# Linear Programming MT

for GAUSS™

Version 4.0

Aptech Systems, Inc.

Documentation Version: May 29, 2002

Part Number: 002952

# Contents

# Chapter 1

# Installation

## 1.1 UNIX

If you are unfamiliar with UNIX, see your system administrator or system documentation for information on the system commands referred to below. The device names given are probably correct for your system.

### 1.1.1 Download

1. Copy the `.tar.gz` file to `/tmp`.
2. Unzip the file.

   `gunzip` *appxxx*`.tar.gz`
3. `cd` to the **GAUSS** or **GAUSS Engine** installation directory. We are assuming `/usr/local/gauss` in this case.

   ```
   cd /usr/local/gauss
   ```
4. Untar the file.

   `tar xvf /tmp/`*appxxx*`.tar`

### 1.1.2 Floppy

1. Make a temporary directory.

   ```
   mkdir /tmp/workdir
   ```

2. `cd` to the temporary directory.

   ```
   cd /tmp/workdir
   ```

3. Use `tar` to extract the files.

   `tar xvf` *device_name*

   If this software came on diskettes, repeat the `tar` command for each diskette.

4. Read the README file.

   ```
   more README
   ```

5. Run the `install.sh` script in the work directory.

   ```
   ./install.sh
   ```

   The directory the files are install to should be the same as the install directory of **GAUSS** or the **GAUSS Engine**.

6. Remove the temporary directory (optional).

The following device names are suggestions. See your system administrator. If you are using Solaris 2.x, see Section 1.1.3.

| Operating System | 3.5-inch diskette | 1/4-inch tape | DAT tape |
|---|---|---|---|
| Solaris 1.x SPARC | `/dev/rfd0` | `/dev/rst8` | |
| Solaris 2.x SPARC | `/dev/rfd0a` (vol. mgt. off) | `/dev/rst12` | `/dev/rmt/1l` |
| Solaris 2.x SPARC | `/vol/dev/aliases/floppy0` | `/dev/rst12` | `/dev/rmt/1l` |
| Solaris 2.x x86 | `/dev/rfd0c` (vol. mgt. off) | | `/dev/rmt/1l` |
| Solaris 2.x x86 | `/vol/dev/aliases/floppy0` | | `/dev/rmt/1l` |
| HP-UX | `/dev/rfloppy/c20Ad1s0` | | `/dev/rmt/0m` |
| IBM AIX | `/dev/rfd0` | `/dev/rmt.0` | |
| SGI IRIX | `/dev/rdsk/fds0d2.3.5hi` | | |

### 1.1.3 Solaris 2.x Volume Management

If Solaris 2.x volume management is running, insert the floppy disk and type

```
volcheck
```

to signal the system to mount the floppy.

The floppy device names for Solaris 2.x change when the volume manager is turned off and on. To turn off volume management, become the superuser and type

```
/etc/init.d/volmgt off
```

To turn on volume management, become the superuser and type

```
/etc/init.d/volmgt on
```

## 1.2 Windows/NT/2000

### 1.2.1 Download

Unzip the `.zip` file into the **GAUSS** or **GAUSS Engine** installation directory.

### 1.2.2 Floppy

1. Place the diskette in a floppy drive.
2. Call up a DOS window
3. In the DOS window log onto the root directory of the diskette drive. For example:

```
A:<enter>
cd\<enter>
```

4. Type: **ginstall** *source_drive target_path*

| | |
|---|---|
| *source_drive* | Drive containing files to install with colon included<br><br>For example: **A:** |
| *target_path* | Main drive and subdirectory to install to without a final \<br><br>For example: **C:\GAUSS** |

A directory structure will be created if it does not already exist and the files will be copied over.

| | |
|---|---|
| *target_path***src** | source code files |
| *target_path***lib** | library files |
| *target_path***examples** | example files |

## 1.3 Differences Between the UNIX and Windows/NT/2000 Versions

- If the functions can be controlled during execution by entering keystrokes from the keyboard, it may be necessary to press *Enter* after the keystroke in the UNIX version.

- On the Intel math coprocessors used by the Windows/NT/2000 machines, intermediate calculations have 80-bit precision, while on the current UNIX machines, all calculations are in 64-bit precision. For this reason, **GAUSS** programs executed under UNIX may produce slightly different results, due to differences in roundoff, from those executed under Windows/NT/2000.

# Chapter 2

# Linear Programming



## 2.1 Introduction

This module contains procedures for solving small scale linear programming problems.

A linear programming problem is a optimization problem presented in the following typical manner:

$$
\begin{array}{lll}
(*) \quad \text{maximize:} & \displaystyle\sum_{j=1}^{n} c_j x_j & \\
\text{subject to:} & \displaystyle\sum_{j=1}^{n} a_{ij} x_j \leq b_i & (i = 1, 2, ...m) \\
& l_j \leq x_j \leq u_j & (j = 1, 2, ...n)
\end{array}
$$

where $a$, $b$, $c$, $l$ and $u$ are user-supplied vectors and matrices. The expression $c \cdot x$ is called the *objective function*, the system $\{a_i \cdot x \leq b_i\}_{i=1}^{m}$ make up the *constraints*, and the inequalities $l_j \leq x_j \leq u_j$ describe the *variable bounds*.

If the constraints in $(*)$ can be satisfied and the problem is not unbounded, then $(*)$ has an *optimal solution* and an *optimal value*. In this case, $x$ is the optimal solution and the value of the expression $c \cdot x$ at the optimal solution is the optimal value.

To solve the above problem and its variations, **LPMT** uses the two-phase standard revised simplex method with an eta factorization similar to the product form of the inverse.

## 2.2 Getting Started

**GAUSS 4.0.x+** is required to use these routines.

### 2.2.1 README Files

The file `README.lp` contains any last minute information on this module. Please read it before using the procedures in this module.

### 2.2.2 Setup

In order to use the procedures in the *LINEAR PROGRAMMING* module the `lpmt` library must be active. This is done by including `lpmt` in the **LIBRARY** statement at the top of your program:

```
library lpmt,pgraph;
```

This enables **GAUSS** to find the procedures contained in this module.

The file `lpmt.sdf` contains the structure definitions and must be "included":

```
#include lpmt.sdf;
```

The version number of each module is stored in a global variable. For the *LINEAR PROGRAMMING* module, this global is:

**_lpmt_ver** 3×1 matrix, the first element contains the major version number, the second element the minor version number, and the third element the revision number.

