collist` array (if a variable has no upper bound, the value `QS_MAXDOUBLE` is given); this field can be `NULL`.

`names` – returns an array of length `num` where the jth entry is a string that specifies the name of variable `collist[j]`; this field can be `NULL`.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

A complete description of the subset of the variables for the problem can be obtained using this function; the indices for the variables are passed into the function via the `collist` array (the indices can be obtained from the corresponding column names by using the function `QSget_column_index()`). Each of the arguments returns an array that is allocated by the function `QSget_columns_list()`; if some piece of information is not needed, the corresponding argument for the function should be set to `NULL`. The calling function should free the memory for each of the arrays (and for the strings in the array `names`) when the information is no longer needed; since the memory for the arrays and strings is allocated by the *QSopt* library, the memory must be freed with calls to `QSfree()` and not by the system `free()` function—see the documentation for `QSfree()`

on page 73. The form of the column information is the same as that used in the `QSload_prob()` function.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */
/* Obtain the column information for column 2.                     */

double *colval = NULL, *obj = NULL, *lower = NULL, *upper = NULL;
int *colcnt = NULL, *colbeg = NULL, *colind = NULL;
char **names = NULL;
int collist[1] = { 2 };
int k, rval;

rval = QSget_columns_list (p, 1, collist, &colcnt, &colbeg, &colind,
                           &colval, &obj, &lower, &upper, &names);
if (rval) {
    fprintf (stderr, "could not obtain column, error code %d\n", rval);
} else {
    printf ("Variable %s, objective %f, bounds %f <= %s <= %f\n",
             names[0], obj[0], lower[0], names[0], upper[0]);
    printf ("Coefficients: ");
    for (k = colbeg[0]; k < colbeg[0] + colcnt[0]; k++) {
        printf ("Row %d = %f  ", colind[k], colval[k]);
    }
    printf ("\n");
}

QSfree (colcnt); QSfree (colbeg); QSfree (colind); QSfree (colval);
QSfree (obj);
QSfree (lower); QSfree (upper);
if (names) {
    QSfree (names[0]);
    QSfree (names);
}
```

QSget_rows

Copy all rows.

Synopsis

```
int QSget_rows (QSprob p, int **rowcnt, int **rowbeg, int **rowind,
    double **rowval, double **rhs, char **sense, char ***names)
```

Arguments

`p` – a handle to an initialized problem.

`rowcnt` – returns an array of length `nrows`, the number of rows in the problem, with the ith entry specifying the number of non-zero coefficients in the ith row of the constraint matrix; this field can be `NULL`.

`rowbeg` – returns an array of length `nrows`, with the ith entry specifying the location of the start of the ith row in the `rowind` and `rowval` arrays; this field can be `NULL`.

`rowind` – returns an array that contains the column indices of the non-zero coefficients in the rows; the indices for the ith row are stored consecutively starting at entry number `rowbeg[i]` (there are `rowcnt[i]` indices for the ith row); this field can be `NULL`.

`rowval` – returns an array that contains the non-zero coefficients in the rows; the coefficients for the ith row are stored consecutively starting at entry number `rowbeg[i]` (there are `rowcnt[i]` coefficients for the ith row); this field can be `NULL`.

`rhs` – returns the right-hand-side values as an array of length `nrows`; this field can be `NULL`.

`sense` – returns an array of length `nrows` where the ith entry specifies the sense of the ith constraint (`'E'` for =, `'L'` for $\leq$; `'G'` for $\geq$); this field can be `NULL`.

`names` – returns an array of length `nrows` where the ith entry is a string that specifies the name of the ith constraint; this field can be `NULL`.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

A complete description of the constraints for the problem can be obtained using this function. Each of the arguments returns an array that is allocated by `QSget_rows()`; if some piece of information is not needed, the corresponding argument for the function should be set to `NULL`. The calling function should free the memory for each of the arrays (and for the strings in the `names` array) when the information is no longer needed; since the memory for the arrays and strings is allocated by the *QSopt* library, the memory must be freed with calls to `QSfree()` and not by the system `free()` function—see the documentation for `QSfree()` on page 73. The form of the row information is the same as that used in the `QSadd_rows()` function.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

double *rowval = NULL, *rhs = NULL;
int *rowcnt = NULL, *rowbeg = NULL, *rowind = NULL;
char *sense = NULL, **names = NULL;
int nrows, i, j, rval;

nrows = QSget_rowcount (p);
rval = QSget_rows (p, &rowcnt, &rowbeg, &rowind, &rowval, &rhs, &sense,
                   &names);
if (rval) {
    fprintf (stderr, "could not obtain rows, error code %d\n", rval);
} else {
    for (i = 0; i < nrows; i++) {
        printf ("%s RHS = %f, SENSE = %c\n", names[i], rhs[i], sense[i]);
        printf ("   Coefficients: ");
        for (j = rowbeg[i]; j < rowbeg[i] + rowcnt[i]; j++) {
            printf ("(%d, %f) ", rowind[j], rowval[j]);
        }
        printf ("\n");
    }
}

QSfree (rowcnt); QSfree (rowbeg); QSfree (rowind); QSfree (rowval);
QSfree (rhs); QSfree (sense);
if (names) {
    for (i = 0; i < nrows; i++) {
        QSfree (names[i]);
    }
    QSfree (names);
}
```

QSget_rows_list

Copy a list of rows.

Synopsis

```
    int QSget_rows_list (QSprob p, int num, int *rowlist, int **rowcnt,
        int **rowbeg, int **rowind, double **rowval, double **rhs,
        char **sense, char ***names)
```

Arguments

p – a handle to an initialized problem.

num – the length of the array rowlist.

rowlist – an array of length num; each entry specifies the index of a row (so a value between 0 and nrows - 1, where nrows is the number of rows in the problem).

rowcnt – returns an array of length num, with the ith entry specifying the number of non-zero coefficients in the row specified by rowlist[i]; this field can be NULL.

rowbeg – returns an array of length num, with the ith entry specifying the location of the start of rowlist[i] in the rowind and rowval arrays; this field can be NULL.

rowind – returns an array that contains the column indices of the non-zero coefficients in the rows specified in the rowlist array; the indices for rowlist[i] are stored consecutively starting at entry number rowbeg[i] (there are rowcnt[i] indices for row rowlist[i]); this field can be NULL.

rowval – returns an array that contains the non-zero coefficients in the rows specified by the rowlist array; the coefficients for rowlist[i] are stored consecutively starting at entry number rowbeg[i] (there are rowcnt[i] coefficients for rowlist[i]); this field can be NULL.

rhs – returns an array of length num, giving the right-hand-side values for rows specified in the rowlist array; this field can be NULL.

sense – returns an array of length num where the ith entry specifies the sense of constraint rowlist[i] ('E' for $=$, 'L' for $\leq$; 'G' for $\geq$); this field can be NULL.

names – returns an array of length num where the ith entry is a string that specifies the name of constraint rowlist[i]; this field can be NULL.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

A complete description of the subset of the constraints for the problem can be obtained using this function; the indices for the constraints are passed into the function via the rowlist array (the indices can be obtained from the corresponding row names by using the function QSget_row_index()). Each of the arguments returns an array that is allocated by QSget_rows_list(); if some piece of information is not needed, the corresponding argument for the function should be set to NULL. The calling function should free the memory for each of the arrays when the information is no longer needed; since the memory for the arrays and strings is allocated by the *QSopt* library, the memory must be freed with calls to QSfree() and not by the system free() function—see the documentation for QSfree() on page 73. The form of the row information is the same as that used in the

`QSadd_rows()` function.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */
/* Obtain the row information for row 0 and row 2.                 */

double *rowval = NULL, *rhs = NULL;
int *rowcnt = NULL, *rowbeg = NULL, *rowind = NULL;
char *sense = NULL, **names = NULL;
int rowlist[2] = { 0, 2 };
int i, j, rval;

rval = QSget_rows_list (p, 2, rowlist, &rowcnt, &rowbeg, &rowind, &rowval,
                        &rhs, &sense, &names);
if (rval) {
    fprintf (stderr, "could not obtain rows, error code %d\n", rval);
} else {
    for (i = 0; i < 2; i++) {
        printf ("%s RHS = %f, SENSE = %c\n", names[i], rhs[i], sense[i]);
        printf ("   Coefficients: ");
        for (j = rowbeg[i]; j < rowbeg[i] + rowcnt[i]; j++) {
            printf ("(%d, %f) ", rowind[j], rowval[j]);
        }
        printf ("\n");
    }
}

QSfree (rowcnt); QSfree (rowbeg); QSfree (rowind); QSfree (rowval);
QSfree (rhs); QSfree (sense);
if (names) {
    for (i = 0; i < 2; i++) {
        QSfree (names[i]);
    }
    QSfree (names);
}
```

QSget_column_index

Obtain the index of a named column.

Synopsis

```
int QSget_column_index (QSprob p, const char *name, int *colindex)
```

Arguments

`p` – a handle to an initialized problem.
`name` – a string containing the name of a variable.
`colindex` – returns the index of the variable in the problem (a value between 0 and `ncols - 1`, where `ncols` is the number of columns in the problem); if the variable is not in the problem,

colindex is set to -1.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

This function returns the index of a variable specified by name (all variables in an initialized QSprob have associated names). The function provides an easy way to keep track of a named variable, particularly after calls that delete columns (and change the indices of the remaining variables).

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */
/* Find the index of the variable z.                               */

int rval, colindex;
const char *name = "z";

rval = QSget_column_index (p, name, &colindex);
if (rval) {
    fprintf (stderr, "could not get index, error code %d\n", rval);
} else {
    if (colindex == -1) {
        printf ("%s is not in the problem\n", name);
    } else {
        printf ("%s is variable number %d\n", name, colindex);
    }
}
```

QSget_row_index

Obtain the index of a named row.