If you call for technical support, you may be asked for the version of your copy of this module.

## 2.3 Solving a Linear Programming Problem

The **lpmt** procedure takes two input structures and returns an output structure. The first input argument is an **LP** structure containing the required matrices, a, b, c, l, and u. The second input structure is an **LPcontrol** structure which contains control information for **lpmt**. Default values for the members of this structure are set by **lpmt** and usually won't need to be reset.

Finally, **lpmt** returns an **LPout** structure.

For example,

```
library lpmt;
#include lpmt.sdf

struct LP lp0;
lp0 = createLP;

lp0.a = { 2 -3 4 1 3,
          1 7 3 -2 1,
          5 4 -6 2 3 };

lp0.b = { 1, 1, 22 };

lp0.c = { 8, -9, 12, 4, 11 };

lp0.l = 0;
lp0.u = 1e200;


struct LPcontrol c0;
c0 = createLPcontrol;

c0.output = 1;

struct LPout out0;
out0 = lpmt(lp0,c0);

call lpmtprt(out0,c0);

output off;
```



As the above sample program illustrates, the arguments $l$ and $u$ may take on the values $+\infty$ or $-\infty$. In **lpmt**, $-\infty$ is represented by $-1e200$ and $+\infty$ by $1e200$. By setting $l = 0$ and $u = 1e200$, the variables $x_j$ are restricted to nonnegative values. Here are examples of two other ways to set up $l$ and $u$:

(1)

```
lp0.l = -1e200;
lp0.u = 50;
```

(2)

```
lp0.l = { 0, -1e200, -50, -1e200 };
lp0.u = { 1e200, 0, 50, 1e200 };
```

In (1), all variables are bounded below by $-\infty$ and above by 50.

In (2), the variables are restricted as follows:

$$\begin{aligned} x_1 &\geq 0 \\ x_2 &\leq 0 \\ -50 \leq x_3 &\leq 50 \\ -\infty \leq x_4 &\leq +\infty \end{aligned}$$

The argument $b$ is used to provide upper and/or lower bounds for the constraint expressions and, if desired, to define constraint types. Usually, though, constraint types ($\leq$, $\geq$, $=$) are defined using global variables. This is discussed next. For more details on defining $b$ see the **lpmt** function definition in Chapter 3. Please note that **lpmt** cannot handle free constraints. Do not set $b_i = \pm 1e200$ for any $i$.

### 2.3.1 The Global Control Variables

Once the arguments to **lpmt** are set up, you probably want to customize your program. Almost all aspects of the linear programming problem, including the constraint type and variable bounds, can be modified by changing the value of one or more of the control variables included in the **LPcontrol** structure. A complete list of all the members of this structure is given under in the reference section for **lpmt** in Chapter 3. Described below are some of the aspects that the user can customize and the structure member used in each case:

- To determine whether **lpmt** should solve the minimization or maximization problem set **minimize**
- The variable bounds. In the case where the user wants simply to define variables as nonnegative, nonpositive or free, the structure member **altConstraint** may be used to indicate variable types, rather than explicitly setting $l$ and $u$. (**altConstraint**)
- For constraint type ($\leq$, $\geq$, $=$) set **constraintType**. ).
- Whether to report a feasible solution only (i.e., terminate with Phase I) or return the optimal solution (i.e., continue on to Phase II) set **feasible**
- For maximum number of iterations that the algorithm is allowed to execute set **maxIterations**.
- For choice of starting value set **start**. This is useful if the user plans to solve several similar problems (e.g., problems which vary only in the $b$ vector).
- The type of output desired. The user can specify whether **lpmt** should produce output suitable for a disk file or the screen. Also, the user may customize the output with a title, variable names and header of his/her choosing. See **output**, **name**, **title**, **altNames**.

Control variables also control more advanced options of the solution process. For a brief discussion on how to control these options, see Section 2.5. The advanced options include:

- To determine tie breaking rules which are used to determine the entering variable and the leaving variable set **rule**.
- For tolerances used in determining the entering and leaving variables set **eps1**, **eps2**.
- For the tolerance used to minimize roundoff errors. (**eps3**, **constraintTol**, **tolerance**)
- For number of solutions returned set **numSolutions**. If the solution first found by **lpmt** is not unique, the user can specify how many more optimal solutions **lpmt** should attempt to find. (**numSolutions**.



**Using the Control Variables**

To use the control variables, simply assign the desired value to the selected control variable in the **GAUSS** program before calling **lpmt**. The control variables are members of an **LPcontrol** structure. You can set the members of this structure to alternate values before passing th **lpmt**. In order to ensure that members of structures are properly re-set to default values when a command file is re-run, assign the structure to its "create" procedure. In the case of the **LPcontrol** structure, set it equal to **createLPcontrol**.

The following is an example of how to solve a minimization problem with equality constraints and free variables:

```
library lpmt;
#include lpmt.sdf;

struct LP lp0;
lp0 = createLP;

lp0.a = trunc(100*rndu(20,30));
lp0.b = 100*ones(20,1);
lp0.c = ones(30,1);

struct LPcontrol c0;
c0 = createLPcontrol;

c0.altConstraint = 0;   /* All variables are free             */
c0.minimize = 2;     /* Solve minimization problem        */
```

```
c0.constraintType = 3;    /* Constraints are all equalities     */
c0.output = 1;   /* Output suitable for disk file      */
c0.name = "Y";  /* Variable name to be used in output */

output file = lp1.out reset;
call lpmtprt(lpmt(lp0,c0),c0);
output off;
```

By setting **c0.minimize** = 2, the minimum value of the objective function is computed. By setting **c0.constraintType** = 3, the constraint equations are treated as equalities. Here **c0.constraintType** is a scalar, but in general it can be a Mx1 vector where each element describes the corresponding equation type. Also note that instead of setting $lp0.l = -1e200$ and $lp0.u = 1e200$, this program uses the global variable **c0.altConstraint** to specify that the variables should be unrestricted. In this case, the values of the LP structure members $l$ and $u$ are ignored. The two methods are equivalent. The user may choose whichever is preferable.

The control variable **c0.output** has been set to specify that information generated during the iterative stages of **lpmt** should be sent to the output file `lp1.out`. The call to **lpmtprt** prints a final report to the same disk file. In general, **output** controls the output produced by the procedure **lpmt** and can take the value 0, 1 or 2, where 0 is no output, 1 is disk output (suitable for an output file) and 2 is screen output (suitable only for the DOS Compatibility Window). Final reports can be generated with either **lpmtprt** (for disk file) or **lpmtview** (for screen display). Both final report formats report the return code, the value of the objective function upon termination, the total number of iterations required by **lpmt**, final solution $x$ (with an indication of which variables are basic upon termination), the quality of the solution, the value of the constraints and the dual variables. If output is directed to the screen, the state of each constraint is also reported.

## 2.4 Example Programs

These and other example programs, `lpmt`$n$`.e`, can be found on the `examples` subdirectory.

#### EXAMPLE 1

This program solves a straightforward linear programming problem. By default, the maximization problem is solved, the variables are restricted to nonnegative values, and the constraints are all of the type $\leq$. Since **c0.output** = 2, information generated during the iterations is formatted for output to the screen. The call to **lpmt** is nested inside a call to **lpmtview**, so the final report also will be suitable only for the screen.

```
/*
**  lpmt1.e
*/

library lpmt;
#include lpmt.sdf


struct LP lp0;
lp0 = createLP;

lp0.a = { 2 -6 2 7 3 8,
          -3 -1 4 -3 1 2,
          8 -3 5 -2 0 2,
          4 0 8 7 -1 3,
          5 2 -3 6 -2 -1 };

lp0.b = { 1, 2, 4, 1, 5 };

lp0.c = { 18, -7, 12, 5, 0, 8 };

lp0.l = 0;
lp0.u = 1e200;


struct LPcontrol c0;
c0 = createLPcontrol;

c0.output = 2;
c0.name ="Y";
c0.title = "lpmt1.e";
c0.minimize = 1;

output file = lpmt1.out reset;

call lpmtview(lpmt(lp0,c0),c0);

output off;
```



**EXAMPLE 2**

A more complicated example might look like this:

```
/*
** lpmt2.e
```

```
*/

library lpmt;
#include lpmt.sdf


struct LP lp0;
lp0 = createLP;

lp0.a = { 3 1 -4 2 5 1,
          -5 4 2 -3 2 3,
           1 1 2 1 1 2 };

lp0.b = { 3, 25, 4 };

lp0.c = { -5, 2, 3, 3, 6, 1 };

lp0.l = { 0, 2, -1e200, -3, -1e200, -1e200 };
lp0.u = { 1e200, 10, 0, 3, 1e200, 1e200 };

struct LPcontrol c0;
c0 = createLPcontrol;

c0.constraintType = { 1, 1, 3 };
c0.output = 1;
c0.minimize = 1;

c0.title = "lpmt2.e";

output file = lpmt2.out reset;

call lpmtprt(lpmt(lp0,c0),c0);

output off;
```

Here **constraintType** is a 3x1 vector which indicates that the first two constraints are $\leq$ constraints and the final constraint is an equality constraint. The variables should all satisfy the inequalities:

$$\begin{aligned} x_1 &\geq 0 \\ 2 \leq x_2 &\leq 10 \\ x_3 &\leq 0 \\ -3 \leq x_4 &\leq 3 \end{aligned}$$

$x_5$ and $x_6$ are free variables.

Results of the algorithm's progress and the final solution report are sent to the output file `lp2.out`.

**EXAMPLE 3**

In this example both the primal and the dual problem are solved. The fundamental principle of linear programming states that the optimal value of the primal is equal to the optimal value of the dual problem (assuming both have optimal solutions). Then the solution to one problem is compared with the dual variables of another.



```
/*
** lpmt3.e
** This example illustrates how to solve
** both a primal and dual problem
*/

library lpmt;
#include lpmt.sdf

struct LP lp0;
lp0 = createLP;

lp0.a = { 4  0 -1  1,
          2  1  4 -1,
         -3  2  0 -8,
          1  1  1  1 };

lp0.b = { 2, 12, -31, 12 };
lp0.c = { -2, -9, -1, 6 };
lp0.l = 0;
lp0.u = 1e200;

struct LPcontrol c0;
c0 = createLPcontrol;

c0.constraintType = { 3, 2, 3, 1 };
c0.output = 1;
c0.title = "PRIMAL PROBLEM";

output file = lpmt3.out reset;

call lpmtprt(lpmt(lp0,c0),c0);


print;
print;

c0 = createLPcontrol;
```

```
c0.minimize = 1;
c0.altConstraint = { 0, 1, 0, -1 };    /*  l and u set to 0 below */
c0.constraintType = 1;
c0.output = 1;
c0.title = "DUAL PROBLEM";
c0.name = "Y";

lp0.a = lp0.a';
lp0.l = 0;
lp0.u = 0;

call lpmtprt(lpmt(lp0,c0),c0);

output off;
```

## 2.5 Advanced Topics Using lpmt

By changing the values of the tolerances **eps1**, **eps2**, **eps3**, **tolerance** and **constraintTol**, the user can affect the speed and accuracy of **lpmt**. Also, if **lpmt** is returning a return code of 5 or 13, these tolerances can be modified to encourage **lpmt** to return a more informative code.

If **lpmt** is returning a return code of 13, **eps1** or **eps2** are probably set too high. Generally, by setting **eps1** lower the number of variables from which **lpmt** chooses the variable entering the basis is increased. The more variables from which **lpmt** has to choose, the more likely it is that it will find one which doesn't cause numerical errors.

Another tolerance which the user might wish to modify is **eps3**. Briefly, **eps3** determines how well $x$, the intermediate solution determined at each iteration, should satisfy a particular expression. By modifying the value of **eps3**, the user can have some affect on how much time **lpmt** requires and on the accuracy of the final solution. In general, increasing the value of **eps3** reduces the amount of time **lpmt** requires, and decreasing **eps3** should improve the accuracy of the solution.

Two other tolerances, **tolerance** and **constraintTol**, are used to determine whether an optimal solution found during Phase I is feasible and whether the $x$ found during a particular iteration satisfies the constraints to within the user's specifications.

In solving a linear programming problem, the user may find that **lpmt** reports that the problem is infeasible, but also reports that the value of the objective function at termination is very small—i.e., less than $10^{-5}$. In this case, the user should consider increasing **tolerance** to at least as large as the value of the objective function returned by **lpmt**. This guarantees that **lpmt** will proceed to Phase II and attempt to find an optimal solution to the problem.