Synopsis

```
int QSget_row_index (QSprob p, const char *name, int *rowindex)
```

Arguments

p – a handle to an initialized problem.
name – a string containing the name of a row.
rowindex – returns the index of the row in the problem (a value between 0 and nrows - 1, where nrows is the number of rows in the problem); if the row is not in the problem, rowindex is set to -1.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

This function returns the index of a constraint specified by name (all rows in an initialized QSprob have associated names). The function provides an easy way to keep track of a named row, particularly after calls that delete rows (and change the indices of those that remain).

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */
/* Find the index of the constraint r1.                            */

int rval, rowindex;
const char *name = "r1";

rval = QSget_row_index (p, name, &rowindex);
if (rval) {
    fprintf (stderr, "could not get index, error code %d\n", rval);
} else {
    if (rowindex == -1) {
        printf ("%s is not in the problem\n", name);
    } else {
        printf ("%s is row number %d\n", name, rowindex);
    }
}
```

QSget_colnames

Copy the names of the columns in the problem.

Synopsis

`int QSget_colnames (QSprob p, char **colnames)`

Arguments

`p` – a handle to an initialized problem.

`colnames` – returns an array of strings specifying the names of the columns (variables); this field should point to an array of length at least `ncols`, the number of columns in the problem.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

A `QSprob` data structure always contains names for each column of the LP problem; if the application does not give explicit column names, then the *QSopt* library will generate them when the problem is loaded and as new columns are added. The `QSget_colnames()` function will allocate memory for each of the strings that are returned via the array of pointers in `colnames`; when the application no longer needs the names, the memory associated with each of the strings should be freed; since the memory for the strings was allocated by the *QSopt* library, the memory must be freed with calls to `QSfree()` rather than by the system `free()` function—see the documentation for `QSfree()` on page 73.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int rval, ncols, j;
char **colnames;

ncols = QSget_colcount (p);
colnames = (char **) malloc (ncols * sizeof (char *));
```

```
rval = QSget_colnames (p, colnames);
if (rval) {
    fprintf (stderr, "Could not get column names, error code %d\n", rval);
} else {
    printf ("Variable Names\n");
    for (j = 0; j < ncols; j++) {
        printf ("%s\n", colnames[j]);
    }

    /* Need to free the individual names */

    for (j = 0; j < ncols; j++) {
        QSfree (colnames[j]);  /* Use QSfree for mem allocated by QSopt */
    }
}

free (colnames);  /* Use free for mem allocated by system malloc */
```

QSget_rownames

Copy the names of the rows in the problem.

Synopsis

```
int QSget_rownames (QSprob p, char **rownames)
```

Arguments

p – a handle to an initialized problem.
rownames – returns an array of strings specifying the names of the rows (constraints); this field should point to an array of length at least `nrows`, the number of rows in the problem.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

A `QSprob` data structure always contains names for each row of the LP problem; if the application does not give explicit row names, then the *QSopt* library will generate them when the problem is loaded and as new rows are added. The `QSget_rownames()` function will allocate memory for each of the strings that are returned via the array of pointers in `rownames`; when the application no longer needs the names, the memory associated with each of the strings should be freed; since the memory for the strings was allocated by the *QSopt* library, the memory must be freed with calls to `QSfree()` rather than by the system `free()` function—see the documentation for `QSfree()` on page 73.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int rval, nrows, i;
char **rownames;
```

```
nrows = QSget_rowcount (p);
rownames = (char **) malloc (nrows * sizeof (char *));

rval = QSget_rownames (p, rownames);
if (rval) {
    fprintf (stderr, "Could not get row names, error code %d\n", rval);
} else {
    printf ("Constraint Names\n");
    for (i = 0; i < nrows; i++) {
        printf ("%s\n", rownames[i]);
    }

    /* Need to free the individual names */

    for (i = 0; i < nrows; i++) {
        QSfree (rownames[i]);  /* Use QSfree for mem allocated by QSopt */
    }
}

free (rownames);  /* Use free for mem allocated by system malloc */
```

QSget_intcount

Obtain the number of integer variables.

Synopsis

```
int QSget_intcount (QSprob p, int *count)
```

Arguments

`p` – a handle to an initialized problem.
`count` – returns the number of integer variables in the problem.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

This function can be used to obtain the number of integer variables in the problem, that is, the number of variables that are required to take on integer values.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int rval, count;

rval = QSget_intcount (p, &count);
if (rval) {
    fprintf (stderr, "could not get integer count, error code %d\n", rval);
} else {
    printf ("Problem has %d integer variables\n", count);
}
```

QSget_intflags

Get flags indicating the integer variables.

Synopsis

```
int QSget_intflags (QSprob p, int *intflags)
```

Arguments

`p` – a handle to an initialized problem.

`intflags` – returns an array of length `ncols` (the number of columns in the problem) of 0 - 1 values; the *j*th entry is 1 if the *j*th variable is an integer variable and the *j*th entry is 0 otherwise; this array must be allocated by the calling routine.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

This function can be used to determine the integer variables in the problem, that is, the variables that are required to take on integer values. The argument `intflags` must point to an array of length at least `ncols`, the number of columns in the problem. The first `ncols` entries of `intflags` will be set to 0 or 1, with the 1's indicating the integer variables.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int j, ncols;
int *intflags;

ncols = QSget_colcount (p);
intflags = (int *) malloc (ncols * sizeof (int));

rval = QSget_intflags (p, intflags);
if (rval) {
    fprintf (stderr, "could not get intflags, error code %d\n", rval);
} else {
    printf ("Integer Variables\n");
    for (j = 0; j < ncols; j++) {
        if (intflags[j] == 1) {
            printf ("%d ", j);
        }
    }
    printf ("\n");
}

free (intflags);
```

Chapter 7

Working with an LP Basis

An LP solution found by the simplex algorithm has an associated *basis*. The basis can be used to restart the solution procedure after a problem is modified (usually allowing the algorithm to save a great amount of time over solving the problem from scratch). In the normal course of solving a problem, modifying the data, and resolving, the *QSopt* solvers will automatically use the basis information to aid in the resolves (the user does not need to provide any additional information to the solvers). However, if a problem is saved (for example to a file) and later reloaded, the basis information is lost. In such a case, it may be useful to save the basis and provide the solver with this information to help in the later solution of the problem. The *QSopt* library provides number of routines for saving and loading the basis information, as well as saving and loading information used by the dual steepest-edge variant of the simplex algorithm.

The *QSopt* functions provide two interfaces to the basis routines. The first interface makes use of the `QSbas` data type to store the associated information, in a manner similar to the use of the `QSprob` data type. The second interface makes use of a set of arrays to store the basis information (allowing the user to easily build and manipulate the data).

		Page
QSget_basis	get the current basis	64
QSwrite_basis	write a basis to a file	65
QSread_basis	read a basis from a file	66
QSload_basis	load a basis stored in a basis structure	67
QSread_and_load_basis	read and load a basis	67
QSfree_basis	free the basis structure	68
QSget_basis_array	copy the current basis into arrays	68
QSload_basis_array	load a basis specified by arrays	70
QSget_basis_and_row_norms_array	copy the basis and row norms	70
QSload_basis_and_row_norms_array	load a basis and row norms	71

QSget_basis

Get the current basis.

Synopsis

```
QSbas QSget_basis (QSprob p)
```

Arguments

p – a handle to an initialized problem.

Returns

An initialized `QSbas` object, providing a handle to the current basis for the LP problem. If an error occurred, the `QSbas` object is set to `NULL`. Note that `QSget_basis()` returns an error code if it is called with a specified problem that has not been solved with one of the optimization functions (`QSopt_dual()` or `QSopt_primal()`) since the last time the problem was loaded or modified.

Description

This function can be used to obtain a copy of the current basis; the basis can later be used with a call to `QSload_basis()` when re-solving the LP problem. When a application no longer requires the `QSbas` data structure, the function `QSfree_basis()` should be call to free the associated memory.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

QSbas B;

B = QSget_basis (p);
if (B == (QSbas) NULL) {
    fprintf (stderr, "Could not obtain the basis\n");
}
```

QSwrite_basis

Write a basis to a file.

Synopsis

```
int QSwrite_basis (QSprob p, QSbas B, const char *filename)
```

Arguments

p – a handle to an initialized problem.

B – a handle to a basis for the problem; this argument can be `NULL`, in which case the basis written will be the current optimal basis for the problem.

`filename` – a string specifying the name of the file to store the basis.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred. Note that if `B` is `NULL`, then `QSwrite_basis()` returns an error code if it is called with a specified problem that has not been solved with one of the optimization functions (`QSopt_dual()` or `QSopt_primal()`) since the last time the problem was loaded or modified.