Due to scaling differences among constraints, the user may wish to allow differences among what it takes to satisfy those constraints. That is, a greater degree of infeasibility may be allowed in those constraints with larger coefficients. **constraintTol** can be set to a scalar, in which case all constraints are satisfied to within the same degree of accuracy, or to an Mx1 vector (M = **rows(***a***)**), in which case each constraint uses the tolerance in the corresponding element of **constraintTol**. A return code of 5 indicates that the algorithm required more iterations than allowed by the global variable **maxIterations**. If cycling is not occurring, simply increase the value of **maxIterations**. If it is suspected that cycling is occurring, change the value of **rule[2]**. Changing the rules used to choose entering and leaving variables may decrease the number of iterations required by **lpmt**. It should be noted, however, that cycling is very rare.



## 2.6 Sparse Constraint Matrices

The constraint matrix $a$ in an instance of the **LP** structure can be either dense or sparse. For very large linear programming problems with many zero elements in the constraint matrix, there are many advantages to storing this matrix in a sparse form. A common storage form is the MPS formatted file. You may also store or generate the matrix using **GAUSS** sparse functions.

### 2.6.1 MPS formatted files

If you have an MPS formatted file, the *mps* function returns an instance of an **LP** structure with the model matrices defined including a sparse constraint matrix. The input to this function is the name of the file. For example,

```
library lpmt;
#include lpmt.sdf

struct LP lp0;
lp0 = mps("adlittle.mps");

struct LPcontrol c0;
c0 = createLPcontrol;

struct LPout out1;
out1 = lpmt(lp0,c0);

call lpmtprt(out1,c0);
```

A keyword function is also available that generates the analysis of the MPS file interactively. From the **GAUSS** command line, type

```
>> library lpmt;
>> solveLP adlittle;
```

This analyzes a linear programming problem stored in a file name *adlittle.mps* and prints results to a file name *adlittle.out*.

### 2.6.2 Alternate Sparse Methods

The constraint matrix can also be stored or generated using **GAUSS** functions. The **GAUSS** function **sparseFD** takes a matrix containing three columns, the element value, its row, and its column, of the nonzero elements of the constraint matrix, and returns a **GAUSS** sparse matrix. For example,

```
ap = { 1 1 2,
       1 1 3,
       1 1 4,
       1 2 5,
       1 2 6,
       1 2 7,
       1 3 8,
       1 3 9,
       1 3 10,
      -1 4 2,
      -1 4 5,
      -1 4 8,
      -1 4 11,
      -1 4 12,
      -1 4 13,
       1 4 1,
       1 5 2,
       1 5 3,
       1 5 4,
       1 5 5,
       1 5 6,
       1 5 7,
       1 5 8,
       1 5 9,
       1 5 10,
       1 5 11,
       1 5 12,
       1 5 13,
       1 6 1,
      -1 6 11,
       2 7 2,
       2 7 3,
```

```
   2 7 4,
 1.2 7 5,
 1.2 7 6,
 1.2 7 7,
 0.7 7 8,
 0.7 7 9,
 0.7 7 20,
   4 8 11,
 2.5 8 12 };

b = { 2754, 850, 855, 0, 5000, 2247, 2440, 4160 };

c = { 72,
       11,   24,   88,
      -13,    0,   64,
      -27,  -14,   50,
       44,
        1,
      -46 };

l = 0;

u = { 1e200,
      1e200,   357,   500,
      1e200,   197,   130,
      1e200,    39,   170,
       1598,
        405,
       1761 };


struct LP lp0;
lp0.a = sparseFP(ap,8,13);
lp0.b = b;
lp0.c = c;
lp0.l = l;
lp0.u = u;
```

The sparse matrix *ap* can be stored as a **GAUSS** matrix file on disk and retrieved whenever necessary.

## 2.7 References

Chvatal, Vašek 1983. *Linear Programming.* New York: W. H. Freeman and Company.

# Chapter 3

# Linear Programming Reference

## Library

`lpmt`

## Purpose

Formats and prints the output from **lpmt**. This printout is suitable for output to a disk file.

## Format

*out0* = **lpmtprt(***out0*,*c0***);**

## Input

*out0* an instance of an **LPout** structure with the following members,

*out0.Solution* $(N + M) \times 1$ vector containing either (1) an optimal solution to the original problem, or (2) the x values which minimize the sum of the infeasiblities or (3), the last solution found before it was determined that the problem is unbounded or that the algorithm was unable to continue. The last $M$ elements contain the values of the slack variables.

*out0.optimumValue* scalar, the value of the objective upon termination of **lpmt**. This may be the optimal value, the minimum sum of the infeasiblities, or the largest value found before it was determined that the problem was unbounded or that the algorithm was unable to continue.

*out0.returncode* scalar, return code:

**0** An optimal solution was found

**1** The problem is unbounded

**2** The problem is infeasible

**5** Maximum number of iterations exceeded. Cycling may be occurring.

**13** Algorithm unable to find a suitable variable to enter the basis. Either set eps1 or eps2 lower, or change rule[1] to another value.

If the return code is negative, than the program terminated in Phase I.

*out0.constraintValues* $M \times 1$ vector. The value of each constraint upon termination of **lpmt**.

*out0.basis* $M \times 1$ vector containing the indices of the variables in the final basis. Normally, the indices returned in basis will be in the range $1 - (M + N)$, but occasionally, however, artificial variables will persist in the basis. In this case, indices will be in the range $(1 - (2 * M + N)$.

*out0.dual* $M \times 1$ vector, the dual variables.

*out0.numIters* $2 \times 1$ vector containing the number of iterations required for each phase of the simplex algorithm. The first and second elements correspond to the number of iterations required by Phase I and Phase II, respectively.

*out0.quality* scalar, reports the quality of the final solution. Quality is judged to be:

**1** POOR
**2** FAIR
**3** GOOD
**4** EXCELLENT

*out0.state* $M \times 1$ vector containing the state of each constraint. The states are:

**-4** Equality constraint has been violated below.
**-3** Equality constraint has been violated above.
**-2** Constraint violates its lower bound
**-1** Constraint violates its upper bound
**0** Constraint strictly between its two bounds
**1** Constraint is at its lower bound.
**2** Constraint is at its upper bound.
**3** Equality constraint is satisfied.

*out0.optSolutions* If numSolutions is greater than 1, this global will be an $(N + M) \times P$ matrix containing $P$ optimal solutions. Otherwise, optSolutions will be set to 0.

*out0.numIters* $2 \times 1$ vector containing the number of iterations required for each phase of the simplex algorithm. The first and second elements correspond to the number of iterations required by Phase I and Phase II, respectively.

*C0* an instance of an **LPcontrol** structure with the following members,

*c0.constraintType* $M \times 1$ vector or scalar used to describe each equation type. The values for each equation type are:

**1** Corresponding constraint is $\leq$.
**2** Corresponding constraint is $\geq$.
**3** Corresponding constraint is $=$.

The default is 1. If constraintType is a scalar, it will be assumed that all constraints are of the same type.

*c0.constraintTol* scalar, tolerances used to determine whether a constraint has been violated or not. This can be a scalar, or $M \times 1$ vector. Default = $10^{-8}$.

*c0.eps1* scalar. This is the smallest value around which a pivot will be performed. If during any iteration, a value exceeds **eps1**, in absolute value, it is a possible candidate for entering the basis. Default: $10^{-5}$.

*c0.eps2* scalar. The algorithm will not divide by a number smaller than this. Default: $10^{-8}$.

*c0.eps3* scalar. This is used to determine how often to refactor the basis. Roughly, if abs(ax-b) is greater than eps3 in any element, in any iteration, the basis will be refactored immediately. Setting this value too low will slow down the algorithm speed, since refactoring is the most time consuming portion of any iteration. If abs(b-ax) never exceeds **eps3**, then the basis will be refactored when either the eta file gets too large for available memory, or when scanning the file becomes too time consuming. Default = $10^{-6}$.

*c0.tolerance* scalar, tolerance used to determine whether a solution is feasible or not. If sum of artificial variables at end of phase I does not exceed fstol, then solution at that point is considered feasible. This may also be a Mx1 vector if you want different feasibilities for each constraint. Default = $10^{-13}$.

*c0.maxIterations* scalar, the maximum number of iterations the simplex algorithm will iterate during either phase. Default = 1000.

*c0.minimize* If 1, the minimum value of the objective function will be calculated. If 2 (default), the maximum will be calculated.

*c0.name* string, Variable name to be used for output purposes. Default = **X**.

*c0.rule* $2 \times 1$ vector. The first element determines which tie breaking rule will be used for the entering variable. The second element determines which rule will be used for the leaving variable. **rule[1]** specifies the tie breaking rule for the entering variable and can have the following values:

**1** Smallest subscript rule.

**2** Largest coefficient rule.

**3** Largest increase rule.

**4** A random selection is made.

**rule[2]** specifies the tie breaking rule for the leaving variable and can have the following values:

**1** Smallest subscript rule.

**2** Lexicographic rule. This rule is very time-consuming and memory-intensive.

**3** A random selection is made.

The rule used to choose the entering variable can have an effect on the number of iterations required before the algorithm finds an optimal solution. Unfortunately, no general rules can be given about

which rule to use. Using the smallest subscript rule for the leaving variable guarantees that off-optimal cycling does not occur. This rule, however, may force the algorithm to go through more iterations than might otherwise be necessary. Default = { 2, 1 }.

*c0.numSolutions* scalar, the number of optimal solutions that **lpmt** should attempt to find. Default = 1.

*c0.feasible* If 1, only a feasible solution will be returned. If 2, an optimal solution will be returned. If you want to input a feasible solution, set **start** to this solution. If feasible = 2, and this solution is feasible, it will be used to start Phase II of the algorithm. Default = 2.

*c0.start* $(N + M) \times 1$ or $(N + M) \times 2$ vector. If start is $(N + M) \times 1$, then it will be used to initialize Phase I. The first $N$ elements are the values of the original variables, and the last $M$ variables are values of slack variables. If start is $(N + M) \times 2$, the first column should contain a solution with which to initilize the algorithm and the second column should contain a 1 if the corresponding first column element is basic, and a zero otherwise. In either case, the initial solution must be feasible with respect to **l** and **u**, but need not be feasible with respect to the constraint equations. Default = 0.

*c0.altConstraint* scalar, or $N \times 1$ vector. This global may be used an alternative to setting input arguments l and u if variables are set to be non-negative, non-positive, or free. Values for this global are:

**-1** Corresponding variable is nonpositive.

**0** Corresponding variable is unrestricted (or free).

**1** Corresponding variable is nonnegative.

If altConstraint is a scalar, it will be assumed that all variables have the same restrictions. If this global has been set to a value other than its default, **l** and **u** will be ignored. In this case, simply pass in 0 for these two input arguments. Default = 2.

*c0.altNames* $N \times 1$ character vector, alternate variable names to be used for printout purposes. These names will be used in the iterations printout and in the final report. By default, the iterations report will use numbers to indicate variables and the final solution report will use the name in **altNames**.

*c0.output* scalar, determines writing to the screen.

**0** Nothing is written.

**1** Iteration information printed to screen.

**2** prints output in DOS compatibility window

*c0.title* string, message printed at the top of the screen and output device by **lpmtprt** and **lpmtview**. By default, a generic title is printed.

*c0.seed* scalar, random number seed.

*c0.scale* scalar, if nonzero, the input matrices will be scaled.

## Output

*out0* an instance of an **LPout** structure identical to the **LPout** instance passed in the first input argument.

## Source

`lpmtprt.src`

## Library

`lpmt`

## Purpose

Creates a screen display of the final results returned from **lpmt**. This display allows the user to page through the values of constraints upon termination, the dual variables and the final solution. The state of each constraint is reported and slack variables are marked.

## Format

*out0* = **lpmtview(***out0*,*c0***);**

## Input

*out0* an instance of an **LPout** structure with the following members,

*out0.Solution* $(N + M) \times 1$ vector containing either (1) an optimal solution to the original problem, or (2) the x values which minimize the sum of the infeasiblities or (3), the last solution found before it was determined that the problem is unbounded or that the algorithm was unable to continue. The last $M$ elements contain the values of the slack variables.

*out0.optimumValue* scalar, the value of the objective upon termination of **lpmt**. This may be the optimal value, the minimum sum of the infeasiblities, or the largest value found before it was determined that the problem was unbounded or that the algorithm was unable to continue.