Description

This can be used to store a basis directly in a file, avoiding the format issues that arise in the use of the `QSget_basis_array()` function. The `B` argument can be used to specify the basis, or `B` can be set to `NULL` to specify that the current optimal basis for the LP problem should be written to the file.

The basis file is written the industry standard MPS Basis format. (This is the format that is used by `QSread_basis()`.)

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int rval;

rval = QSwrite_basis (p, (QSbas) NULL, "binky.bas");
if (rval) {
    fprintf (stderr, "Could not write the basis, error code %d\n", rval);
}
```

QSread_basis

Read a basis from a file.

Synopsis

```
QSbas QSread_basis (QSprob p, const char *filename)
```

Arguments

`p` – a handle to an initialized problem.
`filename` – a string specifying the name of the file containing the basis.

Returns

An initialized `QSbas` object, providing a handle to the basis described in the file. If an error occurred, the `QSbas` object is set to `NULL`.

Description

This function returns the basis in the `QSbas` data structure. The function can be used if an application has stored the basis information and may need to load it several times (for example, when the LP problem is solved in several different ways, or after several different types of modifications). If the application only needs to solve the LP problem a single time and the `QSbas` data structure is being passed directly to the `QSprob p` via a call to `QSload_basis()`, then it may be easier to use the function `QSread_and_load_basis()` to read and load the basis in a single stroke.

When a application no longer requires the `QSbas` data structure, the function `QSfree_basis()` should be call to free the associated memory.

The basis file should contain a description of the basis in the industry standard MPS Basis format. (This is the format that is used by `QSwrite_basis()`.)

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

QSbas B;

B = QSread_basis (p, "binky.bas");
if (B == (QSbas) NULL) {
    fprintf (stderr, "Could not read the basis\n");
}
```

QSload_basis

Load a basis stored in a basis structure.

Synopsis

```
int QSload_basis (QSprob p, QSbas B)
```

Arguments

`p` – a handle to an initialized problem.
`B` – a handle to a basis for the problem.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

Use this function to load a previously saved basis into the `QSprob` data structure. The basis `B` should have been obtained either via a call to `QSget_basis()` or via a call to `QSread_basis()`. The `QSload_basis` function will check that the size of the basis agrees with the size of the LP problem `p`, and return an error code if the two do not match.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem.     */
/* B is an initialized QSbas for p (with size matching the size of p). */

int rval;

rval= QSload_basis (p, B);
if (rval) {
    fprintf (stderr, "Could not load the basis, error code %d\n", rval);
}
```

QSread_and_load_basis

Read and load a basis.

Synopsis

```
int QSread_and_load_basis (QSprob p, const char *filename)
```

Arguments

`p` – a handle to an initialized problem.
`filename` – a string specifying the name of the file containing the basis.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

This function reads a basis from a file an stores in directly in the `QSprob` data structure, avoiding both the need to keep a `QSbas` structure as well as the format issues that arise in the use of the `QSload_basis_array()` function.

The basis file should contain a description of the basis in the industry standard MPS Basis format. (This is the format used by `QSwrite_basis()`.)

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int rval;

rval = QSread_and_load_basis (p, "binky.bas");
if (rval) {
    fprintf (stderr, "Could not read the basis, error code %d\n", rval);
}
```

QSfree_basis

Free the basis structure.

Synopsis

```
void QSfree_basis (QSbas B)
```

Arguments

`B` - a handle to a basis.

Returns

No return value.

Description

When work is complete on a given basis `B`, `QSfree_basis(B)` should be called to free the memory associated with the `QSbas` data structure. Note that after a call to `QSfree_basis(B)`, the specified `QSbas B` will no longer contain any useful information.

Example

```
/* B is an initialized QSbas */

QSfree_basis (B);
```

QSget_basis_array

Copy the current basis into arrays.

Synopsis

```
int QSget_basis_array (QSprob p, char *cstat, char *rstat)
```

Arguments

`p` – a handle to an initialized problem.

`cstat` – returns an array specifying the status of each column (variable) in the basis; this field can be `NULL`, if it is not `NULL` it should point to an array of length at least `ncols`, the number of columns in the problem.

`rstat` – returns an array specifying the status of each row (constraint) in the basis; this field can be `NULL`, if it is not `NULL` it should point to an array of length at least `nrows`, the number of rows in the problem.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

The status of the columns (variables) in the basis are specified in the array `cstat`. Each entry in the `cstat` array is one of the following four values

Value	Constant	Meaning
0	`QS_COL_BSTAT_LOWER`	Variable is non-basic, at lower bound
1	`QS_COL_BSTAT_BASIC`	Variable is basic
2	`QS_COL_BSTAT_UPPER`	Variable is non-basic, at upper bound
3	`QS_COL_BSTAT_FREE`	Variable is non-basic and free

Similarly, the status of the rows (constraints) in the basis (that is, the status of the logical variables [either slack variables or artificial variables] corresponding to the rows) are specified in the array `rstat`. Each entry in the `rstat` array is one of the following two values

Value	Constant	Meaning
0	`QS_ROW_BSTAT_LOWER`	Logical variable is non-basic, at lower bound
1	`QS_ROW_BSTAT_BASIC`	Logical variable is basic
2	`QS_ROW_BSTAT_UPPER`	Logical variable is non-basic, at upper bound

Note that a logical variable has an associated upper bound only if it corresponds to a range constraint.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int rval, ncols, nrows;
char *cstat, *rstat;

ncols = QSget_colcount (p);
nrows = QSget_rowcount (p);

cstat = (char *) malloc (ncols * sizeof (char));
rstat = (char *) malloc (nrows * sizeof (char));

rval = QSget_basis_array (p, cstat, rstat);
if (rval) {
    fprintf (stderr, "Could not get the basis, error code %d\n", rval);
} else {
    /* The basis information is stored in the arrays. */
}

free (cstat);
free (rstat);
```

QSload_basis_array

Load a basis specified by arrays.

Synopsis

```
int QSload_basis_array (QSprob p, char *cstat, char *rstat)
```

Arguments

`p` – a handle to an initialized problem.

`cstat` – an array of length `ncols`, the number of columns in the problem; `cstat[j]` should specify the status of the jth column.

`rstat` – an array of length `nrows`, the number of rows in the problem; `rstat[i]` should specify the status of the ith row.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

The basis should be described by the arrays `cstat` and `rstat`, using the encoding given above in the description of `QSget_basis_array()`. If values other than those listed above are specified in `cstat` or `rstat`, then an error code will be returned by `QSload_basis_array()`.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */
/* cstat and rstat are arrays containing the basis information.    */

int rval;

rval = QSload_basis_array (p, cstat, rstat);
if (rval) {
    fprintf (stderr, "Could not load the basis, error code %d\n", rval);
}
```

QSget_basis_and_row_norms_array

Copy the basis and dual steepest-edge norms.

Synopsis

```
int QSget_basis_and_row_norms_array (QSprob p, char *cstat,
    char *rstat, double *rownorms)
```

Arguments

`p` – a handle to an initialized problem.

`cstat` – returns an array specifying the status of each column (variable) in the basis; this field can be `NULL`, if it is not `NULL` it should point to an array of length at least `ncols`, the number of columns in the problem.

`rstat` – returns an array specifying the status of each row (constraint) in the basis; this field can be `NULL`, if it is not `NULL` it should point to an array of length at least `nrows`, the number of rows in the problem.

`rownorms` – returns an array specifying the dual steepest-edge norm of each row (constraint); this field can be `NULL`, if it is not `NULL` it should point to an array of length at least `nrows`.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

The basis information is returned in the arrays `cstat` and `rstat`, using the encoding given above in the description of `QSget_basis_array()`; the row norms are given as floating point numbers in the array `rownorms`. It is important note that the row norms correspond to the particular basis, so the basis and row norms should be stored together if they are to be used in a later call to `QSload_basis_and_row_norms_array()`.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem  */

int rval, ncols, nrows;
char *cstat, *rstat;
double *rownorms;

ncols = QSget_colcount (p);
nrows = QSget_rowcount (p);

cstat = (char *) malloc (ncols * sizeof (char));
rstat = (char *) malloc (nrows * sizeof (char));
rownorms = (double *) malloc (nrows * sizeof (char));

rval = QSget_basis_and_row_norms_array (p, cstat, rstat, rownorms);
if (rval) {
    fprintf (stderr, "Could not get the basis+norms, error code %d\n", rval);
} else {
    /* The basis and norm information is stored in the arrays. */
}

free (cstat);
free (rstat);
free (rownorms);
```

QSload_basis_and_row_norms_array

Load a basis and dual steepest-edge norms.

Synopsis

```
int QSload_basis_and_row_norms_array (QSprob p, char *cstat,
    char *rstat, double *rownorms)
```

Arguments

`p` – a handle to an initialized problem.

`cstat` – an array of length `ncols`, the number of columns in the problem; `cstat[j]` should specify the status of the jth column.

`rstat` – an array of length `nrows`, the number of rows in the problem; `rstat[i]` should specify the status of the ith row.

`rownorms` – an array of length `nrows`, the number of rows in the problem; `rownorms[i]` should specify the dual steepest-edge norm of the ith row.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

The format for the basis arrays `cstat` and `rstat` is given above in the description of the function `QSget_basis_array()`. The values for `rownorms` should normally be obtained from an earlier call to `QSget_basis_and_row_norms_array()`. Note that the row norms correspond to a particular basis; mixing a given set of row norms with an unmatched basis will most likely lead to poor results in later calls to `QSopt_dual()`.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem.  */
/* cstat and rstat are arrays containing the basis information,     */
/* and rownorms contains the dual steepest-edge norms for the rows. */

int rval;

rval = QSload_basis_and_row_norms_array (p, cstat, rstat, rownorms);
if (rval) {
    fprintf (stderr, "Could not load the basis+norms, error code %d\n", rval);
}
```

Chapter 8

Utility Routines

In this section we describe several utility functions provided in the *QSopt* library, including functions that allow users to set and obtain parameters associated with the solver routines.