*out0.returncode* scalar, return code:

**0** An optimal solution was found

**1** The problem is unbounded

**2** The problem is infeasible

**5** Maximum number of iterations exceeded. Cycling may be occurring.

**13** Algorithm unable to find a suitable variable to enter the basis. Either set eps1 or eps2 lower, or change rule[1] to another value.

If the return code is negative, than the program terminated in Phase I.

*out0.constraintValues* $M \times 1$ vector. The value of each constraint upon termination of **lpmt**.

*out0.basis* $M \times 1$ vector containing the indices of the variables in the final basis. Normally, the indices returned in basis will be in the range $1 - (M + N)$, but occasionally, however, artificial variables will persist in the basis. In this case, indices will be in the range $(1 - (2 * M + N)$.

*out0.dual* $M \times 1$ vector, the dual variables.

*out0.numIters* $2 \times 1$ vector containing the number of iterations required for each phase of the simplex algorithm. The first and second elements correspond to the number of iterations required by Phase I and Phase II, respectively.

*out0.quality* scalar, reports the quality of the final solution. Quality is judged to be:

**1** POOR
**2** FAIR
**3** GOOD
**4** EXCELLENT

*out0.state* $M \times 1$ vector containing the state of each constraint. The states are:

**-4** Equality constraint has been violated below.
**-3** Equality constraint has been violated above.
**-2** Constraint violates its lower bound
**-1** Constraint violates its upper bound
**0** Constraint strictly between its two bounds
**1** Constraint is at its lower bound.
**2** Constraint is at its upper bound.
**3** Equality constraint is satisfied.

*out0.optSolutions* If numSolutions is greater than 1, this global will be an $(N + M) \times P$ matrix containing $P$ optimal solutions. Otherwise, optSolutions will be set to 0.

*out0.numIters* $2 \times 1$ vector containing the number of iterations required for each phase of the simplex algorithm. The first and second elements correspond to the number of iterations required by Phase I and Phase II, respectively.

*C0* an instance of an **LPcontrol** structure with the following members,

*c0.constraintType* $M \times 1$ vector or scalar used to describe each equation type. The values for each equation type are:

**1** Corresponding constraint is $\leq$.
**2** Corresponding constraint is $\geq$.
**3** Corresponding constraint is $=$.

The default is 1. If constraintType is a scalar, it will be assumed that all constraints are of the same type.

*c0.constraintTol* scalar, tolerances used to determine whether a constraint has been violated or not. This can be a scalar, or $M \times 1$ vector. Default = $10^{-8}$.

*c0.eps1* scalar. This is the smallest value around which a pivot will be performed. If during any iteration, a value exceeds **eps1**, in absolute value, it is a possible candidate for entering the basis. Default: $10^{-5}$.

*c0.eps2* scalar. The algorithm will not divide by a number smaller than this. Default: $10^{-8}$.

*c0.eps3* scalar. This is used to determine how often to refactor the basis. Roughly, if abs(ax-b) is greater than eps3 in any element, in any iteration, the basis will be refactored immediately. Setting this value too low will slow down the algorithm speed, since refactoring is the most time consuming portion of any iteration. If abs(b-ax) never exceeds **eps3**, then the basis will be refactored when either the eta file gets too large for available memory, or when scanning the file becomes too time consuming. Default = $10^{-6}$.

*c0.tolerance* scalar, tolerance used to determine whether a solution is feasible or not. If sum of artificial variables at end of phase I does not exceed fstol, then solution at that point is considered feasible. This may also be a Mx1 vector if you want different feasibilities for each constraint. Default = $10^{-13}$.

*c0.maxIterations* scalar, the maximum number of iterations the simplex algorithm will iterate during either phase. Default = 1000.

*c0.minimize* If 1, the minimum value of the objective function will be calculated. If 2 (default), the maximum will be calculated.

*c0.name* string, Variable name to be used for output purposes. Default = **X**.

*c0.rule* $2 \times 1$ vector. The first element determines which tie breaking rule will be used for the entering variable. The second element determines which rule will be used for the leaving variable. **rule[1]** specifies the tie breaking rule for the entering variable and can have the following values:

**1** Smallest subscript rule.

**2** Largest coefficient rule.

**3** Largest increase rule.

**4** A random selection is made.

**rule[2]** specifies the tie breaking rule for the leaving variable and can have the following values:

**1** Smallest subscript rule.

**2** Lexicographic rule. This rule is very time-consuming and memory-intensive.

**3** A random selection is made.

The rule used to choose the entering variable can have an effect on the number of iterations required before the algorithm finds an optimal solution. Unfortunately, no general rules can be given about which rule to use. Using the smallest subscript rule for the leaving variable guarantees that off-optimal cycling does not occur. This rule, however, may force the algorithm to go through more iterations than might otherwise be necessary. Default = { 2, 1 }.

*c0.numSolutions* scalar, the number of optimal solutions that **lpmt** should attempt to find. Default = 1.

*c0.feasible* If 1, only a feasible solution will be returned. If 2, an optimal solution will be returned. If you want to input a feasible solution, set **start** to this solution. If feasible = 2, and this solution is feasible, it will be used to start Phase II of the algorithm. Default = 2.

*c0.start* $(N + M) \times 1$ or $(N + M) \times 2$ vector. If start is $(N + M) \times 1$, then it will be used to initialize Phase I. The first $N$ elements are the values of the original variables, and the last $M$ variables are values of slack variables. If start is $(N + M) \times 2$, the first column should contain a solution with which to initilize the algorithm and the second column should contain a 1 if the corresponding first column element is basic, and a zero otherwise. In either case, the initial solution must be feasible with respect to **l** and **u**, but need not be feasible with respect to the constraint equations. Default = 0.

*c0.altConstraint* scalar, or $N \times 1$ vector. This global may be used an alternative to setting input arguments l and u if variables are set to be non-negative, non-positive, or free. Values for this global are:

**-1** Corresponding variable is nonpositive.

**0** Corresponding variable is unrestricted (or free).

**1** Corresponding variable is nonnegative.

If altConstraint is a scalar, it will be assumed that all variables have the same restrictions. If this global has been set to a value other than its default, **l** and **u** will be ignored. In this case, simply pass in 0 for these two input arguments. Default = 2.