QSfree

Free standard memory allocated by *QSopt* functions.

Synopsis

```
void QSfree (void *ptr)
```

Arguments

`ptr` – a pointer to a block of memory allocated by a *QSopt* function and returned to the user as either the value of a function or as an argument in a function.

Returns

No return value.

Description

This function should be used to free the blocks of memory allocated by *QSopt* functions when the memory is no longer needed by the user. Only pointers to data of standard type (that is, `char`, `int`, `double`) and pointers to pointers of these types should be passed to QSfree; pointers to structures created by *QSopt* functions should be passed to the structure-specific functions, that is, `QSfree_prob ()` or `QSfree_basis ()`.

Examples of blocks of memory allocated by a *QSopt* function are the locations of the strings returned by `QSget_probname ()` or `QSget_objname ()`, or the locations of the strings for the column names returned by `QSget_colnames ()`. Other examples include arguments in `QSget_columns ()`, `QSget_columns_list ()`, `QSget_rownames ()`, `QSget_rows ()`, and `QSget_rows_list ()`.

It is important to note that this function should not be used to free memory allocated by the system `malloc ()` functions.

(The functionality of `QSfree ()` is needed to deal with the partitioned memory system used in the Microsoft Windows operating system. By altering the design of the *QSopt* library, it would be possible to avoid the allocation of standard types by the *QSopt* functions, but only at the expense of having a less intuitive interface for the functions that return collections of strings, like those returning the column names and row names. Under Unix-like systems, you should be able to simply use the standard `free ()` function rather than `QSfree ()` if you prefer to do so.)

Example

```
/* p is a QSprob, a handle to an existing LP problem.              */

char *objname;

objname = QSget_objname (p);
if (objname == (char *) NULL) {
    fprintf (stderr, "Could not obtain the objective name\n");
} else {
    printf ("Objective Name: %s\n", objname);
    QSfree (objname);  /* Use QSfree for mem allocated by QSopt */
}
```

QSset_param

Set the value of a specified parameter.

Synopsis

```
int QSset_param (QSprob p, int whichparam, int newvalue)
```

Arguments

`p` – a handle to an initialized problem.
`whichparam` - specifies the parameter to be set.
`newvalue` – specifies the new value of the parameter.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

This function can be used to set various parameters that determine the behavior of the *QSopt* solver. The whichparam field specifies the parameter to be set and the newvalue field specifies the value for the parameter.

Example

```
/* p is a QSprob, a handle to an existing LP problem.              */
/* Set the QSdual_opt() solver to work with the Dantzig pricing rule. */

int rval;
```

```
rval = QSset_param (p, QS_PARAM_DUAL_PRICING, QS_PRICE_DDANTZIG);
if (rval) {
    fprintf (stderr, "Could not set the parameter, error code %d\n", rval);
}
```

QSget_param

Obtain the value of a specified parameter.

Synopsis

```
int QSget_param (QSprob p, int whichparam, int *value)
```

Arguments

`p` – a handle to an initialized problem.
`whichparam` - specifies the parameter to be obtained.
`value` – returns the value of the specified parameter.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

This function can be used to access parameters associated with the *QSopt* solvers. The `whichparam` argument specifies the parameter and the `value` argument returns that current setting of the specified parameter.

Example

```
/* p is a QSprob, a handle to an existing LP problem.  */
/* Get the current pricing rule used in QSdual_opt().  */

int rval, param;

rval = QSget_param (p, QS_PARAM_DUAL_PRICING, &param);
if (rval) {
    fprintf (stderr, "Could not get the parameter, error code %d\n", rval);
} else {
    switch (param) {
    case QS_PRICE_DDANTZIG:
        printf ("Dual Dantzig Pricing\n");
        break;
    case QS_PRICE_DSTEEP:
        printf ("Dual Steepest-Edge Pricing\n");
        break;
    case QS_PRICE_DMULTPARTIAL:
        printf ("Dual Multiple-Partial Pricing\n");
        break;
    default:
        printf ("Unknown dual pricing rule: %d\n", param);
        break;
    }
}
```

QSset_param_double

Set the value of a numerical parameter.

Synopsis

```
int QSset_param_double (QSprob p, int whichparam, double  newvalue)
```

Arguments

`p` – a handle to an initialized problem.
`whichparam` - specifies the parameter to be set.
`newvalue` – specifies the new value of the parameter.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

This function can be used to set various numerical parameters that determine the behavior of the *QSopt* solver with respect to a given problem. The `whichparam` field specifies the parameter to be set and the `newvalue` field specifies the value for the parameter.

Example

```
/* p is a QSprob, a handle to an existing LP problem.              */
/* Set the time limit of the QSopt solvers to 10.5 seconds.        */

int rval;

rval = QSset_param_double (p, QS_PARAM_SIMPLEX_MAX_TIME, 10.5);
if (rval) {
    fprintf (stderr, "Could not set the parameter, error code %d\n", rval);
}
```

QSget_param_double

Obtain the value of a numerical parameter.

Synopsis

```
int QSget_param_double (QSprob p, int whichparam, double *value)
```

Arguments

`p` – a handle to an initialized problem.
`whichparam` - specifies the parameter to be obtained.
`value` – returns the value of the specified parameter.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

This function can be used to access parameters associated with the *QSopt* solvers. The `whichparam` argument specifies the parameter and the `value` argument returns the current setting of the specified parameter.

Example

```
/* p is a QSprob, a handle to an existing LP problem.             */
/* Get the maximum time (in seconds) allowed in the QSopt solvers.  */

int rval;
double param;

rval = QSget_param_double (p, QS_PARAM_SIMPLEX_MAX_TIME, &param);
if (rval) {
    fprintf (stderr, "Could not get the parameter, error code %d\n", rval);
} else {
    printf (Simplex Maximum Time: %f seconds\n");
}
```

QSwrite_prob

Write the problem to a named file.

Synopsis

```
int QSwrite_prob (QSprob p, const char *filename, const char *filetype)
```

Arguments

`p` – a handle to an initialized problem.
`filename` – a string containing the name of the file to store the problem.
`filetype` – use `"LP"` to write a file in LP format or `"MPS"` to write a file in MPS format.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

Writes problem `p` to a file in LP format or MPS format. The name of the file is specified by `filename`. If `filetype` is specified as `"LP"` (the letters in the string can be any mix of uppercase and lowercase), then the problem is written in LP format (see Chapter 10 for a description of the problem file formats); if `filetype` is specified as `"MPS"` (again, the letters in the string can be any mix of uppercase and lowercase), then the problem is written in MPS format. If `filetype` is specified as something other than `"LP"` or `"MPS"`, then an error code will be returned.

Example

```
/* p is a QSprob, a handle to an existing LP problem */

int rval;

rval = QSwrite_prob (p, "binky.mps", "MPS");
if (rval) {
    fprintf (stderr, "Could not write the MPS file, error code %d\n", rval);
}
```

QSwrite_prob_file

Write the problem to an open file.

Synopsis

```
int QSwrite_prob_file (QSprob p, FILE *file, const char *filetype)
```

Arguments

`p` – a handle to an initialized problem.
`file` – a handle to a file that is open for writing.
`filetype` – use `"LP"` to write a file in LP format or `"MPS"` to write a file in MPS format.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

Writes problem `p` to an open file in LP format or MPS format. The argument `file` should be a handle to a file that is open for writing. The use of the argument `filetype` to specify the format of the output file is the same as in the function `QSwrite_prob()` described above.

This function gives an easy way to write the problem to the screen by specifying the `file` as `stdout` or `stderr`. The function can also be used to store the LP along with other information in a given file. To simply open a file and write the LP, it is usually easier to use the function `QSwrite_prob()`.

Example

```
/* p is a QSprob, a handle to an existing LP problem. */
/* Write the problem to the screen in LP format.      */

int rval;

rval = QSwrite_prob_file (p, stdout, "LP");
if (rval) {
    fprintf (stderr, "Could not write LP to stdout, error code %d\n", rval);
}
```

Chapter 9

Advanced Computational Routines

In this section we describe a number of *QSopt* functions that may be of interest to researchers and expert developers. These functions can be used in the design of branch-and-cut algorithms for integer programming and discrete optimization (they support routines in the *Concorde* traveling salesman problem code of Applegate, Bixby, Chvátal, and Cook).

QSopt_strongbranch

Compute values used with strong-branching.

Synopsis

```
int QSopt_strongbranch (QSprob p, int ncand, int *candidatelist,
    double *xlist, double *down_vals, double *up_vals, int iterations,
    double objbound)
```

Arguments

`p` – a handle to an initialized problem.

`ncand` – the number of variables to be tested.

`candidatelist` – a list of the indices of the variables to be tested; the array of values must have length at least `ncand`.

`xlist` – a list of values to use in determining branches, the down-branch to be tested is $x[candidatelist[i]] \leq \lfloor xlist[i] \rfloor$ and the up-branch to be tested is $x[candidatelist[i]] \geq \lceil xlist[i] \rceil$. The argument `xlist` can be `NULL`; if it is not `NULL` then the array of values should have length at least `ncand`.

`down_vals` – returns the objective function values obtained by the down branches; this field should point to an array of length at least `ncand`.

`up_vals` – returns the objective function values obtained by the up branches; this field should point to an array of length at least `ncand`.

`iterations` – the number of iterations (pivots) of the dual steepest-edge simplex algorithm to use in the tests.

`objbound` – a bound on the objective function (can be `QS_MAXDOUBLE` [for a minimization problem] or `-QS_MAXDOUBLE` [for a maximization problem] if no bound is known).

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

This function is used to implement the *strong-branching* rule used by Applegate, Bixby, Chvátal, and Cook in their TSP research. The idea is to use strong LP information to determine a choice of branching variable that is likely to increase the LP bound for each of the two children created by the branching operation.

For each of the column indices specified in the array `candidatelist` (this array is of length `ncand`), the function will gather information about the bounds that can be obtained by branching on the corresponding variable.

Let j be the ith index in `candidatelist`. In the branching test, variable j first has its upper bound set to $\lfloor xlist[i] \rfloor$ and then variable j has its lower bound set to $\lceil xlist[i] \rceil$, and in both cases the dual steepest-edge simplex algorithm is called with an iteration limit of `iterations` pivots. The resulting objective values are returned in the arrays `down_vals` (for the values where the variables have their upper bounds modified) and `up_vals` (for the values where the variables have their lower bounds modified). If the argument `xlist` is not specified, then `QSopt_strongbranch ()` uses the x-vector in the current LP solution to determine the points where the branches occur. (Note that for 0-1 valued variables, the branches correspond to setting the variables equal to 0 for the down-branch and equal to 1 for the up-branch.)

The parameter `objbound` should be set by the calling routine to be an upper bound on the objective function value for minimization problems, and to a lower bound on the objective function value for maximization problems. (This can be used to terminate the optimization routines early, if the bound is obtained in less than `iterations` pivots. The value of `objbound` can be set to the value of the best available MIP solution.) The values returned in `down_vals` and `up_vals` will be no larger than `objbound` for minimization problems, and no smaller than `objbound` for maximization problems.

Example

```
/* Assume p is a QSprob, a handle to an existing LP problem, that has */
/* been solved with one of the optimization routines.  All variables  */
/* in p will be considered as integer variables, and a candidate      */
/* for a branching variable will be determined using strong-          */
/* branching.  The index of the branching variable is the return      */
/* value of the code fragment.                                        */
/*                                                                    */
/* Assume x is an array containing the current solution to the LP;    */
/* it could be obtained using QSget_x_array ().                       */
/*                                                                    */
/* Assume ObjBound is the objective value of the best integer         */
/* solution known for the problem.                                    */

int rval, i, ncols, ncand, branch = -1;
int *candidatelist = (int *) NULL;
double *xlist, *down_vals, *up_vals;
double bestval;
```

```
ncols = QSget_colcount (p);

candidatelist = (int *) malloc (ncols * (sizeof(int));
xlist = (double *) malloc (ncols * (sizeof(double));

ncand = 0;
for (i = 0; i < ncols; i++) {
    t = x[i] - floor (x[i]);      /* t is the fractional part of x[i]  */
    if (t >= 0.1 && t <= 0.9) {   /* x[i] is at least 0.1 from integer */
        candidatelist[ncand] = i;
        xlist[ncand++] = x[i];
    }
}

if (ncand == 0) {
    free (candidatelist);
    free (xlist);
    return -1;
}

down_vals = (double *) malloc (ncand * sizeof(double));
up_vals = (double *) malloc (ncand * sizeof(double));

rval = QSopt_strongbranch (p, ncand, candidatelist, xlist, down_vals,
                           up_vals, 50, ObjBound);
if (rval) {
    fprintf (stderr, "QSopt_strongbranch failed, error code %d\n", rval);
} else {
    /* Select as a branching variable the candidate that maximizes the */
    /* the function 10*min + max, where min is the smaller of          */
    /* down_vals and up_vals, and max is the larger.                    */

    bestval = -QS_MAXDOUBLE;
    for (i = 0; i < ncand; i++) {
        if (down_vals[i] < up_vals[i]) {
            val = 10 * down_vals[i] + up_vals[i];
        } else {
            val = 10 * up_vals[i] + down_vals[i];
        }
        if (val > bestval) {
            bestval = val;
            branch = candidatelist[i];
        }
    }
}

free (candidatelist);
free (xlist);
free (down_vals);
free (up_vals);
```

```
  return branch;
```

QSopt_pivotin_col

Pivot columns into the basis.

Synopsis

```
int QSopt_pivotin_col (QSprob p, int ccnt, int *clist)
```

Arguments

`p` – a handle to an initialized problem.
`ccnt` – the number of columns to be pivoted into the basis.
`clist` – an array of length `ccnt`; each entry specifies the index of a column (so a value between 0 and `ncols - 1`, where `ncols` is the number of columns in the problem).

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

This function attempts to pivot into the current basis the columns specified in `clist`. The routine sequentially pivots in the columns. The leaving column in each pivot is a column (not one in the list) that results in the least movement from the current solution, chosen from among the columns that yield suitable pivot values.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem.  */
/* Pivot column 2 into the basis.                                   */

int rval;
int collist[1] = { 2 };

rval = QSopt_pivotin_col (p, 1, collist);
if (rval) {
    fprintf stderr, "QSopt_pivotin_col failed with return code %d\n", rval);
}
```

QSopt_pivotin_row

Pivot logical variables into the basis.

Synopsis

```
int QSopt_pivotin_row (QSprob p, int rcnt, int *rlist)
```

Arguments

`p` – a handle to an initialized problem.
`rcnt` – the number of rows to be pivoted into the basis.

rlist – an array of length rcnt; each entry specifies the index of a row (so a value between 0 and nrows - 1, where nrows is the number of rows in the problem).

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

This function attempts to pivot into the current basis the columns for the logical variables associated with the rows specified in rlist. The routine sequentially pivots in the columns. The leaving column in each pivot is a column (not one in the list) that results in the least movement from the current solution, chosen from among the columns that yield suitable pivot values.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem.  */
/* Delete rows 1 and 3 from the LP, but to keep a basis for the     */
/* modified problem, first pivot into the basis the logical         */
/* variables associated with the two rows.                          */

int rval;
int dellist[2] = { 1, 3 };

rval = QSopt_pivotin_row (p, 2, dellist);
if (rval) {
    fprintf stderr, "QSopt_pivotin_row failed with return code %d\n", rval);
} else {
    rval = QSdelete_rows (p, 2, dellist);
    if (rval) {
        fprintf stderr, "QSdelete_rows failed with return code %d\n", rval);
    }
}
```

QScompute_row_norms

Recompute the dual steepest-edge norms.

Synopsis

```
int QScompute_row_norms (QSprob p)
```

Arguments

p – a handle to an initialized problem.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred.

Description

Recomputes the dual steepest-edge norms for problem p. Note that an error will occur if the current dual pricing rule is not QS_PRICE_DSTEEP.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem.  */
/* Assume we are working with the QS_PRICE_DSTEEP pricing rule.     */

int rval;

rval = QScompute_row_norms (p);
if (rval) {
    fprintf (stderr, "Could not compute row norms, error code %d\n", rval);
}
```

QStest_row_norms

Check if dual steepest-edge norms are available.

Synopsis

```
int QStest_row_norms  (QSprob p)
```

Arguments

p – a handle to an initialized problem.

Returns

One if the dual steepest-edge norms exist, and zero otherwise.

Description

This function returns one if dual steepest-edge norms exist, and zero otherwise. If the row norms do not exist (this may happen after some types of problem modifications), then a call to the function `QScompute_row_norms()` or a call to one of the optimization routines will restore them. (Note that row norms will only exist if the LP problem is being solved via the dual steepest-edge simplex algorithm, that is with the dual pricing rule `QS_PRICE_DSTEEP`.)

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem.  */
/* Assume we are working with QS_PRICE_DSTEEP pricing rule.         */

int rval, yesno;

yesno = QStest_row_norms (p);
if (yesno == 1) {
    printf ("Row norms exist\n");
} else {
    printf ("Need to compute row norms\n");
    rval = QScompute_row_norms (p);
    if (rval) {
        fprintf (stderr, "Could not compute norms, error code %d\n", rval);
    }
}
```

QSget_infeas_array

Obtain a certificate of LP infeasibility.