*c0.altNames* $N \times 1$ character vector, alternate variable names to be used for printout purposes. These names will be used in the iterations printout and in the final report. By default, the iterations report will use numbers to indicate variables and the final solution report will use the name in **altNames**.

*c0.output* scalar, determines writing to the screen.

**0** Nothing is written.

**1** Iteration information printed to screen.

**2** prints output in DOS compatibility window

*c0.title* string, message printed at the top of the screen and output device by **lpmtprt** and **lpmtview**. By default, a generic title is printed.

*c0.seed* scalar, random number seed.

*c0.scale* scalar, if nonzero, the input matrices will be scaled.

## Output

*out0* an instance of an **LPout** structure identical to the **LPout** instance passed in the first input argument.

## Source

`lpmtprt.src`

## Library

lpmt

## Purpose

Computes the optimal value of a linear objective function subject to linear inequality constraints and bounds on variables. The problem typically is of the form:

$$\begin{aligned} \text{maximize:} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to:} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad (i = 1, 2, ...m) \\ & l_j \leq x_j \leq u_j \qquad (j = 1, 2, ...n) \end{aligned}$$

## Format

*out0* = **lpmt(***lp0*,*c0***);**

## Input

*lp0* an instance of an **LP** structure with the following members,

*lp0.a* $M \times N$ dense or sparse matrix of constraint coefficients The problem should not be supplied with slack variables.

*lp0.b* $M \times 1$ vector or $M \times 2$ matrix. If *lp0.b* is $M \times 2$, the constraint expressions are bounded below by the first column and above by the second column. That is,

$$lp0.b_{i1} \leq \sum_{j=1}^{n} lp0.a_{ij} x_j \leq lp0.b_{i2} \quad (i = 1, 2, ...m)$$

This format can be used to generate all three constraint types. For example:

$$3x_1 + 4x_2 - 13x_3 \geq 24$$

is equivalent to:

$$24 \leq 3x_1 + 4x_2 - 13x_3 \leq 1e200$$

Note the use of 1e200 for $+\infty$. This is also used below in describing variable ranges.

*lp0.c* $N \times 1$ vector containing the coefficients of the objective function.

*lp0.l* $N \times 1$ vector or a scalar, containing the lower bounds of $x$. Use $-1e200$ for $-\infty$. If *lp0.l* is a scalar, it is assumed that all elements of the solution have the same lower bound.

*lp0.u* $N \times 1$ vector or a scalar, containing the upper bounds of the solution. Use 1e200 for $+\infty$. If *lp0.u* is a scalar, it is assumed that all elements of the solution have the same upper bound.

*c0* an instance of an **LPcontrol** structure with members,

*c0.constraintType* $M \times 1$ vector or scalar used to describe each equation type. The values for each equation type are:

**1** Corresponding constraint is $\leq$.

**2** Corresponding constraint is $\geq$.

**3** Corresponding constraint is $=$.

The default is 1. If constraintType is a scalar, it will be assumed that all constraints are of the same type.

*c0.constraintTol* scalar, tolerances used to determine whether a constraint has been violated or not. This can be a scalar, or $M \times 1$ vector. Default = $10^{-8}$.

*c0.eps1* scalar. This is the smallest value around which a pivot will be performed. If during any iteration, a value exceeds **eps1**, in absolute value, it is a possible candidate for entering the basis. Default: $10^{-5}$.

*c0.eps2* scalar. The algorithm will not divide by a number smaller than this. Default: $10^{-8}$.

*c0.eps3* scalar. This is used to determine how often to refactor the basis. Roughly, if abs(ax-b) is greater than eps3 in any element, in any iteration, the basis will be refactored immediately. Setting this value too low will slow down the algorithm speed, since refactoring is the most time consuming portion of any iteration. If abs(b-ax) never exceeds **eps3**, then the basis will be refactored when either the eta file gets too large for available memory, or when scanning the file becomes too time consuming. Default = $10^{-6}$.

*c0.tolerance* scalar, tolerance used to determine whether a solution is feasible or not. If sum of artificial variables at end of phase I does not exceed fstol, then solution at that point is considered feasible. This may also be a Mx1 vector if you want different feasibilities for each constraint. Default = $10^{-13}$.

*c0.maxIterations* scalar, the maximum number of iterations the simplex algorithm will iterate during either phase. Default = 1000.

*c0.minimize* If 1, the minimum value of the objective function will be calculated. If 2 (default), the maximum will be calculated.

*c0.name* string, Variable name to be used for output purposes. Default = **X**.

*c0.rule* $2 \times 1$ vector. The first element determines which tie breaking rule will be used for the entering variable. The second element

determines which rule will be used for the leaving variable. **rule[1]** specifies the tie breaking rule for the entering variable and can have the following values:

**1** Smallest subscript rule.

**2** Largest coefficient rule.

**3** Largest increase rule.

**4** A random selection is made.

**rule[2]** specifies the tie breaking rule for the leaving variable and can have the following values:

**1** Smallest subscript rule.

**2** Lexicographic rule. This rule is very time-consuming and memory-intensive.

**3** A random selection is made.

The rule used to choose the entering variable can have an effect on the number of iterations required before the algorithm finds an optimal solution. Unfortunately, no general rules can be given about which rule to use. Using the smallest subscript rule for the leaving variable guarantees that off-optimal cycling does not occur. This rule, however, may force the algorithm to go through more iterations than might otherwise be necessary. Default = { 2, 1 }.

*c0.numSolutions* scalar, the number of optimal solutions that **lpmt** should attempt to find. Default = 1.

*c0.feasible* If 1, only a feasible solution will be returned. If 2, an optimal solution will be returned. If you want to input a feasible solution, set **start** to this solution. If feasible = 2, and this solution is feasible, it will be used to start Phase II of the algorithm. Default = 2.

*c0.start* $(N + M) \times 1$ or $(N + M) \times 2$ vector. If start is $(N + M) \times 1$, then it will be used to initialize Phase I. The first $N$ elements are the values of the original variables, and the last $M$ variables are values of slack variables. If start is $(N + M) \times 2$, the first column should contain a solution with which to initilize the algorithm and the second column should contain a 1 if the corresponding first column element is basic, and a zero otherwise. In either case, the initial solution must be feasible with respect to **l** and **u**, but need not be feasible with respect to the constraint equations. Default = 0.