Synopsis

```
int QSget_infeas_array (QSprob p, double *pi)
```

Arguments

`p` – a handle to an initialized problem.

`pi` – returns multipliers for a Farkas-like certificate of LP infeasibility; this should point to an array of length at least `nrows`, the number of rows in the problem.

Returns

A zero value if the function terminated correctly, and a non-zero value if an error occurred. Note that `QSget_infeas_array()` will return an error unless either `QSopt_primal()` reports infeasibility or `QSopt_dual()` reports unboundedness.

Description

This function returns dual multipliers that can be used in a Farkas-like certificate of LP infeasibility. It can be used within an integer programming code to verify that an LP relaxation is really infeasible before pruning a branch-and-bound tree.

Example

```
/* p is an initialized QSprob, a handle to an existing LP problem.  */

int rval, nrows;
double *pi;

nrows = QSget_rowcount (p);
pi = (double *) malloc (nrows * sizeof (double));

rval = QSget_infeas_array (p, pi);
if (rval) {
    fprintf (stderr, "Could not get certificate, error code %d\n", rval);
}

free (pi);
```

Chapter 10

Problem file formats

The *QSopt* library supports two ASCII file formats for specifying LP and MIP problems. The formats are based on two industry standards, the *MPS file format* and the *LP file format.* Both file formats provide convenient mechanisms to define an LP or MIP problem, that is

- its objective and whether it should be minimized or maximized,
- the constraints with their expressions, senses, and right-hand-sides,
- the problem's variables, and their bounds,
- which variables are integer variables, and
- the problem's name.

The *QSopt* reader expects that all problems that it reads from a file have at least one constraint and at least one variable. The objective function that it reads may be empty, that is, it may contain no terms; in such a case, the optimization routines attempt only to find a feasible solution when solving the problem.

10.1 Bounds on Variables

The two file formats do not require that all variable upper and lower bounds be given explicitly. The *QSopt* reader assumes by default that variables are greater than or equal to zero and have no upper bound. Integer variables are assumed to be binary, that is, they have a lower bound of 0 and an upper bound of 1. The bounds sections in LP and MPS files are used to define different variable bounds.

When defining variable bounds keep the following rules in mind.

1. A variable's upper bound must be greater than or equal to its lower bound.
2. Negative infinity may not be used as an upper bound and positive infinity may not be used as lower bound.
3. If a variable's upper bound is defined either as a nonnegative number or as positive infinity, then its lower bound defaults to zero. Thus defining a zero upper bound without defining a lower bound fixes a variable to zero.
4. If a variable's upper bound is defined as negative number its lower bound defaults to negative infinity.

5. If a variable's lower bound is defined either as a negative number or as negative infinity, then its upper bound defaults to zero.

6. If a variable's lower bound is defined either as zero or as a positive number, then its upper bound is assumed to be positive infinity.

10.2 LP Format

An *LP-format* file contains a sequence of sections that state the problem's name, its objective function, and its constraints, and optional sections stating upper and lower bounds on the problem's variables and listing the variables that are required to take on integer values. All LP-format files end with the keyword "`END`".

The LP-format example below defines a linear-programming problem called `smallExample`. The example's objective section defines the objective `obj` to be `x - 2.3y + 0.5z`, where `x` and `y`, and `z` are variables. The problem's objective is to maximize the given linear expression. The constraint section contains two constraints. The first constraint, called `c1`, states that `x - y + s` must be less than or equal to `10.75`. The the second constraint states that `-z + 2x - s` must be greater than or equal to `-100`. The *QSopt* reader will create a name for the second unnamed constraint.

```
Problem
    smallExample
Maximize
    obj: x - 2.3y + 0.5z
Subject
    c1: x - y + s <= 10.75
        -z + 2x - s >= -100
End
```

The objective function in an LP-format description may be empty, that is, it may contain no terms; in such a case, the optimization routines attempt only to find a feasible solution when solving the problem. Constraints may be empty as well; empty constraints are simply ignored. Constraints must always contain a right-hand-side, that is a comparison operator and a number. The *QSopt* reader parses the unnamed constraint

```
<= -1000
```

correctly but ignores it because it contains no terms.

The bounds section is used to define variable bounds that differ from the defaults assumed by the *QSopt* reader (see page 86). The lines

```
Bounds
    x <= 10.5
    y <= -1
    -10 <= z <= 100
    s = 1.0
```

state that `x` ranges from 0 to 10.5, `y` ranges from negative infinity ($-\infty$) to -1, `z` ranges from -10 to 100, and `s` is fixed at 1.

To define a variable `x`'s range to be $[-\infty, +\infty]$ its lower and upper bound must either be explicitly given as

```
-inf<=x<=inf
```

or alternatively as

```
x free
```

where `free, -inf,` and `inf` are keywords. Note that there need not be white space between `-inf`, `inf`, and `x` and the `<=` operators, because identifiers and keywords never contain '`<`' or '`=`' characters. On the other hand, if there were no space between `x` and `free`, `xfree` would be interpreted to be an unknown column name, since the variable `xfree` is used nowhere else in the LP problem statement.

To fix a variable to a specific value, its lower and upper bound must be set to the same value. This can be achieved in two ways:

```
y = 10.1
10.0 <= x <= 10.0
```

By default variables may assume any floating point value within their bounds. If a variable is restricted to integer values (a MIP problem), then it must be listed in the integer section. The lines

```
Integer
    x y
```

state that the variables `x` and `y` are required to take on integer values only. Integer variables are binary in the absence of explicit lower or upper bound definitions, that is, they represent the values 0 or 1.

The syntax rules for LP files are given in Bachus-Naur form in Figure 10.1.

Names in <> are non-terminals. IDENT, NUMBER, STRING, keywords, and literals are the tokens used in LP files.

IDENT is any nonempty string that contains solely lower or upper case letters (`a-z` and `A-Z`) the ten digits (`0-9`), and

```
! " # $ % & ( ) / , ; ? @ _ ‘ ’ { } | ~ .
```

An IDENT must never start with a digit or a period (.).

NUMBER is a floating point number in scientific notation.

STRING is a nonempty sequence of characters that contains no white space characters.

keyword is any of the strings given in double quotes in Figure 10.1. All keywords must appear at the beginning of lines. Characters in keywords may be any mix of upper or lower case. For example "`MIN`" can be written as `min`, `Min`, `MIN`, `mIn`, ...

literal is any of the strings given in single quotes used in Figure 10.1. A literal may appear anywhere on a line. Characters in literals may be any mix of upper or lower case. For example, '`inf`' can we written as `inf`, `INF`, `Inf`, `iNF`, ...

Comments are introduced by the character '\'. The LP file reader ignores all text after '\' until the end of the line.

Tokens may be separated by white space, that is by blanks, tabs (\t), newlines (\n), formfeeds (\f), and control-returns (\r). The LP file reader always tries to match the longest possible string to the next token. Therefore, two tokens must be separated by white space from each other in an LP-format description if their combination becomes a legal token itself. For example, writing

```
Bounds
    10<=xy<=10  \ x and y glued together
```

<A>	→	<B>	means that <A> can be derived to be <B>
<A>	→	<B>	
	→	<C>	means that <A> can be derived to be <B> or <C>
<A>	|	<B>	means either <A> or <B>
(<X>)			means <X>
[<X>]			means 0 or 1 <X>
(<X>)*			means sequence of 0 or more instances of <X>
(<X>)+			means sequence of 1 or more instances of <X>

Figure 10.1. *Convention for BNF notation*

is interpreted as `10 <= xy <= 10` and not as `10 <= x, y <= 10`.

It is legal to use as constraint or variable names the same strings that are used for keywords such as `ST`, `integer`, or `Bound`. If these strings appear at the beginning of a line they are always interpreted as keywords. When using variables and constraint names that look like keywords, LP-format descriptions quickly become very confusing and it is strongly recommended not to do so. Consider the following example, where the objective is to maximize the variable `ST` under the constraint that `ST` is less than or equal to 10.

```
MAX
    ST
ST
    ST <= 10
END
```

The LP-format reader is able to interpret the LP problem as intended, since the first `ST` is not placed at the beginning of its line and is therefore recognized as an identifier. The second `ST` starts a line and is recognized as the keyword that starts the constraint section and the third `ST` becomes the term of the problem's constraint. If the first or third `ST` is moved to the start of its line, then the LP-format description becomes incorrect.

Formally LP-format descriptions can be defined as a sequence of tokens that can be derived from the nonterminal <LP-file> by the syntax rules given in Figure 10.2. In formulating the rules we use the conventions defined in Figure 10.1.

```
<LP-file>              →  [<Problem-Name>]
                          <Min-or-Max> <objective>
                          <constraint-section>
                          [("BOUNDS" | "BOUND") <bounds-section>]
                          [("INTEGER" | "INT") <integer-section>]
                          "END"
<Problem-Name>         →  ("PROBLEM" | "PROB") STRING
<Min-or-Max>           →  "MIN" | "MINIMUM" | "MINIMIZE"
                       →  "MAX" | "MAXIMUM" | "MAXIMIZE"
<objective>            →  <constraint-expr>
<constraint-section>   →  ("ST" | "SUBJECT" | "SUBJECT TO")
                          (<constraint-expr> <sense> <rhs>)+
<constraint-expr>      →  [IDENT ':']
                          [NUMBER IDENT (('+' | '-') NUMBER IDENT)*]
                          % The optional identifier before the ':'
                          % designates a row/constraint name.
                          % All other identifiers are column/variable names.
<sense>                →  <smaller-equal> | <greater-equal> | '='
<smaller-equal>        →  '<' | '<=' | '=<'
<greater-equal>        →  '>' | '>=' | '=>'
<bounds-section>       →  <bounds-item>*
<bounds-item>          →  <num or Inf> <smaller-equal> IDENT
                          [<smaller-equal> <num-or-Inf>]
                       →  <num or Inf> <greater-equal> IDENT
                          [<greater-equal> <num-or-Inf>]
                       →  IDENT <smaller-equal> <num-or-Inf>
                       →  IDENT <greater-equal> <num-or-Inf>
                       →  IDENT '=' NUMBER
                       →  IDENT 'FREE'
<num-or-Inf>           →  NUMBER | '-INF' | '+INF' | 'INF' |
                          '-INFINITY' | '+INFINITY' | 'INFINITY'
<integer-section>      →  IDENT*
```

Figure 10.2. *BNF grammar for LP files*

10.2.1 LP-format Examples

To illustrate the LP-format rules, we give below several additional small examples. The first example contains nine variables, each bounded between 0 and 1, and six constraints. The objective function is the sum of the variables. The constraint names `node_0`, `node_1`, ... , `node_5` are not necessary to describe the problem, but they help to document the formulation and they are useful in viewing dual solutions (if no constraint names are specified, the *QSopt* reader will assign default names).

```
Problem      Example1
Minimize     x0_1 + x0_2 + x0_4 + x1_2 + x1_5 + x2_3 + x3_4 + x3_5 + x4_5
Subject To   node_0: x0_1 + x0_2 + x0_4 = 2
             node_1: x0_1 + x1_2 + x1_5 = 2
             node_2: x0_2 + x1_2 + x2_3 = 2
             node_3: x2_3 + x3_4 + x3_5 = 2
             node_4: x0_4 + x3_4 + x4_5 = 2
             node_5: x1_5 + x3_5 + x4_5 = 2
Bounds       x0_1 <= 1   x0_2 <= 1   x0_4 <= 1
             x1_2 <= 1   x1_5 <= 1   x2_3 <= 1
             x3_4 <= 1   x3_5 <= 1   x4_5 <= 1
End
```

The second example contains four variables, each required to take on integer values, and three constraints. The first variable, `x1`, is bounded between 0 and 2, while the remaining three variables default to binary values, that is, they are bounded between 0 and 1.

```
Problem
 Example2
Maximize
 obj: 4x1 + x2 + 5x3 + 3x4
Subject To
 r1:   x1 -  x2 -  x3 + 3x4 <= 1
 r2:  5x1 +  x2 + 3x3 + 8x4 <= 55
 r3:  -x1 + 2x2 + 3x3 - 5x4 <= 3
Bounds
 0 <= x1 <= 2
Integer
 x1 x2 x3
 x4
End
```

10.3 MPS Format

An MPS file contains a sequence of sections that define an LP or MIP problem. The `ROWS`, `COLUMNS`, `RHS`, and `RANGES` sections define the objective and constraint expressions, the sense of the constraints, and the right-hand-sides values. In the optional `BOUNDS` section, a variable's range may be defined to differ from the default $[0, \infty]$ interval. The `OBJSENSE` section states whether the objective value should be minimized or maximized. The `NAME` section assigns a name to the problem. The `OBJNAME` section picks one of the problem's rows as the objective. Each section starts with the corresponding keyword at the beginning of a line, and, with the exception of the `NAME` section, the keyword is the only item on the line. All MPS-format files end with the keyword "`ENDATA`" at the beginning of a line.

The following example defines an LP problem in MPS format. The line numbers are included for reference, they are not part of the MPS file input.

```
 1  NAME    smallExample
 2  OBJSENSE
 3    MAX
 4  OBJNAME
 5    obj
 6  ROWS
 7    N  obj
 8    L  r1
 9    G  r2
10  COLUMNS
11    x    obj    1   r1  1  r2  2
12    y    obj -2.3   r1 -1
13    z    obj  0.5
14    z    r2    -1
15    s    r2    -1
16    s    r1     1
17  RHS
18    RIGHT    r1 10.75
19    RIGHT    r2  -100
20  ENDATA
```

This example describes the same problem instance as the following LP-format file.

```
Problem
 smallExample
Maximize
 obj:   x - 2.3 y + 0.5 z
Subject To
 r1:   x -  y +  s <= 10.75
 r2:  2.0 x -  z -  s >= -100.0
End
```

Whereas the LP-format file specifies the objective function and constraints row by row, the MPS-format file describes these objects in a column by column fashion.

The following table summarizes the lines in the MPS file that define the different components of `smallExample`'s objective and constraints.

line	11	12	13	14	15	16	8	18	9	19
obj	x	-2.3y	+0.5z							
r1	x	-y				s	$\leq$	10.75		
r2	2x			-z	-s				$\geq$	-100

The remaining lines in the MPS file are interpreted as follows. The first line defines the problem's name to be "`smallExample`". Line 3 states that the objective value should be maximized. Line 5 picks the row `obj` as the objective and line 7 defines `obj` to be an "`N`" row, that is, `obj` has no right-hand-side.

10.3.1 Data Lines

Each section in an MPS file must contain at least one data line, with the exception of the `NAME` section. The data lines are interpreted according to the section to which they belong.

Data lines must start with at least one white space character, that is, with a blank, tab (\t), formfeed (\f), or control-return (\r). Note that, in contrast to LP format, the return character (\n) is not considered to be white space. Data lines contain one or more fields separated by white space. Fields may contain any characters other than white space. Fig. 10.3 lists and interprets the fields of data lines in relation to their sections.

section	field 1	field 2	field 3	field 4,5,...
`OBJNAME`	row name			
`REFROW`	row name			
`OBJSENSE`	"`MAXIMIZE`" or "`MAX`"			
	"`MINIMIZE`" or "MIN"			
`ROWS`	"`N`", "`L`", "`G`", or "`E"`"	row name		
`COLUMNS`	variable name	row name	number	more row name, number pairs may follow
	name	"`'MARKER'`"	"`'INTORG'`"	
	name	"`'MARKER'`"	"`'INTEND'`"	
	name	"`'MARKER'`"	"`'SOSORG'`"	
	name	"`'MARKER'`"	"`'SOSEND'`"	
`RHS`	label	row name	number	more row name, number pairs may follow
`BOUNDS`	"`LO`" or `UP`"	bounds label	variable	number
	"`FX`"	bounds label	variable	number
	"`LI`" or "`UI`"	bounds label	variable	number
	"`BV`"	bounds label	variable	number
	"`MI`" or "`PL`"	bounds label	variable	
	"`FR`"	bounds label	variable	
`RANGES`	label	row name	number	more row name, number pairs may follow

Figure 10.3. *Field interpretation of data lines within sections*

Lines that start with the "`*`" character are comment lines. Also, if field 3 (or field 5, field 7, etc.) starts with the "`$`" character, then it is skipped along with the rest of the line. Empty lines are ignored.

10.3.2 Sections and Section Order

An MPS-format file must contain a `ROWS` section and a `COLUMNS` section. It may also contain optional `NAME`, `OBJSENSE`, `OBJNAME`, `RHS`, `RANGES`, `BOUNDS`, and `REFROW` sections. All MPS files must terminate with an `ENDATA` line.

The following restrictions apply to the ordering of sections.

1. The `ROWS` section must appear before the `COLUMNS` and `RHS` section.
2. The optional `RANGES` section must appear after the `ROWS` section.
3. The optional `BOUNDS` section must appear after the `COLUMNS` section.
4. The `REFROW` section must appear before the `ROWS` section.

10.3.3 Defining the Problem's Name

A problem's name may be defined on the `NAME` line; the first field after `NAME` is used as the name. If no `NAME` section is given, the problem's name defaults to "`unnamed`".

10.3.4 Objective Goal

The *QSopt* reader assumes by default that a program's goal is to minimize the objective function. This goal may be stated explicitly or overwritten in the `OBJSENSE` section, which contains exactly one data line. The data line's first and only field must contain one of the strings "`MAXIMIZE`", "`MAX`", "`MINIMIZE`", or "`MIN`".

10.3.5 Defining Constraints and the Objective Function

The `OBJNAME`, `ROWS`, `COLUMNS`, `RHS`, and `RANGES` sections define a problem's constraints and objective.

The `ROWS` Section

Row names and senses are defined in the `ROWS` section. Each row is declared on its own data line. Field 1 on a row data line contains one of the strings "`N`", "`G`", "`L`", or "`E`" to indicate the row's sense, according to the following table.

field 1	interpretation
`G`	$\geq$
`L`	$\leq$
`E`	$=$
`N`	no right-hand-side.

An MPS-format file may define multiple "`N`" rows; the reader ignores each of them other than the objective row.

The `OBJNAME` Section

By default the first "`N`" row defined in the `ROWS` section becomes a problem's objective; a different objective may be specified in the `OBJNAME` section, which contains exactly one data line that names the objective in field 1. The *QSopt* reader changes the objective constraint chosen to be an "`N`" constraint, if necessary.