*c0.altConstraint* scalar, or $N \times 1$ vector. This global may be used an alternative to setting input arguments l and u if variables are set to be non-negative, non-positive, or free. Values for this global are:

**-1** Corresponding variable is nonpositive.

**0** Corresponding variable is unrestricted (or free).

**1** Corresponding variable is nonnegative.

If altConstraint is a scalar, it will be assumed that all variables have the same restrictions. If this global has been set to a value other than its default, **l** and **u** will be ignored. In this case, simply pass in 0 for these two input arguments. Default = 2.

*c0.altNames* $N \times 1$ character vector, alternate variable names to be used for printout purposes. These names will be used in the iterations printout and in the final report. By default, the iterations report will use numbers to indicate variables and the final solution report will use the name in **altNames**.

*c0.output* scalar, determines writing to the screen.

**0** Nothing is written.

**1** Iteration information printed to screen.

**2** prints output in DOS compatibility window

*c0.title* string, message printed at the top of the screen and output device by **lpmtprt** and **lpmtview**. By default, a generic title is printed.

*c0.seed* scalar, random number seed.

*c0.scale* scalar, if nonzero, the input matrices will be scaled.

## Output

*out0* an instance of an **LPout** structure with members

*out0.Solution* $(N + M) \times 1$ vector containing either (1) an optimal solution to the original problem, or (2) the x values which minimize the sum of the infeasiblities or (3), the last solution found before it was determined that the problem is unbounded or that the algorithm was unable to continue. The last $M$ elements contain the values of the slack variables.

*out0.optimumValue* scalar, the value of the objective upon termination of **lpmt**. This may be the optimal value, the minimum sum of the infeasiblities, or the largest value found before it was determined that the problem was unbounded or that the algorithm was unable to continue.

*out0.returncode* scalar, return code:

**0** An optimal solution was found

**1** The problem is unbounded

**2** The problem is infeasible

**5** Maximum number of iterations exceeded. Cycling may be occurring.

**13** Algorithm unable to find a suitable variable to enter the basis. Either set eps1 or eps2 lower, or change rule[1] to another value.

If the return code is negative, than the program terminated in Phase I.

*out0.constraintValues* $M \times 1$ vector. The value of each constraint upon termination of **lpmt**.

*out0.basis* $M \times 1$ vector containing the indices of the variables in the final basis. Normally, the indices returned in basis will be in the range $1 - (M + N)$, but occasionally, however, artificial variables will persist in the basis. In this case, indices will be in the range $(1 - (2 * M + N)$.

*out0.dual* $M \times 1$ vector, the dual variables.

*out0.numIters* $2 \times 1$ vector containing the number of iterations required for each phase of the simplex algorithm. The first and second elements correspond to the number of iterations required by Phase I and Phase II, respectively.

*out0.quality* scalar, reports the quality of the final solution. Quality is judged to be:

**1** POOR
**2** FAIR
**3** GOOD
**4** EXCELLENT

*out0.state* $M \times 1$ vector containing the state of each constraint. The states are:

**-4** Equality constraint has been violated below.
**-3** Equality constraint has been violated above.
**-2** Constraint violates its lower bound
**-1** Constraint violates its upper bound
**0** Constraint strictly between its two bounds
**1** Constraint is at its lower bound.
**2** Constraint is at its upper bound.
**3** Equality constraint is satisfied.

*out0.optSolutions* If numSolutions is greater than 1, this global will be an $(N + M) \times P$ matrix containing $P$ optimal solutions. Otherwise, optSolutions will be set to 0.

*out0.numIters* $2 \times 1$ vector containing the number of iterations required for each phase of the simplex algorithm. The first and second elements correspond to the number of iterations required by Phase I and Phase II, respectively.

## Remarks

By default, **lpmt** solves the problem:

$$\begin{aligned} \text{maximize:} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to:} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad (i = 1, 2, ...m) \\ & x_j \geq 0 \qquad (j = 1, 2, ...n) \end{aligned}$$

Please note that **lpmt** cannot handle free constraints. Do not set $b_i = \pm 1e200$ for any $i$.

## Example

```
library lpmt;
#include lpmt.sdf

struct LP lp0;
lp0 = createLP;

lp0.a = { 1 -4 3 3,
          1 3 -1 1,
          1 2 3 2,
          1 3 -2 1 };

lp0.b = { 2, -2, 3, -3 };

lp0.c = { 3, 1, 4, 2 };

struct LPcontrol c0;
c0 = createLPcontrol;

c0.title = "THE PRIMAL PROBLEM";
c0.altConstraint = { 0, 0, 1, 1 };
c0.minimize = 1;

call lpmtprt(lpmt(lp0,c0),c0);

struct LP lp1;
lp1 = createLP;

lp1.a = lp0.a';
lp1.b = lp0.b;
lp1.c = lp0.c;
lp1.u = 1e200;

struct LPcontrol c1;
c1 = createLPcontrol;
```

```
c1.title = "THE DUAL PROBLEM";
c1.constraintType = { 3, 3, 2, 2 };
c1.minimize = 2;

call lpmtprt(lpmt(lp1,c1),c1);
```

## Source

lpmt.src

## Library

lpmt

## Purpose

## Format

*lp0* = **mps(***s0***);**

## Input

| | |
|---|---|
| *s0* | string, name of file in MPS format |

## Output

| | |
|---|---|
| *lp0* | an instance of an **LP** structure of |

## Example

```
library lpmt;
#include lpmt.sdf

struct LP lp0;
lp0 = mps("adlittle.mps");

struct LPcontrol c0;
c0 = createLPcontrol;

c0.title = "adlittle problem";

output file = lpmt8.out reset;

struct LPout out1;
out1 = lpmt(lp0,c0);

call lpmtprt(out1,c0);

output off;
```

## Source

lpmt.src

## Library

lpmt

## Purpose

## Format

**solveLP mpsfname outfname**

## Input

*mpsfname* string, name of file in mps format

*outfname* string, name of output file (optional)

## Remarks

If the file extension is omitted from the first argument, an .mps extension is assumed. If the second argument is omitted entirely, an output file is generated with the extension .out appended.

This is a keyword procedure designed to be used interatively, though it could be used from a command file as well. For example, typing the following:

```
>> library lpmt;
>> solveLP adlittle;
```

solves a linear programming problem defined in mps format in a file named *adlittle.mps* the output of which is printed to a file named *adlittle.out*.

## Source

lpmt.src

# Index

## V



## W