The `RHS` and `RANGES` Sections

By default "`L`", "`G`", and "`E`" constraints have a right-hand-side value of 0.0. If values other than 0.0 are needed, they are specified in the `RHS` and `RANGES` sections. The reader ignores all `RHS` and `RANGES` definitions for "`N`" constraints.

Field 1 in a `RHS` or `RANGES` data line is a (possibly empty) label. The reader ignores `RHS` and `RANGES` data lines with a label that differs from the label used in the section's first data line. (This convention makes it possible to study several different models with minimal change to the file; it arose in the days when data was described via card decks.)

`RHS` and `RANGES` data lines contain a row name on field 2, and a floating point value in field 3. Additional row name, number pairs may follow on fields 4 and 5, fields 6 and 7, etc. Each row may be mentioned at most once in a `RHS` data line, and at most once in a `RANGES` data line. In the absence of a `RANGES` definition for a row, the row's right-hand-side either defaults to 0.0 or assumes the value given in its `RHS` data line.

A constraint's `RANGES` definition states that the constraint's expression must evaluate to a number that falls within an interval that depends on the constraint's right-hand-side and the range value, as well as the constraint's sense. If `rhs` is a constraint's right-hand-side value and `range` is the constraint's range value, then the range interval is defined according to the following table:

sense	interval	
G	[rhs, rhs + |range|]	
L	[rhs - |range|, rhs]	
E	[rhs,rhs + range]	if range ≥ 0
	[rhs + range,rhs]	if range < 0

where `|range|` is `range`'s absolute value.

The MPS file excerpt

```
ROWS
  N  obj
  L  lower
  G  greater
  E  equal
RHS
  rhs lower 10
```

states that `equal` is equal to 0.0, `greater` is greater than or equal to 0.0, and `lower` is less than or equal to 10.0. Adding the lines

```
RANGES
  lower    8
  greater -5
  equal   -0.5
```

changes the constraints as to

`lower`'s constraint expression falls inside the interval [2.0, 10.0],

`greater`'s constraint expression falls inside the interval [0.0, 5.0],

`equal`'s constraint expression falls inside the interval [-0.5,0.0].

The `COLUMNS` Section

The objective function and the constraints are treated in the same way in the `COLUMNS` section, which defines their non-zero terms. In a `COLUMNS` data line, field 1 contains a variable name, field 2 contains a row name, and field 3 contains a floating point number; additional row name, number pairs may follow on fields 4 and 5, fields 6 and 7, etc. The *QSopt* reader adds a term $number * variable$ to the specified row for each row name, number pair.

The comment lines in the following MPS file excerpt interpret the `COLUMNS` data lines. Note that the second and third `COLUMNS` data lines both define the term `-5v2` for `r1`'s constraint expression. The *QSopt* reader adds both terms so that `r1`'s expression becomes `-5v2 -5v2` or `-10v2`.

```
ROWS
   N obj
   L r1
   L r2
COLUMNS
*  add term: 1.5v1 to objective
   v1  obj 1.5
*  add term  2.25v2  to objective and
*            -5v2    to r1        and
*            -2.75v2 to r2
   v2  obj 2.25  r1 -5  r2 -2.75
*  add another term  -5v2 to r1
   v2  r1 -5
```

10.3.6 Defining variable bounds

`BOUNDS` data lines are used to define variable bounds that differ from the defaults assumed by the *QSopt* reader (see page 86).

Field 1 of a `BOUNDS` data line contains a bounds indicator, field 2 contains a label, and field 3 a variable name. Depending on the bounds indicator, field 4 is either empty or it contains a floating point number. Similar to `RHS` and `RANGES` data lines, a `BOUNDS` line is ignored if its label differs from the label used in the first `BOUNDS` line.

The following table explains the effects of `BOUNDS` data lines on a variable's bounds. `x` refers to the variable name and `value` refers to the number given on the data line.

indicator	uses field 4	interpretation
`LO`, `LI`	yes	$\texttt{x} \geq \texttt{value}$
`UP`, `UI`	yes	$\texttt{x} \leq \texttt{value}$
`FX`	yes	$\texttt{x} = \texttt{value}$
`BV`	no	$\texttt{x} \in \{0, 1\}$
`MI`	no	$\texttt{x} \geq -\infty$
`PL`	no	$\texttt{x} \leq \infty$
`FR`	no	$-\infty \leq \texttt{x} \leq \infty$

The "`LI`", "`UI`", or "`BV`" bounds indicators declare their variable to be an integer variable as well as defining its bounds.

10.3.7 Integer Variables

Some `BOUNDS` data lines, aside from changing a variable's bounds, declare the variable to be integral (see the above section). Alternatively, a file may contain `'MARKER'` lines in the `COLUMNS` section to declare variables integral.

`MARKER` lines contain three fields. The contents of field 1 are ignored. Field 2 must contain the string "`'MARKER'`" and field 3 contains either the string "`'INTORG'`" or the string "`'INTEND'`". (Note that the single quotes must be part of the fields.) A line with "`'INTORG'`" starts an integer section and a line with "`'INTEND'`" ends an integer section. Integer sections may not be nested. All variables mentioned inside of an integer section are integer (it does not matter whether they are mentioned outside integer sections as well).

The following MPS file excerpt includes line numbers for reference. The `COLUMNS` section contains two integer sections. The first covers the lines 3–7 and the second covers the lines 11–13. The second `MARKER` on line 5 is ignored because it tries to start an integer section from within an integer section. The variables `x2`, `x3`, and `x4` are integer variables since they are mentioned inside an integer section.

```
 1  COLUMNS
 2   x1 obj   1
 3   mm 'MARKER' 'INTORG'
 4   x2 obj   1
 5   mm 'MARKER' 'INTORG'
 6   x3 obj   1
 7   mm 'MARKER' 'INTEND'
 8   x1 row1   3
 9   x2 row1   5
10   x3 row1   5
11   mm 'MARKER' 'INTORG'
12   x4 obj 1 row1 5 row2 10
13   mm 'MARKER' 'INTEND'
```

10.3.8 Miscellaneous

The *QSopt* reader parses `REFROW` sections and “SOS” `MARKER` lines, but does not interpret the input otherwise. Thus files that use these features can be read and are compatible with the MPS format understood by *QSopt*.

A `REFROW` section contains exactly one data line. Its first and only field contains a row name.

QSopt accepts `MARKER` lines which contain the string “`'SOSORG'`” or “`'SOSEND'`” in their third field (see Fig. 10.3). `'SOSORG'` lines start an SOS section and `'SOSEND'` lines terminate an SOS section. SOS sections may not be nested. Each pair of `'SOSORG'` and `'SOSEND'` lines defines a set of variables; the set contains the variables mentioned in that section. All SOS sets defined in an MPS-format file must be disjoint.

10.3.9 MPS-format Examples

We illustrate the MPS-file format with the two examples used in the preceding section of LP format. The first example contains nine variables and six constraints; each of the variables is bounded between 0 and 1. In this small example, the entire columns can be described on single lines in the `COLUMNS` section. This example can be found in LP format on page 91.

```
NAME    Example1
OBJSENSE
  MIN
OBJNAME
  obj
ROWS
 N  obj
 E  node_0
 E  node_1
 E  node_2
 E  node_3
 E  node_4
 E  node_5
COLUMNS
  x0_1    obj    1     node_1    1     node_0    1
  x0_2    obj    1     node_2    1     node_0    1
  x0_4    obj    1     node_4    1     node_0    1
  x1_2    obj    1     node_2    1     node_1    1
  x1_5    obj    1     node_5    1     node_1    1
  x2_3    obj    1     node_3    1     node_2    1
  x3_4    obj    1     node_4    1     node_3    1
  x3_5    obj    1     node_5    1     node_3    1
  x4_5    obj    1     node_5    1     node_4    1
RHS
 RHS    node_0    2    node_1    2
 RHS    node_2    2    node_3    2
 RHS    node_4    2    node_5    2
BOUNDS
 UP BOUND    x0_1    1
 UP BOUND    x0_2    1
 UP BOUND    x0_4    1
 UP BOUND    x1_2    1
 UP BOUND    x1_5    1
 UP BOUND    x2_3    1
 UP BOUND    x3_4    1
 UP BOUND    x3_5    1
 UP BOUND    x4_5    1
ENDATA
```

The second example uses `MARKER` lines to specify that the four variables are required to take on integer values. This example can be found in LP format on page 91.

```
NAME    Example2
OBJSENSE
  MAX
OBJNAME
  obj
ROWS
 N  obj
 L  r1
 L  r2
 L  r3
COLUMNS
 MARK0qs      'MARKER'    'INTORG'
  x1    obj    4   r3    -1   r2    5   r1    1
  x2    obj    1   r3     2   r2    1   r1   -1
  x3    obj    5   r3     3   r2    3   r1   -1
  x4    obj    3   r3    -5   r2    8   r1    3
 MARK4qs      'MARKER'    'INTEND'
RHS
 RHS    r1    1
 RHS    r2    55
 RHS    r3    3
BOUNDS
 UP BOUND    x1    2
ENDATA
```

Index